SCIENCE CHINA
Information Sciences

• REVIEW •

April 2024, Vol. 67, Iss. 4, 141101:1–141101:43
https://doi.org/10.1007/s11432-022-3803-9

When debugging encounters artificial intelligence: state of the art and open challenges

Yi SONG[1†], Xiaoyuan XIE[1*†] & Baowen XU[2*]

[1]*School of Computer Science, Wuhan University, Wuhan* 430072*, China;*
[2]*State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing* 210023*, China*

Received 12 July 2022/Revised 21 September 2022/Accepted 15 March 2023/Published online 21 February 2024

Abstract Both software debugging and artificial intelligence techniques are hot topics in the current field of software engineering. Debugging techniques, which comprise fault localization and program repair, are an important part of the software development lifecycle for ensuring the quality of software systems. As the scale and complexity of software systems grow, developers intend to improve the effectiveness and efficiency of software debugging via artificial intelligence (artificial intelligence for software debugging, AI4SD). On the other hand, many artificial intelligence models are being integrated into safety-critical areas such as autonomous driving, image recognition, and audio processing, where software debugging is highly necessary and urgent (software debugging for artificial intelligence, SD4AI). An AI-enhanced debugging technique could assist in debugging AI systems more effectively, and a more robust and reliable AI approach could further guarantee and support debugging techniques. Therefore, it is important to take AI4SD and SD4AI into consideration comprehensively. In this paper, we want to show readers the path, the trend, and the potential that these two directions interact with each other. We select and review a total of 165 papers in AI4SD and SD4AI for answering three research questions, and further analyze opportunities and challenges as well as suggest future directions of this cross-cutting area.

Keywords software debugging, fault localization, program repair, artificial intelligence, machine learning

1 Introduction

Faults (also known as bugs, errors, defects, and flaws) are growing increasingly common as the software industry expands, and the losses they cause have drawn a lot of attention from software practitioners. The global cost of localizing and removing faults from software systems has risen to $312 billion annually in 2013 [1]. Nevertheless, according to recent research, there can be far more software faults in the world than we will likely ever know about [2]. The software has eaten the world, according to Silicon Valley's entrepreneur Marc Andreessen and researcher Monperrus, but each bite comes with faults [3], which further exacerbates people's anxieties about software quality.

To assure and improve software quality, developers typically employ testing and debugging techniques iteratively throughout the software lifecycle. During software testing, a failure is detected when the actual output differs from the expected result, indicating that the current program has at least one fault [4–7]. The dynamic information about failure(s) collected during testing will be delivered to the debugging process [8], where the potential fault(s) will be localized (fault localization, FL) and fixed (program repair, PR) successively in order to restore the normal function of the program under test. Traditional FL techniques such as setting breakpoints, printing statements, and monitoring variables require a long time yet are still error-prone, whereas traditional PR techniques like manual code inspection rely on the developer's development experience and domain knowledge heavily [9]. To tackle these challenges, a significant number of studies have tried to utilize efficient artificial intelligence (AI) models to improve the effectiveness of FL and PR techniques [10–12]. The mentioned effort, which aims to reduce the labor cost and time cost of the debugging process, has yielded promising results in recent years.

* Corresponding author (email: xxie@whu.edu.cn, bwxu@nju.edu.cn)
† Song Y and Xie X Y have the same contribution to this work, and are co-first authors.

Although AI techniques are becoming increasingly significant in supporting software debugging activities, they are also a sort of software that can contain faults. With the accumulation of massive volume of data and improvements in hardware computing power, AI techniques have been widely implemented in daily life and industrial development including image processing, audio recognition, and even safety-critical fields such as autonomous driving and aerospace. Once AI systems fail, the consequences are typically disastrous since AI applications often exhibit erroneous or unexpected behaviors which can lead to dangerous situations [13]. Evidently, debugging AI systems is critical and urgent, thus studies on this topic have sparked the interest of both academia and industry, and a growing number of researchers are considering how to assure and improve the quality of AI systems.

When artificial intelligence and software debugging collide, two distinct research directions emerge: artificial intelligence for software debugging (AI4SD) and software debugging for artificial intelligence (SD4AI). The former involves incorporating AI techniques into the traditional software debugging process; i.e., AI is used as an auxiliary tool for SD. The latter is to treat AI models as debugging objects so as to localize and repair potential faults that reside in them. In our opinion, comprehensively taking AI4SD and SD4AI into account is non-trivial and the reason partly lies in two points.

• Techniques in the fields of AI4SD and SD4AI could be further improved by each other. An AI-enhanced debugging technique could assist in debugging AI systems more effectively, and a more robust and reliable AI approach could further guarantee and support traditional debugging techniques. For example, a community workshop themed on the intersection of AI and SE (software engineering) was held at ASE'19 (the 34th IEEE/ACM International Conference on Automated Software Engineering), and the participants jointly pointed out that debugging AI-based systems is essential for their interpretability, and an AI-assisted debugging tool to traditional fault localization or repair tasks will be also beneficial from high interpretability [14].

• Despite the encouraging amount of prior literature reviews on a similar topic, they mainly focus on the intersection of AI techniques and software fault prediction [15–17], software testing [18–24], or other tasks in software engineering [25–30]. Their research objects mainly involve AI techniques' application in other software engineering activities, or software engineering practices in the lifecycle of AI systems, without being absorbed in the intersection of artificial intelligence techniques and software debugging (i.e., fault localization and program repair). Besides, although there are some studies emerging recently that involve both SD and AI, they tended to be domain-specific or just summarized some opportunities and challenges based on materials from open-source platforms (e.g., StackOverflow and GitHub) [31–35]. A comprehensive and general literature review on the intersection of AI and SD is still a gap.

In this paper, we first collect a total of 165 primary studies in the fields of AI4SD (131 papers) and SD4AI (34 papers), then analyze opportunities, challenges, and trends as well as suggest future directions of the intersection of AI and SD, to provide a big picture of this hybrid domain[1)].

The rest of this paper is organized as follows. We provide the research background, as well as outline the general workflows of AI4SD and SD4AI in Section 2. We propose our research questions, report on the paper selection process, and analyze the distribution of research popularity in Section 3. Collected papers in the fields of AI4SD and SD4AI are classified and summarized in Sections 4 and 5, respectively. The widely-used experimental configuration is listed in Section 6. Finally, we identify potential future research directions in the intersection of AI and SD in Section 7 and conclude this work in Section 8.

2 Background

The research background, mainstream techniques, and challenges of AI4SD and SD4AI are briefly discussed in this section. The abbreviations used in this paper with high frequency are shown in Table 1.

2.1 Artificial intelligence for software debugging

When a software system fails, a developer has to troubleshoot the root cause(s) of the failure(s) before trying to fix it (them). As for fault localization, developers typically establish linkages between textual documents and root causes depending on the given bug report [36], which records error descriptions,

1) Because the term "artificial intelligence" is too broad to be dived deeply, we only focus on machine learning, deep learning, and some other popular AI techniques (such as evolutionary algorithms and constraint solving). And the scope of debugging is limited to fault localization and program repair in this paper.

Table 1 Frequently used abbreviations in this paper

Abbreviation	Meaning	Abbreviation	Meaning
DNN	Deep neural network	CNN	Convolutional neural network
MLP	Multi-layer perceptron	RNN	Recurrent neural network
LSTM	Long short-term memory	SVM	Support vector machine
RF	Random forest	DT	Decision tree
EA	Evolutionary algorithm	GA	Genetic algorithm
GP	Genetic programming	LtR	Learn to rank
BPNN	Back propagation neural network	SMT	Satisfiability modulo theory

stack traces, and other failure-related information (information retrieval-based FL, IRFL). Alternatively, dynamic information acquired during testing can also be used to sort all program entities[2)] according to their possibility of being faulty (spectrum-based FL, SBFL). As for program repair, an intuitive yet typical method is to manually fix the suspicious code based on the fault context, domain knowledge, or requirement documents, and then re-run the fault-reveal test case to verify if the anomalous behaviors have been eliminated. To achieve automated program repair (APR), developers tend to regard patch generation as a search task, in which an automatic framework searches for candidate patches in a given area followed by evaluating each patch's effectiveness against a set of test cases [37, 38], i.e., determining whether a patch passes or fails [39].

The explosion of data and the surge in hardware computing power liberate a broad class of AI techniques that were born in the last century, which enables them to dramatically enhance the performance of many state-of-the-art techniques in speech recognition, object detection, image processing, and other fields [40]. One of the most significant advantages of AI is that it can mine potential correlations and patterns in data without the need for human intervention, which makes it ideal for many software engineering jobs including FL and APR. As early as 1981, Barr and Feigenbaum [41] discussed the idea of combining AI and software engineering. Furthermore, Durelli et al. [18] pointed out that AI has successfully reduced effort in numerous software engineering activities, and Feldt et al. [42] also believed that there is "ample opportunity" to apply AI models to promote software engineering tasks. Many studies published on leading venues have demonstrated AI techniques' promise in software engineering tasks [43–45].

2.1.1 *Artificial intelligence for fault localization (AI4FL)*

Fault localization is widely acknowledged as one of the most important, but error-prone and resource-consuming tasks in software debugging [46–55]. Many advanced FL approaches have been developed to minimize costs and maximize effectiveness [56], such as slice-based [57–59], spectrum-based [60–64], statistics-based [65], data mining-based [66], and information retrieval-based techniques [67–69]. Although these automated FL techniques can produce promising results with less overhead than manual checking such as setting breakpoints and inserting print statements, they do have many limitations that must be taken into account. For example, slice-based FL may produce dices that contain a large number of irrelevant statements (for static slicing) or fail to capture execution omission errors (for dynamic slicing) [56]. Many entities become indistinguishable due to the same suspiciousness in SBFL because it only uses the coverage information without any other auxiliary complement [70]. Some traditional statistics-based FL techniques require a breadth-first search on the program dependence graph, which adds a considerable amount of time to the process. Despite the fact that data mining-based FL has abundant historical data to endorse it, it is these huge volumes of valuable resources that keep it from being extensively adopted in practice. In the context of IRFL, keywords that are present in faulty elements (file, function, etc.) must also be included in the bug report, otherwise, it will be difficult to get a reasonable sorting result.

In view of the benefits of AI techniques and the needs of FL tasks, Zhang et al. [71] pointed out that, deep learning may open a new perspective for fault localization by using its learning ability of data to construct a localization model. There has been numerous research concerning AI for FL in recent years. For example, coverage information and test case results in classic SBFL are utilized as training data for AI algorithms including BPNN, DNN, CNN, RNN, and MLP, which calculate the suspiciousness of all executable statements using artificially built virtual test cases (each test case covers only one statement) [72–74]. Because the effectiveness of FL at the statement granularity is often low, and the

2) Statement, branch, basic block, function, and file, etc.

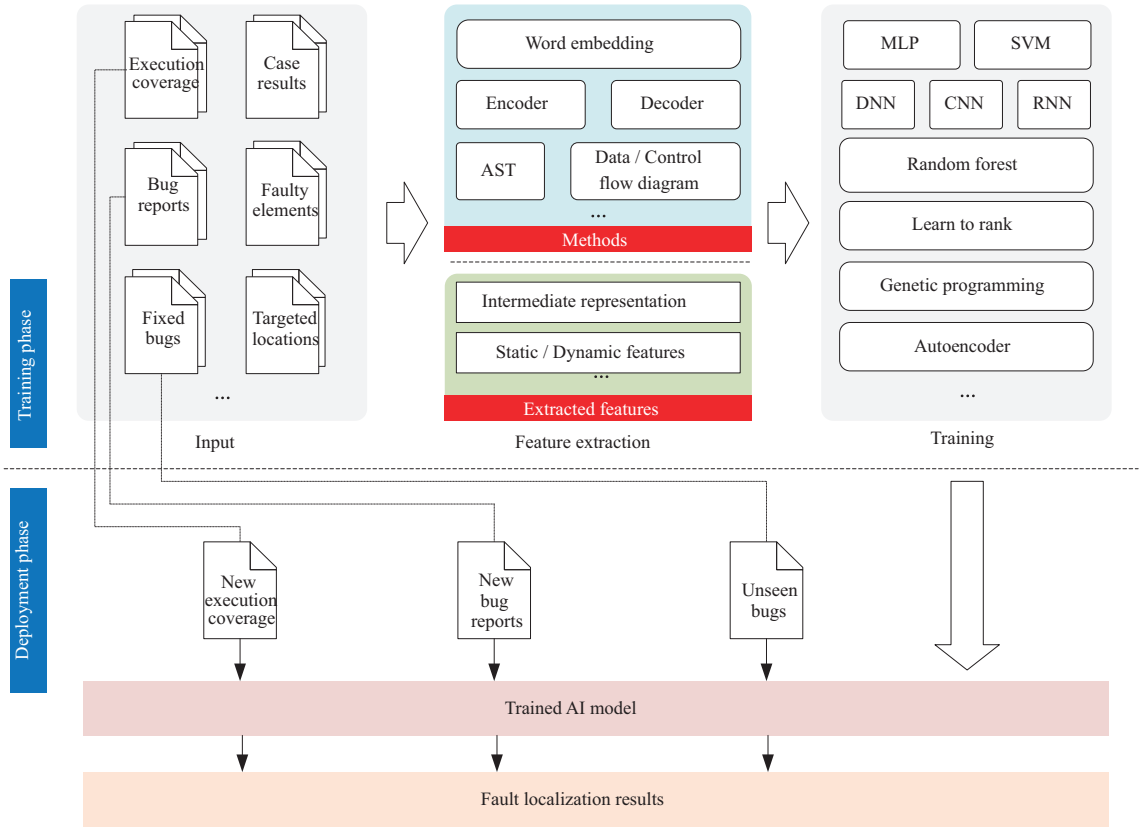

Figure 1 (Color online) General AI4FL workflow.

impact of fault context and developers' subjective experience is generally overlooked, researchers began to focus on file and method granularities. Specifically, they try to use AI techniques like DNN, LSTM, CNN, and LtR to bridge the lexical gap between search queries and retrieved files or methods, so as to establish linkages between semi-structured bug reports and source code with syntactic and semantic features [75–77].

To make use of the data-driven property of AI techniques, researchers and developers typically gather a large amount of unstructured or less structured data from historical software repositories, and then implement a specific technique to extract structured information, i.e., intermediate representation, based on predefined or automatically found features, which is finally fed into a constructed AI model for training its capability to localize faults. For example, execution coverage of test cases and their results can be used as training data and labels, respectively, to train a DL model like CNN or DNN, and then manufactured virtual test cases that each covers one statement can be fed into the trained model to evaluate the risk of program statements being faulty [71, 73]. The general AI4FL workflow is shown in Figure 1.

2.1.2 *Artificial intelligence for automated program repair (AI4APR)*

Once a fault has been detected and localized, the next step is to fix it [78–81]. However, the manual repair usually requires programming experience and domain knowledge, as well as takes a long time for developers. Based on the suspicious location delivered by the FL phase, APR techniques generate a series of candidate patches along with iteratively verifying whether these patches succeed in making the actual output of the program consistent with the expected output without human intervention, until one patch is found that can pass all test cases, or no new patches can be verified [82]. APR techniques have drawn the attention of a rising number of developers from the industry since they can effectively tackle the limitations of high labor costs and time costs seen in manual repair [83], as a result, they have been smoothly implemented in practical development. For example, Facebook, an American multinational technology company, integrates two APR tools, Sapfix [84] and Getafix [85], into its products for obtaining fault patches automatically.

Nevertheless, this emerging technology is also faced with many challenges. According to Motwani et

al. [86], APR techniques may produce some low-quality patches, and they are also difficult to generalize to the intended specification. Smith et al. [87] claimed that the existing APR techniques suffer from overfitting (i.e., generated patches only pass the test cases that determine if they are plausible, but fail the additional independent tests). They further observed that only 68.7%, 72.1%, and 64.2% of patches generated by three popular APR techniques (GenProg [88], TRPAutoRepair [89], and AE [90], respectively) passed independent tests.

The effectiveness of APR depends heavily on the precision of the upstream FL phase, only when the location of the fault is precisely pinpointed can APR techniques have the basis for effective fixing [91,92]. Unfortunately, the FL technique itself may be biased, thus the APR technique based on it faces the risk of "bias plus bias", as Liu et al. [93] argued, and FL is possibly one of the challenging steps in the repair pipeline. Furthermore, although APR techniques have stronger repair capabilities for predefined fault types (a family of faults with the same symptoms, root cause, or solution [94]), their capability to repairing some undefined or other specific types of faults is often limited [95]. As Motwani et al. [96]'s conclusion, APR techniques do not exhibit satisfactory results on faults requiring manual add loops and new function calls, or change method signature to repair.

It makes sense to use AI to help alleviate the bottleneck that APR techniques suffer. For example, researchers observed a vast amount of historical software repair data in issue, pull request, and commitment in open-source communities such as GitHub (or counterparts on other platforms), where many potential fixing patterns (also known as fixing templates) can be extracted. As a result, some strategies use AI techniques such as CNN, RNN, and encoder-decoder to learn the aforementioned templates from prior bug fixes, and then apply them to fix unseen bugs [97,98]. In addition, AI techniques such as Sequence-to-Sequence model and ensemble learning have also been used in research to mine the rich semantic features contained in the fault context [99,100], which can be used to capture long-range dependencies, thus assist in generating higher-quality patches.

In general, AI-based APR techniques represent buggy code, the context of buggy code, and fixed code as intermediate forms guided by specified rules; then input these structured code fragments to the training phase to obtain an AI model that can automatically generate patches for unseen faults. The general AI4APR workflow is illustrated in Figure 2. s_1–s_n is a ranking list of program statements delivered by the FL phase, where all executable statements are sorted by their possibility of being faulty. An APR technique will attempt to fix these statements from the riskiest statement to the safest one (producing m candidate patches for each statement) until the program passes all test cases. The lower the statement's ranking, the less possible it is to contain a fault and to be modified by the APR technique. This intuition is illustrated in Figure 2 by different levels of saturation of suspicious statements and candidate patches.

2.2 Software debugging for artificial intelligence

While AI techniques have been proven to be valuable in a variety of fields, they are also a sort of software that can contain faults [101]. In particular, numerous AI models have been embedded as modules in a wide range of software systems, including safety-critical applications [102, 103]. If these AI models run anomalously, not only will they be interrupted, but the entire system into which they are integrated will be threatened. For example, autonomous driving systems, one of the hottest application fields of AI, have frequently exposed flaws: a tesla driver was killed when his vehicle struck an overturned semi in May, 2021, according to a recent piece of news [104]. As a result, when an AI model fails, it is necessary to localize the root cause(s) and fix it (them) as quickly as possible to assure the quality of the AI model and the related system.

However, in contrast to traditional software that has explicit code and well-defined control flows, an AI model is generally recognized as a black box since the logic that drives its behaviors is inferred from training data [105–107], as a result, developers are only aware of that it can work well in specific tasks, without knowing why it works well. Specifically, a recent trend in the AI field is to use deep neural networks, which further increases the difficulty of localizing and fixing faults that reside in AI systems due to the explosive complexity of the model [108]. As Lourenço et al. [109] concluded, faults in an AI model can reside in many sources including code, input data, and improper parameter settings, which highlighted the difficulties in debugging AI systems.

The factors that influence AI techniques' effectiveness mainly include training data (e.g., wrong formats or dimensions, polluted data), model architecture (e.g., an improper number of layers/neurons and problematic activation functions), and AI libraries (e.g., wrong API usage and incorrect configurations).

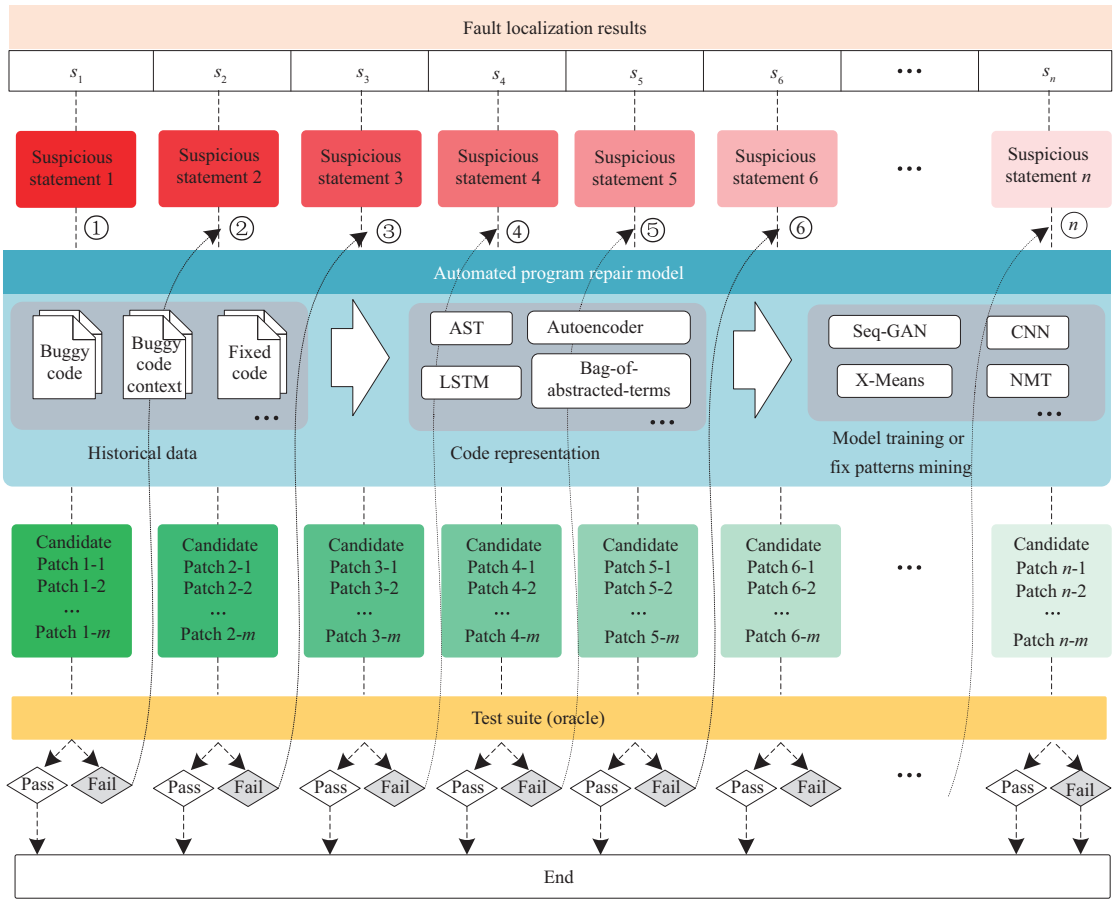

Figure 2 (Color online) General AI4APR workflow.

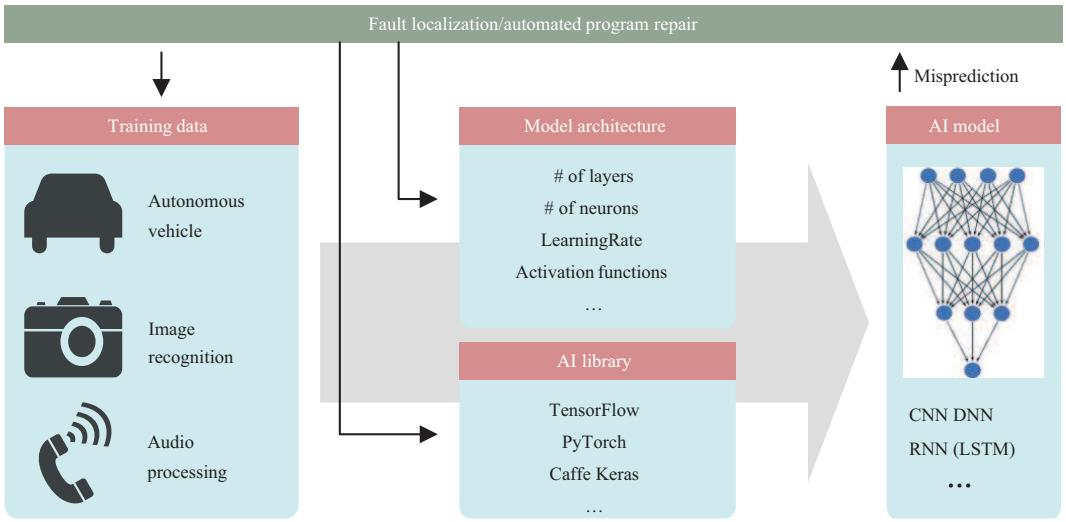

Figure 3 (Color online) An overview of software debugging for AI models.

The majority of existing studies on debugging AI systems are also carried out for these three parts, as shown in Figure 3.

2.2.1 *Training data-oriented debugging*

AI models are driven by data; a high-quality model cannot be produced with low-quality training data. When an AI model gives mispredictions [110], it is reasonable to inspect the training data used in the

model-building process. In general, a developer performs training data-oriented debugging as follows: (1) establish a linkage between training data and mispredictions to determine which part of data contributed the most to the present failure [111, 112], (2) identify and clean up polluted training data (that is, the sub-sets that cause the most misclassifications) [113], and (3) divide the training data multiple times to iteratively repair the AI model [114, 115]. It is worth emphasizing that training data-oriented debugging can be used to not only localize and repair faults in AI models, but also increase the interpretability of predictions [116], which will boost user confidence in AI models thus making developers integrate them into their system without worries.

2.2.2 *Model-oriented debugging*

Although data is crucial in the development of an AI model, the architecture of the model and the parameters it utilizes are both essential factors in determining an AI model's overall performance. To that end, many researchers have proposed new ideas or developed novel approaches to debug AI models. For example, Eniser et al. [117] built hit spectrum for neurons in neural networks to identify and pinpoint highly erroneous neurons, while Guidotti et al. [118] used transfer learning to repair CNN, exemplifying the concept of AI debugs AI.

2.2.3 *AI libraries-oriented debugging*

Open-source libraries on which AI models depend are an essential basis for their quality assurance, we find that researchers tend to analyze and summarize common faults and symptoms that occur in mainstream AI libraries, such as TensorFlow, PyTorch, Caffe, and Keras. The real-world AI applications on GitHub and code snippets on StackOverflow [34, 119–121] are also used to extract the most common types of faults, root causes of faults, and impacts of faults [120]. These artificially summarized fault patterns are abstractions of fault features in real-world AI models; thus they can provide developers with pertinent guidance for localizing and fixing similar faults.

It is worth mentioning that we have not discovered a unified workflow for debugging AI models, indicating that there is not yet a mature framework in this area.

3 Methodology

We present the research questions, introduce the paper selection process, and statistically analyze the selected papers in this section.

3.1 Research questions

The definition of research questions is the core innovation of a secondary study, as they clearly convey the authors' perspective on the subject under investigation and the study's goal [122]. Inspired by the Goal-Question-Metric approach proposed by Basili et al. [123] and Durelli et al.'s previous work [18], we characterize the purpose, the major research topics, and the scope of this paper as follows.

Analyze the AI techniques used in software debugging and the debugging activities aimed at AI models, for the purposes of investigation, summarization, and refinement with respect to the most commonly-used AI algorithms and their usage, the types and features of training data as for AI4SD, the mainstream techniques, the most frequently occurring fault types in AI models, and off-the-shelf libraries as for SD4AI, as well as the popular datasets, programming languages, and metrics employed in experiments, along with the opportunities, challenges, and potential future directions from the point of view of researchers in the context of AI4SD and SD4AI fields.

Three research questions of this survey are presented below according to the preceding definitions.

RQ1: How can AI techniques improve the effectiveness of SD?

A significant number of empirical studies have demonstrated that incorporating AI into SD can considerably improve the latter's effectiveness. However, only a few research has mined in-depth details behind such improvements. This research question focuses on how AI techniques can assist in software debugging from two perspectives: AI techniques and training data.

- RQ1.1: How and which AI techniques are typically used in software debugging?
- RQ1.2: What types of data do researchers tend to use as the input of AI models in the context of software debugging?

RQ2: How can SD be used to assure the quality of AI systems?

To alleviate the difficulty of debugging AI systems caused by their black-box property, various researchers have attempted to leverage traditional SD techniques to guarantee AI systems quality in recent years. The goal of this research question is to extract common points in the field of SD4AI from two perspectives: mainstream techniques and commonly occurring faults.

- RQ2.1: How software debugging techniques are used to localize or repair faults in AI systems?
- RQ2.2: What types of faults occur frequently in AI systems and AI libraries?

RQ3: What configuration do researchers prefer to use in AI4SD and SD4AI experiments?

The evaluation and assessment of novel approaches and techniques are intimately tied to well-designed experiments in both AI4SD and SD4AI. What do researchers have in common when designing experiments? We investigate this research question from two aspects: experimental datasets and metrics.

- RQ3.1: Which datasets (programming languages) are most commonly used?
- RQ3.2: Which metrics are most commonly used?

3.2 Paper selection

We first identify three AI-related terms and three SD-related terms, then combine these two categories of terms by logical ORs to create nine search strings. After specifying the range the papers were published: 2016–March 2021, we deliver these nine search strings to two databases, Google Scholar and Scopus, to collect related papers in the intersection of AI and SD.

The defined search terms are as follows:

[AI] **OR** [Deep learning] **OR** [Machine learning]

AND

[Software debugging] **OR** [Fault localization] **OR** [Program repair].

After obtaining 253 papers, we use backward snowballing to find 65 additional ones. Our initial pool of 318 papers is filtered using the following inclusion criteria (IC) and exclusion criteria (EC):

[✓] **IC1**: Peer-reviewed paper.

[✓] **IC2**: Primary study.

[✓] **IC3**: The paper discusses how AI techniques empower classic debugging processes, or conducts empirical studies or proposes novel approaches in the field of debugging AI software systems.

[✗] **EC1**: The paper focuses on testing or defect prediction, rather than debugging techniques.

[✗] **EC2**: The duplicate version of a selected paper (only the most comprehensive or recent version of each study should be included).

[✗] **EC3**: The length of the paper is less than 6 pages.

[✗] **EC4**: Paper written in a language other than English.

A total of 165 papers are included in the scope of our research. Figure 4 illustrates the year distribution range. The number of papers published in the AI4SD and SD4AI fields has increased by 44% annually in the predefined time range, from a minimum of 16 in 2016 to a maximum of 66 in 2020 and the first quarter of 2021. However, the number of studies on SD4AI is significantly lower than that on AI4SD: 131 and 34 papers relevant to AI4SD and SD4AI were published between 2016 and 2021.3, respectively, with the latter accounting for just 26% of the former, revealing that researchers prefer to employ the AI technique as a tool for tackling difficulties in debugging, instead of debugging AI systems themselves. Nonetheless, we can also observe that researchers are becoming more enthusiastic about SD4AI: the number of SD4AI studies published in 2020 and the first quarter of 2021 exceeds those published between 2016 and 2019.

3.3 Paper analysis

We conduct non-technical analyses of all the 165 papers from three perspectives, i.e., publication venues, the number of citations, and contributions by countries.

3.3.1 *The main venues*

Several well-known journals and conferences in software engineering are listed in Tables 2 and 3, respectively, the selection criterion is how many times they appear in our paper pool. Three conclusions in terms of researchers' submission preferences in the intersection of AI and SD can be derived: (1) *Empirical Software Engineering* (EMSE), *IEEE Transactions on Software Engineering* (TSE), and *Journal of*

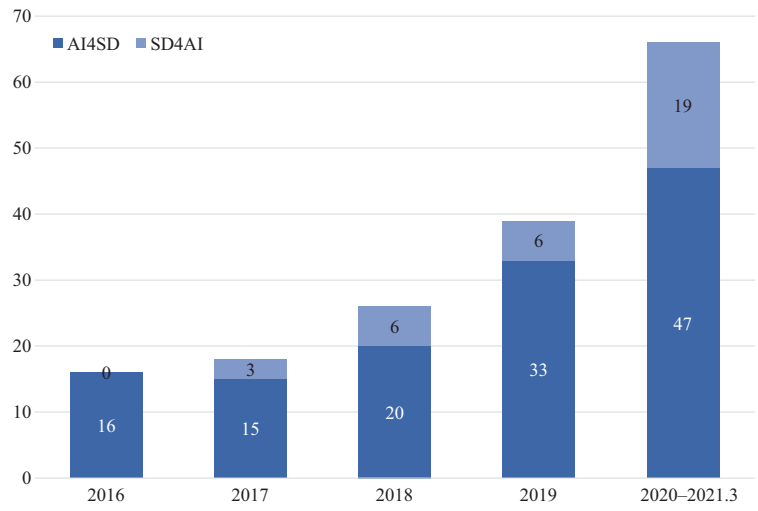

Figure 4 (Color online) Number of publications per year.

Table 2 Main venues (Journal)

Acronym	Name	Number of publications
EMSE	*Empirical Software Engineering*	12
TSE	*IEEE Transactions on Software Engineering*	10
JSS	*Journal of Systems and Software*	7
IST	*Information and Software Technology*	6
TOSEM	*ACM Transactions on Software Engineering and Methodology*	5
ACM PL	*Proceedings of the ACM on Programming Languages*	3
JSEP	*Journal of Software: Evolution and Process*	2

Table 3 Main venues (Conference)

Acronym	Name	Number of publications
ICSE	International Conference on Software Engineering	16
FSE/ESEC	ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference	10
ASE	International Conference on Automated Software Engineering	9
ISSTA	International Symposium on Software Testing and Analysis	5
IJCAI	International Joint Conference on Artificial Intelligence	5
QRS	International Conference on Software Quality, Reliability and Security	5
SANER	International Conference on Software Analysis, Evolution, and Reengineering	4
ICLR	International Conference on Learning Representations	3

Systems and Software (JSS) are the most prevalent journals; (2) International Conference on Software Engineering (ICSE), ACM SIGSOFT Symposium on the Foundation of Software Engineering/European Software Engineering Conference (FSE/ESEC), and International Conference on Automated Software Engineering (ASE) are the most prevalent conferences; (3) researchers prefer to publish their findings through conferences compared with journals.

3.3.2 *Most cited papers*

The number of citations reflects the impact, visibility, and importance of a paper [124]. In order to better understand the academic influence of the intersection of AI and SD as well as the quality of the papers in our pool, we counted all the 165 papers' citations from Google Scholar on June 2, 2022. The total number of citations for all 165 papers is 8739, with an average of 52.96 per paper. More specifically, the 131 papers in AI4SD have earned a total of 5934 citations, with a median of 22, whereas the 34 papers in SD4AI have been cited 2805 times in total, with a median of 21.

Table 4 shows the top-5 most cited papers in AI4SD: Refs. [125] and [126–129] investigated the application of AI techniques in FL and PR, respectively. Table 5 shows the top-5 most cited papers in

Table 4 Top-5 most cited papers (AI4SD)

Year	Title	Venue	Authors	Citations
2016	Automatic patch generation by learning correct code	POPL	Long and Rinard [126]	533
2016	Nopol: automatic repair of conditional statement bugs in Java programs	TSE	Xuan et al. [127]	336
2016	History driven program repair	SANER	Le et al. [128]	323
2016	From word embeddings to document similarities for improved information retrieval in software engineering	ICSE	Ye et al. [125]	304
2017	Precise condition synthesis for program repair	ICSE	Xiong et al. [129]	248

Table 5 Top-5 most cited papers (SD4AI)

Year	Title	Venue	Authors	Citations
2017	Understanding black-box predictions via influence functions	ICML	Koh and Liang [112]	1637
2018	What is wrong with topic modeling? And how to fix it using search-based software engineering	IST	Agrawal et al. [131]	161
2018	An empirical study on TensorFlow program bugs	ISSTA	Zhang et al. [119]	144
2019	A comprehensive study on deep learning bug characteristics	ESEC/FSE	Islam et al. [120]	120
2018	MODE: automated neural network model debugging via state differential analysis and input selection	ESEC/FSE	Ma et al. [130]	101

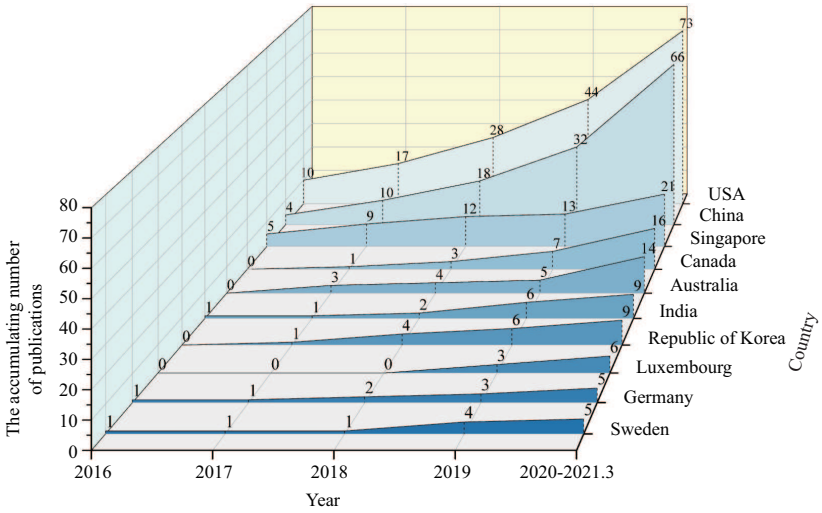

Figure 5 (Color online) Contribution by countries and years.

SD4AI: Refs. [119, 120] were empirical studies on typically occurring faults in AI systems and libraries, and Refs. [112, 130, 131] proposed specific techniques for debugging AI models.

3.3.3 *Contribution by countries*

To see which country's researchers are most active in the intersection of AI and SD, we summarize the countries where the authors of our selected papers are located, including Australia, Austria, Canada, China, Denmark, France, Germany, India, Israel, Italy, Luxembourg, Nigeria, Republic of Korea, Singapore, Sweden, Switzerland, The Netherlands, Turkey, UK, and USA (in alphabetical order). The top-10 countries according to our investigation are depicted in Figure 5.

It can be observed that the numbers of publications from USA and China are 73 and 66, respectively, far exceeding that of Singapore (21 publications), which is ranked third. We further focus on the research trends in USA and China. Since 2016 (the first year of the predefined time range), the numbers of publications in the intersection of AI and SD as of 2016, 2017, 2018, 2019, and 2021.3 are 16, 34, 60, 99, and 165, respectively, according to Figure 4. During the same period, the numbers of publications from USA are 10, 17, 28, 44, and 73, respectively, with the domain shares[3)] of 62.5%, 50.0%, 46.7%, 44.4%, and 44.2%, respectively. Similarly, the numbers of publications from China are 4, 10, 18, 32, and 66, respectively, with the domain shares of 25.0%, 29.4%, 30.0%, 32.3%, and 40.0%, respectively. Although

3) We define the domain share of a country as the proportion of the publications from that country to all publications.

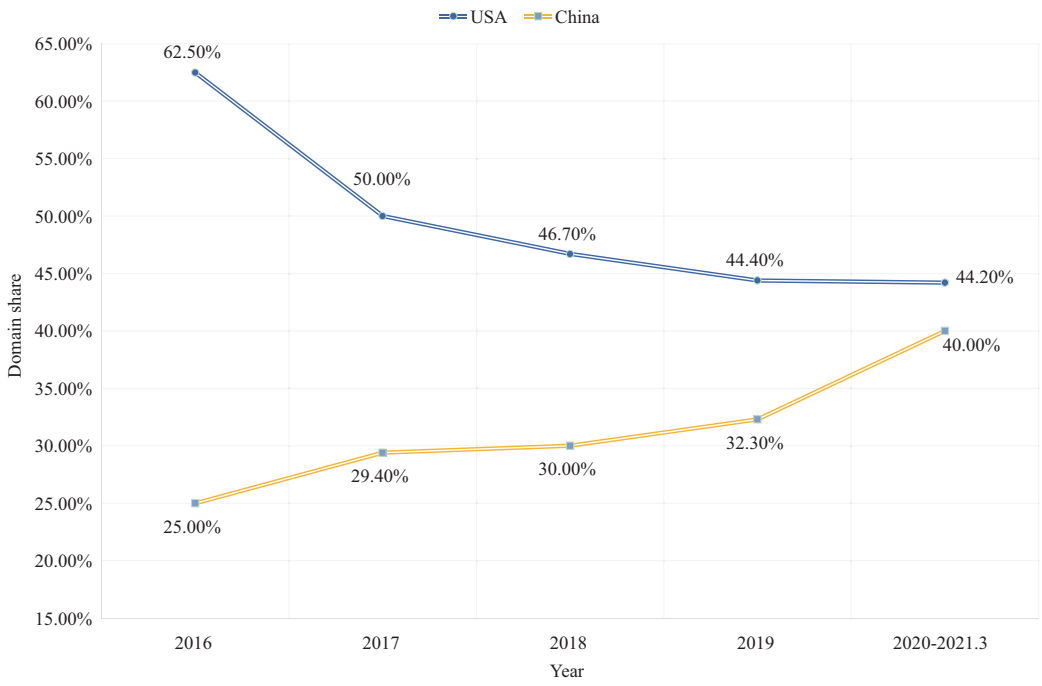

Figure 6 (Color online) Domain shares of USA and China.

China's domain share was lower in the previous period, it shows an upward trend and the gap with the USA is narrowing, as illustrated in Figure 6.

4 How can AI techniques improve the effectiveness of SD? (RQ1)

To answer RQ1, we review 131 studies that investigate AI techniques applied in SD in this section.

4.1 How and which AI techniques are typically used in software debugging? (RQ1.1)

A variety of AI techniques applied in software debugging typically serve two purposes, AI4FL and AI4APR. In general, AI techniques are employed by researchers to mainly tackle six sub-problems in fault localization, i.e., feature extraction (ensemble), suspiciousness calculation, information retrieval, test cases enhancement, multi-fault debugging, and others. For these purposes, CNN, EA, RNN (LSTM), etc. are the most widely-used techniques. On the other hand, AI techniques are employed by researchers to mainly tackle three sub-problems in automatic program repair, i.e., patch generation, patch ranking (filtering), and code representation. For these purposes, RNN (LSTM), EA, Constraint solving, Clustering, etc. are the most widely-used techniques.

4.1.1 *How and which AI techniques are typically used in fault localization?*

The AI techniques are typically used in fault localization in six categories roughly, that is, feature extraction (ensemble) (25%), suspiciousness calculation (24%), information retrieval (23%), test cases enhancement (12%), multi-fault debugging (7%), and others (9%)[4)], as shown in Figure 7(a). Prominent AI techniques in this direction include CNN (13%), EA (9%), RNN (LSTM) (9%), and so on, as shown in Figure 7(b).

(1) Feature extraction (ensemble). To assist in their fault localization jobs, researchers prefer to employ various AI techniques to extract or process failure-relevant features. For example, they intend to automatically extract different forms of features from available sources, or integrate multiple features into a comprehensive evaluation result, with the support of the mining and merging capability of AI

4) If a paper is involved in multiple categories at the same time, we either introduce it in each category independently or only put it in one category.

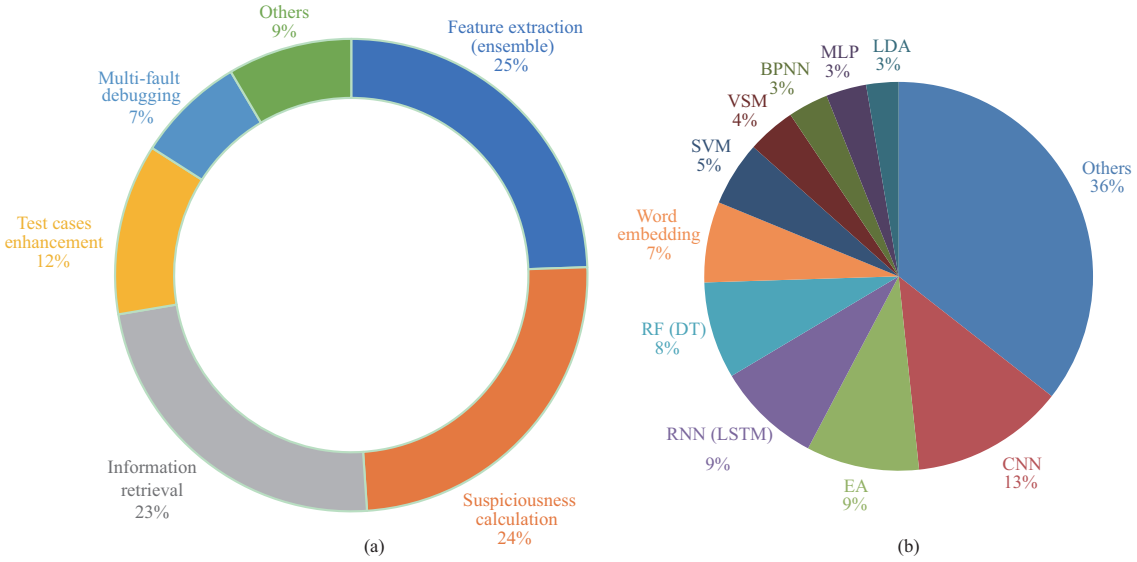

Figure 7 (Color online) (a) Main sub-problems solved by AI techniques and (b) prominent AI techniques in fault localization.

techniques. In this category, the commonly-used AI techniques mainly include RNN (LSTM), CNN, RF (DT), EA, LtR, and so on.

• **RNN (LSTM).** RNN can learn dependencies over time with feedback connections, and the output of the current state is dependent on prior ones. LSTM, a specific RNN architecture, is typically used to handle the vanishing gradient problem. We observe that both of them have been widely used in fault localization tasks. For example, Peng et al. [132] sought to utilize LSTM to extract static features from source code (i.e., tokenize all program statements before feeding them into the model). Huo and Li [76] proposed LS-CNN (LSTM based on CNN), which took advantage of the sequential nature of source code to enhance the unified features for fault localization. Later, Huo et al. [133] stated that in IRFL, apart from sequential information, structural and functional nature can also contain valuable semantics. As such, they presented a novel framework in which LSTM models were utilized to capture such semantic features. With the increases in features' dimension and scale, Li et al. [134] attempted to automatically identify the most effective existing/latent features for precise fault localization. They chose an RNN with an LSTM model for their goal because different feature groups can be treated as inputs for different time steps in RNN. Qi et al. [135] focused on automated fault localization and diagnosis in distributed systems. In their study, the features map of system status can be fed into an RNN model, avoiding human expertise and manual feature extraction.

• **CNN.** CNN can learn implicitly from training data in parallel compared with other models, and it has been integrated into a variety of debugging tasks. To learn a unified feature representation from both natural and programming languages, Huo et al. [136] proposed a novel CNN model that reflected the program structure by convolution operation. TBCNN (tree-based CNN) is a type of tree-structured neural network based on syntax trees, which is designed to improve tree coding in the original CNN model. Liang et al. [137] argued that traditional tree-based FL techniques gave user-defined functions the same importance as system-defined functions. Therefore, they constructed a novel customized abstract syntax tree (AST) and integrated it into a previously proposed TBCNN-based method, to fuse various types of features for fault localization.

• **RF (DT).** Random forests are a type of classifiers that contain multiple decision trees in machine learning. Focusing on factors that influence fault localization techniques' effectiveness, Golagha et al. [138] utilized RF to find the most efficient features for SBFL techniques, concluding that two static, four dynamic, and two test metrics are influential. Gu et al. [139] trained a DT model based on 89 predefined features, for the prediction of whether a software crashing fault lies in given stack traces (i.e., integrating those features into a prediction result using DT given a newly-submitted crash). Experiments demonstrated the promise of their approach in localizing such severe faults.

• **EA.** Both GA and GP are inspired by Darwin's theory of natural selection. GA simulates the natural evolutionary process to find the optimal solution, which has been widely used in fault localization, and GP is a specialized form of GA. Wang and Lo [10] proposed to use version history, similar reports, structure,

stack traces, and reporter information to achieve more accurate FL; they used GA as a composer to tune the weights of the mentioned five features for computing final risky scores. Kim et al. [140] proposed using GP to generate a rank model by combining multiple sets of FL input features (including dynamic features like mutation-based and spectrum-based information, as well as static features like file, function, and statement information). Aiming at recommending relevant locations that need to be changed for supplementary bug fixes, Xia and Lo [141] first extracted change relationship graphs as features and then employed GA to learn a weighting formula for combining each feature's ranking list with regard to candidate change locations.

• **LtR.** LtR is the application of machine learning in the construction of ranking models [142]. Pan et al. [143] modified a LambdaRank model to synthesize static and dynamic features by adding a self-attention mechanism and a transformer encoder, for ranking a list of possible faulty statements. Shi et al. [77] conducted a short survey on IRFL techniques that utilized extra features other than the textual similarity, followed by combining beneficial features using eight LtR algorithms, to further enhance the effectiveness of FL techniques. Ye et al. [144] first defined and extracted 19 classes of features, and then employed LtR techniques to automatically train the weights based on previously fixed bug reports, for the building of the linkage between a newly-received bug report and defective files.

• **Other AI techniques.** SVM is a supervised binary classification model and has been used in fault localization broadly. To rearrange developers' schedules and efforts for more efficient debugging, Yang et al. [145] first defined two types of high-impact faults, i.e., surprise faults and breakage faults, and then managed to identify these risky faults using SVM based on the extracted textual features from bug reports, under the guidance of imbalanced learning strategies. Considering that the effectiveness of IRFL techniques could be harmed when predefined features are correlated with each other, Guo et al. [146] first implemented a ranking-performance-based feature selection approach, and then used logistic regression to identify the relationship between the informative features, for the boost of an off-the-shelf technique [147]. Li et al. [148] treated each program statement's behavior as a feature, and adopted Fisher score, a feature selection technique, to measure its contribution to the test outcomes. Besides, if a test case covers the faulty statement without delivering an unexpected output, the faulty statement will likely be ranked in a lower position. Feyzi [149] proposed further analyzing program statements by an information-theory-based feature selecting algorithm, to tackle the challenge posed by such coincidental correctness. Amar and Rigby [150] utilized the K-nearest neighbor algorithm to extract the essential feature of massive test logs. Their approach can find a large portion of faults with a small inspection space. After conducting extensive experiments for answering an essential question, that is, which features of bug reports should be matched against which features of source code files, Koyuncu et al. [151] presented a novel ensemble approach for more accurate fault localization based on the hypothesis that various IRFL tools have different performances for different types of features, in which prediction probabilities of multiple classifiers are assigned by gradient boosting supervised learning. XGBoost is an effective AI algorithm that can maintain important features during training, Yang et al. [152] integrated it into SBFL and presented a novel XGB-FL approach for fault localization. Nath and Domingos [153] blamed that most existing probabilistic debugging approaches employed simple statistical models and failed to generalize across multiple programs. Therefore, they constructed tractable fault localization models based on suspiciousness and bias term features using the relational sum-product network (RSPN), to infer the bug location.

(2) Suspiciousness calculation. A typical way to fault localization is to calculate the suspiciousness score (also known as the risk value) for program entities. In addition to traditional rule-based suspiciousness calculation strategies, more and more researchers begin to utilize AI techniques to complete such tasks. In this category, the commonly-used AI techniques mainly include MLP, CNN, EA, LtR, RF (DT), and so on.

• **MLP.** MLP is the most basic and original neural network, which connects multiple layers via full connection (the network structure of MLP does not have any loop and the output of each node does not affect the node itself [154]). Li et al. [134] combined their FL approach with the classic MLP model and variants of the classic MLP model tailored for their proposed approach. They directly fed failure-relevant data to MLP and used the produced information to calculate suspiciousness for program elements, concluding that the enhanced MLP outperforms the traditional method for prediction in both effectiveness and efficiency. Notice that MLP is often implemented with DNN, and one of the most important advantages of DNN is that it can model complex non-linear relationships with multiple hidden layers. Zheng et al. [73] presented a DNN-based approach to obtain each statement's suspiciousness in

fault localization tasks with a limited amount of sample data, emancipating the restricted capability to extract complex functions of shallow architecture algorithms. Maru et al. [155] combined BPNN with the traditional SBFL technique, proposing to train the BPNN model using count spectrum instead of hit spectrum (i.e., statements that are covered multiple times should be given a larger weight than those that are only covered once). Dutta et al. [156] attempted to gather the advantages of existing approaches and hence proposed an ensemble of spectrum-based and neural network-based fault localization methods, in which BPNN is employed as a sub-technique, for the suspiciousness estimation.

• **CNN.** Zhang et al. [71] used execution coverage and test results as CNN's training data, so as to combine CNN with SBFL for more effective fault localization. In their approach, the output of the CNN model serves as suspiciousness for program statements. Additionally, Li et al. [157] considered fault localization to be an image pattern recognition problem in crime scene investigation, and leveraged a CNN model to exploit the full details of the code coverage matrix. Program statements will be classified into two categories (i.e., faulty or non-faulty) according to the suspiciousness produced by the trained CNN model. Considering that CNNs have been widely used in fault localization, the authors of [74, 158] evaluated various existing machine learning models by comparing the output suspiciousness. They discovered that deep learning models, particularly CNN, outperformed the other models they investigated.

• **EA.** Mahapatra and Negi used GA to self-adaptively optimize the weights and structure of the radial basis function (RBF) network in fault localization, which has higher precision compared with the baselines with regard to suspiciousness calculation [159]. Sohn and Yoo attempted to overcome the limitation of purely relying on coverage, a common shortcoming of many existing FL techniques, by extending SBFL with code and change metrics as well as utilizing GP to learn the ranking models [160]. This approach was extended by Choi et al. later to have multiple objectives using NSGA-II [161].

• **LtR.** One of the most important trends in AI4SD in recent years is to use statistical techniques to combine different approaches. For example, as the very beginning work of this trend, Xuan and Monperrus combined multiple SBFL risk evaluation formulas by employing RankBoost, an efficient LtR approach, to re-calculate program entities' suspiciousness thus relieving the challenge of no optimal formula for all types of faults [162]. Based on program entities' suspiciousness, Zou et al. [163] extended Xuan and Monperrus' work by trying to use LtR to combine various techniques from different families, rather than a single family, to boost fault localization effectiveness. To automatically localize faulty machines when failures are revealed in software services, Liu et al. [164] deployed an online tool at real-world applications, where an LtR model is trained for ranking all of the potential root cause machines. Li and Zhang [46] argued that mutation-based fault localization (MBFL) techniques will be ineffective if too many or too few mutants can impact the outputs of failed tests, and used rankSVM with the linear kernel to analyze various mutation information (compute different suspiciousness values) for better fault localization. Le et al. [165] utilized rankSVM to construct a model for ranking risky program elements through learning a statistical model based on the likely invariant differences and suspiciousness scores information.

• **RF (DT).** Küçük et al. [166] argued that measuring the correlation between execution profiles and results in statistical fault localization might be problematic. Therefore, they used RF to perform causal inference for alleviating this concern, in which each variable and predicate will be assigned to a unique suspiciousness score. Experiments revealed the promise of their approach on all versions of Defects4J (V2.0). Also inspired by causal inference, Podgurski and Küçük [167] performed simulated value replacement to derive statements' suspiciousness for helping localize faults. In their approach, an RF model is adopted for the prediction of outcomes given a set of variables' values, due to RF's flexibility and robustness.

• **Other AI techniques.** Lou et al. [168] stated that coverage information is often utilized in an oversimplified manner (i.e., binary vectors), and such numerical representation prevented FL techniques from being performed effectively. As such, they first represented coverage as a graph structure and then employed a gated graph neural network (GNN) to rank suspicious code nodes, by mining the valuable relationships between test cases and program entities. Maamar et al. [169] formally defined a fault localization process as a constraint programming problem; namely, they identified the most suspicious statements by solving the arising constraints. To mitigate the negative impacts of potential software faults, Yan et al. [170] integrated fault prediction and fault localization into a novel just-in-time framework. They first identified a given change as buggy or clean by training a logistic regression classifier, then calculated source code lines' suspiciousness in buggy changes through an N-gram model. Zaman et al. [171] developed an online tool to estimate the likelihood of a system call sequence (with its relevant functions) being faulty in concurrency environments. During execution, they adopted principal

component analysis and frequent pattern mining techniques to tackle the large scale of collected system call traces. Hoang et al. [172] considered employing multi-modal information to cluster bug reports and methods by applying network lasso regularization, they also developed an adaptive learning model based on a Newton method to rank methods based on the likelihood of being buggy.

(3) Information retrieval. Bug reports and other sources written in natural languages can provide valuable information for developers in the context of fault localization. But such a human-friendly form hinders their use in automatic debugging. Researchers intend to employ AI techniques to tackle this challenge. In this category, the commonly-used AI techniques mainly include DNN, CNN, EA, Word embedding, and so on.

• **DNN.** Cheng et al. [173] calculated the similarity between bug reports and source files based on information retrieval and word embedding techniques, and then used DNN to integrate them. Lam et al. [75] used two DNNs for features reduction and relevancy estimation between a bug report and a source file, which can assist in solving the lexical mismatch problem. Pradel and Sen [174] thought that the natural language part in source code (e.g., variable and function names) contains useful information, thus they proposed a name-based fault localization approach based on feed forward neural network (FFNN), to determine whether a piece of code suffers from name-related bug patterns.

• **CNN.** Liu et al. [175] employed CNN to joint lexical and semantic information of source files to learn how to localize faults from bug reports and source files. Xiao et al. [176] argued that semantic information in bug reports and source code is useful but underutilized by many off-the-shelf approaches, so they exploited two CNN models, one for bug reports and the other for source files, to detect features from vectors with semantic information and then fed the hybrid features into an enhanced CNN model to localize faulty files. Li et al. [177] stated that identifiers (i.e., terms written in natural languages) in source code conveyed rich semantics; therefore they preprocessed and fed such data into CNN models to automatically identify suspicious return statements.

• **EA.** Instead of adopting the common one level of abstraction strategy in IRFL, Zhang et al. [178] proposed to represent code snippets and textual information at multiple abstraction levels by (1) combining vector space model (VSM) and latent Dirichlet allocation (LDA) to comprehensively analyze word frequency and semantics of text, and (2) employing GA to find the optimal configuration in each abstraction level. Mills et al. [179] employed GA to explore the relationship between bug reports and queries in IRFL. They blamed that some prior experiments in this field are improper and inflated, and recommended that stakeholders reformulate queries in a more careful manner. Assuming that API documentation conveys more useful information than the natural language part of source code in the scenario of IRFL, Almhana et al. [180] employed multi-objective GA to more accurately build linkages between the given bug report and relevant classes. Experiments indicated that their method can both maximize localization effectiveness and minimize the labor of manual inspection. Moreover, aiming to more pertinently identify relevant methods given a bug report, Almhana et al. [181] further proposed a heuristic framework to perform both global and local searches, in which GA and a simulated annealing algorithm were employed to find risky classes and methods, respectively.

• **Word embedding.** Word embedding refers to the re-representation of words for text analysis, which encodes words into a vector space so that similar words are close to each other [182]. Due to its strong mapping capability, word embedding has received broad attention of researchers in AI4SD. For example, Ye et al. [125] trained word embeddings to bridge the lexical gap between natural and programming languages, so that a more accurate linkage between buggy files and bug reports could be established for IRFL based on the semantic similarity. To detect the faults in boundary conditions (for instance, "$\leqslant$" is misused as "$<$"), Briem et al. [183] trained a Code2Vec-like model on enormous historical code snippets, and their approach has been proven to be effective on real-world unseen code. Liu et al. [184] proposed a novel three-module FL framework; specifically, they proposed to calculate the surface lexical similarity between bug reports and source files using VSM in the first module, and calculate the semantic similarity using Skip-gram and GloVe in the second module. The final score is determined based on these two similarities to rank source files in the third module. Employing word embedding and TF-IDF, Zhang et al. [185] used the semantic similarity, the temporal proximity, and the call dependency to achieve method expansion (that is, augmenting a short-length representation of a method to tackle the representation sparseness challenge), which was beneficial for enhancing method-level fault localization according to their conclusion. Zhu et al. [186] argued that existing cross-project fault localization techniques typically dedicate effort to transferring common characteristics, while little attention is paid to preserving the private information of each project. Therefore, they developed an adversarial transfer learning framework

to tackle this challenge, where the pre-trained GloVe was exploited for embedding, and a bidirectional LSTM model was used to encode the sequence of bug reports.

• **Other AI techniques.** Zhong and Mei [187] chose LDA to classify bug reports by measuring the similarity between a topic and a document; then a graph-based classifier integrating AdaBoost and DT is used to determine whether a program node is buggy or clean. Moreover, Jonsson et al. [188] took historical bug reports as input, and utilized the supervised linear Bayesian LDA model to determine whether a fault localization prediction result is reliable. To help developers pinpoint which packages are buggy and need to be fixed, Huang et al. [189] developed an ensemble learning-based framework to deliver a ranking list of packages, which built linkages between a given bug report and a series of packages that are possibly affected. Le et al. [190] were concerned that if faulty elements appear low in the ranking list, developers could distrust the fault localization technique. As such, they first trained four individual prediction models and then combined them via bagging-based ensemble classification, for automatically predicting the effectiveness of IRFL tools. Considering that neighbors of suspicious files (i.e., call relationship) may also be suspicious, Li et al. [191] proposed a multi-label distribution learning approach to mine such dependencies for building more accurate linkages between a bug report and program code. Safdari et al. [12] pointed out that previously fixed faults and their reports could provide guidance for localizing current faults; thus they collected large numbers of existing bug reports for the training of an LtR model, enabling this model to learn the dependency relationship between historical faults and further localize unseen buggy files. Rahman and Roy [192] attempted to reformulate queries in IRFL for more accurately pinpointing relevant files. Specifically, they first categorized a piece of bug report into one of three classes, namely, stack traces, program elements, and natural language parts, and then applied the PageRank algorithm to the identification of key information in it.

(4) Test cases enhancement. The quality of test suites determines the effectiveness and efficiency of fault localization to some extent. Researchers attempt to enhance their test suites using existing AI techniques, making them have a stronger capability of revealing or localizing faults. In this category, the commonly-used AI techniques mainly include EA, RF (DT), and so on.

• **EA.** Li et al. [193] developed a GA-based framework comprising test generation and fault localization, and experiments revealed that higher code coverage achieved by their approach can more effectively and efficiently test and debug programs in software product lines. Chatterjee et al. [194] provided GA-based test suite generation frameworks with a novel fitness function. The improved fault localization effectiveness indicated that the newly-designed metric had a strong capability to measure the diagnosability of test suites.

• **RF (DT).** Elmishali et al. [195] proposed an RF-based approach that generated additional tests when diagnosis accuracy was not good enough. For a small but diverse test suite in debugging Simulink models, Liu et al. [196] defined four test objectives to run a search-based test generation algorithm, followed by training decision trees to determine the stop point of adding new test cases.

• **Other AI techniques.** To balance the numbers of failed test cases and passed test cases, Zhang et al. [197] proposed an iterative oversampling approach to reduce the negative effect caused by an imbalanced test suite. They integrated their strategy into four state-of-the-art deep learning-based FL techniques, namely, CNN-FL [71], MLP-FL [198], BPNN-FL [199], and BiLSTM-FL [200], and demonstrated that the effectiveness of these models was enhanced significantly. For more efficiently debugging single-fault and multi-fault programs, Xia et al. [201] proposed a novel diversity maximization speedup algorithm to select and order a smaller number of test cases. The results demonstrated that existing fault localization techniques can achieve comparable accuracy with less effort with the assistance of their approach. Liu et al. [202] first gave a theoretical analysis of how coincidental correct test cases harm the effectiveness of SBFL techniques, and then proposed to identify and manipulate such test cases based on the K-nearest neighbor algorithm, for higher accuracy of SBFL. To complement existing SBFL techniques that only differentiate program entities, Zhang et al. [203] extended them by differentiating tests and proposed a novel PageRank-based SBFL technique with a richer spectrum. Chen et al. [204] proposed a twofold approach to isolate compiler faults. They first conduct structural mutation instead of traditional local mutation for better altering the control-flow of test programs, and then used reinforcement learning to prevent such a mutation process from generating test programs that, though passing, were useless for isolation. To make use of unlabeled test cases in dynamic testing, Zhang et al. [205] proposed a novel suspiciousness probability-based classifier to predict their labels according to execution information, which is expected to enhance off-the-shelf fault localization techniques. Gupta et al. [206] presented a twofold approach to localize faults without running the program. Specifically, in the first phase, they trained

a tree CNN for the prediction of the execution result (failed or passed) given a test case. And in the second phase, they queried a neural prediction attribution technique [207] to identify which lines result in a faulty program (i.e., the execution result predicted as failed in the first phase).

(5) Multi-fault debugging. The assumption that an abnormal program contains only one fault is becoming unrealistic due to the increasing scale and complexity of software systems. Debugging in multi-fault scenarios could suffer from many challenges, such as fault interference and faults number estimation. Research on this topic shows a growing trend recently, and many of them are supported by AI techniques. In this category, the commonly-used AI techniques mainly include CNN, EA, and so on.

• **CNN.** To identify different expressions between bug reports with the same root cause, He et al. [208] used word embedding to translate each piece of bug report into a 2D matrix, then fed bug report pairs into the proposed dual-channel CNN-based duplication detection model for obtaining the similarity among all pairs. To classify faults based on their different root causes, Ni et al. [209] proposed a TBCNN model that represented the fix trees of bug fixes and extracted the features of these fix trees. Their results demonstrated the existence of a relation between the bug fix and the cause of bugs.

• **EA.** Yan et al. [210] presented a GA framework along with a newly-designed fitness function to measure the likelihood of failed test cases being coupled in programs containing more than one fault. Considerable improvements in terms of multi-fault localization effectiveness were observed in their experiments. Zheng et al. [211] also adopted a GA framework to solve multi-fault localization problems. Specifically, they encoded a binary vector as a candidate solution, in which each value indicates whether a program entity should be assumed buggy, and evolved such vectors by their approach for identifying multiple root causes simultaneously.

• **Other AI techniques.** Gao et al. [212] performed an empirical study on the performance of 22 different machine learning models for multiple fault localization; they discovered that RF has higher accuracy and more significant localization efficiency than the others. Seeing that the co-existence of multiple faults in a program could decrease the effectiveness of fault localization techniques, Gao and Wong [48] divided failed test cases according to their root cause by using clustering, a commonly-used unsupervised learning scheme. Such a failure indexing process has shown to be promising in tackling multi-fault localization problems. In addition, many researchers have coupled RNN with multi-fault debugging, for instance, an RNN-based method was proposed to predict the number of faults in a program for software reliability assessment [213].

(6) Others. There are also many studies that introduce AI techniques to fault localization for tackling other sub-problems.

In addition to advanced FL techniques, traditional logging statements can also assist developers in pinpointing the location of root causes. However, according to Li et al. [214], properly helping developers decide finer-grained logging locations for different situations remains an unsolved challenge. Therefore, they first manually summarized six common categories of logging locations and then proposed an LSTM-based logging location recommender. Experimental results showed that their approach achieved promising results in both within-project and cross-project scenarios. Vasic et al. [215] presented multi-headed pointer networks based on LSTM for program debugging, in which one head is responsible for fault localization.

Chappelly et al. [216] attempted to use a simple CNN model to identify a suitable abstraction for static analysis, for the improvements of its scalability and precision. To alleviate high false-positive ratio in code static analysis, Yang et al. [217] reported a novel tool based on an incremental linear SVM mechanism that can learn to identify spurious false alarms from more serious matters. Lin et al. [218] concluded five prevalent types of omission faults in Defects4J, and further trained a three-layer neural network to recommend where to break when existing FL techniques were stuck at the dead end caused by such special fault types. Yu et al. [219] presented the Bayesian network-based program dependence graph and introduced it into fault localization. Their approach achieved promising results due to the better reasoning ability than rivals.

Besides, AI techniques can also be applied to localize faults that resided in domain-specific programs or concurrency environments. For example, inspired by qualitative reasoning, a subfield of AI, Hofer et al. [220] managed to automatically extract abstract models and then used them to achieve semi-automated fault localization in spreadsheets. Existing concurrency faults isolation approaches report how failed executions are different from passed ones. However, Terra-Neves et al. [221] stated that such reports were generally of an unnecessarily large size; thus, they used the maximum SMT to generate minimal reports for better performances of the isolation process.

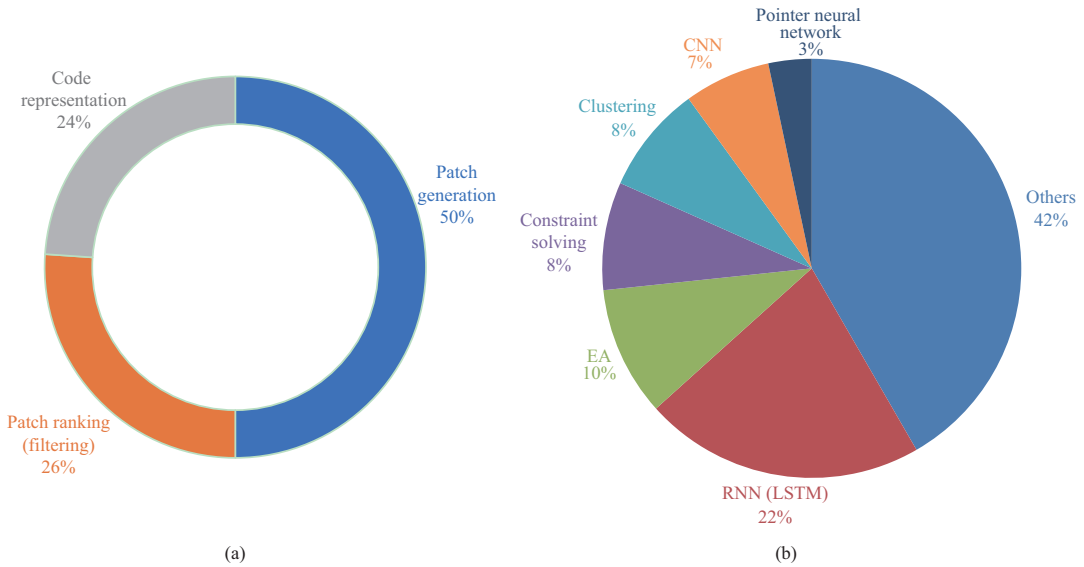

Figure 8 (Color online) (a) Main sub-problems solved by AI techniques and (b) prominent AI techniques in program repair.

4.1.2 *How and which AI techniques are typically used in automated program repair?*

The AI techniques are typically used in automated program repair in three categories roughly, that is, patch generation (50%), patch ranking (filtering) (26%), and code representation (24%)[5)], as shown in Figure 8(a). Prominent AI techniques in this direction include RNN (LSTM) (22%), EA (10%), Constraint solving (8%), Clustering (8%), and so on, as shown in Figure 8(b).

(1) Patch generation. Generating patches for faulty code snippets is an essential way to program repair, which is typically tedious and time-consuming. Researchers intend to integrate AI techniques into this task for making it fully automatic, and many studies have emerged recently. In this category, the commonly-used AI techniques mainly include RNN (LSTM), EA, Constraint solving, Clustering, DNN, and so on.

• **RNN (LSTM).** Many researchers consider program repair to be the process of translating a buggy program to a fixed one, the purpose of which is in line with the property of sequence-to-sequence (Seq2Seq) models. For example, gathering historical correct and incorrect code, Pang [11] proposed to generate candidate patches by building an LSTM-based encoder-decoder architecture. Similarly, Chen et al. [99] also developed a data-driven APR system by training a Seq2Seq network with an encoder-decoder model based on LSTM gates, to automatically fix faulty programs through generating one-line patches. Besides, Tufano et al. [98] trained an RNN encoder-decoder model for their empirical study, which analyzed the applicability of neural machine translation (NMT) to learn patches from available repositories. Pointing out that developers are likely to adopt similar measures to fix code when encountering build-time compilation faults, Mesbah et al. [222] developed an LSTM-based NMT model to translate compile error messages into corresponding delta changes.

• **EA.** To automatically fix vulnerable smart contracts, a type of programs typically running on top of blockchain systems [223], Yu et al. [224] used genetic programming search to generate and validate patches in split sub-spaces without breaking previously passed test cases. To better explore GenProg's search space, Yuan and Banzhaf [225] transformed APR into a multi-objective search problem for generating multi-edit patches, and their approach has shown to be effective in multi-location repair. Based on this, they later implemented a stronger repair model for Java code by using GP. Specifically, they first reduced the search space by excluding some irrelevant ingredients, then represented patches in a lower granularity, and finally designed a novel fitness function to guide the search process of GP [226]. Similarly, for better patch evolution in GP-based program repair, Oliveira et al. [227] introduced new insights into two GP components, i.e., repair representation as well as operators of crossover and mutation, and they offered empirical evidence in terms of both effectiveness and efficiency of their strategy. Le et al. [128] utilized GP to generate candidate patches followed by measuring the degree to which they are similar to existing

5) If a paper is involved in multiple categories at the same time, we either introduce it in each category independently or only put it in one category.

ones. The intuition behind this strategy is that correct patches generally have common characteristics. For giving high-quality feedback when logical faults are exposed in functional programming assignments, Lee et al. [228] first localized the potential root cause built upon dynamic executions, and then fixed it by performing the enumerative search with a search space pruning strategy.

• **Constraint solving.** Constraint solving problems are referred to as a set of objects whose state must satisfy a number of limitations [6)]. Xuan et al. [127] proposed a constraint solving-based tool, which involves fix localization, runtime trace collection, and patch synthesis, for the automatic repair of faulty "if-then-else" statements. Machado et al. [229] proposed to isolate a small set of elements responsible for observed failures, as well as to reformulate a non-failing alternative by using SMT. Pan et al. [230] focused on finding minimal modifications to fix regular expressions. Specifically, taking a program under test (a faulty regular expression), positive instances (strings the tested regular expression should match), and negative instances (strings the tested regular expression should reject) as inputs, their tool utilized two SMT encodings to efficiently explore a pruned search space.

• **Clustering.** Clustering is the task of dividing data samples with samples in the same group being more similar to each other than to those in other groups [7)]. For mining reusable and separate fix patterns, Koyuncu et al. [231] utilized an iterative clustering strategy to mine atomic changes, that is, break patches into atomic units. Experimental results have shown that the mined patterns could lead an APR system to generate correct patches with a higher probability. Gulwani et al. [232] made use of the wisdom of the crowd for more effective automated program repair. Specifically, they clustered together similar correct programs to form classes, and then fixed incorrect ones by generating the smallest patches based on their nearest classes. To overcome the challenges caused by false positives in static analysis tools, Liu et al. [97] proposed to mine fix patterns on over 250 million violations. In particular, they first used an AST differencing tool [233] to collect a huge number of violation fixing changes, then utilized an X-means clustering algorithm to group similar changes. They applied the identified fix patterns to unfixed violations, finding that the majority of the generated patches will be accepted.

• **DNN.** Inspired by the hypothesis that similar faults have similar repairs, Sakkas et al. [234] trained two separate DNN-based classifiers, that is, a binary classifier to predict fault locations, and a multi-class classifier to predict fix templates for given faults. White et al. [235] presented a novel DNN-based approach to reason about repair ingredients with respect to code similarities, for prioritizing and transforming statements in a codebase for patch generation.

• **Other AI techniques.** Yi et al. [236] first proposed a method to automatically fix high floating-point faults in numerical libraries, and then employed the least squares method, a search optimization strategy, to reduce its time overhead. Xiong et al. [129] developed a supervised learning-based program repair tool, which applied the locality of variable uses to program repair and analyzed the documentation of programs, thus automatically mined predicates from source code. Their approach provides only one patch for one defect due to the goal of fully automatic defect repair, and has been proven to be significantly better than the existing SOTA approaches. By offline mining the characteristics of a corpus of existing patches, Jiang et al. [237] first implemented a novel APR tool that produces two separate search spaces based on existing patches and source code, respectively, and then took the intersection of them for a more precise search domain. Experiments showed that their tool successfully fixed bugs that have never been fixed by other techniques. Jiang et al. [238] proposed an NMT-based APR technique comprising three main components, namely, a human being-like programming style learner, a compilable patches finder, and a smaller but stronger search space generator. Their technique can outperform many advanced approaches on well-recognized benchmarks according to their report. Koyuncu et al. [239] developed a four-component framework (including fault localization, fix pattern matching, patch generation, and patch validation) to improve APR tools' effectiveness jointly, in which an ensemble learning approach proposed by them previously, D&C [151], was leveraged to produce a more pertinent fix location for better patch generation. Zhu et al. [240] developed a novel provider/decider architecture based on TreeGen [241], a tree-based Transformer, to produce program edits with ensuring the syntactic correctness according to the given faulty statement and the context.

(2) Patch ranking (filtering). Not all patches generated by APR techniques are correct. To make users have higher confidence with respect to the suggested repair operation, as well as to help achieve the goal of fully automatic defect repair, many studies attempt to employ AI techniques to rank or filter

6) Wikipedia. Constraint satisfaction problem. 2022.
7) Wikipedia. Cluster analysis. 2022.

output patches according to their possibility of being correct. In this category, the commonly-used AI techniques mainly include Constraint solving, and so on.

• **Constraint solving.** For relieving the overfitting problem in APR, Shariffdeen et al. [242] implemented a tool in which plausible but incorrect patches can be identified by solving the arising constraints based on user-provided specifications. Lee et al. [243] encoded the repair for memory deallocation faults in C programs as boolean satisfiability (SAT), and they used a partial MAX-SAT solver to determine the optimal patch in many generated ones.

• **Other AI techniques.** To avoid waste caused by repetitive fixes for a specific bug category, Bader et al. [85] proposed to learn from prior fixes for generating human-like patches. They presented a novel hierarchical and agglomerative clustering algorithm, which organized fix patterns into a hierarchical structure and was able to offer top-most fix(es) to developers. Li et al. [244] exploited a CNN-based binary classifier model, in which each candidate patch is represented as a set of character vectors, to re-rank them and push the correct patch onto the top of the ranking list. To tackle the search space explosion problem, Wen et al. [245] devised a prioritization scheme to estimate candidate patches' likelihood of being correct, one of the key steps of which is to learn faults' context information extracted from substantial open source projects. Wang et al. [246] summarized eight types of static code features that can be used to identify correct patches from overfitting ones, followed by integrating them through six machine learning schemes, including decision table [247], J48 [248], logistic regression [249], naiveBayes [248], random forest [250], and SMO [251]. Existing automated program repair techniques are generally restricted with weak test suites (i.e., an incomplete oracle), which often leads APR techniques to deliver plausible but incorrect patches. For predicting whether generated patches are correct or not, Xiong et al. [252] exploited the behavior similarity of test case executions by using the nearest-neighbor algorithm, and experimental results have demonstrated the accuracy of their approach in terms of filtering out incorrect patches. Moreover, an Eclipse plugin tool to filter out incorrect patches in an interactive way was also implemented [253]. Long and Rinard [126] tried to learn correct code for generating patches automatically, and they presented a novel parameterized discriminative probabilistic model to predict the probability of each candidate patch being correct. To repair faults in object-oriented programs, Saha et al. [254] first extracted four features from the program context and then leveraged a logistic regression model to rank candidate patches. Here we also give some studies that alleviate overfitting problems that exist in AI-based APR systems using manual approaches. For example, Tan et al. [255] provided seven anti-patterns based on manual inspection for better search-based program repair. In their approach, a generated patch fallen into anti-patterns will be rejected even if it has passed all given test cases. By integrating such rules into two existing APR techniques, GenProg [88] and SPR [256], they demonstrated the efficiency and efficacy of this search space-reduction strategy. Le et al. [257] generated patches using provided training test cases and evaluated these candidate patches using held-out cases, to explore the overfitting problem that exists in semantics-based APR techniques, including a supervised learning-based tool [237].

(3) Code representation. In APR tasks, code snippets typically need to be converted to an intermediate form before they can be fed to AI models. This is partly because (i) the input rule specified by different AI models could also be distinct, and (ii) some non-explicit (e.g., semantic and syntactic) features of code snippets are valuable and need to be mined. To that end, many studies aim to utilize AI techniques to better represent raw code snippets. In this category, the commonly-used AI techniques mainly include RNN (LSTM), and so on.

• **RNN (LSTM).** Li et al. [244] treated program repair as a code transformation learning task, and leveraged tree-based RNN to represent source code as corresponding ASTs. Yasunaga and Liang [258] utilized an LSTM model to encode faulty source code, and then trained their repair model using the represented code as well as error feedback, an erroneous line index, and the repaired line. While most neural program embedding techniques are based on syntactic features, Wang et al. [259] focused on developing a novel semantic program embedding approach built upon RNN to improve the performance of existing APR systems. Gupta et al. [260] designed a deep reinforcement learning-based framework for the repair of syntactic faults, in their approach, an LSTM network is employed to embed the program text. Notice that the motivation of this study is that compiler error messages might inaccurately localize the root cause and are sometimes difficult to understand [261]. For addressing a newly identified type of faults, variable-misuse faults, Vasic et al. [215] quoted the enumerative strategy proposed by Allamanis et al. [262] and concluded its drawbacks, then presented a multi-headed pointer networks model that jointly and directly localized and repaired variable-misuse faults. In their approach, tokens are embedded and processed using an LSTM model.

• **Other AI techniques.** Gember-Jacobson et al. [263] cast the configuration repair in network control planes as a constraint solving problem, and used SMT to handle it. In their work, the solution to the constraint is dependent on a digraph-based representation, which is able to capture dependencies between traffic classes. Arguing that existing APR techniques typically depend on hard-code patterns, Lutellier et al. [100] presented a novel context-aware NMT to represent the buggy code and its surrounding context separately. To detect and repair faults in Javascript programs, Dinella et al. [264] represented a buggy program as a graph and then exploited GNN to map the graph into a representation in fixed dimensional vector space, along with further presenting an end-to-end framework that consisted of fault localization, fixing patterns prediction, and patch generation. Gupta et al. [265] proposed a novel neural program generation technique that consisted of synthesis, execution, and debugging components. In contrast to previous approaches that output the final program directly, their framework employed a CNN model and an LSTM-based neural program debugger to fix the candidate program when it failed, where a CNN-based execution trace embedding technique is used to better reveal the semantic errors. In addition, Tian et al. [266] proposed employing four distributed representation learning techniques to learn deep features of the correctness property of patches.

Finding 1.1

— Answer to RQ1.1 (How and which AI techniques are typically used in software debugging?)

▶ Various AI techniques have been extensively integrated into existing fault localization tasks, and they are typically used in six sub-problems: (1) Feature extraction(ensemble). In this category, RNN (LSTM), CNN, RF (DT), EA, LtR, etc. are commonly used. (2) Suspiciousness calculation. In this category, MLP, CNN, EA, LtR, RF (DT), etc. are commonly used. (3) Information retrieval. In this category, DNN, CNN, EA, Word embedding, etc. are commonly used. (4) Test cases enhancement. In this category, EA, RF (DT), etc. are commonly used. (5) Multi-fault debugging. In this category, CNN, EA, etc. are commonly used. (6) Others. We find that in the AI4FL domain, 25%, 24%, and 23% of the studies used AI techniques for feature extraction (ensemble), suspiciousness calculation, and information retrieval, respectively, indicating that the application of AI techniques in these three sub-problems is relatively balanced. Besides, 12%, 7%, and 9% of the studies used AI techniques for test cases enhancement, multi-fault debugging, and others, respectively, implying that AI techniques may still have great potential for solving these sub-problems.

▶ Various AI techniques have also been extensively integrated into existing automatic program repair tasks, and they are typically used in three sub-problems: (1) Patch generation. In this category, RNN (LSTM), EA, Constraint solving, Clustering, DNN, etc. are commonly used. (2) Patch ranking (filtering). In this category, Constraint solving, etc. are commonly used. (3) Code representation. In this category, RNN (LSTM), etc. are commonly used. We find that in the AI4APR domain, 50%, 26%, and 24% of the studies used AI techniques for patch generation, patch ranking (filtering), and code representation, respectively, suggesting that AI techniques can be more used to solve the latter two sub-problems in the future.

▶ We find a trend that the diversity of AI techniques used in fault localization tasks is higher than that in automated program repair tasks. The crux of FL is mainly to pinpoint the location of the underlying fault(s) according to given clues in various forms, such a requirement of building linkages could be satisfied by many AI models, for example, artificial neural networks can be used to link high-level failure information to fine-grained code for their capability of mining implicit feature, and evolutionary algorithms can adjust the FL features' weight for their characteristic of heuristics. The learning approach of AI4FL models mainly involves supervised learning (e.g., mining historical data and extracting fault-related patterns) and unsupervised learning (e.g., employing clustering to determine the linkage between failures and multiple root causes).

The main purpose of APR is to generate and validate patches for suspicious code snippets, which are generally performed in a manual way since it might be hard to find a proper type of AI models that fit the need. Nonetheless, RNN (LSTM) is preferred for its advantage of capturing previous information and relieving the long-term dependency problem, which is suitable for extracting the fault's context information in APR. Besides, AI models are also used in other phases in APR, such as generated patches re-ordering, code snippet representation, etc. The learning approach of AI4APR models mainly involves supervised learning (e.g., extracting the fixed pattern based on faulty-fixed code instances for unseen faults).

Besides, according to our investigation, models in both AI4FL and AI4APR are typically trained and tested by two strategies, namely, using all tests as training samples and virtual tests as testing samples,

or using part of tests as training samples and the remaining as testing samples.

► In our investigation, we observe that many researchers have released a tool for implementing their AI4FL and AI4APR techniques. To provide stakeholders with a convenience index, we extract and list these tools in Tables A1 and A2 in Appendix A, where the link to the tool is also given.

4.2 What types of data do researchers tend to use as the input of AI models in the context of software debugging? (RQ1.2)

One of the most significant properties of AI models is data-driven, therefore, apart from the architecture and training strategy of AI techniques, the data fed into AI models deserve special attention. Based on the concept of different types of input data which are suited for different debugging approaches and AI models, we roughly categorize input data into three types: execution-based, code-based, and text-based ones.

4.2.1 *Execution-based input*

Execution-based data are mainly referred to as characteristics (e.g., trace, slice, invocation, and their derivations) produced during dynamic testing, which involves both test cases and the program under test. Kim et al. [140] presented a novel LtR model that used MBFL and SBFL features for effective fault localization. In particular, these two groups of features extract dynamic characteristics of target programs based on the execution information, such as mutation-based scores (calculated by mutants from MBFL) and risk values (calculated by spectrum from SBFL). Likewise, Pan et al. [143] considered a series of dynamic features for accurate fault localization, including raw spectrum information, eleven risk evaluation formulas of SBFL, three MBFL techniques, stack trace, and dynamic program slicing based on statements, and then fed them into a neural ranking model. Li et al. [157] argued that fault localization is useful for developers at both method and statement levels; thus they constructed spectrum-based and mutation-based matrices at these two granularities, respectively, followed by inputting these vector/matrix representations to the CNN model. To predict FL techniques' effectiveness, Golagha et al. [138] collected 18 dynamic features to investigate the key factors in FL tasks using five machine learning algorithms.

In particular, program coverage is a matrix consisting of the numbers 0 and 1, which records the execution trace of each statement against the test suite and has been broadly used to train AI models. Mahapatra and Negi first utilized the coverage matrix to train a GA-RBF model, then constructed a set of virtual test cases, and fed them into the trained model to obtain statement suspiciousness scores [159]. Stating that a limited amount of samples may restrict AI models' capability to learn complex functions, Zheng et al. [73] and Zhang et al. [71] proposed to train a DNN model and a CNN model using program coverage and test cases' results (passed/failed) in 2016 and 2019, respectively, then they both generated a virtual test suite for the trained model to obtain the probability of each test case being failed. Two years later, Zhang et al. [74] further exploited the coverage matrix and corresponding test results on three DL models, RNN, MLP, and CNN, to assess the advantage of deep learning for fault localization.

Execution-based input has been broadly used in fault localization and automated program repair tasks, especially in spectrum-based fault localization. For example, numerous SBFL risk evaluation formulas were proposed by human intelligence and artificial intelligence, to incorporate code coverage, a classical type of execution-based input. In addition, there are also many studies employing deep learning approaches to handle code coverage. In our opinion, solely developing novel debugging techniques is not enough to improve debugging effectiveness, because the underlying data source, such as code coverage, is still poor due to its simplicity. Therefore, we recommend that future researchers focus on both proposing a new technique and extracting more diverse execution-based information.

4.2.2 *Code-based input*

In addition to execution information, several static features of faulty source files can also be extracted and utilized to train AI models. Code-based data is mainly referred to as characteristics extracted from source code at different levels (e.g., files, methods, statements, and tokens) in varied forms, such as dependency information and call graphs. For example, apart from the aforementioned MBFL and SBFL features, Kim et al. [140] gathered and fed predefined code-based static features into the LtR model. These features are at file, function, and statement levels, such as fan-in and fan-out of a file dependency graph, the number

of global/local variables a function reads/writes, and the number of operators/variables a statement uses. Similarly, Pan et al. [143] first extracted 5 types of static features, i.e., indentation level, # of lines, # of comments, # of tokens, and # of chars of statements, and then combined them with the other dynamic features through the attention mechanism for better fault localization. In like manner, Golagha et al. [138] first defined 15 static features (e.g., mean cyclomatic complexity and mean depth of inheritance hierarchy) and other 55 features, then selected five common machine learning techniques to explore which metrics have the strongest influence on FL techniques' performance. In addition to the traditional program spectrum, Zhang et al. [203] also used static call graphs to mine the connections among program entities, their enhanced spectrum based on the PageRank algorithm is proven to be beneficial to the performance of traditional SBFL techniques. Yang et al. [267] considered extracting useful information from stack traces, source code, and bug reports, and fed these features into an autoencoder and a CNN model successively to establish linkages between bug reports and source code. Besides, they also converted source code into multiple lines with tokens and employed a Seq-GAN to perform program repair.

A number of primary studies also attempted to learn the relationship between correct code and buggy code for automated program repair. Li et al. [244] regarded the APR process as a code transformation learning task, where source code was first fed into a tree-based RNN to generate the corresponding AST, and then transformation learning layers were trained on these trees. Based on the concept of features in successful patches can help with finding correct patch candidates in search space, Long and Rinard [126] took successful human patches as training data to learn and exploit properties of correct code by the parameterized discriminative probabilistic model. Saha et al. [254] pointed out that reporters sometimes write about possible fixes in bug reports; thus they extracted features from previous bug fixes and bug reports, used them to train a logistic regression model offline, and then employed information in bug reports to prioritize candidate patches.

Code-based input is typically extracted from program code snippets directly, without the dynamic execution. It often serves as static features in fault localization or patch ingredients in program repair, many previous studies have demonstrated its promise in these tasks. In our opinion, there is a trade-off between the availability and the effectiveness of the code-based input. Specifically, we can obtain a diversity of code-based information without much computational cost; for example, the static call graph, the number of variables, etc. can be collected in an uncomplicated way. But the capability of code-based information to help with software quality assurance could also be limited; for example, static analysis tools often report false alarms. Therefore, we recommend that future researchers utilize code-based information according to their requirements (e.g., effectiveness first or efficiency first). A stronger technique that makes full use of code-based information, or better combines code-based information and dynamic execution information, is also expected.

4.2.3 *Text-based input*

Text-based data is mainly referred to as bug reports and natural language parts in source code. A piece of bug report is typically described in natural language, which mainly comprises a summary, meta fields, descriptions, and comments related to faults. Based on the concept of the more relevant one source file is to the bug report, the more suspicious it is, many IRFL techniques take bug reports as input to find the faulty element(s). However, terms used by developers to describe faults in a bug report are often different from their counterparts (i.e., terms and code tokens) in source code. To address this lexical mismatch problem, Lam et al. [75] linked bug reports to the corresponding buggy files and non-buggy files, respectively, features extracted from them were fed into an autoencoder to learn the relevancy of files with respect to bug reports. Moreover, Wang and Lo [10] managed to complement rich information sources for better fault localization. They put together version history, similar report, structure, reporter information, and stack trace, and then employed a genetic algorithm to tune the five components' weights. In addition to bug reports, Ye et al. [125] also selected API documents, tutorials, and reference documents to train word embeddings; specifically, they managed to learn embeddings of natural language words and source code tokens by Skip-gram model. For extracting useful information hidden in the natural language part of code, Pradel and Sen [174] first translated code examples into vectors via a Word2Vec network, then tried to learn name-based bug detectors from correct and incorrect code.

We observe a trend toward more and more widespread use of text-based input. For example, in fault localization tasks, many information retrieval-based approaches intend to build linkages between the natural part of bug reports and source files. And in program repair tasks, documents/specifications

written in natural languages can also be used to guide patch generation or correctness identification. Despite the fact that using text-based information to boost FL and APR techniques still suffers from many challenges, for example, the lexical gap between natural languages and programming code is hard to narrow, we believe text-based information has great potential in future AI4SD research, considering the need for fully automatic debugging, the broad prospects of human-computer interaction, and the rapid development of natural language processing techniques.

Finding 1.2

— Answer to RQ1.2 (What types of data do researchers tend to use as the input of AI models in the context of software debugging?)

▶ For fault localization, researchers typically utilize three types of data to train AI models. (1) Execution-based input. Dynamic characteristics collected during program execution against a given test suite, such as coverage, dynamic program slices, invocation, and so on, are frequently fed into AI models. (2) Code-based input. Researchers often extract useful features from source code, or convert source code into diverse intermediate representations by AI techniques to serve fault localization. (3) Text-based input. Bug reports, API documents, and other descriptions written in natural languages can also be linked to suspicious program entities through AI models.

▶ For program repair, researchers typically utilize the following two types of data to train AI models. (1) Code-based input. Developers prefer to mine historical data (such as buggy and fixed code) in software repositories, then construct a specific AI model to extract automated fixing patterns based on these code snippets. (2) Text-based input. Bug reports sometimes contain possible fixes; thus developers also use them to train AI models for prioritizing candidate patches.

▶ The three forms of inputs all have their own strengths and weaknesses. For example, execution-based data are more commonly used in software debugging since they can dynamically reflect or monitor the state of programs, however, the resulting high time costs prevent it from being adopted broadly in practice. Code-based data are often utilized as a representation of code snippets, which can reduce the complexity of debugging tasks to an extent, however, it typically needs to manually define and extract code features. Text-based data are in a lightweight form and are easier to get than execution-based data, but it is mostly used at method and file levels, since text-based data are typically too coarse-grained to be linked to a statement.

5 How can SD be used to assure the quality of AI systems? (RQ2)

To answer RQ2, we review 34 studies that investigate software debugging techniques applied in AI systems in this section. Among them, 20 studies are technical studies that involve fault localization or automated program repair for AI systems (see Subsection 5.1), while the remaining 14 are empirical studies in terms of summarization or investigation of common faults/symptoms in AI systems or libraries (see Subsection 5.2).

5.1 How software debugging techniques are used to localize or repair faults in AI systems? (RQ2.1)

In this subsection, we summarize how researchers technically apply software debugging in AI systems from two separate perspectives, FL4AI and APR4AI.

5.1.1 *Fault localization techniques applied in AI systems*

Many researchers are concerned with localizing faults at different granularities (e.g., neuron, layer, function, pipeline, etc.) for assuring AI systems quality; relevant studies include [109, 117, 268–270]. For example, to eliminate debugging challenges caused by the black-box property, Wardat et al. [270] utilized historic trends in values propagated between layers to (1) determine whether a given deep learning model is buggy or not, and (2) determine in which layer and phases the fault exists. Eniser et al. [117] presented a novel technique that first created a hit spectrum for each neuron, and then employed SBFL formulas to obtain risk scores for generating a ranking list of suspicious neurons.

Besides, to detect faults in AI programs written in dynamic languages (such as Python), Dolby et al. [271] first translated normal language AST into common AST, then employed two internal representation (IR) generation strategies to convert common ASTs into WALA's IR, which enables WALA, a static

framework, to be applied in AI code that is built upon TensorFlow. Focusing on code-level faults in AI algorithms, Cheng et al. [272] attempted to simulate faults in AI systems by mutation testing for manifesting these faults as many as possible. Instead of testing deep learning models, Pham et al. [269] stated that multiple implementations of the same deep learning algorithm can detect inconsistencies among these implementations, thus they managed to find and localize faults in deep learning libraries based on this hypothesis. To infer root causes of failures in complex computational pipelines of machine learning tasks, Lourenço et al. carried out two pieces of research on machine learning pipeline debugging. They first attempted to iteratively execute multifarious pipeline instances to find minimal definitive root causes in 2019 [109], and then proposed to find root causes by executing fewer pipeline instances in 2020 [268].

5.1.2 *Automated program repair techniques applied in AI systems*

Existing AI model repair techniques are mainly designed from two aspects, i.e., data and neural network connectivity [34].

• **Data-oriented repair.** Developers conduct AI model repair predominantly in terms of training data, for its data-driven property [111–114,130]. Most researchers first divide datasets into several subsets to identify which one has the most impact on the prediction bias, and then remove a specific subset to reduce its negative impact on the model prediction. For example, Zhang and Chan [114] presented to divide the datasets into multi subsets and retrained the model over them to generate reduced models, then tagged them as correct or incorrect for further optimizing the original model. Krishnan and Wu [111] sought to isolate a small set of training samples that can most perturb labels or features, to address mispredictions that occurred in DNN models. Realizing the potential risk caused by data pollution, Cao et al. [113] proposed a novel approach and a corresponding system to clean up polluted training data by checking whether the previously misclassified samples were correctly classified after removing suspicious subsets. Additionally, the model predictions can also be influenced by modifying the model labels. For example, Koh and Liang [112] proposed to use the influence function to find the relationship between training data and prediction. Specifically, they attempted to change the training data through upweighting and perturbing: the former is to delete a training sample and observe the effect, whereas the latter focuses on the problem of "how would the model's predictions change if a training input were modified". Targeting underfitting and overfitting problems in machine learning models' training process, Ma et al. [130] proposed to debug neural network models via state differential analysis (for identifying fault-related internal features) and training input selection (inspired by the program input selection in regression testing), which outperformed the state-of-the-art technique in terms of both effectiveness and efficiency according to their results. Focusing on the correctness of labels in predictive model construction, Wu et al. [273] observed many mislabeled samples in security bug report prediction datasets. They first manually corrected these problematic labels, and then measured the performance of classification models against corrected and uncorrected datasets, respectively, the results of which highlighted the negative influence caused by noisy data. Seeing the potential harm of problematic input word embeddings to the accuracy of RNN models, Tao et al. [274] presented a novel technique to analyze such negative influence and provided an embedding regulation/tunning algorithm for the repair.

• **Neural network connectivity-oriented repair.** Some researchers try to restructure the AI model to perform fault repair. For example, Guidotti et al. [118] argued that as long as the accuracy of the hybrid network (replace non-linear layers with linear ones) is comparable or outperforms that of the original model, then (1) the former can replace the latter, and (2) repairing the latter can be replaced by repairing the former. Based on this hypothesis, they further proposed to tackle the repair of CNNs using transfer learning and convex programming, exemplifying the concept of AI debugs AI. Similarly, Kim et al. [275] utilized random forests to extract rules that were correlated with failures of the object detector. The mined rules have been demonstrated to be non-trivial for the model's correct rate. Sotoudeh and Thakur [276] proposed decoupled DNNs, a new architecture that can be converted from any existing DNN, to reduce the repair for DNN to a linear programming problem. Song et al. [277] integrated two-stream self-attention into NMT models, by doing so the previous tokens can be corrected by the content stream and the next token can be predicted by the query stream at the same time. Zhang et al. [278] assured AI systems quality at the architecture level; they proposed a novel static analysis technique that used abstract interpretation to identify numerical faults in DL software, which could help prevent the potential loss before the model training process. Schoop et al. [279] implemented a novel system to identify and repair bugs in DL programs, in which the model structure and behavior of DL programs are

checked against existing heuristics used by experts, enabling to pinpoint the root cause(s) and provide fixes suggestions automatically. Zhang et al. [280] summarized five typical types of faults in DL models' training phase, namely, vanishing gradient, exploding gradient, dying ReLU, oscillating loss, and slow convergence, followed by proposing a novel tool to monitor the DL model (e.g., record neuron activations, and layers and their configurations) and automatically fix the detected fault(s).

In addition to the mentioned two categories, there are also some studies fixing faults in AI models in other ways. For example, Agrawal et al. [131] utilized differential evolution, a search-based optimizer, to improve the performance of LDA, a widely-used while non-deterministic algorithm in the field of topic modeling, by tuning its parameters in a heuristic way. Sun et al. [281] proposed to improve the consistency of translators based on the black-box repair. In their approach, the utilization of mutation and metamorphic testing enables the repair to run without training data and the source code of systems under test.

Finding 2.1

— Answer to RQ2.1 (How software debugging techniques are used to localize or repair faults in AI systems?)

▶ Some researchers conduct technical studies that localize or repair faults in AI-related modules by employing traditional software debugging techniques, or even AI approaches.

▶ We have not extracted explicit common points in localizing AI systems' faults since the diversity and individuality of the investigated techniques. Nonetheless, we can conclude that the localization activity spreads from AI libraries to AI models since both of them may contain faults, and various strategies or techniques are typically utilized in this process, such as mining model's historical behaviors, conducting multiple implementations (similar to N-version programming), and obtaining the probability of a neuron being suspicious.

▶ As for program repair, the majority of researchers focus on fixing faulty AI models from the perspective of data (for example, isolating fault-prone datasets or retraining the original model using subdatasets), with some studies also attempting to restructure neural network connectivity or tune the parameters.

▶ In our investigation, we observe that many researchers have released a tool for implementing their SD4AI technique. To provide stakeholders with a convenience index, we extract and list these tools in Table A3 in Appendix A, where the link to the tool is also given.

5.2 What types of faults occur frequently in AI systems and AI libraries? (RQ2.2)

Apart from technical work, a large number of researchers also empirically analyze common fault types and fix patterns in AI systems or mainstream deep learning libraries. For example, Islam et al. [34] investigated the bug fix patterns in five deep learning libraries and revealed the distinction between AI models repair and traditional software repair. Jebnoun et al. [282] explored code smells in AI applications and discovered a co-existence between code smells and faults contained in AI code. A substantial number of AI repositories have grown on GitHub and other open-source platforms with more and more AI researchers releasing their code and datasets. Fan et al. [283] collected over 1000 academic AI repositories and attempted to mine the common features among them; they discovered that having too many copy-paste snippets can lower both the popularity of repositories and the quality of corresponding AI systems. Apart from explicit faults, software systems can also contain a variety of prospective faults. Liu et al. [284] conducted empirical studies to investigate different types of technical debt in seven deep learning frameworks. They first highlighted that defect debt (unresolved defects) could lead to unstable data dependencies or unexpected behavior in flawed framework-based AI systems, and then further concluded that defect debt is one of the fastest removed debt types [285]. To investigate potential software crashes or other unexpected severe problems caused by improper versions of deep learning libraries, Han et al. [286] extracted open-source projects that relied on TensorFlow, PyTorch, and Theano for empirically analyzing the dependency networks that existed among them.

Additionally, Sun et al. [287] studied the faults associated with three open-source AI projects, Scikit-learn, Paddle, and Caffe, where they summarized seven fault types and found the most frequent type is Variable faults, accounting for 29.79%. Islam et al. [120] investigated faults in deep learning models built upon Caffe, Keras, TensorFlow, Theano, and Torch and found that (1) data bugs appear most of the time in all libraries, (2) structural logic bugs are the second major fault type. Furthermore, they indicated that most of the faults in deep learning programming happen at the data preparation stage,

with 32% of faults occurring in this stage. Besides, observing that some tasks on Microsoft Philly fail after a long execution time due to faults, Zhang et al. [288] sought to have a systematic understanding of failures in deep learning jobs; thus they investigated failed jobs on Philly and found that 39.4% of faults caused by Path not found in execution environment. Aiming at constructing a taxonomy for real faults in deep learning systems, Humbatova et al. [121] analyzed thousands of faults that resided in TensorFlow, Keras, and PyTorch, and summarized five first-level fault types, namely, Model, GPU usage, API, Tensor & inputs, and Training. They further divided these types into 11 second-level, 8 third-level, and 93 fourth-level fault types, and concluded a series of root causes in deep learning models, such as wrong tensor shape and suboptimal network structure.

Zhang et al. [119] and Jia et al. [289] made comprehensive studies on faults related to TensorFlow. On the one hand, Zhang et al. summarized four types of symptoms and found Error (analogous to exceptions or crashes in conventional applications) is the most common among them, accounting for up to 64%. On the other hand, Jia et al. found that Functional error (a program does not function as designed) is the most common symptom inside TensorFlow with 35.64%, compared with the other five symptoms, namely, Crash, Hang, Performance degradation, Build failure, and Warning-style error. Meanwhile, they also noted that 34.91% of faults were localized in the Contribution component, and 26.42% and 11.79% in API (interface) and Kernel (deep learning algorithms) components, respectively.

Apart from deep learning libraries, some researchers investigated faults in specific application domains. For example, Garcia et al. [290] made an empirical study on faults in autonomous vehicle software systems and found that Incorrect algorithmic implementations and Incorrect configurations are the most common bug categories. Chen et al. [291] classified the deployment faults of mobile deep learning Apps according to five phases in deployment. They observed that 48.4% and 36.2% of faults occurred during the Model conversion stage and Inference stage, respectively, with the remaining faults occurring during Deep learning integration, Data preparation, and Model update stage.

Finding 2.2

— Answer to RQ2.2 (What types of faults occur frequently in AI systems and AI libraries?)

▶ Researchers tend to mine software repositories or code snippets on platforms like GitHub and StackOverflow, to manually collect historical and real-world faults and analyze classic root causes, symptoms or fix patterns in AI models or deep learning libraries.

▶ There are two fault types that frequently occur in AI systems, data faults and logic faults. Data faults primarily refer to the wrong data input formats or dimensions, wrong data labels, and wrong file paths. Logic faults primarily refer to the wrong function definition (including the wrong usage of variables and parameters), and the wrong connection structure. Additionally, wrong API usage and wrong GPU usage also occur frequently and are worth attention.

▶ Faults in AI systems appear largely in the data preparation and model inference phases; developers are suggested to pay more attention during these stages of development.

▶ We observe that the authors of these empirical studies tend to manually collect datasets for their investigation, and some of them make the collected datasets available for other researchers' replication or further exploration. We summarize and list these manually collected datasets in Table A3 in Appendix A for the convenience of interested readers.

6 What configuration do researchers prefer to use in AI4SD and SD4AI experiments? (RQ3)

To answer RQ3, we summarize the common experimental settings in AI4SD and SD4AI from two perspectives: datasets (programming languages) and metrics.

6.1 Which datasets (programming languages) are most commonly used? (RQ3.1)

As for AI4SD, most researchers tend to use publicly available datasets for their high stability, reliability, and comparability. For example, we observe that Defects4J, a Java benchmark released by Just et al. in 2014 [292], was adopted by most researchers. In addition, AspectJ, SWT, JDT, Tomcat, Eclipse, and Siemens Suite are also preferred to be chosen by authors. The number of papers adopting these datasets is shown in Figure 9. Among them, Siemens Suite is a little bit old thus not the first choice, albeit it played a dominant role in prior research [293–297]. Defects4J has received more and more attention since it was published in 2014 [292], and it is being in widespread use in the fields of both FL and APR.

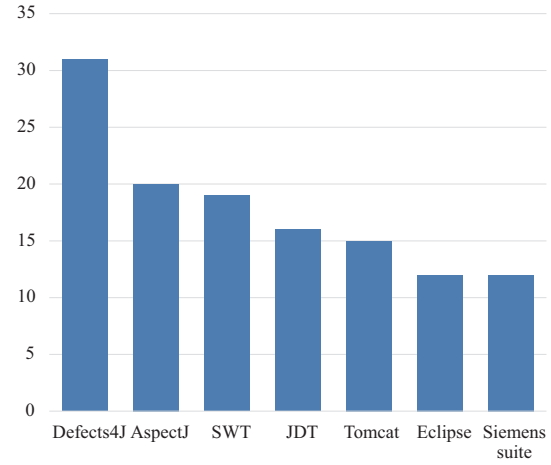

Figure 9 (Color online) Most commonly used datasets in AI4SD.

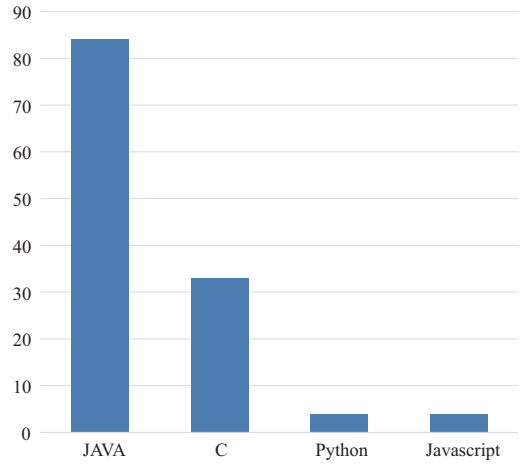

Figure 10 (Color online) Most commonly used programming languages in AI4SD.

Besides, researchers should also consider the potential problems of datasets in use. For example, projects in Siemens Suite are generally too small to support evaluating an approach sufficiently, and Defects4J has a benchmark overfitting problem for FL, i.e., FL techniques may perform extremely well on Defects4J but lose their capability on unseen faults. Researchers have presented some other datasets (e.g., Bears [298] and Bugs.jar [299]) to solve this problem.

As for programming language, researchers in AI4SD primarily focus on Java and C programs, with only a few of them conducting experiments on programs written in Python, Javascript, or other languages. The number of papers that used datasets written in these languages is shown in Figure 10.

As for SD4AI, the authors of empirical studies tend to collect the relevant data from GitHub and StackOverflow manually, thus we only focus on technical studies that investigate how to localize or repair faults that resided in AI models in this section.

According to our investigation, the most widely-used open-source datasets of the selected SD4AI papers include MNIST, CIFAR-10, ImageNet, and Enron-spam. Considering most experimental datasets used in SD4AI are in image or text forms, rather than traditional programs written in a specific programming language, we do not summarize the most popular programming languages.

We further point out three important dataset-related problems based on our investigation. Firstly, many existing benchmarks suffer from the class imbalance problem. The lack of faulty program elements (e.g., statements, functions, or files) could mislead a trained classifier in AI4SD into preferring the majority, since labeling a faulty element as innocent is low-cost, resulting in a poor capability of generalization. The second is the coincidental correctness problem. If a test case returns the testing result of passed albeit it covers the faulty statement, it is difficult to reveal the masked fault since we can hardly link it to observed failures, especially in multi-fault scenarios. Thirdly, the oracle problem is worthy of discussion because the expected outputs are generally difficult or even impossible to obtain for faulty programs, which hinders datasets' practicality since the testing results cannot be labeled properly. We believe effort that aims at relieving or eliminating the above challenges will be beneficial to the robustness of AI4SD/SD4AI techniques.

Finding 3.1

— Answer to RQ3.1 (Which datasets (programming languages) are most commonly used?)

▶ Defects4J is the most prominent dataset in AI4SD experiments, while MNIST, CIFAR-10, ImageNet, and Enron-spam are typically selected by researchers to conduct experiments in SD4AI.

▶ Java and C programs serve as benchmarks in most AI4SD experiments.

6.2 Which metrics are most commonly used? (RQ3.2)

As for AI4SD, we list the most commonly-used metrics in AI4FL and AI4APR, respectively. Table 6 shows four typically used metrics in fault localization tasks. Mean average precision (MAP) is the most popular one among them, as evidenced by the fact that it is used in over 30% of the selected papers. Apart from the metrics listed in Table 6, the performance of novel approaches in AI4FL is also

Table 6 Most typically used metrics in AI4FL

Name	Description
Mean average precision (MAP)	The mean of the average precision of all faults [77, 136, 158, 165, 185]
Recall at Top-n	The number of faulty programs that at least one buggy element appears in the top-n position of the ranking list [157, 158, 239]
Mean reciprocal rank (MRR)	The reciprocal of the position of the first buggy element in the ranking list [77, 158, 185]
Acc@n	The number of faults localized within top-n elements of the ranking list [140, 161, 165]

Table 7 Most typically used metrics in AI4APR

Name	Description
Accuracy	The percentage of patches that correctly fix faults [234]
Overfitting rate	The number of overfitting patches out of the number of total patches generated [257]
Top-k	The number of correct patches that can be found by considering top-k statements delivered by the employed FL technique [239]

Table 8 Most typically used metrics in SD4AI

Name	Description
Precision	The fraction of identified root causes that are in fact faults [109]
Recall	The fraction of AI models for which at least one/all root causes is (are) pinpointed [109, 268]
Contrast	The loss and accuracy of AI models before and after fixing [117, 118, 270]

evaluated by nearly 40 other metrics. We find that these prevalent metrics are usually related to the sub-problem of suspiciousness calculation. Specifically, they are often based on the extent to which the faulty program element(s) can be pushed onto the top of the ranking list, which is generated according to the calculated suspiciousness. Such a phenomenon indicates that the core of fault localization is to measure the likelihood of program entities being faulty. Table 7 shows three frequently used metrics in automated program repair tasks, these metrics are primarily built upon the number of correct patches generated by the APR technique under evaluation. We find that these prevalent metrics are usually related to the sub-problems of patch generation and patch ranking (filtering), namely, the extent to which the output patch can fix the observed failures, and the extent to which these correct patches can be delivered to developers with high priority.

We list the most popular experimental metrics in SD4AI in Table 8, finding that Precision, Recall, and Contrast are typically utilized by researchers. Among them, the functionality of the first two is similar to that of the metrics in AI4FL, while the last one is specially designed for SD4AI tasks.

Finding 3.2

— Answer to RQ3.2 (Which metrics are most commonly used?)

▶ MAP, Recall at Top-n, MRR, and Acc@n are typically used to evaluate the performance of novel AI4FL techniques, and researchers tend to use Accuracy, Overfitting rate, and Top-k to assess AI4APR techniques' effectiveness.

▶ Precision, Recall, and Contrast are often used to demonstrate the effectiveness of novel approaches in SD4AI.

7 Challenges and opportunities

In this section, we discuss the challenges and potential future directions in the intersection of AI and SD.

7.1 In the field of AI4SD

Features employed by AI techniques for debugging should be prioritized. We find that researchers tend to mine and use a large number of features with AI techniques to improve the effectiveness of existing fault localization techniques. For example, static features like title, summary, description, and stack trace of bug reports, dynamic features like code coverage, risk value, and mutation-based information, as well as code-based features like the package name, class name, variable name, length of statements (bytes), and the number of operators that a statement uses, are utilized to improve the effectiveness of existing FL techniques including IRFL and SBFL. However, we also note that many studies further indicate that various features contribute differently to the improvement of FL techniques'

effectiveness [10,138,140,151]. In other words, some features may not have a positive impact on the technique into which they are integrated but may increase the complexity. Therefore, we suggest researchers evaluate the effectiveness of features employed by FL tasks before using them. Furthermore, we believe that finer investigations of features for cross-project fault localization or specific task-oriented domains are future research directions in AI4FL.

AI4SD techniques should be evaluated under real-world scenarios. Many approaches that apply AI techniques to FL or APR tasks are evaluated in datasets downloaded from SIR (Software Infrastructure Repository) or other toy benchmarks, in which most of the faults are artificially seeded. Although some researchers used mutation-based strategies to generate new faults to expand the scale of their dataset, the evaluation of novel AI4SD techniques can still be biased since mutated faults are distinct from real faults, in other words, techniques that work well in artificial-fault environments may not work as well in real-fault environments. As a result, we suggest future researchers evaluate their novel AI4SD techniques in a real-fault scenario to demonstrate the practicality. More importantly, we believe that a paucity of real-fault datasets is one of the reasons for researchers' preference for artificial faults. For example, Defects4J, one of the few real-fault Java datasets, has been exhaustively utilized to evaluate a variety of novel AI4SD techniques, whereas additional real-fault datasets that are equally well-recognized are hard to access. Therefore, we believe that building more real-fault datasets in C, C++, and other languages is a promising future topic.

Utilizing AI techniques to fill the gap between plausible and correct patches. Most of the existing AI4APR techniques use AI techniques to generate plausible patches that can pass all existing test cases. However, a plausible patch may not be able to pass additional and unseen test cases. This overfitting problem has become a bottleneck for the further development of APR techniques. We believe that AI techniques can be used not only to generate candidate patches, but also to predict the possibility of plausible patches being correct at the same time. For example, an AI model can be used to develop a probability model that establishes a linkage between static information such as program specifications and candidate patches. The strength of the linkage (that is, the similarity between a patch and a specification) determines the likelihood of a plausible patch being correct. This extended application of AI techniques in APR can effectively expand the space for the development of AI4APR.

Conducting APR in agile development with AI techniques. In addition to the effectiveness, the efficiency of APR techniques is also nontrivial. In actual software development processes, the speed of program repair will directly affect the iteration and release of the product. We believe that in the future, it is worth employing AI techniques to combine the advantages of software healing and software repair, so that existing APR techniques can be applied to agile development. Specifically, the goal of software healing is to turn a failed execution into a passed one at runtime (applied on the deployed application without modifying source code), whereas the goal of software repair is to change the source code to remove the root cause (often deployed in-house such as the debugging process). AI approaches can be used to improve the efficiency of APR techniques, which enables the latter to be performed during runtime like software healing, and thus achieves just-in-time program repair in agile development.

Enhancing the generalization ability of AI4SD models. Albeit AI-integrated debugging techniques are shown to be highly effective in experiments, adapting them to a broader circumstance that is independent of training setups is more essential. For example, an AI model trained for fault localization or program repair may not perform as well on unseen faults (e.g., written in different languages, having distinct contexts, etc.) as on the training set. There are many possible reasons for such degradation, including underfitting and overfitting. For the former, future researchers can consider adding new features from different levels (such as files, functions, statements, or even tokens) and different sources (such as coverage, data and control flows, or text descriptions), or feature crosses. For the latter, manually selecting which failure/fault-related features to keep, reducing the magnitude of features, and regularizing, are all worth trying.

Debugging in multi-fault scenarios. Almost all existing research hypothesizes that a buggy program contains only one fault, despite the fact that it is often unrealistic with the increases in software complexity and scale. In multi-fault scenarios, failed test case(s) can be triggered by distinct root causes instead of a certain one, such one-to-many or even many-to-many mapping relations could reduce debugging techniques' effectiveness. Only a limited number of studies are found to employ AI techniques to tackle the challenge introduced by the co-existence or interference among multiple faults [300, 301]. Future researchers are encouraged to leverage AI techniques' strength of mining complicated mapping relations to localize or fix multiple faults in parallel.

7.2 In the field of SD4AI

Debugging AI models still lacks attention. We find that most researchers tend to apply existing AI techniques to debug traditional programs. While some researchers have conducted mutual studies on debugging AI systems, the majority of them carried out empirical studies of failure symptoms and root causes in specific domains or platforms. Besides, according to our investigation, 51 of the 131 papers in AI4SD organize a novel and unseen solution to an existing problem, or propose a new open problem, accounting for 38.9% of the total. On the contrary, such papers make up 64.7% of SD4AI (22 of the 34 papers). This implies that a large portion of the studies in AI4SD are incremental, while many research directions in SD4AI remain unexplored. These findings (1) reveal the reasons why AI systems encounter quality crisis to some extent [34, 290], (2) point out the insufficient attention in the field of AI system quality assurance, and (3) provide traditional software debugging researchers with a new direction, i.e., explore new ways to localize or fix faults in AI models.

Alleviating the oracle problem that exists in debugging AI. In traditional software testing and debugging, we can easily identify a failure if the oracle is violated (i.e., the actual output deviates from the expected output). However, when testing and debugging AI systems, such oracles are difficult to obtain, because we can hardly say that a wrong output necessarily indicates a fault in the AI system, considering that no AI models can guarantee 100% correct prediction. To alleviate the oracle problem, Chen et al. [302] proposed a novel testing strategy, Metamorphic Testing (MT), which uses properties of functions (also known as metamorphic relations) such that it is possible to predict expected changes to the output for particular changes to the input [303]. Since it provides the solution to the oracle problem with a feasible way, many researchers introduce MT to AI systems quality assurance, yielding a large number of highly-quality studies [304–306]. Despite its rapid development, there are still many directions that can be explored for better debugging AI using MT. For example, how to extract metamorphic relations in an automated way? How to more properly generate follow-up test cases according to both metamorphic relations and the characteristics of AI models? How to construct a comprehensive and reusable metamorphic relation repository for the community of SD4AI? We believe that these questions are worth to be answered in the future. Moreover, AI models are quite tolerant of some types of faults (such as calculation faults), which can be successfully executed but cause the training to get stuck in a local optimum. How to capture, further localize and fix such faults is also an interesting topic.

Paying more attention to the quality of training data. Traditional software typically has concrete logic flows, thus the functionality and the output can be easily traced back to each line of source code. But AI systems are usually data-driven, that is to say, the training data can also have a significant impact on the AI systems' performance. Therefore, if we are not satisfied with the trained model, we may need to inspect both the source code (e.g., model construction and hyperparameter tuning) and the training data (e.g., data normalization and data pollution), where the latter is a problem that distinguishes debugging AI systems from debugging traditional software. In the future, we recommend that researchers and practitioners pay more attention to the collection of high-quality training data, making the model training process have a good foundation. Monitoring the data flow/neuron coverage during the training for better debugging AI models can also be taken into account.

Reproducing failures in AI systems. Training an AI model involves many random factors. For example, the initialization of parameters typically relies on a random number generator, and the different input orders of the same set of training data can also result in a different model. When it comes to traditional fault localization or program repair, the failures that are often discussed by researchers are bohrbugs (i.e., those that can be reproduced) [307]. But in debugging AI models, the mentioned randomness could make the observed failures irreproducible, which hinders further testing and debugging for them. We believe that this problem is important, because enabling faults in AI systems to be triggered consistently under some well-defined conditions will enhance developers' confidence in ensuring the reliability of AI systems.

Automatically fixing the faults caused by the API version change. According to the existing conclusion, if the new API version is not backward compatible with its previous version, faults could occur due to the inconsistencies between (1) the old and the new API names, and (2) the old and the new orders of arguments. Such identifier faults account for 40.9% of issues in GitHub projects based on the investigation of a survey [119]. We think that a potential solution to this challenge is to propose an identifier analysis technique based on natural language processing. Specifically, although the name and the order of arguments in the API have been updated, the functionality implemented by the API can be

coherent, therefore, the signature of the API can be consistently reflected by some essential tokens. Based on this, we can extract the API change history and further mine the semantics of faulty identifiers in old and new contexts using natural language processing techniques, for automatically generating patches for fixing such identifier faults.

8 Conclusion

In this paper, we collect and review 165 primary studies published between 2016 and 2021.3 from the intersection of AI and SD. Many of these papers come from top journals or top conferences, which highlight the most advances in the intersection of AI and SD as well as support this survey with high-quality resources.

After categorizing these studies into two directions, AI4SD and SD4AI, we point out that the latter is far less popular than the former, along with three research questions being designed to explore the two fields: (1) How can AI techniques improve the effectiveness of SD? (2) How can SD be used to assure the quality of AI systems? (3) What configuration do researchers prefer to use in AI4SD and SD4AI experiments? Our conclusions include: (1) AI techniques are typically used in fault localization tasks in six ways, namely, feature extraction (ensemble), suspiciousness calculation, information retrieval, test cases enhancement, multi-fault debugging, and others. In these sub-problems, CNN, EA, RNN (LSTM), etc. are commonly used. On the other hand, AI techniques are typically used in automated program repair tasks in three ways, namely, patch generation, patch ranking (filtering), and code representation. In these sub-problems, RNN (LSTM), EA, Constraint solving, Clustering, etc. are commonly used. Researchers tend to use execution-based, code-based, and text-based information as the input of AI models. (2) The localization activity spreads from AI libraries to AI models since both of them may contain faults, while researchers typically repair faulty AI models from the perspectives of data and neural network connectivity. Researchers tend to manually investigate faults or symptoms in AI systems or libraries through empirical studies: data faults and logic faults occur frequently, while data preparation and model inference are risky phases during development. (3) As for AI4SD, researchers tend to use Defects4J as datasets, as well as use Java and C programs in experiments. Seven metrics, such as MAP, Recall at Top-n, and Accuracy, are often used to evaluate novel approaches. As for SD4AI, MNIST, CIFAR-10, and other two datasets are often employed as benchmarks. Prominent metrics include Precision, Recall, and Contrast.

Besides, we further highlight opportunities and challenges as well as suggest potential directions for future researchers. We also extract and list a series of tools and public repositories in AI4SD and SD4AI to provide stakeholders with a convenience index.

Acknowledgements This work was partially supported by National Natural Science Foundation of China (Grant Nos. 62250610-224, 61972289, 61832009). We sincerely appreciate the valuable suggestions from the anonymous reviewers for our paper.

References

1 Garousi V, Rainer A, Lauvås jr P, et al. Software-testing education: a systematic literature mapping. J Syst Software, 2020, 165: 110570
2 Lou Y, Ghanbari A, Li X, et al. Can automated program repair refine fault localization? A unified debugging approach. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. 75–87
3 Monperrus M. Automatic software repair: a bibliography. ACM Computing Surveys, 2018, 51: 1–24
4 Zakari A, Lee S P, Abreu R, et al. Multiple fault localization of software programs: a systematic literature review. Inf Software Tech, 2020, 124: 106312
5 Lu G Z, Xu L, Yang Y B, et al. Predictive analysis for race detection in software-defined networks. Sci China Inf Sci, 2019, 62: 062101
6 Fang C R, Chen Z Y, Xu B W. Comparing logic coverage criteria on test case prioritization. Sci China Inf Sci, 2012, 55: 2826–2840
7 Zhou Y M, Leung H, Song Q B, et al. An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. Sci China Inf Sci, 2012, 55: 2800–2815
8 Wang G, Shen R, Chen J, et al. Probabilistic delta debugging. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021. 881–892
9 Jiang J J, Xiong Y F, Xia X. A manual inspection of Defects4J bugs and its implications for automatic program repair. Sci China Inf Sci, 2019, 62: 200102
10 Wang S, Lo D. AmaLgam+: composing rich information sources for accurate bug localization. J Software Evolu Process, 2016, 28: 921–942
11 Pang N. Deep learning for code repair. Vancouver: University of British Columbia, 2018. https://people.ece.ubc.ca/qhanam/papers/npang_thesis_2018.pdf
12 Safdari N, Alrubaye H, Aljedaani W, et al. Learning to rank faulty source files for dependent bug reports. In: Proceedings of Big Data: Learning, Analytics, and Applications, 2019. 109890B

13 Zhang Z, Xie X. On the investigation of essential diversities for deep learning testing criteria. In: Proceedings of IEEE 19th International Conference on Software Quality, Reliability and Security, 2019. 394–405
14 Devanbu P, Dwyer M, Elbaum S, et al. Deep learning & software engineering: state of research and future directions. 2020. ArXiv:2009.08525
15 Pandey S K, Mishra R B, Tripathi A K. Machine learning based methods for software fault prediction: a survey. Expert Syst Appl, 2021, 172: 114595
16 Ranjan P, Kumar S, Kumar U. Software fault prediction using computational intelligence techniques: a survey. Ind J Sci Tech, 2017, 10: 1–9
17 Batool I, Khan T A. Software fault prediction using data mining, machine learning and deep learning techniques: a systematic literature review. Comput Electrical Eng, 2022, 100: 107886
18 Durelli V H S, Durelli R S, Borges S S, et al. Machine learning applied to software testing: a systematic mapping study. IEEE Trans Rel, 2019, 68: 1189–1212
19 Mahapatra S, Mishra S. Usage of machine learning in software testing. In: Proceedings of Automated Software Engineering: A Deep Learning-Based Approach, 2020. 39–54
20 Braiek H B, Khomh F. On testing machine learning programs. J Syst Software, 2020, 164: 110542
21 Zhang J M, Harman M, Ma L, et al. Machine learning testing: survey, landscapes and horizons. IEEE Trans Software Eng, 2022, 48: 1–36
22 Riccio V, Jahangirova G, Stocco A, et al. Testing machine learning based systems: a systematic mapping. Empir Software Eng, 2020, 25: 5193–5254
23 Wang Y, Jia P, Liu L, et al. A systematic review of fuzzing based on machine learning techniques. Plos One, 2020, 15: e0237749
24 Chen J, Patra J, Pradel M, et al. A survey of compiler testing. ACM Comput Surv, 2021, 53: 1–36
25 Li X, Jiang H, Ren Z, et al. Deep learning in software engineering. 2018. ArXiv:1805.04825
26 Ferreira F, Silva L L, Valente M T. Software engineering meets deep learning: a mapping study. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021. 1542–1549
27 Yang Y, Xia X, Lo D, et al. A survey on deep learning for software engineering. 2020. ArXiv:2011.14597
28 Serban A, van der Blom K, Hoos H, et al. Adoption and effects of software engineering best practices in machine learning. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2020. 1–12
29 Arpteg A, Brinne B, Crnkovic-Friis L, et al. Software engineering challenges of deep learning. In: Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications, 2018. 50–59
30 Zhang X, Yang Y, Feng Y, et al. Software engineering practice in the development of deep learning applications. 2019. ArXiv:1910.03156
31 Yang Y, Xia X, Lo D, et al. Predictive models in software engineering: challenges and opportunities. ACM Trans Softw Eng Methodol, 2022, 31: 1–72
32 Lertvittayakumjorn P, Toni F. Explanation-based human debugging of NLP models: a survey. Trans Assoc Comput Linguistics, 2021, 9: 1508–1528
33 Zhang Q, Zhao Y, Sun W, et al. Program repair: automated vs. manual. 2022. ArXiv:2203.05166
34 Islam M J, Pan R, Nguyen G, et al. Repairing deep neural networks: fix patterns and challenges. In: Proceedings of IEEE/ACM 42nd International Conference on Software Engineering, 2020. 1135–1146
35 Zhong W, Li C, Ge J, et al. Neural program repair: Systems, challenges and solutions. 2022. ArXiv:2202.10868
36 Feng Y, Liu Q, Dou M Y, et al. Mubug: a mobile service for rapid bug tracking. Sci China Inf Sci, 2016, 59: 013101
37 Zhang Z Y, Chen Z Y, Gao R Z, et al. An empirical study on constraint optimization techniques for test generation. Sci China Inf Sci, 2017, 60: 012105
38 Zhao Y, Feng Y, Wang Y, et al. Quality assessment of crowdsourced test cases. Sci China Inf Sci, 2020, 63: 190102
39 Staats M, Whalen M W, Heimdahl M P. Programs, tests, and oracles: the foundations of testing revisited. In: Proceedings of the 33rd International Conference on Software Engineering, 2011. 391–400
40 LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521: 436–444
41 Barr A, Feigenbaum E A. The Handbook of Artificial Intelligence. Oxford: Butterworth-Heinemann, 1981
42 Feldt R, de Oliveira Neto F G, Torkar R. Ways of applying artificial intelligence in software engineering. In: Proceedings of IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, 2018. 35–41
43 Mou L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, 2016
44 Gu X, Zhang H, Zhang D, et al. Deep API learning. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 631–642
45 Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: Proceedings of IEEE/ACM 38th International Conference on Software Engineering, 2016. 297–308
46 Li X, Zhang L. Transforming programs and tests in tandem for fault localization. In: Proceedings of the ACM on Programming Languages, 2017. 1–30
47 Xie X, Chen T Y, Kuo F C, et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans Softw Eng Methodol, 2013, 22: 1–40
48 Gao R, Wong W E. MSeer—an advanced technique for locating multiple bugs in parallel. IEEE Trans Software Eng, 2019, 45: 301–318
49 Wang X Y, Jiang S J, Gao P F, et al. Cost-effective testing based fault localization with distance based test-suite reduction. Sci China Inf Sci, 2017, 60: 092112
50 Wang Y, Huang Z Q, Li Y, et al. Lightweight fault localization combined with fault context to improve fault absolute rank. Sci China Inf Sci, 2017, 60: 092113
51 Tu J, Xie X, Chen T Y, et al. On the analysis of spectrum based fault localization using hitting sets. J Syst Software, 2019, 147: 106–123
52 Xu Z, Ma S, Zhang X, et al. Debugging with intelligence via probabilistic inference. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 1171–1181
53 Tu J, Xie X, Zhou Y, et al. A search based context-aware approach for understanding and localizing the fault via weighted call graph. In: Proceedings of the 3rd International Conference on Trustworthy Systems and their Applications, 2016. 64–72
54 Cao J, Yang S, Jiang W, et al. BugPecker: locating faulty methods with deep learning on revision graphs. In: Proceedings

of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 1214–1218
55 Wen M, Wu R, Cheung S C. Locus: locating bugs from software changes. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 262–273
56 Wong W E, Gao R, Li Y, et al. A survey on software fault localization. IEEE Trans Software Eng, 2016, 42: 707–740
57 Weiser M D. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Dissertation for Ph.D. Degree. Ann Arbor: University of Michigan, 1979
58 Zhang X, He H, Gupta N, et al. Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging, 2005. 33–42
59 Wotawa F. Fault localization based on dynamic slicing and hitting-set computation. In: Proceedings of the 10th International Conference on Quality Software, 2010. 161–170
60 Xie X, Xu B. Essential Spectrum-Based Fault Localization. Berlin: Springer, 2021
61 Laghari G, Murgia A, Demeyer S. Fine-tuning spectrum based fault localisation with frequent method item sets. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 274–285
62 Zhang L, Li Z, Feng Y, et al. Improving fault-localization accuracy by referencing debugging history to alleviate structure bias in code suspiciousness. IEEE Trans Rel, 2020, 69: 1021–1049
63 Zhang L, Yan L, Zhang Z, et al. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. J Syst Software, 2017, 129: 35–57
64 Wen M, Chen J, Tian Y, et al. Historical spectrum based fault localization. IEEE Trans Software Eng, 2021, 47: 2348–2368
65 Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation. SIGPLAN Not, 2005, 40: 15–26
66 Nessa S, Abedin M, Wong W E, et al. Software fault localization using n-gram analysis. In: Proceedings of International Conference on Wireless Algorithms, Systems, and Applications, 2008. 548–559
67 Guo Z Q, Zhou H C, Liu S R, et al. Information retrieval based bug localization: research problem, progress, and challenges (in Chinese). J Software, 2020, 31: 2826–2854
68 Zou W, Li E, Fang C. BLESER: bug localization based on enhanced semantic retrieval. 2021. ArXiv:2109.03555
69 Ren Z, Jiang H, Xuan J, et al. Automated localization for unreproducible builds. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 71–81
70 de Souza H A, Chaim M L, Kon F. Spectrum-based software fault localization: a survey of techniques, advances, and challenges. 2016. ArXiv:1607.04347
71 Zhang Z, Lei Y, Mao X, et al. CNN-FL: an effective approach for localizing faults using convolutional neural networks. In: Proceedings of IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, 2019. 445–455
72 Wong W E, Qi Y U. BP neural network-based effective fault localization. Int J Soft Eng Knowl Eng, 2009, 19: 573–597
73 Zheng W, Hu D, Wang J. Fault localization analysis based on deep neural network. Math Problems Eng, 2016, 2016: 1–11
74 Zhang Z, Lei Y, Mao X, et al. A study of effectiveness of deep learning in locating real faults. Inf Software Tech, 2021, 131: 106486
75 Lam A N, Nguyen A T, Nguyen H A, et al. Bug localization with combination of deep learning and information retrieval. In: Proceedings of IEEE/ACM 25th International Conference on Program Comprehension, 2017. 218–229
76 Huo X, Li M. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, 2017. 1909–1915
77 Shi Z, Keung J, Bennin K E, et al. Comparing learning to rank techniques in hybrid bug localization. Appl Soft Computing, 2018, 62: 636–648
78 Chen Z F, Ma W W Y, Lin W, et al. A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features. Sci China Inf Sci, 2018, 61: 012107
79 Le X B D, Le Q L, Lo D, et al. Enhancing automated program repair with deductive verification. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2016. 428–432
80 Gopinath D, Wang K, Hua J, et al. Repairing intricate faults in code using machine learning and path exploration. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2016. 453–457
81 Roychoudhury A, Xiong Y F. Automated program repair: a step towards software automation. Sci China Inf Sci, 2019, 62: 200103
82 Kong X, Zhang L, Wong W E, et al. Experience report: how do techniques, programs, and tests impact automated program repair? In: Proceedings of IEEE 26th International Symposium on Software Reliability Engineering, 2015. 194–204
83 Wen M, Liu Y, Cheung S C. Boosting automated program repair with bug-inducing commits. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, 2020. 77–80
84 Marginean A, Bader J, Chandra S, et al. SapFix: automated end-to-end repair at scale. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, 2019. 269–278
85 Bader J, Scott A, Pradel M, et al. Getafix: learning to fix bugs automatically. In: Proceedings of the ACM on Programming Languages, 2019. 1–27
86 Motwani M, Soto M, Brun Y, et al. Quality of automated program repair on real-world defects. IEEE Trans Software Eng, 2022, 48: 637–661
87 Smith E K, Barr E T, Goues C L, et al. Is the cure worse than the disease? Overfitting in automated program repair. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, 2015. 532–543
88 Le Goues C, Dewey-Vogt M, Forrest S, et al. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering, 2012. 3–13
89 Qi Y, Mao X, Lei Y. Efficient automated program repair through fault-recorded testing prioritization. In: Proceedings of IEEE International Conference on Software Maintenance, 2013. 180–189
90 Weimer W, Fry Z P, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, 2013. 356–366
91 Kim J, Kim J, Lee E. VFL: variable-based fault localization. Inf Software Tech, 2019, 107: 179–191
92 Wang S, Liu K, Lin B, et al. Beep: fine-grained fix localization by learning to predict buggy code elements. 2021. ArXiv:2111.07739
93 Liu K, Koyuncu A, Bissyandé T F, et al. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification, 2019. 102–113
94 Monperrus M. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software

Engineering, 2014. 234–242
95 Wang S, Mao X, Niu N, et al. Multi-location program repair strategies learned from past successful experience. 2018. ArXiv:1810.12556
96 Motwani M, Sankaranarayanan S, Just R, et al. Do automated program repair techniques repair hard and important bugs? Empirical Software Eng, 2018, 23: 2901–2947
97 Liu K, Kim D, Bissyande T F, et al. Mining fix patterns for FindBugs violations. IEEE Trans Software Eng, 2021, 47: 165–188
98 Tufano M, Watson C, Bavota G, et al. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans Softw Eng Methodol, 2019, 28: 1–29
99 Chen Z, Kommrusch S J, Tufano M, et al. SEQUENCER: sequence-to-sequence learning for end-to-end program repair. IEEE Trans Software Eng, 2021. doi: 10.1109/TSE.2019.2940179
100 Lutellier T, Pham H V, Pang L, et al. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. 101–114
101 Cao J, Li M, Chen X, et al. DeepFD: automated fault diagnosis and localization for deep learning programs. 2022. ArXiv:2205.01938
102 Li Z, Ma X, Xu C, et al. Operational calibration: debugging confidence errors for dnns in the field. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 901–913
103 Yan S, Tao G, Liu X, et al. Correlations between deep neural network model coverage criteria and model quality. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 775–787
104 Brown L. Tesla driver killed in crash posted videos of himself driving hands-free. 2021. https://www.marketwatch.com/story/tesla-driver-killed-in-crash-posted-videos-of-himself-driving-hands-free-11621220917
105 Marijan D, Gotlieb A. Software testing for machine learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2020. 34: 13576–13582
106 Shen W, Li Y, Han Y, et al. Boundary sampling to boost mutation testing for deep learning models. Inf Software Tech, 2021, 130: 106413
107 Shen G, Liu Y, Tao G, et al. Backdoor scanning for deep neural networks through k-arm optimization. In: Proceedings of International Conference on Machine Learning, 2021. 9525–9536
108 Meng L, Li Y, Chen L, et al. Measuring discrimination to boost comparative testing for multiple deep learning models. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 385–396
109 Lourenço R, Freire J, Shasha D. Debugging machine learning pipelines. In: Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning, 2019. 1–10
110 Feng Y, Shi Q, Gao X, et al. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. 177–188
111 Krishnan S, Wu E. PALM: machine learning explanations for iterative debugging. In: Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, 2017. 1–6
112 Koh P W, Liang P. Understanding black-box predictions via influence functions. In: Proceedings of International Conference on Machine Learning, 2017. 1885–1894
113 Cao Y, Yu A F, Aday A, et al. Efficient repair of polluted machine learning systems via causal unlearning. In: Proceedings of the Asia Conference on Computer and Communications Security, 2018. 735–747
114 Zhang H, Chan W. Apricot: a weight-adaptation approach to fixing deep learning models. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, 2019. 376–387
115 Shen W, Li Y, Chen L, et al. Multiple-boundary clustering and prioritization to promote neural network retraining. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 410–422
116 Zhang X, Yin Z, Feng Y, et al. NeuralVis: visualizing and interpreting deep learning models. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, 2019. 1106–1109
117 Eniser H F, Gerasimou S, Sen A. DeepFault: fault localization for deep neural networks. 2019. ArXiv:1902.05974
118 Guidotti D, Leofante F, Pulina L, et al. Verification and repair of neural networks: a progress report on convolutional models. In: Proceedings of International Conference of the Italian Association for Artificial Intelligence, 2019. 405–417
119 Zhang Y, Chen Y, Cheung S C, et al. An empirical study on TensorFlow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018. 129–140
120 Islam M J, Nguyen G, Pan R, et al. A comprehensive study on deep learning bug characteristics. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019. 510–520
121 Humbatova N, Jahangirova G, Bavota G, et al. Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 1110–1121
122 Kitchenham B A, Budgen D, Brereton P. Evidence-Based Software Engineering and Systematic Reviews: Volume 4. Boca Raton: CRC Press, 2015
123 Basili V R, Caldiera G, Rombach H D. The goal question metric approach. In: Encyclopedia of Software Engineering. 1994. 528–532
124 Colanzi T E, Assunção W K G, Farah P R, et al. A review of ten years of the symposium on search-based software engineering. In: Proceedings of International Symposium on Search Based Software Engineering, 2019. 42–57
125 Ye X, Shen H, Ma X, et al. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 404–415
126 Long F, Rinard M. Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016. 298–312
127 Xuan J, Martinez M, DeMarco F, et al. Nopol: automatic repair of conditional statement bugs in Java programs. IEEE Trans Software Eng, 2017, 43: 34–55
128 Le X B D, Lo D, Le Goues C. History driven program repair. In: Proceedings of IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, 2016. 213–224
129 Xiong Y, Wang J, Yan R, et al. Precise condition synthesis for program repair. In: Proceedings of IEEE/ACM 39th International Conference on Software Engineering, 2017. 416–426

130 Ma S, Liu Y, Lee W C, et al. MODE: automated neural network model debugging via state differential analysis and input selection. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. 175–186
131 Agrawal A, Fu W, Menzies T. What is wrong with topic modeling? And how to fix it using search-based software engineering. Inf Software Tech, 2018, 98: 74–88
132 Peng Z, Xiao X, Hu G, et al. ABFL: an autoencoder based practical approach for software fault localization. Inf Sci, 2020, 510: 108–121
133 Huo X, Li M, Zhou Z H. Control flow graph embedding based on multi-instance decomposition for bug localization. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence, 2020. 4223–4230
134 Li X, Li W, Zhang Y, et al. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019. 169–180
135 Qi G, Yao L, Uzunov A V. Fault detection and localization in distributed systems using recurrent convolutional neural networks. In: Proceedings of International Conference on Advanced Data Mining and Applications, 2017. 33–48
136 Huo X, Li M, Zhou Z H. Learning unified features from natural and programming languages for locating buggy source code. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence, 2016. 1606–1612
137 Liang H, Sun L, Wang M, et al. Deep learning with customized abstract syntax tree for bug localization. IEEE Access, 2019, 7: 116309
138 Golagha M, Pretschner A, Briand L C. Can we predict the quality of spectrum-based fault localization? In: Proceedings of IEEE 13th International Conference on Software Testing, Validation and Verification, 2020. 4–15
139 Gu Y, Xuan J, Zhang H, et al. Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence. J Syst Software, 2019, 148: 88–104
140 Kim Y, Mun S, Yoo S, et al. Precise learn-to-rank fault localization using dynamic and static features of target programs. ACM Trans Softw Eng Methodol, 2019, 28: 1–34
141 Xia X, Lo D. An effective change recommendation approach for supplementary bug fixes. Autom Softw Eng, 2017, 24: 455–498
142 Mohri M, Rostamizadeh A, Talwalkar A. Foundations of Machine Learning. Cambridge: MIT Press, 2018
143 Pan Y, Xiao X, Hu G, et al. ALBFL: a novel neural ranking model for software fault localization via combining static and dynamic features. In: Proceedings of IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications, 2020. 785–792
144 Ye X, Bunescu R, Liu C. Mapping bug reports to relevant files: a ranking model, a fine-grained benchmark, and feature evaluation. IEEE Trans Software Eng, 2016, 42: 379–402
145 Yang X L, Lo D, Xia X, et al. High-impact bug report identification with imbalanced learning strategies. J Comput Sci Technol, 2017, 32: 181–198
146 Guo Z, Li Y, Ma W, et al. Boosting crash-inducing change localization with rank-performance-based feature subset selection. Empir Software Eng, 2020, 25: 1905–1950
147 Wu R, Wen M, Cheung S C, et al. ChangeLocator: locate crash-inducing changes based on crash reports. Empir Software Eng, 2018, 23: 2866–2900
148 Li A, Lei Y, Mao X. Towards more accurate fault localization: an approach based on feature selection using branching execution probability. In: Proceedings of IEEE International Conference on Software Quality, Reliability and Security, 2016. 431–438
149 Feyzi F. CGT-FL: using cooperative game theory to effective fault localization in presence of coincidental correctness. Empir Software Eng, 2020, 25: 3873–3927
150 Amar A, Rigby P C. Mining historical test logs to predict bugs and localize faults in the test logs. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering, 2019. 140–151
151 Koyuncu A, Bissyandé T F, Kim D, et al. D&C: a divide-and-conquer approach to IR-based bug localization. 2019. ArXiv:1902.02703
152 Yang B, He Y, Liu H, et al. A lightweight fault localization approach based on XGBoost. In: Proceedings of IEEE 20th International Conference on Software Quality, Reliability and Security, 2020. 168–179
153 Nath A, Domingos P. Learning tractable probabilistic models for fault localization. In: Proceedings of the AAAI Conference on Artificial Intelligence: Volume 30. 2016
154 Popescu M C, Balas V E, Perescu-Popescu L, et al. Multilayer perceptron and neural networks. WSEAS Trans Circuits Syst, 2009, 8: 579–588
155 Maru A, Dutta A, Kumar K V, et al. Effective software fault localization using a back propagation neural network. In: Proceedings of Computational Intelligence in Data Mining, 2020. 513–526
156 Dutta A, Pant N, Mitra P, et al. Effective fault localization using an ensemble classifier. In: Proceedings of International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering, 2019. 847–855
157 Li Y, Wang S, Nguyen T N. Fault localization with code coverage representation learning. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 661–673
158 Polisetty S, Miranskyy A, Başar A. On usefulness of the deep-learning-based bug localization models to practitioners. In: Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering, 2019. 16–25
159 Mahapatra R, Negi A. Effective software fault localization using GA-RBF neural network. J Theor Applied Inform Technol, 2016, 90: 168–174
160 Sohn J, Yoo S. FLUCCS: using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017. 273–283
161 Choi K, Sohn J, Yoo S. Learning fault localisation for both humans and machines using multi-objective GP. In: Proceedings of International Symposium on Search Based Software Engineering, 2018. 349–355
162 Xuan J, Monperrus M. Learning to combine multiple ranking metrics for fault localization. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2014. 191–200
163 Zou D, Liang J, Xiong Y, et al. An empirical study of fault localization families and their combinations. IEEE Trans Software Eng, 2021, 47: 332–347
164 Liu P, Chen Y, Nie X, et al. FluxRank: a widely-deployable framework to automatically localizing root cause machines for software service failure mitigation. In: Proceedings of IEEE 30th International Symposium on Software Reliability Engineering, 2019. 35–46

165 Le T D B, Lo D, Goues C L, et al. A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016. 177–188
166 Küçük Y, Henderson T A, Podgurski A. Improving fault localization by integrating value and predicate based causal inference techniques. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 649–660
167 Podgurski A, Küçük Y. CounterFault: value-based fault localization by modeling and predicting counterfactual outcomes. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2020. 382–393
168 Lou Y, Zhu Q, Dong J, et al. Boosting coverage-based fault localization via graph-based representation learning. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021. 664–676
169 Maamar M, Lazaar N, Loudni S, et al. Fault localization using itemset mining under constraints. Autom Softw Eng, 2017, 24: 341–368
170 Yan M, Xia X, Fan Y, et al. Just-In-Time defect identification and localization: a two-phase framework. IEEE Trans Software Eng, 2022, 48: 82–101
171 Zaman T S, Han X, Yu T. SCMiner: localizing system-level concurrency faults from large system call traces. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, 2019. 515–526
172 Hoang T, Oentaryo R J, Le T D B, et al. Network-clustered multi-modal bug localization. IEEE Trans Software Eng, 2019, 45: 1002–1023
173 Cheng S, Yan X, Khan A A. A similarity integration method based information retrieval and word embedding in bug localization. In: Proceedings of IEEE 20th International Conference on Software Quality, Reliability and Security, 2020. 180–187
174 Pradel M, Sen K. DeepBugs: a learning approach to name-based bug detection. Proc ACM Program Lang, 2018, 2: 1–25
175 Liu G, Lu Y, Shi K, et al. Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance. IEEE Access, 2019, 7: 131304–131316
176 Xiao Y, Keung J, Bennin K E, et al. Improving bug localization with word embedding and enhanced convolutional neural networks. Inf Software Tech, 2019, 105: 17–29
177 Li G, Liu H, Jin J, et al. Deep learning based identification of suspicious return statements. In: Proceedings of IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, 2020. 480–491
178 Zhang Y, Lo D, Xia X, et al. Fusing multi-abstraction vector space models for concern localization. Empir Software Eng, 2018, 23: 2279–2322
179 Mills C, Parra E, Pantiuchina J, et al. On the relationship between bug reports and queries for text retrieval-based bug localization. Empir Software Eng, 2020, 25: 3086–3127
180 Almhana R, Mkaouer W, Kessentini M, et al. Recommending relevant classes for bug reports using multi-objective search. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 286–295
181 Almhana R, Kessentini M, Mkaouer W. Method-level bug localization using hybrid multi-objective search. Inf Software Tech, 2021, 131: 106474
182 Mikolov T, Sutskever I, Chen K, et al. Distributed representations of words and phrases and their compositionality. In: Proceedings of Advances in Neural Information Processing Systems, 2013. 3111–3119
183 Briem J A, Smit J, Sellik H, et al. OffSide: learning to identify mistakes in boundary conditions. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020. 203–208
184 Liu G, Lu Y, Shi K, et al. Mapping bug reports to relevant source code files based on the vector space model and word embedding. IEEE Access, 2019, 7: 78870–78881
185 Zhang W, Li Z, Wang Q, et al. FineLocator: a novel approach to method-level fine-grained bug localization by query expansion. Inf Software Tech, 2019, 110: 121–135
186 Zhu Z, Li Y, Tong H, et al. CooBa: cross-project bug localization via adversarial transfer learning. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence, 2020. 3565–3571
187 Zhong H, Mei H. Learning a graph-based classifier for fault localization. Sci China Inf Sci, 2020, 63: 162101
188 Jonsson L, Broman D, Magnusson M, et al. Automatic localization of bugs to faulty components in large scale software systems using Bayesian classification. In: Proceedings of IEEE International Conference on Software Quality, Reliability and Security, 2016. 423–430
189 Huang Q, Lo D, Xia X, et al. Which packages would be affected by this bug report? In: Proceedings of IEEE 28th International Symposium on Software Reliability Engineering, 2017. 124–135
190 Le T D B, Thung F, Lo D. Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. Empir Software Eng, 2017, 22: 2237–2279
191 Li Z, Jiang Z, Chen X, et al. Laprob: a label propagation-based software bug localization method. Inf Software Tech, 2021, 130: 106410
192 Rahman M M, Roy C K. Improving IR-based bug localization with context-aware query reformulation. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. 621–632
193 Li X, Wong W E, Gao R, et al. Genetic algorithm-based test generation for software product line with the integration of fault localization techniques. Empir Software Eng, 2018, 23: 1–51
194 Chatterjee P, Chatterjee A, Campos J, et al. Diagnosing software faults using multiverse analysis. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence, 2020. 1629–1635
195 Elmishali A, Stern R, Kalech M. An artificial intelligence paradigm for troubleshooting software bugs. Eng Appl Artif Intelligence, 2018, 69: 147–156
196 Liu B, Nejati S, Lucia S, et al. Effective fault localization of automotive Simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. Empir Software Eng, 2019, 24: 444–490
197 Zhang Z, Lei Y, Mao X, et al. Improving deep-learning-based fault localization with resampling. J Software Evolu Process, 2021, 33: e2312
198 Japkowicz N, Stephen S. The class imbalance problem: a systematic study1. Intell Data Anal, 2002, 6: 429–449
199 Graves A, Mohamed A R, Hinton G. Speech recognition with deep recurrent neural networks. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, 2013. 6645–6649
200 Jarrett K, Kavukcuoglu K, Ranzato M, et al. What is the best multi-stage architecture for object recognition? In: Proceedings of IEEE 12th International Conference on Computer Vision, 2009. 2146–2153
201 Xia X, Gong L, Le T D B, et al. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs.

Autom Softw Eng, 2016, 23: 43–75
202 Liu Y, Li M, Wu Y, et al. A weighted fuzzy classification approach to identify and manipulate coincidental correct test cases for fault localization. J Syst Software, 2019, 151: 20–37
203 Zhang M, Li Y, Li X, et al. An empirical study of boosting spectrum-based fault localization via PageRank. IEEE Trans Software Eng, 2021, 47: 1089–1113
204 Chen J, Ma H, Zhang L. Enhanced compiler bug isolation via memoized search. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 78–89
205 Zhang X Y, Zheng Z, Cai K Y. Exploring the usefulness of unlabelled test cases in software fault localization. J Syst Software, 2018, 136: 278–290
206 Gupta R, Kanade A, Shevade S. Neural attribution for semantic bug-localization in student programs. In: Proceedings of Advances in Neural Information Processing Systems, 2019. 32
207 Sundararajan M, Taly A, Yan Q. Axiomatic attribution for deep networks. In: Proceedings of International Conference on Machine Learning, 2017. 3319–3328
208 He J, Xu L, Yan M, et al. Duplicate bug report detection using dual-channel convolutional neural networks. In: Proceedings of the 28th International Conference on Program Comprehension, 2020. 117–127
209 Ni Z, Li B, Sun X, et al. Analyzing bug fix for automatic bug cause classification. J Syst Software, 2020, 163: 110538
210 Yan X B, Liu B, Wang S H. A test restoration method based on genetic algorithm for effective fault localization in multiple-fault programs. J Syst Software, 2021, 172: 110861
211 Zheng Y, Wang Z, Fan X, et al. Localizing multiple software faults based on evolution algorithm. J Syst Software, 2018, 139: 107–123
212 Gao M, Li P, Chen C, et al. Research on software multiple fault localization method based on machine learning. In: Proceedings of MATEC Web of Conferences: volume 232, 2018. 01060
213 Behera R K, Shukla S, Rath S K, et al. Software reliability assessment using machine learning technique. In: Proceedings of International Conference on Computational Science and Its Applications. Springer, 2018. 403–411
214 Li Z, Chen T H, Shang W. Where shall we log? Studying and suggesting logging locations in code blocks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 361–372
215 Vasic M, Kanade A, Maniatis P, et al. Neural program repair by jointly learning to localize and repair. 2019. ArXiv:1904.01720
216 Chappelly T, Cifuentes C, Krishnan P, et al. Machine learning for finding bugs: an initial report. In: Proceedings of IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation, 2017. 21–26
217 Yang X, Yu Z, Wang J, et al. Understanding static code warnings: an incremental AI approach. Expert Syst Appl, 2021, 167: 114134
218 Lin Y, Sun J, Tran L, et al. Break the dead end of dynamic slicing: localizing data and control omission bug. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 509–519
219 Yu X, Liu J, Yang Z, et al. The Bayesian network based program dependence graph and its application to fault localization. J Syst Software, 2017, 134: 44–53
220 Hofer B, Nica I, Wotawa F. AI for localizing faults in spreadsheets. In: Proceedings of IFIP International Conference on Testing Software and Systems, 2017. 71–87
221 Terra-Neves M, Machado N, Lynce I, et al. Concurrency debugging with MaxSMT. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence, 2019. 1608–1616
222 Mesbah A, Rice A, Johnston E, et al. DeepDelta: learning to repair compilation errors. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019. 925–936
223 Zou W, Lo D, Kochhar P S, et al. Smart contract development: challenges and opportunities. IEEE Trans Software Eng, 2021, 47: 2084–2106
224 Yu X L, Al-Bataineh O, Lo D, et al. Smart contract repair. ACM Trans Softw Eng Methodol, 2020, 29: 1–32
225 Yuan Y, Banzhaf W. ARJA: automated repair of Java programs via multi-objective genetic programming. IEEE Trans Software Eng, 2018, 46: 1040–1067
226 Yuan Y, Banzhaf W. Toward better evolutionary program repair. ACM Trans Softw Eng Methodol, 2020, 29: 1–53
227 Oliveira V P L, Souza E F, Goues C L, et al. Improved representation and genetic operators for linear genetic programming for automated program repair. Empir Software Eng, 2018, 23: 2980–3006
228 Lee J, Song D, So S, et al. Automatic diagnosis and correction of logical errors for functional programming assignments. Proc ACM Program Lang, 2018, 2: 1–30
229 Machado N, Quinta D, Lucia B, et al. Concurrency debugging with differential schedule projections. ACM Trans Softw Eng Methodol, 2016, 25: 1–37
230 Pan R, Hu Q, Xu G, et al. Automatic repair of regular expressions. Proc ACM Program Lang, 2019, 3: 1–29
231 Koyuncu A, Liu K, Bissyandé T F, et al. FixMiner: mining relevant fix patterns for automated program repair. Empir Software Eng, 2020, 25: 1980–2024
232 Gulwani S, Radiček I, Zuleger F. Automated clustering and program repair for introductory programming assignments. SIGPLAN Not, 2018, 53: 465–480
233 Falleri J R, Morandat F, Blanc X, et al. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014. 313–324
234 Sakkas G, Endres M, Cosman B, et al. Type error feedback via analytic program repair. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020. 16–30
235 White M, Tufano M, Martinez M, et al. Sorting and transforming program repair ingredients via deep learning code similarities. In: Proceedings of IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, 2019. 479–490
236 Yi X, Chen L, Mao X, et al. Efficient automated repair of high floating-point errors in numerical libraries. Proc ACM Program Lang, 2019, 3: 1–29
237 Jiang J, Xiong Y, Zhang H, et al. Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018. 298–309
238 Jiang N, Lutellier T, Tan L. CURE: code-aware neural machine translation for automatic program repair. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 1161–1173
239 Koyuncu A, Liu K, Bissyandé T F, et al. iFixR: bug report driven program repair. In: Proceedings of the 27th ACM Joint

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019. 314–325
240 Zhu Q, Sun Z, Xiao Y a, et al. A syntax-guided edit decoder for neural program repair. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021. 341–353
241 Sun Z, Zhu Q, Xiong Y, et al. TreeGen: a tree-based transformer architecture for code generation. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence, 2020. 8984–8991
242 Shariffdeen R, Noller Y, Grunske L, et al. Concolic program repair. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021. 390–405
243 Lee J, Hong S, Oh H. MemFix: static analysis-based repair of memory deallocation errors for C. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. 95–106
244 Li Y, Wang S, Nguyen T N. DLFix: context-based code transformation learning for automated program repair. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 602–614
245 Wen M, Chen J, Wu R, et al. Context-aware patch generation for better automated program repair. In: Proceedings of IEEE/ACM 40th International Conference on Software Engineering, 2018. 1–11
246 Wang S, Wen M, Lin B, et al. Automated patch correctness assessment: how far are we? In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 968–980
247 Ziarko W, Shan N. Machine learning through data classification and reduction. Fundamenta Informaticae, 1997, 30: 373–382
248 Patil T R, Sherekar S S. Performance analysis of Naive Bayes and J48 classification algorithm for data classification. Int J Comput Sci Appl, 2013, 6: 256–261
249 Kleinbaum D G, Dietz K, Gail M, et al. Logistic Regression. Berlin: Springer, 2002
250 Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, 2011. 1–10
251 Platt J. Sequential minimal optimization: a fast algorithm for training support vector machines. 1998. https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/
252 Xiong Y, Liu X, Zeng M, et al. Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 789–799
253 Liang J, Ji R, Jiang J, et al. Interactive patch filtering as debugging aid. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2021. 239–250
254 Saha R K, Lyu Y, Yoshida H, et al. Elixir: effective object-oriented program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017. 648–659
255 Tan S H, Yoshida H, Prasad M R, et al. Anti-patterns in search-based program repair. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 727–738
256 Long F, Rinard M. Staged program repair with condition synthesis. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, 2015. 166–178
257 Le X B D, Thung F, Lo D, et al. Overfitting in semantics-based automated program repair. Empir Software Eng, 2018, 23: 3007–3033
258 Yasunaga M, Liang P. Graph-based, self-supervised program repair from diagnostic feedback. In: Proceedings of International Conference on Machine Learning, 2020. 10799–10808
259 Wang K, Singh R, Su Z. Dynamic neural program embedding for program repair. 2017. ArXiv:1711.07163
260 Gupta R, Kanade A, Shevade S. Deep reinforcement learning for syntactic error repair in student programs. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence, 2019. 930–937
261 Traver V J. On compiler error messages: what they *say* and what they *mean*. Adv Hum-Comput Interaction, 2010, 2010: 1–26
262 Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proceedings of International Conference on Learning Representations, 2018
263 Gember-Jacobson A, Akella A, Mahajan R, et al. Automatically repairing network control planes using an abstract representation. In: Proceedings of the 26th Symposium on Operating Systems Principles, 2017. 359–373
264 Dinella E, Dai H, Li Z, et al. Hoppity: learning graph transformations to detect and fix bugs in programs. In: Proceedings of International Conference on Learning Representations, 2020
265 Gupta K, Christensen P E, Chen X, et al. Synthesize, execute and debug: learning to repair for neural program synthesis. 2020. ArXiv:2007.08095
266 Tian H, Liu K, Kaboré A K, et al. Evaluating representation learning of code changes for predicting patch correctness in program repair. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 981–992
267 Yang G, Min K, Lee B. Applying deep learning algorithm to automatic bug localization and repair. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020. 1634–1641
268 Lourenço R, Freire J, Shasha D. BugDoc: algorithms to debug computational processes. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2020. 463–478
269 Pham H V, Lutellier T, Qi W, et al. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering, 2019. 1027–1038
270 Wardat M, Le W, Rajan H. DeepLocalize: fault localization for deep neural networks. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 251–262
271 Dolby J, Shinnar A, Allain A, et al. Ariadne: analysis for machine learning programs. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2018. 1–10
272 Cheng D, Cao C, Xu C, et al. Manifesting bugs in machine learning code: an explorative study with mutation testing. In: Proceedings of IEEE International Conference on Software Quality, Reliability and Security, 2018. 313–324
273 Wu X, Zheng W, Xia X, et al. Data quality matters: a case study on data label correctness for security bug report prediction. IEEE Trans Software Eng, 2022, 48: 2541–2556
274 Tao G, Ma S, Liu Y, et al. TRADER: trace divergence analysis and embedding regulation for debugging recurrent neural networks. In: Proceedings of IEEE/ACM 42nd International Conference on Software Engineering, 2020. 986–998
275 Kim E, Gopinath D, Pasareanu C, et al. A programmatic and semantic approach to explaining and debugging neural network based object detectors. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020.

11128–11137
276 Sotoudeh M, Thakur A V. Provable repair of deep neural networks. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021. 588–603
277 Song K, Tan X, Lu J. Neural machine translation with error correction. 2020. ArXiv:2007.10681
278 Zhang Y, Ren L, Chen L, et al. Detecting numerical bugs in neural network architectures. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 826–837
279 Schoop E, Huang F, Hartmann B. UMLAUT: debugging deep learning programs using program structure and model behavior. In: Proceedings of the CHI Conference on Human Factors in Computing Systems, 2021. 1–16
280 Zhang X, Zhai J, Ma S, et al. AUTOTRAINER: an automatic DNN training problem detection and repair system. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 359–371
281 Sun Z, Zhang J M, Harman M, et al. Automatic testing and improvement of machine translation. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 974–985
282 Jebnoun H, Braiek H B, Rahman M M, et al. The scent of deep learning code: an empirical study. In: Proceedings of the 17th International Conference on Mining Software Repositories, 2020. 420–430
283 Fan Y, Xia X, Lo D, et al. What makes a popular academic AI repository? Empir Software Eng, 2021, 26: 2
284 Liu J, Huang Q, Xia X, et al. Is using deep learning frameworks free? Characterizing technical debt in deep learning frameworks. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society, 2020. 1–10
285 Liu J, Huang Q, Xia X, et al. An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. Empir Software Eng, 2021, 26: 16
286 Han J, Deng S, Lo D, et al. An empirical study of the dependency networks of deep learning libraries. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2020. 868–878
287 Sun X, Zhou T, Li G, et al. An empirical study on real bugs for machine learning programs. In: Proceedings of the 24th Asia-Pacific Software Engineering Conference, 2017. 348–357
288 Zhang R, Xiao W, Zhang H, et al. An empirical study on program failures of deep learning jobs. In: Proceedings of IEEE/ACM 42nd International Conference on Software Engineering, 2020. 1159–1170
289 Jia L, Zhong H, Wang X, et al. An empirical study on bugs inside TensorFlow. In: Proceedings of International Conference on Database Systems for Advanced Applications. Berlin: Springer, 2020. 604–620
290 Garcia J, Feng Y, Shen J, et al. A comprehensive study of autonomous vehicle bugs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 385–396
291 Chen Z, Yao H, Lou Y, et al. An empirical study on deployment faults of deep learning based mobile applications. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering, 2021. 674–685
292 Just R, Jalali D, Ernst M D. Defects4j: a database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the International Symposium on Software Testing and Analysis, 2014. 437–440
293 Abreu R, Zoeteweij P, van Gemund A J. An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, 2006. 39–46
294 Wong W E, Qi Y, Zhao L, et al. Effective fault localization using code coverage. In: Proceedings of the 31st Annual International Computer Software and Applications Conference, 2007. 449–456
295 Rao P, Zheng Z, Chen T Y, et al. Impacts of test suite's class imbalance on spectrum-based fault localization techniques. In: Proceedings of the 13th International Conference on Quality Software, 2013. 260–267
296 Shu T, Ye T, Ding Z, et al. Fault localization based on statement frequency. Inf Sci, 2016, 360: 43–56
297 Feyzi F, Parsa S. Inforence: effective fault localization based on information-theoretic analysis and statistical causal inference. Front Comput Sci, 2019, 13: 735–759
298 Madeiral F, Urli S, Maia M, et al. BEARS: an extensible Java bug benchmark for automatic program repair studies. In: Proceedings of IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, 2019. 468–478
299 Saha R K, Lyu Y, Lam W, et al. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs. In: Proceedings of the 15th International Conference on Mining Software Repositories, 2018. 10–13
300 Song Y, Xie X, Liu Q, et al. A comprehensive empirical investigation on failure clustering in parallel debugging. J Syst Software, 2022, 193: 111452
301 Song Y, Xie X, Zhang X, et al. Evolving ranking-based failure proximities for better clustering in fault isolation. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022
302 Chen T, Cheung S, Yiu S. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report hkust-cs98-01. Hong Kong University of Science and Technology, 1998
303 Xie X, Ho J W K, Murphy C, et al. Testing and validating machine learning classifiers by metamorphic testing. J Syst Software, 2011, 84: 544–558
304 Xie X, Zhang Z, Chen T Y, et al. METTLE: a METamorphic testing approach to assessing and validating unsupervised machine learning systems. IEEE Trans Rel, 2020, 69: 1293–1322
305 Xie X, Ho J, Murphy C, et al. Application of metamorphic testing to supervised classifiers. In: Proceedings of the 9th International Conference on Quality Software, 2009. 135–144
306 Chen S, Jin S, Xie X. Testing your question answering software via asking recursively. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, 2021. 104–116
307 Grottke M, Trivedi K S. A classification of software faults. J Reliab Engin Assoc Japan, 2005, 27: 425–438

Appendix A

Table A1 AI4FL tools (material)

Paper	Description	Website
[229]	Symbiosis: A concurrency debugging technique based on differential schedule projections	https://github.com/nunomachado/symbiosis
[160]	FLUCCS: A fault localization technique that learns to rank program elements based on both existing SBFL techniques and source code metrics	https://bitbucket.org/teamcoinse/fluccs
[145]	An approach to identifying high-impact bug reports	https://github.com/goddding/JCST
[189]	PkgRec: A tool to predicting the confidence score of a package being affected by a new bug report	http://github.com/tkdsheep/MultiPackage
[174]	DeepBugs: A learning approach to name-based bug detection	https://github.com/michaelpradel/DeepBugs
[206]	NeuralBugLocator: A deep learning-based technique that can localize bugs without running the program	https://bitbucket.org/iiscseal/nbl/
[218]	Tregression: An Eclipse plugin to visualizing the bugs in Defects4J repository	https://github.com/llmhyy/tregression
[228]	FixML: A system for automatically generating feedback on logical errors in functional programming assignments	https://github.com/kupl/FixML
[147]	ChangeLocator: A method to automatically locating crash-inducing changes for a given bucket of crash reports	https://bitbucket.org/rongxin/changelocator-dataset
[69]	RepLoc: An automated framework to localizing the problematic files for unreproducible builds	https://reploc.bitbucket.io/
[171]	SCMiner: An online bug diagnosis tool	https://github.com/Tarannum-Zaman/Scminer
[134]	DeepFL: A deep learning approach to automatically learning the most effective existing/latent features for fault localization	https://github.com/DeepFL/DeepFaultLocalization
[176]	A tool to constructing datasets for evaluating AI-based fault localization techniques	https://github.com/yanxiao6/BugLocalization-dataset
[139]	CraTer: An automatic approach for predicting whether a crashing fault resides in stack traces	http://cstar.whu.edu.cn/p/crater/
[151]	D&C: A divide-and-conquer approach for IR-based fault localization	https://github.com/d-and-c/d-and-c
[109]	MLDebugger: A method that iteratively finds minimal definitive root causes	https://github.com/raonilourenco/MLDebugger
[163]	CombineFL: An infrastructure for evaluating and combining fault localization techniques	https://damingz.github.io/combinefl/index.html
[64]	HSFL: A historical spectrum-based fault localization technique	https://github.com/justinwm/HSFL
[146]	ChangeRanker: A fault localization tool with feature selection technique	https://github.com/Naplues/ChangeRanker
[194]	Ulysis: A metric to capturing the quality of test suites for diagnosability	https://github.com/EvoSuite/evosuite/pull/293
[183]	OffSide: A technique to identifying mistakes in boundary conditions	https://github.com/SERG-Delft/ml4se-offside
[143]	ALBFL: A neural ranking model that combines static and dynamic features for fault localization	https://github.com/indigo-99/ALBFL
[197]	A test case resampling tool for boosting the effectiveness of deep learning-based fault localization techniques	https://github.com/zhuozhangNUDT/test-case-resampling
[204]	RecBi: A reinforcement compiler bug isolation tool	https://github.com/haoyang9804/RecBi
[138]	JDCallgraph: A tool to generating dynamic call graphs for fault localization	https://github.com/dkarv/jdcallgraph
[170]	A two-phase framework for just-in-time defect identification and localization	https://github.com/MengYan1989/JIT-DIL
[168]	Grace: A fault localization tool using graph-based representation learning	https://github.com/yilinglou/Grace
[8]	ProbDD: A probabilistic delta debugging algorithm	https://github.com/Amocy-Wang/ProbDD
[101]	DeepFD: A learning-based fault diagnosis and localization framework	https://github.com/ArabelaTso/DeepFD

Table A2 AI4APR tools (material)

Paper	Description	Website
[128]	A technique for history-based program repair	https://github.com/xuanbachle/bugfixes
[255]	A set of anti-patterns that can be used on search-based repair tools	https://anti-patterns.github.io/search-based-repair/
[127]	Nopol: An approach to automatic repair of buggy conditional statements	http://github.com/SpoonLabs/nopol/
[129]	ACS: A program repair system to generating conditions at faulty locations	https://github.com/Adobee/ACS
[252]	An approach that heuristically determines the correctness of the generated patches	https://github.com/Ultimanecat/DefectRepairing
[225]	ARJA: A new GP based repair approach for automated repair of Java programs	https://github.com/yyxhdy/arja
[259]	A semantic program embedding for program repair	https://github.com/keowang/dynamic-program-embedding
[232]	Clara: An automated program repair algorithm for introductory programming assignments	https://github.com/iradicek/clara
[237]	SimFix: An automatic program repair approach that utilizes both existing patches and similar code	https://github.com/xgdsmileboy/SimFix
[245]	CapGen: A context-aware patch generation approach	https://github.com/justinwm/CapGen
[239]	iFixR: A debugging pipeline incorporating both patch generation and patch validation	https://github.com/SerVal-DTF/iFixR
[99]	SEQUENCER: An end-to-end approach to program repair based on sequence-to-sequence learning	https://github.com/kth/SequenceR
[98]	A tool to automatically learning bug-fixes in the wild for patch generation	https://sites.google.com/view/learning-fixes
[235]	DeepRepair: An approach for sorting and transforming program repair ingredients	https://sites.google.com/view/deeprepair
[260]	RLAssist: A deep reinforcement learning-based program repair technique	https://bitbucket.org/iiscseal/rlassist
[258]	DrRepair: A self-supervised learning-based program repair technique	https://github.com/michiyasunaga/DrRepair
[100]	CoCoNuT: An ensemble learning-based patch generation and validation tool	https://github.com/lin-tan/CoCoNut-Artifact
[244]	DLFix: A two-tier APR tool based on code transformation learning	https://github.com/ICSE-2019-AUTOFIX/ICSE-2019-AUTOFIX
[224]	SCRepair: A smart contract repair tool	https://screpair-apr.github.io/
[226]	ARJA-e: An evolutionary repair tool for Java code	https://github.com/yyxhdy/arja/tree/arja-e
[97]	A tool to mining fix patterns for FindBugs violations	https://github.com/FixPattern/findbugs-violations
[238]	CURE: An NMT-based APR tool	https://github.com/lin-tan/CURE
[240]	Recoder: A syntax-guided APR tool	https://github.com/pkuzqh/Recoder
[263]	CPR: An APR tool for network control planes	https://bitbucket.org/uw-madison-networking-research/arc

Table A3 SD4AI tools (material)

Paper	Description	Website
[112]	A technique to identifying training points most responsible for a given prediction	http://bit.ly/gt-influence
[113]	KARMA: A causal unlearning system that uses several mechanisms to determine the set of polluted data sample	https://github.com/CausalUnlearning/KARMA
[131]	LDADE: A search-based tool that tunes LDA's parameters to fix its systematic errors	https://github.com/ai-se/Pits_lda/
[277]	An error correction mechanism for NMT	https://github.com/StillKeepTry/ECM-NMT
[274]	TRADER: A tool for debugging RNN models	https://github.com/trader-rnn/TRADER
[268]	BugDoc: A framework for finding root causes of errors in computational pipelines	https://github.com/ViDA-NYU/BugDoc
[278]	DEBAR: A static analysis tool for detecting numerical bugs in neural architectures	https://github.com/ForeverZyh/DEBAR
[270]	DeepLocalize: A tool to identifying the root causes for DNN errors	https://github.com/Wardat-ISU/DeepLocalize
[119]	175 faults gathered from GitHub and StackOverflow	https://github.com/ForeverZyh/TensorFlow-Program-Bugs
[290]	499 faults from 16851 commits in Baidu Apollo and Autoware	http://tiny.cc/cps_bug_analysis
[289]	202 TensorFlow bug fixes (with some bug reports)	https://github.com/fordataupload/tfbugdata
[34]	667 DNN instances (including bug and repair) for DNN repair	https://github.com/lab-design/ICSE2020DNNBugRepair
[121]	1981 commits, 1392 issues/pull requests and 2653 posts for delivering a taxonomy of real faults in deep learning systems	https://github.com/dlfaults/dl_faults
[282]	59 deep learning repositories on GitHub for investigating the scent of deep learning code	https://github.com/Hadhemii/DLCodeSmells
[291]	304 deployment faults gathered from GitHub and StackOverflow	https://github.com/chenzhenpeng18/icse2021
[283]	1149 labeled academic AI repositories (popular or unpopular)	https://github.com/YuanruiZJU/academic-ai-repos