SCIENCE CHINA
Information Sciences

• REVIEW •

January 2025, Vol. 68, Iss. 1, 111102:1–111102:88
https://doi.org/10.1007/s11432-023-4127-5

Deep learning-based software engineering: progress, challenges, and opportunities†

Xiangping CHEN[2*], Xing HU[3*], Yuan HUANG[4], He JIANG[5*], Weixing JI[6], Yanjie JIANG[1*], Yanyan JIANG[7*], Bo LIU[6], Hui LIU[6], Xiaochen LI[5], Xiaoli LIAN[8*], Guozhu MENG[9*], Xin PENG[10*], Hailong SUN[11*], Lin SHI[11*], Bo WANG[12*], Chong WANG[10], Jiayi WANG[7], Tiantian WANG[13*], Jifeng XUAN[14*], Xin XIA[15], Yibiao YANG[7*], Yixin YANG[11], Li ZHANG[8], Yuming ZHOU[7*] & Lu ZHANG[1*]

[1]*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, School of Computer Science, Peking University, Beijing 100871, China;*
[2]*School of Journalism and Communication, Sun Yat-sen University, Guangzhou 510275, China;*
[3]*School of Software Technology, Zhejiang University, Hangzhou 310058, China;*
[4]*School of Software Engineering, Sun Yat-sen University, Guangzhou 510275, China;*
[5]*School of Software, Dalian University of Technology, Dalian 116024, China;*
[6]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China;*
[7]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China;*
[8]*School of Computer Science and Engineering, Beihang University, Beijing 100191, China;*
[9]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100864, China;*
[10]*School of Computer Science, Fudan University, Shanghai 200433, China;*
[11]*State Key Laboratory of Complex & Critical Software Environment (CCSE), School of Software, Beihang University, Beijing 100191, China;*
[12]*School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China;*
[13]*School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China;*
[14]*School of Computer Science, Wuhan University, Wuhan 430072, China;*
[15]*Huawei Technologies, Hangzhou 310056, China*

Received 26 September 2023/Revised 31 December 2023/Accepted 1 April 2024/Published online 24 December 2024

Abstract Researchers have recently achieved significant advances in deep learning techniques, which in turn has substantially advanced other research disciplines, such as natural language processing, image processing, speech recognition, and software engineering. Various deep learning techniques have been successfully employed to facilitate software engineering tasks, including code generation, software refactoring, and fault localization. Many studies have also been presented in top conferences and journals, demonstrating the applications of deep learning techniques in resolving various software engineering tasks. However, although several surveys have provided overall pictures of the application of deep learning techniques in software engineering, they focus more on learning techniques, that is, what kind of deep learning techniques are employed and how deep models are trained or fine-tuned for software engineering tasks. We still lack surveys explaining the advances of subareas in software engineering driven by deep learning techniques, as well as challenges and opportunities in each subarea. To this end, in this study, we present the first task-oriented survey on deep learning-based software engineering. It covers twelve major software engineering subareas significantly impacted by deep learning techniques. Such subareas spread out through the whole lifecycle of software development and maintenance, including requirements engineering, software development, testing, maintenance, and developer collaboration. As we believe that deep learning may provide an opportunity to revolutionize the whole discipline of software engineering, providing one survey covering as many subareas as possible in software engineering can help future research push forward the frontier of deep learning-based software engineering more systematically. For each of the selected subareas, we highlight the major advances achieved by applying deep learning techniques with pointers to the available datasets in such a subarea. We also discuss the challenges and opportunities concerning each of the surveyed software engineering subareas.

Keywords deep learning, software engineering, software benchmark, software artifact representation, survey

Citation Chen X P, Hu X, Huang Y, et al. Deep learning-based software engineering: progress, challenges, and opportunities. Sci China Inf Sci, 2025, 68(1): 111102, https://doi.org/10.1007/s11432-023-4127-5

* Corresponding author (email: chenxp8@mail.sysu.edu.cn, xinghu@zju.edu.cn, jianghe@dlut.edu.cn, yanjiejiang@pku.edu.cn, jyy@nju.edu.cn, lianxiaoli@buaa.edu.cn, mengguozhu@iie.ac.cn, pengxin@fudan.edu.cn, sunhl@buaa.edu.cn, shilin@buaa.edu.cn, wangbo_cs@bjtu.edu.cn, wangtiantian@hit.edu.cn, jxuan@whu.edu.cn, yangyibiao@nju.edu.cn, zhouyuming@nju.edu.cn, zhanglucs@pku.edu.cn)
† All authors have the same contribution to this work.

1 Introduction

In recent years, deep learning, first proposed by Hinton et al. [1] in 2006, has achieved highly impressive advances [2]. Because of the unsupervised-layer-wise training proposed by Hinton et al. [3], the major obstacle to the training of deep neural networks has been removed. Since then, most artificial intelligence (AI) researchers have turned to deep learning, constructing deep neural networks containing dozens, thousands, and even millions of layers [4]. They have also proposed various novel structures of deep neural networks, such as convolutional neural networks (CNNs) [5], recurrent neural networks (RNNs) [6], long short-term memory networks (LSTMs) [7], bidirectional LSTM [8], and Transformer [9]. With the advances and popularity of deep learning, hardware vendors like NVIDIA release more powerful computing devices specially designed for deep learning. All of these together significantly push forward machine learning techniques and make deep learning-based AI one of the most promising techniques in the 21st century.

Given significant advances in deep learning, various deep learning techniques have been employed to fulfill software engineering tasks [10]. Although natural language processing (NLP), image and video processing, and speech processing are the major targets of current deep learning techniques, deep learning has been successfully applied to a wide range of various domains, including data mining [11], machine manufacturing [12], biomedical engineering [13], and information security [14]. Concerning software engineering, researchers have successfully exploited various deep learning techniques for various important tasks, such as code generation [15], code completion [16], code summarization [17], software refactoring [18], code search [19], fault localization [20], automated program repair [21], vulnerability detection [22], and software testing [23]. In all such tasks, deep learning techniques have been proven useful, substantially improving the state-of-the-art. One possible reason for the success of deep learning-based software engineering is the significant advances in deep learning techniques. Another possible reason is that various and massive software engineering data are publicly available for training advanced neural models. With the popularity of open-source software applications, developers share massive software requirements, source code, documents, bug reports, patches, test cases, online discussions, and logs and trace relationships among different artifacts. All such data make training specialized deep neural models for software engineering tasks feasible. To our knowledge, there have already been several surveys on deep learning for software engineering (e.g., Yang et al. [10], Watson et al. [24] and Niu et al. [25]). Although these surveys provide some overall pictures of the applications of deep learning for software engineering, there is still a lack of detailed analyses of the progress, challenges, and opportunities of deep learning techniques from the perspective of each subarea of software engineering influenced by deep learning.

In this paper, we present a detailed survey covering the applications of deep learning techniques in major software engineering subareas. We choose to provide one survey to cover the technical research of deep learning for the whole discipline of software engineering instead of several surveys for different individual subareas for the following reasons. First, software engineering has one central objective, and researchers follow a divide-and-conquer strategy to divide software engineering into different subareas. However, the advances of deep learning may provide us an opportunity to break the boundaries of research in different subareas to push forward the software engineering discipline as a whole. That is, deep learning may provide a common means to revolutionize software engineering in the future. Therefore, we believe that a survey of deep learning in all software engineering subareas may be more beneficial for software engineering researchers. Second, deep learning belongs to representation learning, and deep learning techniques for software engineering are thus highly specific to different artifacts in software development. As different subareas of software engineering typically share some common artifacts, putting different subareas into one survey would help researchers from different subareas understand the strengths and weaknesses of different deep learning techniques. For example, a deep learning technique based on source code may impact all subareas related to the comprehension, generation, and modification of codes.

One issue that raises in preparing this survey is that software engineering is a big discipline and the applications of deep learning techniques may thus touch some subareas in an uninfluential way. When facing such a subarea, where there are very few deep learning papers, we feel that these papers can hardly reflect the essence of deep learning research in that subarea. Therefore, instead of covering all subareas of software engineering, we focus on subareas where deep learning has already had significant impacts. Fortunately, deep learning has deeply impacted software engineering, and through the subareas we selected for surveying in this paper, we can already form a general and insightful picture of deep learning for software engineering. To achieve our goal, for each of the selected subareas, we highlight

Table 1 Software engineering tasks covered by the survey.

	Software engineering task	Number of surveyed papers
1	Requirements engineering	28
2	Code generation	46
3	Code search	40
4	Code summarization	55
5	Software refactoring	19
6	Code clone detection	53
7	Software defect prediction	32
8	Bug finding	114
9	Fault localization	42
10	Program repair	64
11	Bug report management	51
12	Developer collaboration	57
	Total	601

major technical advances, challenges, and opportunities. As far as we know, our survey is the first task-oriented survey on deep learning-based software engineering, providing a technical overview of software engineering research driven by deep learning.

Another issue is the alignment between software engineering and deep learning. From the perspective of software engineering, research efforts are primarily divided into five phases (i.e., requirements, design, implementation, testing, and maintenance) according to the software development lifecycle, where each phase may be further divided into different software development activities. Meanwhile, from the perspective of deep learning, research efforts are grouped by the target learning tasks, where the most distinctive characteristic of a learning task is its input and output as deep learning is typically of an end-to-end fashion. That is, using the same form of input-output pairs is typically viewed as being the same task, but there may be several different tasks for achieving the same goal. To balance the two perspectives, we grouped the research efforts according to their goals. That is, we viewed research for achieving the same goal or a set of similar goals as being of one subarea, and accordingly divided the research efforts into 12 subareas (Table 1 for the numbers of surveyed papers in each subarea): requirements engineering, code generation, code search, code summarization, software refactoring, code clone detection, software defect prediction, bug finding, fault localization, program repair, bug report management, and developer collaboration. We believe that this organization is friendly to readers from both software engineering and deep learning fields. Each subarea is of clear semantics in software engineering, and the tasks in each subarea are highly related from the perspective of deep learning. We further arranged the subareas in an order consistent with the order of the five phases of software development. An additional benefit is that this organization naturally prevents one subarea from being overcrowded. However, this organization also demonstrates the following drawbacks. From the software development lifecycle perspective, many subareas belong to software maintenance, but fewer subareas belong to other phases in our survey. We would like to emphasize that although this may roughly reflect that more research efforts have been devoted to software maintenance, it does not mean that software maintenance is of more importance than the other phases in software engineering.

In collecting papers for our survey, we focused on publications in major conferences and journals on software engineering and artificial intelligence between 2000 and 2023. Tables 2 and 3 present the conferences and journals we searched for papers on deep learning for software engineering. Notably, as we are not performing a systematic literature review, we did not follow a strict procedure for data collection (i.e., paper collection). Therefore, although Tables 2 and 3 provide the conferences and journals from where papers were surveyed, the selection of papers for our survey also underwent subjective judgment from the authors for appropriation. That is, the primary goal of our survey is to provide readers with a general picture of technical advances in deep learning-based software engineering but not to characterize a precise distribution of research efforts across different software engineering subareas. Because there are quite many researchers interested in deep learning-based software engineering, the number of papers increases quickly. Consequently, the number of papers discussed in this survey has already significantly surpassed the number of surveyed papers in recent systematic literature reviews (i.e., Yang et al. [10] and Watson et al. [24]). Further, this survey notably belongs to the theme of AI4SE (where various techniques of artificial intelligence are applied to software engineering), whereas there is also intensive

Table 2 Conferences covered by the survey.

#ID	Abbreviation	Conference
1	AAAI	AAAI Conference on Artificial Intelligence
2	ACL	Annual Meeting of the Association for Computational Linguistics
3	APSEC	Asia-Pacific Software Engineering Conference
4	ASE	IEEE/ACM International Conference on Automated Software Engineering
5	COMPSAC	IEEE Annual Computers, Software, and Applications Conference
6	EASE	Evaluation and Assessment in Software Engineering
7	ESEC/FSE	ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
8	ICDM	IEEE International Conference on Data Mining
9	ICLR	International Conference on Learning Representations
10	ICMLA	International Conference on Machine Learning and Applications
11	ICML	International Conference on Machine Learning
12	ICPC	International Conference on Program Comprehension
13	ICSE	IEEE/ACM International Conference on Software Engineering
14	ICSME	IEEE International Conference on Software Maintenance and Evolution
15	ICSR	International Conference on Social Robotics
16	ICST	IEEE International Conference on Software Testing, Verification and Validation
17	ICTAI	IEEE International Conference on Tools with Artificial Intelligence
18	IJCAI	International Joint Conference on Artificial Intelligence
19	IJCNN	International Joint Conference on Neural Networks
20	Internetware	Asia-Pacific Symposium on Internetware
21	ISSRE	IEEE International Symposium on Software Reliability Engineering
22	ISSTA	ACM SIGSOFT International Symposium on Software Testing and Analysis
23	IWoR	International Workshop on Refactoring
24	IWSC	International Workshop on Software Clones
25	MSR	International Conference on Mining Software Repositories
26	NeurIPS	Annual Conference on Neural Information Processing Systems
27	NIPS	Advances in Neural Information Processing Systems
28	OOPSLA	Object-Oriented Programming, Systems, Languages, and Applications
29	QRS	International Conference on Software Quality, Reliability and Security
30	RE	International Conference on Requirements Engineering
31	SANER	International Conference on Software Analysis, Evolution and Reengineering
32	SEKE	International Conference on Software Engineering and Knowledge Engineering
33	S&P	IEEE Symposium on Security and Privacy
34	WCRE	Working Conference on Reverse Engineering
35	WWW	Web Conference

research in another related theme of SE4AI (where techniques for software engineering are applied to enhance artificial intelligence systems). An important goal of our subjective judgment is to avoid the inclusion of SE4AI papers in our paper collection.

2 Related work

Recently, Yang et al. [10] performed a systematic literature review to summarize, classify, and analyze relevant papers in the field of software engineering that leverage deep learning techniques. They collected in total of 250 relevant papers published in 32 major software engineering conferences and journals since 2006. Based on the papers, they analyzed the development trends of deep learning, provided a classification of deep learning models, and summarized the research topics tackled by these relevant papers. The major task of the systematic literature review is to figure out which and how deep learning techniques have been applied to software engineering. Their findings suggest that four categories of deep neural networks (DNNs) (CNN, LSTM, RNN, and feed-forward neural network (FNN)) were frequently employed by more than 20 studies. In addition, they summarized three types of DNN-based model selection strategies, i.e., characteristic-based selection, prior study-based selection based on, and using multiple feasible DNNs where the first strategy (characteristic-based selection) is by far the most popular

Table 3 Journals covered by the survey.

#ID	Abbreviation	Journal
1	ASE	Automated Software Engineering
2	COLA	Journal of Computer Languages
3	CSUR	ACM Computing Surveys
4	ESE	Empirical Software Engineering
5	FCS	Frontiers in Computer Science
6	IETS	IET Software
7	IST	Information and Software Technology
8	JSS	Journal of Systems and Software
9	NCA	Neural Computing and Applications
10	PACMPL	ACM on Programming Languages
11	RE	Requirements Engineering
12	SCIS	Science China Information Sciences
13	TC	IEEE Transactions on Computers
14	TOSEM	ACM Transactions on Software Engineering and Methodology
15	TSE	IEEE Transactions on Software Engineering
16	TSUSC	IEEE Transactions on Sustainable Computing
17	–	Soft Computing
18	–	IEEE Transactions on Reliability
19	–	IEEE Access
20	–	Expert Systems with Applications
21	–	Journal of Software: Evolution and Process
22	–	Advances in Engineering Software

one. Our survey differs from the systematic literature review by Yang et al. [10] in that Yang et al. focused more on deep learning techniques whereas we focus more on software engineering tasks. Yang et al. explained what deep learning techniques have been applied to software engineering whereas we analyze each software engineering task to explain how deep learning techniques could provide help for the task, and what kind of challenges could be encountered. In total, we analyze more than 500 papers that leverage deep learning techniques for software engineering, providing a more comprehensive view of this emerging research field.

Watson et al. [24] conducted another systematic literature review on the application of deep learning techniques in software engineering. They collected and analyzed 128 papers from software engineering and deep learning conferences and journals. The major task of the paper is to answer five questions, i.e., what types of software engineering tasks have been addressed by deep learning techniques, what deep learning techniques have been applied to software engineering, how requested training data are collected, how well software engineering tasks have been addressed by deep learning techniques, and what common factors influence the replicability of deep learning applied to software engineering tasks. Their findings suggest that deep learning-based techniques have been applied in a diverse set of tasks where program synthesis, code comprehension, and source code generation are the most prevalent. They also found that a number of different data preprocessing techniques have been utilized. Tokenization and neural embeddings are the two most frequently employed data pre-processing techniques. Besides that, their analysis revealed seven major types of deep learning architectures that have been employed for software engineering tasks, including RNNs, encoder-decoder models, CNNs, FNNs, AutoEncoders, Siamese neural networks, and highly tailored architectures. Our survey differs from Watson et al. [24] in that Watson et al. took software engineering as a whole to discuss the advances and challenges in applying deep learning techniques to software engineering. In contrast, we discuss the advances and challenges for each software engineering task concerning how deep learning techniques could be employed to resolve the given software engineering task.

The survey conducted by Niu et al. [25] reviews pre-trained models used in software engineering. In total, they identified and analyzed 20 pre-trained models developed for software engineering tasks. They classified the models with four dimensions, i.e., the underlying network architecture, the number of input modalities, the tasks used for pretraining, and whether they are pre-trained on a single programming language or multiple programming languages. It also investigated how the models were pretrained for different software engineering tasks. Their goal is to raise the awareness of AI technologies, including the

development and use of pre-trained models and the successful applications of such models in resolving software engineering tasks. Their findings suggest that code pre-training models are a promising approach to a wide variety of software engineering tasks. Our survey differs from the survey conducted by Niu et al. [25] in that the survey by Niu et al. is confined to pre-trained models whereas our survey covers all deep neural networks employed for software engineering tasks.

Zhang et al. [26] conducted a systematic survey to summarize the current state-of-the-art research in the large language model (LLM)-based software engineering (SE) community. They summarized 30 representative LLMs of source code across three model architectures, 15 pre-training objectives across four categories, and 16 downstream tasks across five categories. They presented a detailed summarization of the recent SE studies for which LLMs are commonly utilized. Furthermore, they summarized existing attempts to empirically evaluate LLMs in SE, such as benchmarks and empirical studies. Finally, they discussed several critical aspects of optimization and applications of LLMs in SE. Our survey differs from the survey conducted by Zhang et al. [26] in that our survey focuses on all deep neural networks employed for software engineering tasks. However, the survey conducted by Zhang et al. specifically concentrates on different representative LLMs.

As a conclusion, although a few surveys have been made concerning the synergy between software engineering and deep learning, we still lack a clear picture of the advances, potentials, and challenges concerning deep learning-driven attempts at various software engineering tasks. To this end, in this paper, we select the most fundamental and most challenging software engineering tasks, and for each of them we analyze the advances, potentials, and challenges of its deep learning-based solutions.

3 Requirements engineering

Requirements engineering (RE) is the process of eliciting stakeholder needs and desires and developing them into an agreed-upon set of detailed requirements that can serve as a basis for all subsequent development activities. Unlike solution-oriented SE tasks (such as software design and program repair) that aim to ensure "doing the thing right", requirements engineering is a problem-oriented SE task that aims to ensure "doing the right thing", i.e., to make the problem that is being stated clear and complete, and to guarantee that the solution is correct, reasonable, and effective [27].

Recently, more and more RE researchers employ deep learning techniques to elicit, analyze, trace, validate, and manage software requirements. In this section, we will introduce the progress of deep learning (DL)-related RE literature from three perspectives, i.e., RE task taxonomy, datasets, and DL models. Then, we will summarize the challenges and opportunities faced by DL-related RE literature.

3.1 Requirements elicitation

The task of requirements elicitation aims to gather accurate and complete information about what the system should do, how it should behave, and what constraints and limitations it should adhere to. This process is critical in software development because the success of a project depends on clearly understanding requirements and translating them into solutions that meet stakeholder needs.

Huang et al. [28] proposed a CNN-based approach for automatically classifying sentences into different categories of intentions: feature request, aspect evaluating, problem discovery, solution proposal, information seeking, information giving, and meaningless. They sped up the training process by integrating batch normalization. They also optimized the hyper-parameters through an automatic hyper-parameter tuning method in order to improve accuracy. Pudlitz et al. [29] presented an automated approach for extracting system states from natural language requirements using a self-trained named-entity recognition model with bidirectional LSTMs and CNNs. They presented a semi-automated technique to extract system state candidates from natural language requirements for the automotive domain. The results show that their automated approach achieves an F1-score of 0.51, with only 15 min of manual work, while the iterative approach achieves an F1-score of 0.62 with 100 min. Furthermore, manual extraction took 9 h, demonstrating that machine learning approaches can be applied with a reasonable amount of effort to identify system states for requirements analysis and verification.

Li et al. [30] proposed a technique called DEMAR based on deep multi-task learning, which can discover requirements from problem reports, and solve the limitations that requirements analysis tasks usually rely on manually coded rules or insufficient training data. Through the three steps of data augmentation, model building, and model training, their experimental results show that the multi-task learning mode of

DEMAR has higher performance than the single-task mode. Meanwhile, DEMAR also outperforms the other selected existing techniques. DEMAR provides directions for exploring the application of multi-task learning to other software engineering problems. Guo et al. [31] proposed Caspar, a technique for extracting and synthesizing app question stories from app reviews. Caspar first extracts ordered events from acquired app reviews and ranks them using NLP techniques. Caspar then classifies these events into user actions or application questions, and synthesizes action-question pairs. Finally, an inference model is trained on the operation-problem pairs. For a given user action, Caspar can infer possible program problem events, thus enabling developers to understand possible problems in advance to improve program quality. They experimented with SVM, USE+SVM, and Bi-LSTM networks, respectively; and the results show that the Bi-LSTM model performs better than the other two. Mekala et al. [32] proposed a deep learning-supported artificial intelligence pipeline that can analyze and classify user feedback. This technique includes a sequence classifier. Their experimental results show that the BERT-based classifier performs the best overall and achieves good performance. Their experimental results further demonstrate that pre-trained embeddings of large corpora are a very effective way to achieve state-of-the-art accuracy in the context of low-capacity datasets. Tizard et al. [33] proposed a technique for linking forum posts, issue trackers, and product documentation to generate corresponding requirements. They scraped product forum data for VLC and Firefox, performed similarity calculations using the universal sentence encoder (USE), and matched forum posts to issue trackers. Furthermore, they demonstrated that applying clustering to USE results in impressive performance on matching forum posts with product documents. Shi et al. [34] proposed a technique called FRMiner to detect feature requests with high accuracy from a large number of chat messages via deep Siamese networks. In particular, they used Spacy to perform word segmentation, lemmatization, and lowercase processing on sentences, randomly selected 400 conversations in three open-source projects for data sampling, and annotated them with the help of an inspection team. Then, they built a context-aware dialogue model using a bidirectional LSTM structure, and used a Siamese network to learn the similarity between dialogue pairs. Experimental results show that FRMiner significantly outperforms two-sentence classifiers and four traditional text classification techniques. The results confirm that FRMiner can effectively detect hidden feature requests in chat messages, thus helping obtain comprehensive requirements from a large amount of user data in an automated manner.

Pan et al. [35] introduced an automated developer chat information mining technique called F2CHAT, which aims to solve the challenge of retrieving information from online chat rooms. They constructed a thread-level taxonomy of nine categories by identifying different types of messages in developer chats. Furthermore, they proposed an automatic classification technique F2CHAT, which combines hand-crafted non-textual features with deep textual features extracted by neural models. The results show that F2CHAT outperforms FRMiner and achieves high performance in cross-project verification, indicating that F2CHAT can be generalized across various projects.

3.2 Requirements generation

The task of requirements generation aims to recommend requirements drafts on top of very limited information (e.g., keywords or structured models).

Most existing studies on automated requirements generation concentrate on transforming (semi-) structured models (e.g., business process models [36,37], i* framework [38,39], KAOS and Objectiver [40–42], UML models [43–45], or other representations like security goals in [46]) into specific syntactic pattern-oriented natural language requirements specifications, based on a set of pre-defined rules. Recently, Zhao et al. [47] proposed an approach called ReqGen that aims to recommend requirements drafts based on a few given keywords. Specifically, they first selected keyword-oriented knowledge from the domain ontology and injected it into the basic unified pre-trained language model (UniLM). Next, they designed a copy mechanism to ensure the occurrence of keywords in the generated statements. Finally, they proposed a requirement-syntax-constrained decoding technique to minimize the semantic and syntax distance between candidate and reference specifications. They evaluated ReqGen on two public datasets and demonstrated its superiority over six existing natural language generation approaches. Koscinski et al. [48] investigated the usage of relational generative adversarial networks (RelGAN) in automatically synthesizing security requirements specifications, and demonstrated promising results with a case study.

3.3 Requirements analysis

The purpose of requirements analysis is to systematically identify, understand, and document the software requirements for a given system or software project.

Li et al. [49] proposed a new technique named RENE, which uses the LSTM-CRF model for requirements entity extraction and introduces general knowledge to reduce the need for labeled data. This technique can more efficiently extract requirements entities from textual requirements, thereby reducing labor costs. They introduced the construction and training process of RENE, and evaluated RENE on an industrial requirements dataset, showing good results. Furthermore, they explored the value of general-purpose corpora and unlabeled data, and provided an effective practical approach that can inspire how to further improve performance on specific tasks.

3.4 Smelly requirements detection

The task of smelly requirements detection is to identify the potential issues in software requirements that might threaten the success of software projects.

Casillo et al. [50] proposed to utilize a pre-trained CNN to identify personal, and private disclosures from short texts to extract features from user stories. They constructed a user-story privacy classifier by combining the extracted features with those obtained from a privacy dictionary. Ezzini et al. [51] proposed an automated approach for handling anaphoric ambiguity in requirements, addressing both ambiguity detection and anaphora interpretation. They developed six alternative solutions based on the choices of (1) whether to use hand-crafted language features, word embeddings or a combination thereof for classification, (2) whether pre-trained language models like BERT are a viable replacement for the more traditional techniques, and (3) whether a mashup of existing (and often generic) NLP tools would be adequate for specific RE tasks. Wang et al. [52, 53] proposed a deep context-wise semantic technique to resolve entity co-reference in natural-language requirements, which consists of two parts. One is a deep fine-tuned BERT context model for context representation, and the other is a Word2Vec-based entity network for entity representation. Then they proposed to utilize a multi-layer perceptron (MLP) to fuse two representations. The input of the network is a requirement contextual text and related entities, and the output is a predicted label to infer whether the two entities are co-referent. Ezzini et al. [54] proposed QAssist, a question-answer system that provides automated assistance to stakeholders during the analysis of NL requirements. The system retrieves relevant text passages from the analyzed requirements document as well as external domain knowledge sources and highlights possible answers in each retrieved text passage. When domain knowledge resources are missing, QAssist automatically constructs a domain knowledge resource by mining Wikipedia articles. QAssist is designed to detect incompleteness and other quality issues in requirements without the tedious and time-consuming manual process.

3.5 Requirements classification

The purpose of requirements classification is to categorize and organize the collected requirements based on certain criteria or characteristics. By classifying requirements, the goal is to improve the understanding, management, and communication of requirements throughout the software development process.

Baker et al. [55] proposed to classify software requirements into five categories by using machine learning techniques: maintainability, operability, performance, security, and availability. They conducted experimental evaluations on two widely used software requirements datasets for CNN and artificial neural network (ANN) evaluation. The results show that this technique is effective, reaching high precision and recall. In addition, they explored potential applications of the technique in the software engineering life cycle, including automating the process of analyzing NFRs, reducing human errors and misunderstandings, and reducing potential requirements-related defects and errors in software systems. Hey et al. [56] proposed an automated requirements classification technique called NoRBERT, which uses the transfer learning capabilities of BERT. They evaluated NoRBERT on different tasks, including binary classification, multi-class classification, and classification in terms of quality. Their experimental results show that NoRBERT performs well when dealing with natural language requirements and can effectively improve the performance of automatic classification methods. Luo et al. [57] proposed a new requirements classification technique named PRCBERT, which is based on BERT's pre-trained language model

and applies flexible prompt templates to achieve accurate classification. They also conducted experiments on a large-scale demand dataset to compare it with other techniques, showing that the technique significantly improves accuracy and efficiency. Winkler et al. [58] proposed an automated approach for classifying natural language requirements by their potential verification techniques. They proposed to use a convolutional neural network for text classification, and trained it on a dataset created by industry requirements specifications. They performed 10-fold cross-validation on a dataset containing about 27000 industrial requirements, achieving a high F1 score. AlDhafer et al. [59] used bidirectional gated recurrent neural networks (BiGRU) to classify requirements into binary labels (functional and non-functional) and multiple labels (operational, performance, security, usability, etc.).

3.6 Requirements traceability

The task of requirements traceability is to build the associations between software requirements and other artifacts, which may be different types of requirements, design artifacts, source code, test cases, and other artifacts. Compared to other tasks in RE, much more research uses DL techniques for requirements traceability.

To the best of our knowledge, the first DL for requirements traceability research was from Guo et al. [60] in 2017. They proposed the technique of TraceNN to build the links between software requirements and design documents. In particular, they modeled the task of traceability construction as a classification problem. They embedded the features of text sequences based on the RNN and used the MLP to conduct the classification task. They evaluated two types of RNN models, namely LSTM and gated recurrent unit (GRU) on large-scale industrial datasets, and showed GRU achieved better mean average precision (MAP). In contrast, RNNs can only encode one side of contextual information, which will be weakened for long sequences [61]. With the wide application of BERT since 2018, different variants such as BioBERT [62] and CodeBERT [63] have been developed for various domains. Lin et al. proposed [64] TraceBERT to construct the trace links between requirements and source code based on CodeBERT, and modeled the traceability as a code search problem. They designed a three-fold procedure of pre-training, intermediate-training, and fine-tuning toward their tasks. Particularly, they first pre-trained CodeBERT on source code to build TraceBERT. Then in the intermediate-training phase, they provided adequate labeled training examples to train the model to address the code search problem with the expectation that the model can learn general traceability knowledge. Finally, in the fine-tuning phase, they applied the model to the specific "issue (natural language)-commit (programming language) tracing" problem to improve the tracing effect. They evaluated three commonly used BERT architectures (i.e., single, twin, and Siamese) on open-source projects. Their experimental results showed that the single architecture achieves the best accuracy, while the Siamese architecture achieves similar accuracy with faster training time. Tian et al. [65] proposed a technique named DRAFT to build the traceability between new requirements and other requirements in different abstraction levels, during the system evolution process. They performed a second-phase pre-training on BERT based on the project-related corpus for the purpose of project-related knowledge transformation. Then, they designed 11 heuristic features and embedded them with requirements text. The performance of DRAFT has been evaluated with eight open-source projects. Lin et al. [66] explored the performance of information retrieval and deep learning techniques in building trace links in 14 English-Chinese projects. The involved approaches include the vector space model (VSM), latent semantic indexing (LSI), latent Dirichlet allocation (LDA), and various models that combine mono- and cross-lingual word embeddings with the generative vector space model (GVSM), and a deep-learning approach based on a BERT language model. They showed that their TraceBERT performed best on large projects, while IR-based GVSM worked best on small projects.

From the perspective of paper distribution over different RE tasks, the most four tasks using DL techniques are requirements elicitation, requirements traceability, smelly requirements detection, and requirements classification, with 9, 7, 5, and 5 papers, respectively. For the other tasks, including requirements generation and analysis, the involved papers are few. We did not find any papers related to requirements management and requirements validation.

3.7 DL models

Figure 1 shows the DL models involved in our surveyed studies on requirements engineering. We can see that there are over 10 different kinds of models utilized in these DL4RE studies. Among these models,

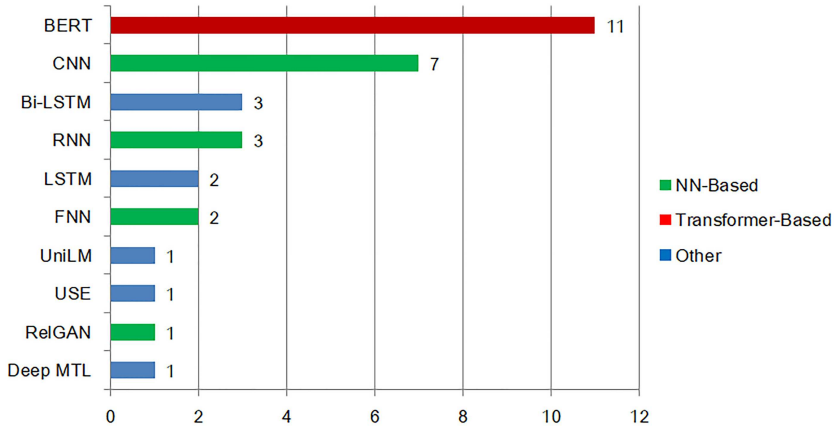

Figure 1 (Color online) Deep learning models involved in the related work.

BERT is the most widely-used one, followed by CNN, and the number of papers related to other models is far smaller than these two. This may be because BERT and CNN are relatively mature and universal deep learning models, which can adapt to different RE scenarios and datasets with high performance and reproducibility. It may also be due to the active and extensive research community of BERT, which can provide rich reference materials and the latest developments, stimulating the interest and innovation of researchers.

We can also see that the numbers of papers on the transformer-based model and NN-based model are 11 and 13, respectively, indicating that they have certain competition and complementarity in requirements engineering, and have certain research value and practical significance. There is no absolute difference between good and bad, but it is necessary to select or design appropriate in-depth learning models according to the specific RE tasks.

Usage of DL models. Figure 2 illustrates the ways of using deep learning models in requirements engineering. Primarily, there are three types: direct training, pre-trained, and fine-tune. Among them, 43% of the studies directly build neural networks from training. The advantage of direct training is that it can train a model from scratch according to the dataset of the target task, without being limited to the pre-trained model, and can customize the network structure and parameters towards specific domains and tasks. The disadvantage is that if the dataset of the target task is not large enough, direct training may lead to problems such as model non-convergence, overfitting, and low generalization ability. Considering that most of the studies use the way of direct training, it may indicate that this way is relatively simple and effective. Another possible reason is the lack of suitable pre-trained models or domain knowledge.

36% of studies use pre-trained DL models. The advantage is that it can use the model parameters trained by large-scale datasets, saving time and computing resources, and improving computational efficiency and accuracy. The disadvantage is that if the dataset of the pre-trained model and that of the target task are not highly similar, the effect of the pre-trained model may not be satisfactory, because different tasks need to extract different features.

21% of studies use fine-tune DL models. The advantage of fine-tune is that it can adjust part or all of the parameters based on the pre-trained model according to the dataset of the target task, retaining the ability of the pre-trained model to extract general features and increasing the ability of the model to adapt to new task features. The disadvantage is that it needs to choose a suitable pre-trained model, a suitable fine-tuning level and range, a suitable learning rate, and other hyperparameters. Otherwise, it may affect the effect of fine-tuning. Pre-training and fine-tuning each account for approximately a quarter, possibly because the use of pre-training and fine-tuning has certain limitations and complexity. Pre-training needs to select appropriate pre-trained models and objective functions, and the matching degree and difference between pre-training models and RE problems or data need to be considered. Fine-tuning needs to select appropriate fine-tuning strategies and parameter settings, and consider the similarities and differences between different RE tasks or datasets.

Performance of DL models. Most papers use three indicators to measure the performance of deep learning models: precision, recall, and F1. We find that the precision and recall of deep learning models are often above 80%, and the F1 score is often above 75%. The performance of deep learning models is related to many factors, such as the structure and parameters of the model, the training methods,

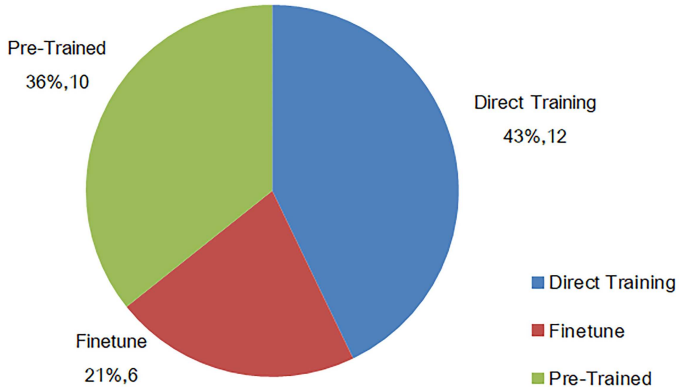

Figure 2 (Color online) Proportion of different usage of DL models.

and the size and quality of the dataset. Researchers often collect as many data as possible when using deep learning models to improve the expected performance of the models. However, it is not necessarily true that the larger the dataset is, the better the model performance is. The size of the selected dataset in [54] is only 387, but the average recall and precision of its model are over 90% and 84%. The dataset size used in [55] is 914, with a precision between 82% and 94%, a recall between 76% and 97%, and an F1-score between 82% and 92%. In [35], the size of the used dataset is 2959, the largest one is among these three work, but its F1-score is only 62.8%, smaller than the other two. We can see that there are many factors that affect the performance of deep learning models, and the size of the dataset is just one of them. Different factors also affect and constrain each other. We need to analyze and experiment based on specific situations.

3.8 Datasets

Datasets are a very important part of deep learning, as they are the foundation for training deep learning models. We focus on the selected datasets for the DL4RE studies and summarize the results into a bubble chart, as shown in Figure 3. The x-axis indicates the RE tasks that the studies focused on, and the y-axis shows the size of data entries used in training and testing the involved DL models. The size of the bubble indicates the number of publications. As shown in Figure 3, we can see that 61% of the selected publications have a dataset size of less than 6000, which may be due to the difficulty and cost of data collection. In some fields, data may be difficult to obtain and require professional equipments, personnel, or licenses.

We can also find that the datasets used in the tasks of requirements detection, requirement generation, and requirement classification are relatively smaller, which may be due to the phenomenon of data-hungriness in the RE community. As software requirements often embed the value that software products aim to deliver to their users, the requirements documents are typically private and confidential. Thus, it is difficult for DL4RE studies to obtain enough data for training effective DL models. Requirements generation is a text generation problem that can be improved by utilizing pre-trained language models without requiring a large amount of domain data. Besides, some techniques, such as data enhancement, transfer learning, and miniaturization networks, can be leveraged to reduce corpus size. In the meanwhile, requirements elicitation studies have larger datasets since it is a comprehensive task involving knowledge in multiple fields and multiple information sources. Moreover, many studies leverage the data in the open-source community, such as feature requests or live chats, to build their deep-learning models.

3.9 Challenges and opportunities

3.9.1 *Challenges*

• **Low transparency of public requirements.** Unlike other software development activities, the publicly accessible data for requirements typically are small in volume and lack details. For proprietary software, since requirements reflect the delivered value of the software, organizations usually consider requirements as confidential assets and are reluctant to open them. For open-source software (OSS), their requirements are scattered in massive informal online discussions, such as issue reports [34] or live community chats [30]. Although there are some widely-used requirements benchmarks, such as

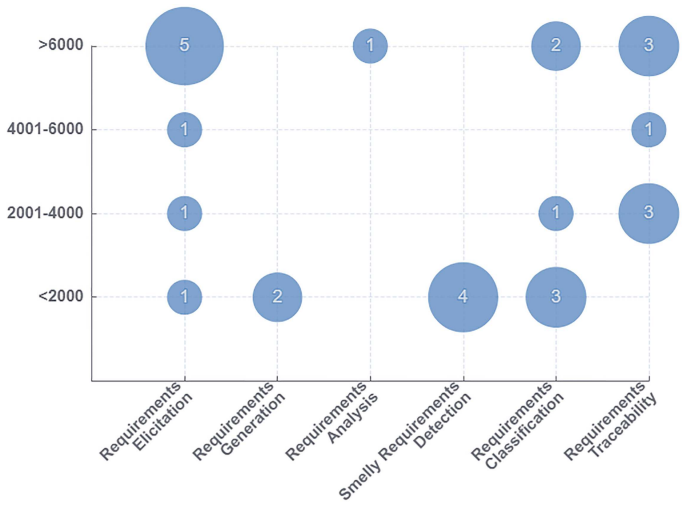

Figure 3 (Color online) Size distribution of the datasets involved in the DL4RE studies.

PROMISE and MODIS, the scale cannot be compared to that of open-source code in GitHub. Besides, many publicly accessible benchmarks depict requirements in textual and entry-oriented formats, lacking details on project-level information, such as detailed background, stakeholder preferences, and graphic presentations. Thus, it becomes difficult for RE researchers to comprehensively understand the rationale behind the textual requirements, as well as efficiently train DL models for RE.

• **High diversity in representations.** Software requirements are typically depicted in a variety of formats, including textual formats, data-entry formats, user stories, use cases, as well as (semi-) structured models. Notably, the representation within each format can also vary significantly. For instance, when it comes to modeling requirements, representations might take the form of UML, systems modeling language (SysML), goal-oriented models, etc. Although guideline standards like ISO/IEC/IEEE 29148:2018 [67] and easy approach to requirements syntax (EARS) [68] are provided, their adoption in the industry is limited. It is reported that nearly half (47%) of RE practitioners are unaware of these standards [69]. Consequently, the representation of requirements often aligns with individual writing practices rather than standardized guidelines. This significant diversity in representations poses a considerable challenge for DL models in accurately interpreting requirements. Additionally, it presents a notable obstacle in training DL models too, compounding the complexity of achieving reliable performance across varied formats.

• **Hidden domain-specific knowledge and experiences.** Many requirements-related tasks (such as specification, modeling, and analysis) hinge on domain-specific knowledge and the accumulated experience of experts. For example, it is commonly accepted that high-quality requirements must embody three key characteristics: correctness, consistency, and completeness — collectively known as the '3Cs'. Yet, these attributes are not clearly and formally defined in the industry, nor are there universally recognized detailed metrics to evaluate whether a set of requirements meets the 3C standard. Consequently, this aspect of the work often depends on the nuanced insights of experienced engineers. Regrettably, requirements engineers typically do not document this critical domain knowledge or their hands-on engineering experiences. This documentation is not typically required by project management protocols. Moreover, the challenge of accurately capturing the objective essence of such experiential knowledge remains unresolved. This gap means that DL models are currently difficult to absorb and apply such valuable expertise to support RE tasks.

3.9.2 *Opportunities*

• **Needs for research on piecing scattered OSS requirements into a big picture.** The influence and momentum of OSS on other SE tasks, like code generation and repair, are substantial and revolutionary. To advance AI4RE research, we appeal to construct evolving requirements for OSS. As the above discussion, OSS requirements are scattered in massive informal online discussions. Therefore, it is crucial to embark on research focused on pinpointing discrete pieces of requirements information, assembling them into coherent and intelligible requirements documents, and ensuring their evolution in

tandem with changes in the code base.

• **Exploring the feasibility and usability of AI-based RE approaches in real industrial settings.** While AI-based approaches have demonstrated efficacy in a multitude of tasks, their effectiveness for RE-tasks in real-world industrial settings remains uncertain. Indeed, there was even evidence that practicing engineers are reluctant to rely on AI models for higher-level design goals [70]. This presents a significant opportunity to delve into and expand upon this uncharted area of RE research.

• **Leveraging the interaction capability of LLMs to enhance requirements activities.** High-quality requirements lay the foundation for robust software products. In essence, the success of LLMs in various SE tasks is contingent upon the quality of these underlying software requirements. Given the impressive interactive capabilities of LLMs, a promising avenue is to harness these models to engage with diverse stakeholders intelligently. Through such interactions, LLMs can be leveraged to efficiently carry out tasks like requirements discovery, negotiation, and validation, anchored by rapid prototyping.

4 Code generation

In practice, a highly efficient code generation model may significantly boost developers' productivity, enabling them to complete programming tasks by simply inputting target descriptions into code generators [71]. Therefore, automated code generation can be deemed as an important objective for developers, highlighting the crucial role of the code generation task in the SE field. In addition, code completion is one of the most widely used code generation tasks in integrated development environments (IDEs) [72]. According to the study on the Eclipse IDE by Murphy et al. [73], code completion is one of the top ten commands used by developers. Recent years have seen increasing interest in code completion systems using deep learning techniques [74, 75].

To achieve this goal, within the academic community, a range of studies [76] have put forth DL-based models for automatically generating code fragments from input utterances. To gain a deeper understanding of these advanced techniques, in this section, we survey different studies on DL-based code generation. In particular, we survey the following three types of research: (1) research on proposing new techniques, (2) research on empirical evaluations, and (3) research on constructing datasets. Due to the specific characteristics of code, it is necessary to invent specialized deep learning techniques to deal with code. Therefore, we can see a number of influential deep learning techniques for code stemming from the area of code generation. Table 4 [77–107] highlights the main technical contributions in code generation. Since code completion is one of the most important code generation applications, we also survey deep learning-based code completion.

The primary contributions of the analyzed studies can be classified into nine distinct categories, highlighting the diverse range of advancements and innovations within code generation.

4.1 Enhancing code structure information

Six studies highlight the oversight of rich structural information in many code generators [106]. To address this concern, diverse approaches have been proposed for incorporating additional code structure information into their DL-based models, aiming to enhance the overall performance. Rabinovich et al. [77] introduced abstract syntax networks, a DL-based model that incorporates supplementary structural information from abstract syntax trees (ASTs). Their model utilizes an encoder-decoder BiLSTM with hierarchical attention, aiming to generate well-formed and executable Python code fragments. Jiang et al. [81] observed that the standard Seq2Tree model translates the input natural language description into a sequence based on the pre-order traversal of an AST. However, this traversal order might not be optimal for handling multi-branch nodes in certain cases. To address this, they put forward the idea of enhancing the Seq2Tree model with a context-based branch selector, enabling it to dynamically determine the optimal expansion orders for multi-branch nodes.

4.2 Special code generation

Four studies employ DL techniques to generate code fragments for less common programming languages or in unusual logical forms. Yu et al. [83] introduced a novel DL model specifically designed for generating structured query language (SQL) code for test code scenarios. Yang et al. [85] introduced a pre-trained model to generate assembly code from NL descriptions.

Table 4 Characteristics of studies within the design research category.

Contribution	Model type	Programming language (PL)	Ref.
Enhanced code structural info	DL	Python	[77]
	DL	Java	[78]
	DL	Java, Python	[79]
	DL	Python, SQL	[80]
	Pre-trained	Python	[81]
Special code generation	DL	Conditional statement	[82]
	DL	SQL (Test Code)	[83]
	DL	Pseudo-code	[84]
	Pre-trained	Assembly, Python	[85]
Multi-mode-based	DL	Java, Python	[86, 87]
	Pre-trained	Python	[88, 89]
	Pre-trained	Go, Java, JavaScript, PHP, Python, Ruby	[90]
Compilability	DL	SQL	[91]
	Pre-trained	Python	[92]
	Pre-trained	SQL, Vega-Lite, SMCalFlow	[93]
Dual learning-based	DL	Java, Python	[94]
	Pre-trained	Java	[95]
	Pre-trained	Python, SQL	[96]
Search-based	DL	Python	[97]
	DL	C++	[98]
	Pre-trained	Java, Python	[99]
Context-aware	DL	Java	[100, 101]
	Pre-trained	Python	[102]
Practicality	DL	python, SQL	[103]
	DL	Javascript	[104]
Long dependency problem	DL	Python	[105–107]

4.3 Multi-mode based code generation

The studies in this category construct the code generators by taking into account multiple code artifacts in a comprehensive manner. Le et al. [89] noticed that most code generators overlook certain crucial yet potentially valuable code specifications, such as unit tests, which frequently lead to subpar performance when addressing intricate or unfamiliar coding tasks. They thus introduced a new generation procedure with a critical sampling strategy that allows a model to automatically regenerate programs based on feedback from example unit tests and NL descriptions. Wang et al. [90] integrated NL descriptions with the specific attributes of programming languages, such as token types, to construct CodeT5, which is a unified pre-trained encoder-decoder transformer model designed for generating code fragments.

4.4 Compilability

Three studies concentrate on the development of DL-based code generators with the objective of generating executable code fragments across multiple programming languages. Sun et al. [91] identified a significant number of inaccuracies and non-executable SQL code generated as a result of the mismatch between question words and table contents. To mitigate this problem, they took into account the table structure and the SQL language syntax to train a DL-based SQL generator. Wang et al. [92] and Poesia et al. [93] leveraged the powerful capability of LLMs to improve the compilability of generated code.

4.5 Dual-learning-based code generation

Three studies capitalize on the dual connections between code generation (CG) and code summarization (CS) to generate accurate code fragments from NL descriptions. Wei et al. [94] exploited the duality between code generation and code summarization tasks to propose a DL-based dual training framework to train the two tasks simultaneously. Ahmad et al. [95] and Ye et al. [96] leveraged the inherent relationship between CG and CS and utilized pre-trained models to enhance the accuracy of the code generation task.

4.6 Code generation on top of existing code

Several studies have observed that generating code based on existing related code fragments can yield superior performance compared to generating code from scratch. As a result, these studies incorporate code search techniques and leverage DL to construct their code generators. For instance, Hashimoto et al. [97] and Kulal et al. [98] employed DL-based retrieval models to generate Python and C++ code. Additionally, Parvez et al. [99] utilized information retrieval techniques along with pre-trained models to develop a code generator capable of generating code in multiple programming languages, such as Java and Python.

4.7 Context-aware code generation

Recognizing that the code fragments generated by many existing code generators may not be directly applicable in software, several studies have proposed to incorporate code contexts to enhance the accuracy of code generation. Guo et al. [101] introduced a context-aware encoder-decoder model with a latent variable in their code generator, enabling it to incorporate the contextual environment during code generation. Li et al. [102] proposed SKCODER, an approach for sketch-based code generation, aiming to simulate developers' code reuse behavior. SKCODER retrieves a similar code snippet based on an NL requirement, extracts relevant parts as a code sketch, and then modifies the sketch to generate the desired code.

4.8 Practicality

Two relevant studies aim to improve the practicality of code generators. Dong et al. [103] introduced a structure-aware neural architecture for code generation that exhibits adaptability to diverse domains and representations. Shen et al. [104] proposed a task augmentation technique that integrates domain knowledge into code generation models, making their model the first domain-specific code generation system adopted in industrial development environments.

4.9 Long dependency

Many deep learning-based code generators are trained using RNNs such as LSTM[1)], BiLSTM[2)], and GRU [108]. To overcome the long-term dependency problem, three studies introduce novel techniques to tackle this challenge. Sun et al. [106] proposed a novel tree-based neural architecture and applied the attention mechanism of transformers to alleviate the long-dependency problem. Xie et al. [107] utilized mutual distillation learning to train a code generator in order to avoid the occurrence of this problem.

4.10 Code completion

Table 5 [109–114] shows code completion techniques published in the premier publication venues (i.e., ASE, ICSE, and FSE) from 2020 to 2023. Since 2020, there have been six papers focusing on the code completion task, in which three of them exploit the pre-trained models or large language models. GPT-2 is widely used in completing source code. Among all programming languages, Java and Python are the most popular programming languages. Wang et al. [113] conducted an empirical study that investigated developers' perspectives on code completion. Liu et al. [109] and Izadi et al. [110] integrated the multi-task learning techniques by learning different types of information of the source code (e.g., token sequences and ASTs). Tang et al. [111] introduced the retrieval-augmented language model to conduct domain adaption and improve the performance of existing LLMs (e.g., ChatGPT and UniXcoder) on the code completion task. Their results show that retrieval techniques can be seamlessly integrated with black-box code completion models and as a plugin to further enhance the performance of LLMs.

4.11 Empirical studies

Four studies [115–118] undertook empirical investigations to examine the characteristics of existing code generators. Dahal et al. [115] leveraged text-to-tree, linearized tree-to-tree, and structured tree-to-tree code generation models to perform an empirical analysis of the significance of input forms for code generation. They found that using a structure-aware model improves the performance of models on

1) https://en.wikipedia.org/wiki/Long_short-term_memory.
2) http://www.gabormelli.com/RKB/Bidirectional_LSTM_(BiLSTM)_Model.

Table 5 Deep learning-based code completion studies.

Year	Venue	Ref.	Language	Model
2020	ASE	[109]	Java, TypeScript	Multi-task learning, pre-trained models
2022	ICSE	[110]	Python	Multi-task learning, GPT-2
2023	ASE	[111]	Java, Python	Retrieval-augmented language model
2023	FSE	[112]	Java, Python	GPT-2, CodeT5
2023	FSE	[113]	–	Empirical study
2023	ICSE	[114]	Java	Transformer

Table 6 Detailed information of studies within the dataset construction category.

PL	Dataset name	Ref.
Java	CONCODE	[100]
Python	Lyra	[119]
Python	APPS	[120]
Python, Java, PHP, JavaScript, Ruby, Go, C/C++	CodeXGLUE	[121]

both two datasets. Norouzi et al. [116] examined whether a generic transformer-based Seq2Seq model can achieve competitive performance with the minimal design of code-generation-specific inductive bias. They observed that it is possible for a transformer-based Seq2Seq model with minimal specific prior knowledge to achieve results that are superior to or on par with state-of-the-art models specifically tailored for code generation. Mastropaolo et al. [117] presented an empirical study to investigate whether different but semantically equivalent NL descriptions yield the same code fragments. The experimental results demonstrate that modifying the description leads to generating different code in approximately 46% of cases. Furthermore, differences in semantically equivalent descriptions can have an impact on the correctness of the generated code (±28%). Xu et al. [118] conducted a comprehensive user study on code generation and retrieval within an IDE, developing an experimental harness and framework for analysis. They noticed that developers raise concerns about the potential side effects of code generators on their workflow, encompassing aspects such as time efficiency, code correctness, and code quality.

4.12 Datasets

There are four studies that construct available benchmark datasets for the code generation task. Table 6 [100, 119–121] provides detailed information about the studies within the dataset construction category. Specifically, Iyer et al. [100] emphasized the significance of code contexts in the code generation task. To achieve accurate code fragment generation based on corresponding code contents and natural language descriptions, they developed CONCODE, a dataset including over 100000 examples comprising Java classes sourced from online code repositories. Liang et al. [119] introduced a novel code generation task: to generate a program in a base imperative language with an embedded declarative language, given a natural language comment. To support this task, they created a dataset (i.e., Lyra) consisting of 2000 carefully annotated database manipulation programs extracted from real-world projects. Each program is associated with both a Chinese comment and an English comment. To accurately assess the performance of code generation, Hendrycks et al. [120] introduced APPS, a benchmark specifically tailored for code generation in more restricted settings compared with the prior benchmarks. Their benchmark comprises 10000 problems, spanning from simple online solutions to substantial algorithmic challenges. Lu et al. [121] developed a comprehensive dataset known as CodeXGLUE. This dataset covers a diverse set of 10 tasks across 14 different datasets, encompassing eight programming languages, and it serves as a platform for evaluating and comparing models in the field of code generation.

4.13 Challenges and opportunities

In this section, we highlight some challenges and opportunities for future research in deep learning techniques for code generation.

4.13.1 *Challenges*

- **Unsafe code generation.** Deep learning techniques, especially recently proposed LLMs, are (pre-) trained on massive code bases and then applied to code generation. The code bases may include vulnerable

code snippets that lead to generation of unsafe code. Thus, how to generate functionally correct and safe code is challenging.

• **Benchmarks.** Existing benchmarks for code generation mainly include hand-written programming problems and their corresponding solutions (such as HumanEval). However, there is a huge difference between these human-written benchmarks and real projects. In addition, the human-written benchmarks are time-consuming and strongly dependent on experts' knowledge. Thus, constructing a benchmark from real projects automatically is important for code generation.

• **Hallucination of LLMs.** Recently, many LLMs have been exploited for code generation, such as Copilot. Existing studies have shown that LLMs, such as ChatGPT, often generate fabricated or inaccurate responses, which are commonly referred to as the hallucination phenomena [122]. Hallucination makes LLMs not always reliable in generating code snippets. It is crucial to combine the capabilities of ChatGPT with human expertise to ensure the quality and reliability of the generated code.

4.13.2 *Opportunities*

• **Knowledge-augmented code generation.** Existing studies have shown that recently proposed LLMs can generate code effectively. To better adapt LLMs to generate code for a specific domain, knowledge-augmented code generation is helpful. It thus is important to integrate different information, such as project information and similar code snippets, to boost existing LLMs.

• **Managing datasets as software.** Datasets (including the training data and benchmarks) are important in training and evaluating a code generation model. As more and more datasets have been proposed in recent years, we need to better manage the datasets for code generation models. Nowadays, the datasets are also evolved and hosted in collaborating platforms, such as GitHub and Hugging Face. Similar to software, we should improve dataset productivity, quality, and security.

5 Code search

Code search is the process of finding relevant code snippets from online or local code repositories based on query statements, typically expressed in natural language or code itself.

5.1 Natural language-based code search

5.1.1 *Information retrieval*

Early code search engines primarily relied on information retrieval (IR) techniques to match query keywords with code snippets. These techniques assess the relevance of queries and code based on their textual similarity [121, 123–125]. Subsequent approaches enhance code search by delving into the structural aspects of source code. They consider diverse relationships between code entities and seek matches between relevant APIs. A notable trend is the representation of code as directed graphs, effectively transforming the search task into a graph exploration problem. For instance, McMillan et al. [126] introduced Portfolio, a tool that pinpoints potential methods containing query keywords within an API call graph. The tool then ranks the results using a combination of PageRank scores to evaluate node importance and spreading activation network (SAN) scores to gauge query relevance. In a similar vein, Li et al. [127] presented RACS, a technique founded on relations. It parses natural language queries into action graphs and codes into relation invocation graphs. This enables a structural alignment between the two graph representations, thereby elevating the accuracy of matches. However, these approaches encounter limitations stemming from the pronounced disparities between programming languages and natural languages. Consequently, the comprehension of semantics remains challenging for IR-based approaches.

5.1.2 *Deep learning*

To establish more robust semantic connections between natural language queries and code, researchers have increasingly turned to deep learning models to tackle code search tasks. The fundamental approach involves encoding both the query statement and code into separate vector representations, then assessing the semantic correlation between the two through vector similarity analysis. Gu et al. [19] innovatively employed perceptrons and RNNs to embed various code elements such as method names, API call sequences, and code sequences into a shared high-dimensional space. This lays the foundation for DeepCS, a code search tool. Sachdev et al. [128] introduced neural code search (NCS), tailored for extensive

code repositories. NCS combines word embeddings with TF-IDF to generate vector representations for code snippets and query statements. It then gauges relevance through vector distances, simulating the significance of code snippets in relation to queries.

As deep learning progresses, subsequent research integrates more complex representations and more sophisticated models for code vectorization. Zou et al. [129] introduced a novel source code search technique employing graph embedding. It involves creating a code graph from a software project's source code, representing code elements using graph embedding, and then utilizing this structure to answer natural language queries by returning relevant subgraphs composed of code elements and their relationships. Gu et al. [130] proposed CRaDLe, a novel approach for code retrieval based on statement-level semantic dependency learning. CRaDLe distills code representations by merging dependency and semantic information at the statement level, ultimately learning unified vector representations for code-description pairs to model their matching relationship. Wan et al. [131] introduced MMAN, a multi-modal attention network designed for semantic source code retrieval. They created a holistic multi-modal representation by utilizing LSTM for sequential tokens, Tree-LSTM for code's AST, and GGNN for its control flow graph (CFG), followed by a multi-modal attention fusion layer that combines and assigns weights to different components for an integrated hybrid representation. Ling et al. [132] introduced an end-to-end deep graph matching and searching (DGMS) model for semantic code retrieval. They represented query texts and code snippets as unified graph-structured data, and used the DGMS model to retrieve the most relevant code snippet by capturing structural information through graph neural networks and fine-grained similarity through cross-attention-based semantic matching operations. Liu et al. [133] presented GraphSearchNet, a neural network framework that improves source code search accuracy by simultaneously learning from source code and natural language queries. They introduced bidirectional GGNN (BiGGNN) to create graphs for code and queries, capturing local structural details, and enhanced BiGGNN using a multi-head attention module to incorporate global dependencies for enhanced learning capacity. Li et al. [134] introduced CodeRetriever, which obtains function-level code semantic representations via extensive code-text contrastive pre-training. This involves unimodal contrastive learning that uses function names and documentation to build code pairs, and bimodal contrastive learning that utilizes code comments and documentation for code-text pairs, both contributing to effective pre-training using a vast code corpus. Jiang et al. [135] introduced ROSF, a technique that enhances code snippet recommendations by combining information retrieval and supervised learning. The approach involves two stages: generating a candidate set using information retrieval and then re-ranking the candidates based on probability values predicted by a trained model, resulting in improved code snippet recommendations for developers.

In recent years, significant progress has been made in the realm of large pre-trained models based on the transformer architecture, driving advancements across numerous NLP tasks. This progress leads to the emergence of code understanding pre-training models leveraging transformers, fostering the growth of code intelligence. For instance, Feng et al. [63] introduced CodeBERT, a pioneering large-scale pre-trained model that integrates natural language and programming language understanding across multiple programming languages. CodeBERT harnesses masked language modeling (MLM) to capture the semantic relationship between natural language and code. Researchers have explored the incorporation of multiple modal representations of source code into the transformer paradigm to gain a comprehensive understanding. Guo et al. [136] developed the GraphCodeBERT model, seamlessly combining the variable sequence from data flow graphs with the code token sequence. This model undergoes training via MLM, edge prediction (EP), and node alignment (NA) tasks to encompass both code structures and data dependencies. Similarly, Guo et al. [137] introduced UnixCoder, which fuses serialized ASTs with comment text sequences. By utilizing MLM, unidirectional language modeling (ULM), denoising (DNS), multi-modal contrastive learning (MCL), and cross-modal generation (CMG), this model enriches its comprehension of code syntax and semantics. Some researchers further leveraged contrastive learning to enhance model performance. Shi et al. [138] introduced CrossCS, a technique that improves code search through cross-modal contrastive learning. They devised a novel objective considering both inter- and intra-modality similarity, used data augmentation for semantic consistency, and boosted pre-trained models by ranking code snippets with weighted similarity scores based on retrieval and classification scores. Bui et al. [139] presented Corder, a self-supervised contrastive learning framework for source code models. It aims to reduce the need for labeled data in code retrieval and summarization tasks by training the model to differentiate between similar and dissimilar code snippets using contrastive learning and semantic-preserving transformations. Additionally, Shi et al. [140] introduced CoCoSoDa, which employs

contrastive learning for code search, incorporating soft data augmentation and negative samples. They also applied multimodal contrastive learning to enhance code-query pair representations.

5.1.3 *Query expansion and refinement*

Significant differences in expression and vocabulary between natural languages and code are key factors contributing to the mismatch between high-level intents implied in natural languages and low-level code implementations [19], impacting the accuracy of code search. Improving the query statement or the candidate code has been proven to be an essential approach for enhancing code search effectiveness. Bajracharya et al. [141] introduced Sourcerer, an open-source code search engine that extracts detailed structural information from code and stores it in a relational model. This information facilitates the implementation of CodeRank and supports search forms beyond traditional keyword-based searches. Lu et al. [142] introduced an approach that extends queries using synonyms from WordNet, which involves extracting natural language phrases from source code identifiers, matching expanded queries with these phrases, and sorting the search results. Lv et al. [143] introduced CodeHow, a code search technique capable of recognizing potential APIs referenced in a user query. After identifying relevant APIs, CodeHow expands the query with these APIs and performs code retrieval using the extended boolean model, incorporating both text similarity and potential APIs for improved search. Rahman et al. [144] used context-awareness and data analysis to apply appropriate term weighting in query reformulation, thereby enhancing code search. Hill et al. [145] presented a search technique based on method signature analysis, involving the rewriting of code method names and subsequent matching of the altered method names with queries to facilitate the search process. Additionally, Liu et al. [146] introduced the NQE model, which predicts keywords related to the query keywords in the corpus based on natural language queries. This technique expands query statements and subsequently improves code search effectiveness. Alongside utilizing identifiers in source code, researchers explore leveraging search logs from platforms like stack overflow to enhance code search. Cao et al. [147] analyzed large-scale search logs from stack overflow to identify patterns in query reformulation. They constructed a corpus encompassing both original queries and their reconstructed versions, and then trained a model using this corpus. The trained model can generate a list of candidate reconstructed queries when provided with a user query, offering improved search options. Li et al. [148] introduced a generation-augmented query expansion framework that utilizes code generation models to enhance code retrieval. Instead of relying solely on documentation queries, the approach involves augmenting queries with generated code snippets from the code generation model, drawing inspiration from the human retrieval process of sketching an answer before searching.

5.2 Code-to-code search

In addition to searching for code based on natural language input, code snippets are also utilized as input for code search, divided into searching within the same programming language and across different programming languages. A notable work for searching within the same language is Aroma proposed by Luan et al. [149]. Aroma takes incomplete code snippets as input and searches for similar complete code snippets from pre-indexed open-source projects. Compared to searching within the same language, cross-language code search is more challenging due to syntactic and semantic differences across languages. Mathew et al. [150] introduced the COSAL approach, which performs non-dominated sorting based on similarities between code snippets, including AST structures and input-output behaviors, to facilitate code search within the same language and across languages. Additionally, cross-language search is used for code translation, such as converting Java code into Python code with the same functionality. Perez et al. [151] employed LSTM networks to model clone similarity between cross-language code snippets based on ASTs, and Nguyen et al. [152] utilized the API2Vec model, inspired by Word2Vec, to embed APIs into high-dimensional vectors for cross-language code translation. Chen et al. [153] introduced BigPT, a technique for interactive cross-language retrieval from Big Code, involving a predictive transformation model based on auto-encoders to aid program translation using retrieved code representations. Users are able to further refine the retrieval results to improve the process.

5.3 Datasets

The following datasets in Table 7 [154,176,182,187,202,211,212] are commonly used for natural language based code search.

Table 7 Natural language-based code search datasets.

Dataset	Language	Size	Source	Release year
StaQC [130]	Python, SQL	267k	SO	2018
CoNaLa [132]	Python, Java	2.8k	SO	2018
FB-Java [138]	Java	287	SO, GitHub	2019
CodeSearchNet [139]	Python, Java, Ruby, Go, PHP, JavaScript	2M	GitHub	2019
SO-DS [140]	Python	2.2k	SO, GitHub	2020
CosBench [148]	Java	52	SO, GitHub	2020
CodeXGLUE [121]	Python	281k	Bing, GiHub	2020
CoSQA [154]	Python	20k	Bing, GitHub	2021
xCodeEval [155]	C#, C++, C, D, Go, Haskell, Java, Javascript, Kotlin, Ocaml, Pascal, Perl, PHP, Python, Ruby, Rust, Scala	11k	Codeforces	2023

The StaQC dataset [130] is tailored for predicting the suitability of code snippets in addressing specific queries. Comprising (question, code) pairs, it was curated by filtering Python and SQL Stack Overflow posts tagged with "how-to" questions, resulting in 147546 Python pairs and 119519 SQL pairs.

The CoNaLa dataset [132] consists of 2379 training and 500 test examples, manually annotated with natural language intents and corresponding Python snippets.

The FB-Java dataset [138] comprises 287 natural language queries and relevant code snippet answers from Stack Overflow threads tagged with "java" or "android". Additionally, it includes code snippet examples from the search corpus, sourced from public repositories on GitHub, that correctly answer the corresponding queries.

The CodeSearchNet corpus [139] is an extensive collection of approximately 6 million functions automatically gathered from open-source code spanning six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). It includes 2 million functions with query-like natural language descriptions obtained via scraping and preprocessing associated function documentation. Furthermore, it contains 99 natural language queries with around 4000 expert relevance annotations of likely results from the CodeSearchNet Corpus.

The SO-DS corpus [140] consists of code snippets mined from Stack Overflow posts with the most upvoted posts labeled with "python" and tags related to data science libraries such as "tensorflow," "matplotlib," and "beautifulsoup." The ground truth is collected by creating queries from duplicate Stack Overflow posts, resulting in 2225 annotated queries.

The CosBench corpus [148] comprises 475783 Java files and 4199769 code snippets (Java methods) extracted from the top-1000 popular Java projects on GitHub. It includes 52 queries with ground truth code sbippets indicating three types of intentions: bug resolution, code reuse, and API learning, chosen from Stack Overflow.

The CodeXGLUE [121] serves as a benchmark dataset and an open challenge for code intelligence, encompassing various code intelligence tasks. For NL-based code search, it includes two sub-datasets: AdvTest and WebQueryTest. AdvTest, constructed from the CodeSearchNet corpus [139], uses the first paragraph of documentation as a query for the corresponding function. WebQueryTest is a testing set of Python code questions answered with 1046 query-code pairs and expert annotations.

The CoSQA dataset [154] is based on real user queries collected from Microsoft's Bing search engine logs. It encompasses 20604 labels for pairs of natural language queries and code snippets, each annotated by at least 3 human annotators.

The xCodeEval [155] is recognized as one of the most extensive executable multilingual multitask benchmarks, encompassing seven code-related tasks that span across 17 programming languages. Derived from a pool of 25 million openly available samples from codeforces.com, a platform hosting competitive programming contests, this dataset comprises 7514 distinct problems. In terms of code retrieval, xCodeEval introduces a novel and more demanding task, specifically centered on matching a natural language problem description to the most relevant and accurate code within a candidate pool containing similar solutions. To facilitate this, all submitted code snippets and their associated test cases are aggregated for each programming language, creating a retrieval corpus and a suite of test cases. The primary objective is to evaluate the correctness of these code snippets against the provided test cases. In this context, the natural language problems serve as queries, and the correct solutions, verified by successful execution outcomes (PASSED), are considered the ground truth.

For code-to-code search, existing datasets designed for code clone detection, featuring clusters of semantically equivalent implementations, can be utilized. One such dataset is the BigCloneBench provided within CodeXGLUE [121]. In these datasets, a group typically comprises variations of the same implementation. In practice, one implementation within a group can serve as a query, while the remaining implementations within the cluster act as the ground truth. Additionally, xCodeEval [155] offers a dataset specifically tailored for code-to-code tasks. This dataset consists of 9508 queries, created from correct submitted solutions to the same natural language problems, adding diversity to the evaluation of code-to-code search capabilities.

5.4 Challenges and opportunities

5.4.1 *Challenges*

• Quality assurance of search results. Ensuring the quality of code search results goes beyond merely matching search intent. Factors such as correctness, security, and timeliness must be considered to guarantee the reliability and suitability of the returned code snippets.

• Long tail issues. Addressing the challenges posed by less common, long-tail issues is essential. Code search systems need to effectively handle diverse and infrequent queries, ensuring comprehensive coverage across a spectrum of coding scenarios.

• Result interpretability. Achieving interpretability in code search results is also a challenge. Only little research effort (e.g., [156]) in this direction. It involves presenting search outcomes in a clear and understandable manner, aiding developers in comprehending the context and relevance of the returned code snippets.

• Integrating retrieved code into development context. Effectively utilizing code search results poses a significant challenge. Typically, developers expend considerable effort in adapting and integrating a retrieved code snippet into the current context of their code development.

• Ambiguity in search intents. The inherent ambiguity in certain search intents poses a challenge. Code search systems need to navigate and interpret vague or imprecise queries to provide relevant and accurate results.

• Dataset quality issues. Existing datasets used for training and evaluating code search models, such as CodeSearchNet [139], often consider method comments as search queries, which fails to align well with real-world code search scenarios [157]. While this can lead to strong model performance during evaluation, it may not be effectively transferred into practical use.

• Efficiency vs. effectiveness. Learning-based code search has shown promising search accuracy, but it often demands substantial computing and storage resources. Although certain approaches [158, 159] have been proposed to address efficiency concerns in learning-based code search, there is still significant room for improving the trade-off between efficiency and effectiveness.

5.4.2 *Opportunities*

• Incorporating richer input information. Expanding the scope of input information beyond the query itself presents an opportunity. Incorporating details from IDE development contexts, historical patterns, and personal preferences can enhance the accuracy and relevance of code search results.

• Enhanced code search with LLMs. Leveraging advancements in LLMs offers an opportunity to augment code search capabilities. Focusing on result interpretability, code auto-adaptation, and harnessing the advanced natural language and code semantic understanding abilities of LLMs can elevate the efficiency of code search systems.

• Improved intent clarification techniques. Advancing techniques for intent clarification in code search represents an opportunity. Developing techniques to better understand and refine user search queries contributes to a more streamlined and effective code search experience.

• High-quality dataset construction. This presents an opportunity for the community to enhance the quality of current datasets [157] or create more reliable ones for code search evaluation.

6 Code summarization

Code summarization, also known as code comment generation, is a process that aims to enhance the understanding and documentation of source code by automatically generating concise and informative

summaries for software artifacts [63, 160–162]. It helps address the challenge of comprehending large and complex code repositories by providing developers with high-level descriptions that capture the code's essential functionality and usage patterns.

Code summarization has been a hot research topic in software engineering in recent years. Initially, researchers explore template-based methods like SWUM [163] and Stereotypes [164] for generating code comments automatically. Meanwhile, information retrieval-based techniques such as VSM [165] and LSI [166] are also applied for code summarization. However, with the rapid advancements in deep neural networks within machine learning, deep learning-based approaches have gained momentum and become predominant in code summarization research.

Typically, researchers leverage deep learning models to capture implicit relationships between relevant information within source code and natural language descriptions. These approaches have significantly contributed to the development of code summarization, facilitating more effective comprehension and documentation of software products.

Deep learning-based approaches for comment generation primarily mimic the neural machine translation (NMT) [167] models in natural language processing. However, compared to translation tasks in natural language, source code typically has a much greater length than comments and contains rich structural information. Most deep learning-based research takes the source code token sequence as the input to the model, while some studies also consider other information sources, such as ASTs and API. We divide these studies into five categories based on different sources of information: techniques that utilize source code sequences as the model input, techniques that employ AST sequences as the model input, techniques that use tree structures as the model input, techniques that utilize graph structures as the model input, and techniques that consider other sources of information. These techniques generate comments for code snippets (class level, function level or line level), as well as comments for code commits (i.e., commit messages).

6.1 Using source code sequences as model input

A source code sequence refers to a simple stitching of a code snippet into a sequence with a token as a basic unit. Using source code sequences as model input is simple and convenient and could preserve the most original semantic information of the code.

Iyer et al. [17] proposed CODE-NN, the first deep learning model in code summarization, which uses LSTM network structures and attention mechanisms to generate natural language descriptions of C# code and SQL code. Allamanis et al. [168] applied CNNs with attention mechanisms to an encoder that helps detect long-range topical attention features and local time-invariant features of code sequences. Ahmad et al. [169] first used the transformer model for source code summarization, innovatively adding an attention layer to the encoder for replicating rare tokens of the source code. Wang et al. [170] proposed Fret, which combines transformer and BERT to bridge the gap between source code and natural language descriptions and alleviate the problem of long dependencies. Zhang et al. [171] tried to fuse two techniques: deep learning and information retrieval. Specifically, they proposed Rencos, which first trains an encoder-decoder model based on a training corpus. Subsequently, two code segments are selected from the training corpus according to the syntax and semantic similarity. Finally, the input code segment and the retrieved two similar code segments are encoded and decoded to generate comments. LeClair et al. [172] explored the orthogonality of different code summarization techniques and proposed an integration model to exploit this for better overall performance. Gong et al. [173] proposed SCRIPT, which first obtains the structural relative position matrix between tokens by parsing the AST of the source code, and then encodes this matrix during the computation of the self-attention score after the source code sequence is input into the encoder.

Some research considers both comment generation and code generation tasks. Chen et al. [174] focused on both code retrieval and comment generation tasks, and they proposed a framework, BVAE, which allows a bidirectional mapping between code and natural language descriptions. The approach attempts to construct two VAEs (variational autoencoders), where C-VAE mainly models code and L-VAE primarily models the natural language descriptions in comments. The technique jointly trains these two VAEs to learn the semantic vectors of code and natural language representation. Similarly, Wei et al. [94] considered code summarization and code generation as dyadic tasks, as there is a correlation between the two tasks. They proposed a dual framework to train both tasks simultaneously. They exploited the pairwise nature and the duality between probability and attention weights. Then they designed

corresponding regularization terms to constrain this duality. Clement et al. [88] focused on the Python language and proposed PYMT-5. They also focused on dual tasks: code generation from signatures and documentation generation from method code.

Some researchers focused on comment generation for commits. Jiang et al. [175, 176] used NMT to generate concise summaries of commits while designing a filter to ensure that the model is trained only on higher-quality commit messages. Jiang [177] preprocessed code changes into more concise inputs, explicitly using a code semantic analysis approach for the dataset, which can significantly improve the performance of the NMT algorithm. Liu et al. [178] used a modified sequence-to-sequence model to automatically generate pull request (PR) descriptions based on submission information and source code comments added in PRs. Bansal et al. [179] proposed a project-level encoder to generate vector representations of selected code snippets in software projects to improve existing code summarization models. Xie et al. [180] considered method names as refined versions of code summaries. Their approach first uses the prediction of method names as an auxiliary training task and then feeds the generated and manually written method names into the encoder separately. Finally, the outputs are fused into the decoder.

6.2 Using AST sequences as model input

An AST is a hierarchical representation of the syntactic structure of a program or code snippet. An AST represents the structure of the code by breaking it down into its constituent parts and organizing them in a tree-like format. ASTs are commonly used in computer science and programming language theory to analyze and manipulate code. Each node in an AST corresponds to a syntactic element of the code, such as a statement, expression, or declaration.

Hu et al. [181] proposed Deepcom, a technique to preserve the structural information of the code intact by parsing the source code into an AST. The authors designed a new traversal strategy, structure-based traversal (SBT), which solves the problem that the source code cannot be effectively restored from the AST sequence. Subsequently, they proposed Hybrid-DeepCom [182] based on DeepCom, which mainly improves DeepCom in three aspects. First, it uses a combination of code information and AST sequence information. Second, the OOV problem is mitigated by subdividing the identifier into multiple words based on the camel naming convention. Finally, Hybrid-DeepCom uses beam search to generate code comments. Huang et al. [183] proposed a statement-level AST traversal approach that preserves both code text information and AST structure information, and achieves good results in code snippet-oriented comment generation tasks. Tang et al. [184] proposed AST-Trans, which exploits two node relationships in ASTs: ancestor-descendant and sibling relationships. The authors applied the attention of a tree structure to assign weights to related nodes dynamically. Liu et al. [185] proposed ATOM, which explicitly incorporates an AST of code changes and utilizes a hybrid sorting module to prioritize the most accurately retrieved and generated messages based on a single code change.

Some approaches receive both AST and source code sequences as input. Wan et al. [186] combined an LSTM that receives code sequences and an LSTM that receives ASTs to extract a hybrid vector representation (named Hybrid-DRL) of the target code synthetically. It further uses a reinforcement learning framework (i.e., actor-critic network) to obtain better performance. LeClair et al. [187] proposed ast-attend gru, which also considers two representations of the code: a word-based text sequence and an AST-based tree structure. It processes each data source as a separate input and later merges the vectors produced by the attention layer. Xu et al. [188] proposed CoDiSum to extract AST structures and code semantics from source code changes and then jointly model these two sources of information to learn the representation of code changes better. Zhou et al. [189] designed a new semantic parser, SeCNN, using two CNN components that receive source code and AST, respectively, and proposed a new AST traversal technique ISBT to encode structural information more sufficiently. Specifically, they used the serial number of the AST via pre-order traversal to replace the brackets in the SBT sequence. Gao et al. [190] proposed M2TS, which uses cross-modal fusion further to combine AST features with the missing semantic information and highlight the key features of each module. Zhou et al. [189] proposed GSCS, which uses a graphical attention network to process AST sequences and a multi-head attention mechanism to learn features of nodes in different representation subspaces.

6.3 Using tree structure as model input

Unlike the approaches discussed in Subsections 6.1 and 6.2, which transforms the AST into a sequence, approaches discussed in this subsection retain the tree structure of the AST directly as input.

Liang et al. [190] proposed Code-RNN based on tree-LSTM, which is for the case where a node has multiple children, thus overcoming the restriction of converting ASTs into binary trees. For decoding, code-GRU is used. Wang et al. [191] built a tree structure based on code indentation, where the nodes of the tree are statements in the code, and statements with the same indentation are sibling nodes. They then fed this tree structure into a tree-transformer-based encoder. Lin et al. [192] partitioned the AST into several subtrees according to the control flow graph of the method, and then fed the subtrees into a tree LSTM for pre-training to obtain their vector representations. They used these representations in the subsequent comment generation task. Similarly, Shi et al. [193] used user-defined rules to split the AST tree hierarchically. The model learns the representation of each subtree using a tree-based neural model, i.e., RvNN. The difference is that RvNN finally combines the representations of all subtrees by reconstructing the split AST to capture the structural and semantic information of the whole tree.

6.4 Using graph structure as model input

A graph is a versatile and powerful data structure that captures complex relationships and interconnections among entities. Some approaches treat the source tokens as graph vertices and represent the relationships between tokens by edges.

Fernandes et al. [194] added graph information to sequence encoding. Source code is modeled as a graph structure, which helps infer long-distance relationships in weakly structured data (e.g., text). LeClair et al. [195] used a graph neural network (GNN)-based encoder to model the graph form of an AST and an RNN-based encoder to model the code sequences. Liu et al. [196] constructed a code property graph (CPG) based on an AST while augmenting it with CPGs of ASTs of the retrieved similar code snippets. Then, the CPGs are input into a graph neural network for training. Liu et al. [197] proposed a graph convolutional neural network (GCN) based on a hierarchical attention mechanism for encoding graphs parsed from code sequences. The code encoded by the sequence encoder is further combined with the document text information for decoding. Cheng et al. [198] designed three encoders that receive source code sequences, code structure information, and code context information. A bipartite graph is used to represent the structure information evolved from the AST, with the addition of a keyword guidance module.

Guo et al. [199] proposed CODESCRIBE, which models a code snippet's hierarchical syntactic structure (i.e., AST) by introducing new triadic positions. Then, they used a graph neural network and transformer to preserve the structural and semantic information of the code, respectively. Ma et al. [200] proposed MMF3, which uses a graph convolutional network to encode AST graph embeddings while fusing the sequence of source code features to determine the matching relationship between each token in the code sequence and each leaf node in the AST by comparing the position order. Wang et al. [201] proposed GypSum to introduce specific edges associated with the control flow of code snippets into the AST for building graphs, and designed two encoders for learning from the graph and source code sequences.

6.5 Considering other sources of information

Other sources of information include APIs, control flow graphs, unified modeling languages, and bytecode. Hu et al. [202] argued that APIs called within code may provide certain information, and they proposed TL-CodeSum, which first trains the mapping relationship between APIs and code comments and subsequently migrates the learned knowledge to the code summarization task. Shahbazi et al. [203] generated comments using API documentation, code snippets, and abstract syntax trees. They showed that API documentation is an external knowledge source, and the performance improvement is negligible. Gao et al. [204] proposed GT-SimNet, a code semantic modeling approach based on local application programming interface (API) dependency graphs (local ADG). This approach is accomplished by computing the correlation coefficients between dependency graphs and AST nodes.

Zhou et al. [205] proposed ContextCC to obtain ASTs by parsing code to find methods and their associated dependencies (i.e., contextual information) and then generate code comments by combining the filtered contextual information. Wang et al. [206] constructed a type-augmented abstract syntax tree (Type-augmented AST) and extracted CFGs as an alternative syntax-level representation of the code, with a hierarchical attention network to encode this data. Wang et al. [207] introduced class names and associated unified modeling languages (UMLs) for method comment generation, where the UMLs are fed into the graph neural network as graph forms. Son et al. [208] found that program dependency graphs (PDGs) can represent the structure of code snippets more effectively than ASTs, proposed an

Table 8 Datasets for code summarization.

Dataset	Ref.	Language	Size
TL-CodeSum	[202]	Java	87136
Deepcom	[182]	Java	588108
Funcom	[187]	Java	2.1 million
CodeSearchNet	[154]	Go, Java, JavaScript, PHP, Python, and Ruby	2 million
code-docstring-corpus	[211]	Python	150370
SCGen	[212]	Java	600243
commitMessage	[176]	Java	2027734

enhancement module (PBM) that encodes PDGs as graph embeddings, and designed a framework for implementing PBMs with existing models. Zhang et al. [209] proposed Re_Trans to enhance structural information by adding data flow and control flow edges to the AST and using GCN to encode the entire AST.

Huang et al. [210] explored the feasibility of using bytecode as a source of information to generate code comments. They used pre-order traversal to serialize the bytecode control flow graph, and combined it with a bytecode token sequence as model input to achieve automatic code summarization in a scenario without available source code.

6.6 Datasets

The following datasets in Table 8 [154, 176, 182, 187, 202, 211, 212] are commonly used for automatic code summarization.

TL-CodeSum [202] comprises 69708 method-comment pairs obtained by crawling Java projects developed between 2015 and 2016, each having a minimum of 20 stars on GitHub. The average lengths of Java methods, API sequences, and comments are 99.94, 4.39, and 8.86, respectively.

Deepcom [182] is collected from GitHub's Java repositories created from 2015 to 2016 considering only those having more than 10 stars to filter out low quality repositories. It uses the first sentences of the Javadoc as the target comments and excludes the setter, getter, constructor, and test methods. After the preprocessing, there are 588108 method-comment pairs in total.

Funcom [187] constitutes a compilation of 2.1 million method-summary pairs derived from the Sourcerer repository. After the removal of auto-generated code and exact duplicates, the dataset is partitioned into training, validation, and test sets by project.

CodeSearchNet [154] is a large well-formatted dataset collected from open source libraries hosted on GitHub. It contains 2 milllion code-summary pairs and about another 4 million functions without an associated documentation, spanning six programming languages (i.e., Go, Java, JavaScript, PHP, Python, and Ruby).

code-docstring-corpus [211] is a dataset downloaded from repositories on GitHub, retaining Python 2.7 code. The dataset contains 150370 code-comment pairs. The vocabulary size of code and comment is 50400 and 31350, respectively.

SCGen [212] is a dataset of Java code snippets constructed from 959 Java projects of GitHub. Data cleaning is performed to filter out invalid data according to templates, such as comments in setter and getter methods or comments generated by the template predefined in the IDE comment plugin. Using a comment scope detection approach, 600243 code snippet-comment pairs are collected.

commitMessage [176] contains 967 commits from the existing work and all the commits from the top 1000 popular Java projects in Github. The rollback commits, merge commits, and the commits with messages that are empty or have non-English letters are filtered. In the end, there are 2027734 commits in the dataset.

6.7 Challenges and opportunities

6.7.1 *Challenges*

• High-quality dataset. The code summarization techniques based on deep learning need a high-quality dataset to improve their performance. Although several datasets have been published in this area, the data is selected from the perspective of the project popularity. As a result, there may be duplicate data

and machine-generated comments in the dataset. Developing practical techniques to identify high-quality comments that really reflect the code intent is challenging.

• Evaluation metrics. Many studies employ the BLEU metric to evaluate the performance of the code summarization models. However, there are many variations of BLEU, which results in different ways of calculation, such as BLEU-L and BLEU-C. On one hand, due to the different emphasis of each variation of BLEU, the performance of the same model shows significant differences under different BLEU metrics. On the other hand, it is possible that the BLEU score does not accurately reflect the actual effect of the generated comment because the BLEU score is based on the repetition of tokes in two sentences. Two sentences with the same semantics but different words have a low BLEU score, which is unreasonable.

• Adaptation ability. Code summarization needs to be adaptable to various programming languages, each with its own syntax and semantics. Developing a universal summarization model that performs well across diverse languages is a significant challenge.

6.7.2 *Opportunities*

At present, most code summarization models cannot be directly applied to the production practice, and are still in the experimental prototype. It comes down to the fact that the model is not powerful enough to apply. There is still a lot of room for improvement.

• Utilizing more implementation information. When the information at the source code level (e.g., ASTs, tokens, APIs, CFGs) is almost mined, a feasible way may be to mine more useful information from outside the source code (e.g., bytecode, API documents, design documents) to characterize the internal patterns of the code, and further improve the performance of code comment generation models.

• Considering richer information in the comments. Most code summarization datasets take the first sentences of a comment or commit message as the code summary because the first sentences are considered to describe the functionalities of Java methods according to Javadoc guidance. With the development of deep learning models, other information in the summary (e.g., the intent or rationale of the code) may be extracted and used for summary generation.

7 Software refactoring

Software refactoring is to improve software quality by changing its internal structures whereas its external behaviors are kept intact [213]. Ever since Opdyke proposed the concept of software refactoring in 1992, researchers have tried to automate software refactorings aiming to reduce the cost of software refactoring and to improve the safety of refactorings. Thanks to such hard work, automatic or semi-automatic software refactoring has been provided as a default feature in all mainstream IDEs, such as Eclipse, IntelliJ IDEA, and Visual Studio, thus significantly increasing the popularity of software refactorings.

Various techniques have been exploited for software refactorings [214–217] whereas recently deep learning techniques have become the main force in this field [218, 219]. Traditional software refactoring heavily depends on static code analysis, code metrics, and expert-defined heuristics to identify code smells [214, 215] (i.e., what should be refactored) and to recommend refactoring opportunities. However, it is challenging to formalize complex refactorings with human-defined simple heuristics, making traditional heuristics-based refactoring less accurate. In contrast, deep learning techniques with complex networks and numerous weights, have the potential to learn complex refactorings [216]. Consequently, various deep learning techniques have been recently employed for software refactoring.

However, applying deep learning techniques to software refactorings is nontrivial, encountering a sequence of challenges. The first challenge is to collect a large number of high-quality items requested for training. Since deep models often contain a large number of parameters, they usually request a large number of labeled items as training data. However, we lack such large-scale high-quality datasets in the field of software refactoring. The second challenge is to figure out how deep models (deep learning techniques) could be adapted for different categories of software refactorings. There are various deep learning techniques (e.g., CNN, LSTM, and GNN), which are originally designed for tasks (e.g., natural language processing or image processing) other than software refactoring. Consequently, such techniques should be substantially adapted for this specific task.

7.1 Detection of code smells

Code smell detection is often taken as the first step in software refactoring because code smells often indicate the problems of source code as well as their solutions (i.e., refactorings). Traditional approaches usually distinguish software entities associated with code smells from smell-free code by code metrics, taking the task of code smell detection as a binary classification problem. Since deep learning techniques have proven effective in classification tasks [220, 221], it is reasonable to investigate deep learning-based detection of code smells.

To the best of our knowledge, the automated approach to detecting feature-envy smells proposed by Liu et al. [216] is the first attempt at deep learning-based code smell detection. They exploited traditional code metrics, e.g., coupling between code entities, by a CNN, and exploited the identifiers of code entities (i.e., names of the to-be-tested method and names of its enclosing class as well as its potential target class) by another CNN. The outputs of the two CNNs are fed into a dense layer (fully-connected network, FCN) and its output generates how likely the method should be moved from its enclosing class to the given target class. Their evaluation results suggest that trained with automatically generated data, the deep learning-based approach is more accurate than traditional approaches that do not leverage deep learning techniques. Based on the success, Ref. [219] expanded the deep learning-based detection to additional categories of code smells (i.e., long methods, large classes, and misplaced classes), and their evaluation results suggest that deep learning techniques have the potential to improve the state of the art in code smell detection.

Barbez et al. [222] applied CNN to capture the evolution of God classes. For a class to be tested, the proposed approach (called CAME) extracts the latest n versions of this class, and for each version, it extracts the selected code metrics (e.g., the complexity of the class). As a result, it expresses the evolution of the class as a matrix $X_{n,m}$ where n is the number of versions and m is the number of involved code metrics. This matrix is fed into a CNN whose output is forwarded to an MLP (multilayer perceptron). The MLP will make the final prediction, i.e., whether the class under test is a God class. Notably, although this approach depends on deep neural networks, it leverages only 71 real-world God classes for training and testing.

Yu et al. [223] employed a GNN to detect feature-envy smells. They represented methods as nodes in the graph and the calling relationships among methods as edges. They leveraged a GNN technique to extract features and vectorize the nodes, and finally classified the nodes (methods) as smelly methods or smell-free ones. Their evaluation results suggest that their GNN-based approach is more accurate than the CNN and FCN-based approach proposed by Liu et al. [219] for detecting feature-envy smells.

Kurbatova et al. [217] proposed a hybrid approach to identifying feature-envy smells by leveraging both deep learning techniques and traditional machine learning techniques (i.e., SVM). The approach first represents methods and classes as vectors with Code2Vec [224]. Code2Vec parses a method into an AST, and represents each path between two AST leaves as a vector. Based on the resulting vectors that represent patches within the AST, Code2Vec represents the whole method as a vector. A class is presented as a vector that equals the average of the vectors of methods within this class. The vector of the to-be-tested method and the vector of its potential target class are fed into an SVM-based binary classifier to predict whether the method should be moved to the given target class. Note that, the deep learning model employed by this approach (i.e., Code2Vec) is unsupervised. Consequently, it does not request a large number of labeled training data, which is a significant advance of the hybrid approach. Similarly, Cui et al. [225] also leveraged Code2Vec (or Code2Seq) to turn a method (i.e., AST) into a vector, and employed graph embedding techniques to represent its dependency with other methods. With the resulting embeddings, they also leveraged a traditional machine learning technique (Naive Bayes) to make predictions.

Code smell detection, if taken as a binary classification, often encounters serious class imbalances because software entities associated with code smells (noted as positive items) are often significantly fewer than smell-free entities (noted as negative items). Fuzzy sampling, proposed by Yedida and Menzies [226], is a novel technique to handle class imbalance. It adds points concentrically outwards from points (items) of the less popular class. The oversampling thus may push the decision boundary away from these points if the newly added points belong to the same class. As a result, the classifier trained with additional items may learn better to identify similar items belonging to the less popular class. Yedida and Menzies [227] validated whether this novel technique can boost deep learning-based code smell detection. Their results demonstrate that fuzzy sampling boosts the deep learning-based approaches (proposed by Liu et al. [219])

to detect feature envy, long methods, large class, and misplaced classes by fuzzy sampling. That is to say, the results suggest that fuzzy sampling does improve the state of the art in deep learning-based code smell detection.

7.2 Recommendation of refactoring opportunities

Although we may suggest where and which refactorings should be applied by automated detection of code smells as introduced in the preceding section, we may also need to recommend refactoring opportunities (and even detailed refactoring solutions) directly without code smell detection. For example, Liu et al. [228] proposed an automated approach to recommending renaming opportunities based on renaming refactorings that developers recently conducted. In this approach, they do not detect any specific code smells but recommend refactorings similar to what has been conducted recently. The approach proposed by Liang et al. [229] is similar to the approach by Liu et al. [228] in that both approaches recommend renaming opportunities according to the evolution history of the source code. The key difference is that Liang et al. [229] employed deep learning techniques whereas Liu et al. [228] depended on heuristics and static source code analysis. Notably, Liang et al.'s approach recommends renaming opportunities on methods only. For a given method, it leverages BERT [230] and textCNN [231] to vectorize the method. After that, it employs an MLP classifier to predict whether the method should be renamed. The predicted method is renamed only if at least one of its closely related entities has been renamed recently. Liu et al. [232] employed unsupervised deep learning techniques to identify and recommend renaming opportunities. First, for each method in the given corpus, it vectorizes method names and method bodies by CNN and paragraph vector [233], respectively. For a given method whose method name is mn, and whose method body is md, it retrieves the top k most similar method names from the corpus and top k method names whose corresponding method bodies are the most similar to md. If the two sets of method names are highly similar, method name mn is consistent with the corresponding method body md. Otherwise, they are inconsistent, and thus it selects a method name from the latter set with a set of heuristics and recommends replacing mn with it.

Since useful refactorings should be frequently applied by various developers on various software applications, it is likely that we may infer such refactorings from the rich evolution histories of software applications without knowing exactly what the refactorings are in advance. Consequently, by applying advanced learning or mining techniques to evolution histories, we may learn (discover) some less-known refactorings, and can even learn where and how such refactorings could be applied. For example, Tufano et al. [234] proposed a deep learning-based technique to learn from pull requests and to infer how code is changed. Among the most frequent changes learned by this approach, refactorings are dominating. After training the deep neural model with various pull requests, the technique applies the resulting model to predict expected changes in a given application. Most of the predicted changes are refactorings, and thus the prediction could be viewed as an automated recommendation of refactorings.

Nyamawe et al. [235] suggested that refactoring activities, as well as other software development activities, should be traced to software requirements as well as their changes. Based on this assumption, they proposed a novel technique to recommend refactorings based on feature requests. It first associates feature requests with code smells and refactorings by mining software evolution histories. For a new feature request, it employs various machine learning-based models to predict the required refactorings based on the feature request as well as code smells associated with related source code.

AlOmar et al. [236] proposed the first just-in-time recommendation approach for extract-method refactorings based on copy-and-paste actions. When developers copy and paste a piece of source code, the approach determines whether the copied fragment of source code should be extracted as a new method. It leverages a large number of code metrics and uses a CNN to classify code fragments based on the code metrics.

Chi et al. [237] proposed a novel and more reliable approach called ValExtractor to conduct extract-local-variable refactorings. The primary challenge in automating extract-local-variables refactorings is the efficient identification of side effects and potential exceptions between extracted expressions and their contexts without resorting to time-consuming dynamic program execution. ValExtractor addresses this challenge by utilizing lightweight static source code analysis to validate the side effects of the selected expressions. It also identifies occurrences of the selected expression that can be extracted together without introducing program semantics or potential exceptions.

Besides the generic approaches that could be applied to various software applications, deep learning-based refactoring recommendations have also been proposed for some special domains, e.g., microservices. Desai et al. [238] proposed a deep learning-based approach to recommending refactoring opportunities, i.e., extracting some classes from a monolith application as micro-services. The approach represents classes as nodes and invocation among them as edges. It also identifies entry points (i.e., APIs of web applications) and represents such information as attributes of the classes. It then employs a graph neural network to cluster the nodes (i.e., classes), aiming to minimize the effect of outlier nodes. The resulting outlier nodes are finally recommended to be extracted as micro-services.

We conclude that both supervised and unsupervised deep learning techniques have been applied to recommending generic and domain-specific refactoring opportunities. However, we also notice that existing approaches support only a limited number of refactoring categories, and thus it is potentially fruitful to recommend more categories of refactoring opportunities by deep learning techniques in the future. We also notice that some latest advances in deep learning techniques, like large language models (e.g., GPT) have not yet been fully exploited in automated software refactoring approaches.

7.3 Datasets

Lacking of large-scale and high-quality training datasets is one of the biggest obstacles to deep learning-based software refactoring. Notably, most existing datasets for software refactorings are built manually [239] or built by mining refactoring histories [240]. For example, to validate the automated move-method refactorings, JMove [215] requested developers to manually check the suggested refactoring opportunities, and thus such manually confirmed items could serve as training or testing datasets for future research in this line. However, such a manually constructed dataset is often too small for sufficient training of deep neural networks. It is also challenging to enlarge such datasets because the manual identification of refactoring opportunities is time-consuming and error-prone. Another way to construct datasets of software refactorings is to leverage automated refactoring miners to discover actually conducted refactorings recorded in open-accessed version control systems. A few approaches, e.g., RefactoringMiner [241], RefDiff [242], and Ref-Finder [243], have been proposed for such purpose. Although such mining tools could identify a large number of refactorings from real-world applications, they often result in false positives, making the resulting dataset unsuitable for model training.

To this end, Liu et al. [216] proposed a novel approach to generating large-scale training data for the training of deep learning-based refactoring models. In the paper, the authors focused on a single category of code smells (i.e., feature envy) and its corresponding refactoring (i.e., move-method refactoring). To create positive items (i.e., methods associated with feature envy smells), they randomly moved methods across classes (with precondition checking of Eclipse move method refactoring), and took the moved methods as positive items because they had better be moved back to the original place (i.e., their enclosing classes before the movement). Methods that could be moved (i.e., satisfying the precondition of move method refactorings) but have not been moved could be taken as negative items, i.e., methods not associated with feature envy smells. By applying this novel approach to high-quality open-source applications, they generated huge data sets of refactorings (code smells) where the ratio of positive (negative) items could be accurately controlled as well. Based on the generated data, they trained a CNN-based deep neural model to detect feature envy smells and to suggest solutions (i.e., where the associated methods should be moved). Their evaluation results suggest that the resulting model significantly outperforms the state-of-the-art approaches. Later, Liu et al. [219] successively expanded this approach to more categories of code smells, i.e., long methods, large classes, and misplaced classes. Long methods are created by automated inline refactorings that merge multiple methods into a single one, and large classes are created by merging multiple classes whereas misplaced classes are created by moving classes across packages. Their evaluation results suggest that employing such automatically generated large datasets to train deep neural networks could significantly improve the state of the art in code smell detection and automated recommendation of refactoring opportunities [219]. Currently, this automated data generation has been employed by almost all deep learning-based refactoring approaches that request labeled training items [227, 244].

Although the quantity of automatically generated training items is satisfying, their quality may still be questionable. Because the code smells (i.e., positive items) are automatically generated, they could be significantly different from code smells introduced unconsciously by developers. As a result, deep neural models trained with such generated artificial data may learn only how to identify artificial smells instead

of real-world code smells. To this end, Liu et al. [240] aimed to improve the precision of refactoring miners, and thus their discovered real-world code smells and refactorings could be taken directly as high-quality training data. The key to their approach is to leverage a sequence of heuristics and machine learning-based classifiers to exclude false positives. Notably, they employed a traditional machine learning technique (i.e., decision trees) instead of deep learning techniques to exclude false positives. The major reason for the selection is that traditional machine learning techniques may work well with small (but high-quality) training data whereas deep learning ones often request much larger datasets that they were unable to provide. Their evaluation results suggest that by filtering out false positives with their approach, the precision of the employed refactoring miner (RefactoringMiner [241]) is able to reach a high level comparable to human experts in discovering move-method refactorings. Compared to the artificial feature-envy methods automatically generated by previous approach [216], such real-world feature-envy methods discovered by the proposed approach could significantly improve the performance of the deep learning-based model in detecting feature-envy smells and in recommending move-method opportunities.

Most of the current refactoring detection approaches often result in non-negligible false positives and false negatives. To solve this problem, Liu et al. [245] proposed a novel refactoring detection approach (called ReExtractor). The rationale of ReExtractor is that an entity matching algorithm takes full advantage of the qualified names, the implementations, and the references of software entities. Compared to the state of the art, it improves the accuracy of entity matching between two successive versions and thus substantially reduces false positives and false negatives in refactoring detection.

Based on the preceding analysis, we conclude that large-scale and high-quality training data are critical for deep learning-based refactoring, and data collection remains an open question that deserves further investigation.

7.4 Challenges and opportunities

Based on the preceding analysis of deep learning-based software refactoring, we present here a list of potential challenges and opportunities for future research in deep learning-based software refactorings.

7.4.1 *Challenges*

• Large-scale high-quality dataset. It remains challenging to collect large-scale and high-quality refactoring data to train deep learning models. Although generating refactorings to be automatically reversed as suggested by Liu et al. [216] should result in large-scale refactoring data, the quality and representativeness are in question. In contrast, discovering refactoring histories in open-source applications may result in high-quality real-world refactoring data, such data are often small and lack diversity.

• Generalization across different paradigms. Most of the code bases are written in various programming languages, each with its own syntax and semantics. Developing deep learning models that generalize well across different languages is a considerable challenge. Currently, most of the studies in deep learning-based refactoring use applications written in Java. Thus, there is little or no slow adoption in other programming languages. Another challenge concerning the generalization is to make the approaches applicable to all categories of code smells. The identification of code smells is very crucial in the process of software refactoring. It has been proven challenging to have a general deep learning model to detect code smells for refactoring as different models behave differently for specific smells.

• Generic classification and feature engineering. It is very challenging to design a general classifier which may be used for the process of software engineering as different features may be needed for different refactoring processes.

• Complexity of code patterns. Code bases often contain complex patterns and structures. Capturing and representing these patterns effectively for training deep learning models can be difficult, especially when dealing with large and diverse code bases.

• Context sensitivity. Refactoring decisions are often context-dependent, considering the broader system architecture, design patterns, and usage scenarios. Deep learning models might struggle to capture and understand such context-sensitive information.

• Interpretability. Deep learning models are known for their "black-box" nature, making it challenging to understand the rationale behind their refactoring recommendations. This lack of interpretability can be a significant hurdle for developers who need to trust and adopt these suggestions.

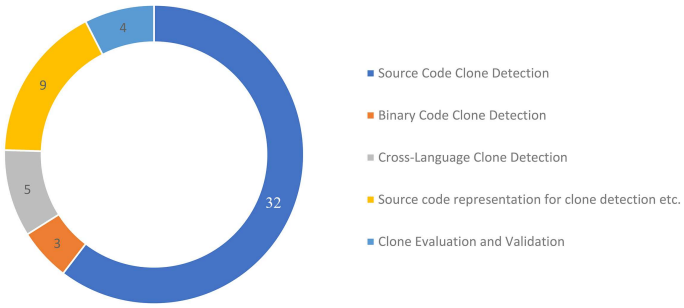

Figure 4 (Color online) Code clone research tasks where deep learning has been applied.

7.4.2 *Opportunities*

• Large language model-based refactoring. Large language models have great potential in smell detection and refactoring suggestions. Up to date, various deep learning techniques have been proposed to detect code smells, and/or to suggest solutions to identified code smells. However, to the best of my knowledge, large language models have not yet been applied to such tasks. Since large language models are good at understanding and generating source code, it is likely that they can be employed to format source code and to identify abnormal parts of source code (i.e., code smells). They may also be employed to discover (and measure) the relationship (like coupling, similarity, cohesion, and dependency) among software entities. Such a relationship, currently measured by statistic code metrics only, is critical for the detection of code smells as well as suggestion of refactoring solutions.

• Automated discovery of new refactorings. Up to date, the academic community focuses on well-known code smells and refactorings. All such smells and refactorings are coined by human experts. However, with the development of programming languages and new paradigms, new categories of code smells as well as new categories of refactorings are emerging. One possible way to discover such new smells and new refactorings is to mine the evolution history of open-source applications. With advanced deep learning techniques, it is potentially feasible.

8 Code clone detection

Code clones are code fragments that have the same or similar syntax and semantics. The wide presence of code clones in open-source and industrial software systems makes clone detection fundamental in many software engineering tasks, e.g., software refactoring, evolution analysis, quality management, defect prediction, bug/vulnerability detection, code recommendation, plagiarism detection, copyright protection, and program comprehension.

The recent research in code clones has attracted the use of deep learning techniques. As shown in Figure 4, most of the research focuses on source code clone detection and source code representation for clone detection. A little of research focuses on the binary clone detection, cross language clone detection, and clone evaluation and validation.

8.1 Source code clone detection

Source code clones are typically classified as follows based on their degree of similarity.

Type1: Duplicate code fragments, except for differences in white space, comments, and layout.

Type2: Syntactically identical code fragments, except for differences in variable names, literal values, white space, formatting, and comments.

Type3: Syntactically similar code fragments with statements added, modified, or deleted.

Type4: Syntactically different code fragments implementing the same functionality.

As the boundary between Type3 and Type4 clones is often ambiguous, in benchmarks like BigCloneBench [246] researchers further divide these two clone types into three categories: strongly Type3, moderately Type3, and Weakly Type3/Type4. Each type is harder to detect than the former one. Weak Type3 and Type4 clones are usually called semantic clones.

The research efforts on source code clone detection are shown in Table 9 [247–284]. As can be seen from Table 9, most of recent research tries to leverage deep neural networks to effectively capture complex

Table 9 Source code clone detection.

Year	Venue	Ref.	Language	Code representation	Deep learning models	Clone type	Benchmark	Baseline
2022	ICSME	SSCD [247]	Java, C/C++	token	CodeBERT, GraphBERT	Type3 and Type4	BigCloneBench, CompanyC, CompanyC++	SourcererCC [248]
2022	IWSC	[249]	Java	token	CodeBERT	All	BigCloneBench, SemanticCloneBench, etc.	NA
2022	TSE	Holmes [250]	Java	PDG	GNN, LSTM	All	GoogleCodeJam, SeSaMe, BigCloneBench	TBCCD [268]
2022	ICSR	SEED [251]	Java, C, C++	Semantic graph	GMN	Type4	OJClone, BigCloneBench	TBCCD [268], ASTNN [278]
2022	ADES	CCCD-DL [252]	Java	AST, CFG, FCG	DNN	Type4	BigCloneBench, open source projects	LV-CCD [279], FCCA [260], and TBCNN [255]
2021	QRS	CSEM [253]	C	Event Embedding Tree	GAT	Type3, Type4	OJClone	CCLearner [280], Deckard [281], CloneWork [282], SourcerCC [248]
2021	APSEC	[254]	Java, C	AST	BiC-CNN	Type3 and Type4	BigCloneBench, OJClone	Deckard [281], DLC [276], CDLH [275], Deepsim [272], ASTNN [278], etc.
2021	Applied Sciences	[255]	Java	AST	TBCNN	All	BigCloneBench	Deckard [281], RtvNN [276], CDLH [275], DeepSim [272]
2020	IEICE T INF SYST	[256]	Java	AST	CNN, Siamese	All	BigCloneBench	NiCad [283]
2020	ASE	SCDetector [257]	Java, C/C++	CFG	GRU, Siamese	All	GCJ, BigCloneBench	RtvNN [276], ASTNN [278], Deckard [281], SourcererCC [248]
2020	APSEC	Sia-RAE [258]	Java	AST	recursive autoencoders (RAE), Siamese	All	BigCloneBench	DeepSim [272], CDLH [275], weighted RAE [265]
2020	DSA	[259]	Java	CFG	Bi-RNN, GCN	Type4	open source projects	Nicad [283]
2020	T-R	FCCA [260]	Java	token, AST, CFG	RNN	All	BigCloneBench	Deckard [281], CDLH [275], DeepSim [272], DLC [276], SourcerCC [248], TBCCD [268]
2020	SANER	FA-AST-GMN [261]	Java, C/C++	FA-AST	GNN	All	GCJ, BigCloneBench	Deckard [281], RtvNN [276], CDLH [275], ASTNN [278]
2020	ISSTA	[262]	C++	AST, CFG	DNN	Type4	OJClone	Deckard [281], DLC [276], CDLH [275], ASTNN [278], DeepSim [272]
2020	IEEE Access	[263]	Java	AST	CNN	Type1, Type2	BigCloneBench	SourcererCC [248], NiCad [283], Deckard [281], CCLearner [280], Oreo [271]
2020	Complexity	[264]	Java, C/C++	AST	BiLSTM	All	OJClone, BigCloneBench	RAE [276], CDLH [275], ASTNN [278]
2019	IEEE Access	weighted RAE [265]	Java	AST	RAE	All	BigCloneBench	Oreo [271], DeepSim [272], CCLearner [280], CDLH [275], Nicad [283] Deckard [281], SourcerCC [248], CloneWorks [282]
2019	AAAI	ACD [266]	Java, C	AST	LSTM	All	OJClone, BigCloneBench	Deckard [281], SourcerCC [248], CDLH [275]
2019	SANER	[267]	Java	AST	RvNN, Siamese Network	Type4	BigCloneBench	Deckard [281]
2019	ICPC	TBCCD [268]	Java, C	AST, token	PACE	All	OJClone, BigCloneBench	CDLH [275], Deckard [281], SourcerCC [248], DLC [276]
2019	ISSTA	Go-Clone [269]	Golang	LSFG	DNN	NA	From Github	NA
2019	TII	[270]	C	AST	GCN	NA	Open source projects	VUDDY [284], LSTM
2018	ESEC/FSE	Oreo [271]	Java	metrics, token	DNN with Siamese architecture	All	Open source projects	Nicad [283], Deckard [281], SourcerCC [248], CloneWorks [282]
2018	ESEC/FSE	DeepSim [272]	Java	Semantic features matrix	Feed-forward nerual network	Type3, Type4	GCJ, BigCloneBench	CDLH [275], Deckard [281], RtvNN [276], etc.
2018	ICMLA	CCDLC [273]	Java	BDG, PDG, AST	CNN	NA	Open source projects	NA
2018	IJCAI	CDOU [274]	Java, C	AST	LSTM.word2vec	All	OJClone, BigCloneBench	CDLH [275], Deckard [281], SourcerCC [248], RtvNN [276] etc.
2017	IJCAI	CDLH [275]	Java, C	AST	LSTM	All	OJClone, BigCloneBench	Deckard [281], SourcerCC [248], RtvNN [276], etc.
2017	ICSME	CCLearner [280]	Java	token	DNN	Type1, Type2, Type3	BigCloneBench	Deckard [281], SourcerCC [248], Nicad [283]
2016	ASE	RtvNN [276]	Java	AST	RvNN	All	Open source projects	Deckard [281]
2016	ICMLA	[277]	Java	metrics	MLP	All	BigCloneBench	Deckard [281], SourcerCC [248], Nicad [283], CCFiner, IClone

Table 10 Source code representation learning for clone detection.

Year	Venue	Ref.	Language	Code representation	Deep learning models	Clone type	Benchmark	Baseline
2022	ICSE	[285]	Java, C/C++	token	CodeBERT, GraphCodeBERT	All	OJClone, BigCloneBench	CDLH [275], FA-AST-GMN [261], TBCCD [268], etc.
2022	SANER	[286]	C	features, token, AST, CPG	BiLSTM, LSTM, Transformer, Tree-LSTM, Code2Vec, GAT, GCN, GGNN	NA	OJClone	NA
2022	EMSE	[287]	Java	AST	code2vec	NA	opensource projects	NA
2021	ICSE	InferCode [288]	Java, C/C++	AST	TBCNN	All	OJClone, BigCloneBench	code2vec, code2seq, Deckard [281], SourcererCC [248], RtvNN [276]
2021	ICONIP	HGCR [289]	C, Java	SG, EDFG	T-GCN, E-GAT	All	GCJ, BigCloneBench,	ASTNN [278], FA-AST, FCCA
2019	ICSE	ASTNN [278]	Java, C	AST	ASTNN	All	OJClone, BigCloneBench	TBCNN [293], CDLH [275], RAE [276], etc.
2019	ICCF	TBCAA [290]	Java, C	AST	Tree-based Convolution	All	OJClone, BigCloneBench	CDLH [275], Deckard [281], SourcerCC [248], DLC [276]
2019	ICSME	TECCD [291]	Java	AST	Sentence2Vec	Type3	BigCloneBench, open source projects	CCLearner [280], Nicad [283], CCAligner [294]
2018	ICSE	[292]	Java	identifier, ASTs, CFGs, and Bytecode	RNN	All	Qualitas.class Corpus [295]	NA

semantic information in code fragments, so as to improve the effectiveness of semantic clone detection.

The code clone detection process begins by modeling the semantics of the source code. To achieve this goal, diverse program representations such as tokens, ASTs, CFGs, data flow graphs (DFGs), and PDGs are being used to learn program features.

Various deep learning models, such as CodeBERT, GraphCodeBERT, GNN, graph attention network, CNN, GCN, ASTNN, tree-based convolutional neural network (TBCNN), recursive autoencoders (RAE), RNN, recursive neural network (RvNN), LSTM, have been used.

These studies have achieved higher recall and better precision than the best classical approaches.

8.2 Code representation learning for clone detection

The studies in Table 10 [248, 261, 268, 275, 276, 278, 280, 281, 283, 285–295] focus on learning source code representation, so as to automatically capture the syntactic and semantic information from source code. Then the embedding is applied to code clone detection, code classification tasks, etc. Code similarities can be learned from diverse representations of the code, such as identifiers, tokens, ASTs, CFGs, DFGs, and bytecode.

Siow et al. [286] performed an empirical study on code representation. They found that the graph-based representation is superior to the other selected techniques across these tasks. Different tasks require task-specific semantics to achieve their highest performance; however, combining various program semantics from different dimensions such as control dependency, data dependency can still produce promising results. Tufano et al. [292] demonstrated that combined models relying on multiple representations can be effective in code clone detection and classification.

Table 11 Cross-language code clone detection.

Year	Venue	Ref.	Language	Code representation	Deep learning models	Benchmark	Baseline
2023	Computers	CLCD-I [296]	Java, Python	AST	Siamese architecture [300]	Code from programming competition	LSTM
2022	ICPC	UAST [297]	C, C++, Java, Python, JavaScript	token	Bi-LSTM, GCN	JC, dataset collected from leecode	InferCode [288]
2019	SANER	[298]	Java, C++	AST	Bi-NN	OJClone, opensource projects	TBCNN [293], etc.
2019	MSR	[151]	Java, Python	token, AST	tree-based skip-gram, LSTM	Code from programming competition	Sequential input model
2019	ASE	CLCDSA [299]	Java, Python, C#	Metrics	DNN	Code from programming competition	LICCA [301], CLCMiner [151, 302]

Table 12 Binary code clone detection.

Year	Venue	Ref.	Code representation	Deep learning models	Benchmark	Baseline
2018	MASES	[303]	Visualization graph	CNN	GoogleCodeJam, etc.	Shallow Neural Net
2018	FEAST	Clone-Slicer [304]	CFG, DDG	RNN	SPEC2006	CloneHunter [306]
2017	CCS	Gemini [305]	ACFG	Structure2vec	Dataset from [307], etc.	Genius [307]

8.3 Cross-language code clone detection

The above work on source code clone detection focuses on clones in a single programming language. However, software systems are increasingly developed on a multi-language platform on which similar functionalities are implemented across different programming languages [151, 296].

The main challenge of cross-language code clone detection is how to reduce the feature gap between different programming languages.

The studies on cross-language code clone detection are shown in Table 11 [151,288,293,296–302]. These studies focus on extracting syntactic and semantic features of different programming languages. Nafi et al. [299] used a Siamese architecture to learn the metric features. Perez et al. [151] used an unsupervised learning approach for learning token-level vector representations and an LSTM-based neural network to predict clones. Bui et al. [298] proposed a Bi-NN framework to learn the semantic features of two different programming languages. Wang et al. [297] proposed a unified abstract syntax tree neural network. Yahya et al. [296] used AST embeddings from InferCode [288] as input of the Siamese architecture.

8.4 Binary code clone detection

Binary code clone detection can be used in the context of cross-platforms as well as legacy applications that are already deployed in several critical domains.

Research on binary code clone detection is shown in Table 12 [303–307].

Xue et al. [304] combined program slicing and a deep learning-based binary code clone modeling framework to identify pointer-related binary code clones. Xu et al. [305] proposed a neural network-based approach to compute the embedding based on the control flow graph of each binary function, and then to measure the distance between the embeddings for two binary functions. Marastoni et al. [303] tackled the problem of binary code similarity using deep learning applied to binary code visualization techniques. They found that it is important to further investigate how to build a suitable mapping from executables to images.

8.5 Clone evaluation and validation

Mostaeen et al. [308] proposed a machine learning-based approach for predicting the user code clone validation patterns. The proposed method works on top of any code clone detection tools for classifying the reported clones as per user preferences. The automatic validation process can accelerate the overall process of code clone management.

Saini et al. [309] presented a semi-automated approach to facilitating precision studies of clone detection tools. The approach merges automatic mechanisms of clone classification with manual validation of clone pairs, so as to reduce the number of clone pairs that need human validation during precision experiments.

Liu et al. [310] proposed an evaluation methodology that can systematically measure the cross-functionality generalizability of neural clone detection. They also conducted an empirical study and the results indicate that the studied neural clone detectors cannot generalize well as expected. They found that the performance loss on unseen functionalities can be reduced by addressing the out-of-vocabulary problem and increasing training data diversity.

Table 13 Source code clone detection.

Dataset	Ref.	Language
BigCloneBench	[246]	Java
OJClone	[293]	C
GoogleCodeJam (GCJ)	[272]	Java
SemanticCloneBench	[313]	Java, C, C#, Python
SeSaMe	[314]	Java
JC	[298]	Java, C++
Leetcode	[297]	C, C++, Java, Python, JavaScript

Yu et al. [311] presented an experimental study to show that BigCloneBench typically includes semantic clone pairs that use the same identifier names, which however are not used in non-semantic-clone pairs. To alleviate these issues, they abstracted a subset of the identifier names (including type, variable, and method names) in BigCloneBench to result in AbsBigCloneBench and used AbsBigCloneBench to better assess the effectiveness of deep learning models on the task of detecting semantic clones.

Krinke and Ragkhitwetsagul [312] performed a manual investigation on BigCloneBench. They demonstrated that the way BigCloneBench being constructed makes it problematic to use BigCloneBench as the ground truth for learning code similarity. BigCloneBench fails to label all clone pairs. Moreover, only a small set of true negatives has been created and, for most of the possible pairs in the dataset, the ground truth is unknown. This leads to a strong impact on the validity of the ground truth for Weakly Type3 and Type4 clone pairs, threatening the validity of results for evaluations in the Weakly Type3 and Type4 category and approaches to learning code similarity.

8.6 Datasets

We summarize datasets used in clone detection in Table 13 [246, 272, 293, 297, 298, 313, 314].

BigCloneBench is a benchmark of inter-project clones from IJaDataset [246], a big Java source code repository. It has about 8M labeled clone pairs, as well as 260000 false clone pairs, covering 43 functionalities. BigCloneBench divides Type3 and Type4 clones into four categories: Very-Strongly Type3, Strongly Type3, Moderately Type3, and Weakly Type3/Type4.

OJClone is generated based on the OJ dataset [293] covering 104 functionalities. Each functionality is a programming question with 500 verified solutions written in C, submitted by students. Two solutions to the same question can be considered a clone pair.

GoogleCodeJam (GCJ) is a benchmark similar to OJClone. It contains 1669 solutions written in Java for 12 functionalities collected from GoogleCodeJam.

SemanticCloneBench is a benchmark of semantic clone pairs [313]. It consists of four thousand clone pairs, each for four programming languages (i.e., Java, C, C#, and Python). The clone pairs are collected from the StackOverflow answers. The method pairs to the same questions on Stack Overflow are considered semantic clones.

The SeSaMe dataset consists of 857 semantically similar method pairs mined from 11 open-source Java repositories [314].

The JC dataset includes 10 categories of programs crawled by Bui et al. [298] from GitHub. It contains 5822 Java files and 7019 C++ files. The code files for each category implement the same function.

The Leetcode dataset contains 50 categories of programs from Leetcode, each of which contains 400 semantically similar solutions of five different programming languages, with a total of 20000 files [297].

8.7 Challenges and opportunities

This subsection presents the challenges and opportunities for further work on code clone detection.

8.7.1 *Challenges*

• Challenge to build comprehensive learning-oriented code clone detection datasets. Most of the existing code clone datasets are of a limited scale due to the effort required in manually constructing the benchmarks. BigCloneBench is a large-scale dataset and becomes a standard to evaluate and compare the performance of clone detection tools. Many researchers also use it to train deep learning models. However, as pointed out by Krinke and Ragkhitwetsagul, the incomplete ground truth and the bias and

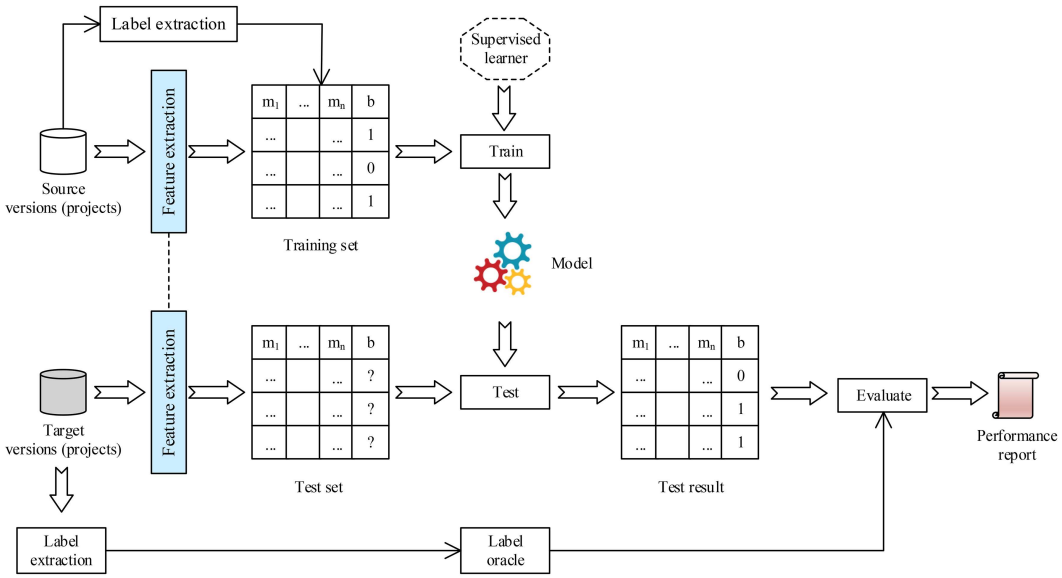

Figure 5 (Color online) Overall structure of supervised defect prediction.

imbalance of the ground truth will have a strong impact on deep learning approaches for code similarity that learned from Big-CloneBench's ground truth [312]. Besides, there still lack standard datasets for cross-language clone detection. Also, there is no clear taxonomy about cross-language clones. Therefore, it is necessary to build large, cross-language, learning oriented clone datasets.

• Challenge to cross-functionality generalizability of deep learning-based clone detection. Most of the deep learning-based code clone detection approaches are proposed for detecting semantic clones, and they have achieved impressive results based on the tested benchmarks. However, according to the research by Liu et al., these studies are limited in detecting clones whose functionalities have never been previously observed in the training dataset [310]. Further research on deep learning algorithms for improving the cross-functionality generalizability is required. Moreover, cross-functionality generalizability should also be considered in detection result evaluation.

• Challenge to selecting suitable source code representation. Current deep learning-based clone detection approaches use different representations of source code like tokens, AST, CFG, and PDG. These represent variations in clone detection scalability, efficiency, and effectiveness. For example, approaches representing code fragments as sequences of tokens may be more efficient, but they may not be generalized to source code having out-of-vocabulary tokens, as the vocabulary of tokens is unlimited. Approaches representing code fragments as PDGs help in semantic analysis. However the low efficiency may prevent them from being used on a large code base. Further research on source code representation is still needed to deeply analyze the scalability, efficiency, and effectiveness.

8.7.2 *Opportunities*

• Opportunity to explore deep learning in more clone related tasks. A software may contain a lot of clones but not all clones need to be manipulated. History of clone management helps extract useful features that can help make various decisions of clone related tasks. For example, deep learning-based recommendations can help automatically identify important clones for refactoring. Deep learning-based methods can also help predict the code clone quality, detect clone related bugs, and reduce the maintenance cost caused by harmful or risky clones. However, only a few existing studies focus on these tasks. Research can be further conducted in these areas.

• Opportunity to explore large language model in detecting cross-language code clones. In recent years, large language models have experienced rapid development. By pre-training on a large amount of corpus, the model is able to "remember" knowledge from the corpus, including syntax and semantics, thus typically being able to understand the syntax and semantics of different programming languages. This lays a good foundation for detecting cross-language code clones.

9 Software defect prediction

Software defect prediction (SDP) aims to forecast potential defect locations in a software project, predicting which modules (such as files, classes, and functions) may contain defects. This predictive information is crucial for the software quality assurance process. On one hand, it prioritizes modules awaiting inspection or testing, facilitating early identification of defective modules in the software project under test. On the other hand, it guides testing personnel to allocate code review or testing resources more efficiently and sensibly for each module. Allocating more review resources to modules with a higher likelihood of defects can help quality assurance personnel discover as many software defects as possible within the given budget.

As depicted in Figure 5, in the supervised defect prediction scenario, a learner is utilized to establish the connections between features and labels using the training set. Subsequently, the learned model is applied to the test set to predict defect-proneness. For each instance in the test set, the corresponding features are used to calculate the probability of being defect-prone. If the probability surpasses a predefined threshold (typically set at 0.5), the instance is classified as "buggy"; otherwise, it is labeled as "clean". Traditionally, numerous manually crafted features have been employed, such as size metrics, complexity metrics, cohesion metrics, and coupling metrics. However, these conventional hand-crafted features primarily rely on syntactic information. Consequently, they fail to capture semantic information, resulting in limited predictive capability for defect-proneness. To address this limitation, a variety of deep learning techniques have been adopted to generate powerful semantic features for defect prediction.

Table 14 [315–346] provides a summary of notable studies that employ deep learning techniques in defect prediction. Table 14 includes information on the publication year and the type of defect prediction addressed in the first and second columns, respectively. To simplify the presentation, "WPDP" represents within-project defect prediction, while "CPDP" represents cross-project defect prediction. The third column presents the granularity at which defect prediction is conducted, spanning from file-level to statement-level. Note that GDRC (graph of defect region candidates) corresponds to an area of the code file. The fourth column specifies the utilized deep learning technique. The fifth to ninth columns outline the inputs provided to the deep learning model for extracting powerful semantic features. An entry marked with "•" or non-blank signifies that the corresponding information is used as input, while a blank entry indicates its absence. The last column provides the reference for each study.

From Table 14, we can see that numerous studies have emerged, employing a variety of deep learning techniques for defect prediction. Among these techniques, CNN, RNN (including LSTM as a variant), and GCN stand out as the most prominent ones, each offering distinct capabilities that researchers find advantageous for defect prediction. CNN proves highly effective in capturing localized patterns and spatial dependencies within the data. RNN excels at capturing sequential and long-term dependencies, which are particularly valuable in understanding the temporal aspects of defect occurrences. GCN demonstrates remarkable proficiency in capturing the structural information and intricate inter-dependencies within code, attributes that are crucial for accurate defect prediction. Overall, the adoption of deep learning techniques has significantly advanced defect prediction research, offering valuable insights into defect detection.

Overall, in the defect prediction community, significant attention has been dedicated to utilizing deep learning in order to generate expressive features and enhance the effectiveness of defect prediction. Through the application of deep learning, remarkable progress has been made in automatically generating features that are both highly informative and discriminative. This transition from manual feature engineering to automated feature extraction using deep learning signifies a fundamental change in approaches to predicting defect-proneness in software systems. The objective is to construct models capable of comprehending the fundamental traits and complexities of the code, thereby enabling more dependable and accurate defect predictions.

9.1 Using manually crafted features as the input

In the literature, numerous manually crafted features have been proposed for defect prediction. However, these features often focus on specific characteristics of a module, and their high-level semantic relationships are not adequately captured, limiting their defect prediction capability. To address this limitation, Yang et al. [315] utilized DBN to generate expressive features from a set of initial change features. This approach results in more powerful change-level defect prediction models. Similarly, Tong et al. [320]

Table 14 Overview of prominent defect prediction models utilizing deep learning techniques.

Year	Type	Granularity	Deep learning technique	Input to deep learning					Ref.
				Hand-crafted features	AST	CFG/ CPG	Raw code (token)	Other	
2015	WPDP	Change	DBN	•					Yang et al. [315]
2017	WPDP	File	CNN			CFG			Phan et al. [316]
	WPDP	File	CNN		•				Li et al. [317]
2018	WPDP/CPDP	File	CNN				•	Comments	Huo et al. [318]
	WPDP	File	RNN	•				Change history	Liu et al. [319]
	WPDP	File/Function	SDAEs	•					Tong et al. [320]
2019	CPDP	File	CNN		•				Qiu et al. [321]
	WPDP	Change	CNN					Commit log + change	Hoang et al. [322]
	WPDP	File/Function	Deep Forest	•					Zhou et al. [323]
	WPDP	File	DNN	•					Xu et al. [324]
	WPDP	File	Layered RNN	•					Turabieh et al. [325]
	WPDP/CPDP	File	LSTM		•				Dam et al. [326]
	CPDP	File	Bi-LSTM		•				Li et al. [327]
2020	WPDP/CPDP	File	CNN				• (image)		Chen et al. [328]
	WPDP/CPDP	Change	DAE-CNN	•					Zhu et al. [329]
	WPDP/CPDP	File/Change	DBN		•				Wang et al. [330]
	WPDP	File	Bi-LSTM	•					Deng et al. [331]
	WPDP/CPDP	File	Bi-LSTM +Attention		•				Shi et al. [332]
	WPDP	Statement	LSTM	•			•		Majd et al. [333]
	WPDP	File	LSTM	•				Change history	Wen et al. [334]
2021	WPDP/CPDP	File	CNN		•				Shi et al. [335]
	CPDP	File	DNN	•					Xu et al. [336]
	WPDP	File	GCN		•				Xu et al. [337]
	WPDP	File	GCN	•				CDN	Zeng et al. [338]
	CPDP	GDRC	GCN			CPG			Xu et al. [339]
	WPDP	File	LSTM		•				Wang et al. [340]
	CPDP	File	MDA	•					Zou et al. [341]
	WPDP	File/Function	SSDAE	•					Zhang et al. [342]
2022	WPDP/CPDP	File	Bi-LSTM				•		Uddin et al. [343]
	WPDP	Change	TCN	•					Ardimento et al. [344]
2023	WPDP	Code line	HAN				•		Pornprasit et al. [345]
	WPDP	File	CNN		•				Qiu et al. [346]

employed SDAEs to extract expressive features from traditional software metrics. Following their work, a range of deep learning techniques have been explored for this purpose, including deep forest [323], DNN [324, 336], Layered RNN [325], DAE-CNN [329], Bi-LSTM [331], MDA [341], SSDAE [342], and TCN [344].

9.2 Using raw source code as the input

Traditional manually crafted features in source code analysis tend to be syntax-based, disregarding the valuable semantic information embedded in the code. To overcome this limitation, Chen et al. [328] introduced a novel technique that visualizes the source code of programs as images and then employs CNN to extract image features. These extracted features are utilized for building more effective defect prediction models. Uddin et al. [343] leveraged Bi-LSTM to capture expressive features from the embedded token vectors obtained through BERT applied to the source code. This technique allows effectively harnessing the semantic information within the code for defect prediction. Furthermore, Pornprasit et al. [345] adopted HAN to learn semantic features from the surrounding tokens and lines of code. This approach enables the prediction of defective lines more accurately by considering the context and relationships between different parts of the code.

9.3 Using abstract syntax trees as the input

One significant drawback of using raw source code as input is the disregard for crucial structural information when extracting expressive features. To tackle this issue, Li et al. [317] introduced a novel approach, utilizing a program's AST as input to generate semantic features. In their approach, Li et al. initially extracted token vectors from the program's AST and then transformed them into numerical vectors through mapping and word embedding. Subsequently, they fed these numerical vectors into a

CNN, allowing the model to learn both semantic and structural features for defect prediction. Following this breakthrough, several other deep learning techniques have been explored for the same purpose, including LSTM [326, 340], Bi-LSTM [327, 332], DBN [330], and GCN [337]. These approaches aim to enhance defect prediction by considering the inherent structural characteristics of the source code, leading to more effective and accurate models.

9.4 Using graphical representations as the input

Phan et al. [316] emphasized the importance of utilizing precise graphs that represent program flows to capture the complex semantics of programs accurately. They argued that tree structures like ASTs might fall short in this regard. To address this issue, Phan et al. adopted CFGs and applied GCN to extract expressive features. By leveraging CFGs, they were able to capture intricate program dependencies and interactions more effectively. The GCN allows the generation of expressive features that incorporate both local and global information from the program's control flow. As a result, their approach improves the accuracy of defect prediction models by considering the intricate semantics of the code. In a related study, Xu et al. [339] proposed an alternative approach using CPGs as input to GCN. This approach aims to extract even more expressive features for defect prediction, further enhancing the capabilities of the model in predicting software defects.

9.5 Using hybrid-source information as the input

In addition to source code, other information such as comments, commit logs, and change history contains valuable insights for defect prediction. Recognizing this, researchers have proposed innovative approaches to leverage this diverse information for more accurate defect prediction models. Huo et al. [318] introduced a technique that jointly learns semantic features from both source code and comments for defect prediction. Likewise, Liu et al. [319] utilized the historical version sequence of manually crafted features from continuous software versions as input for RNN in defect prediction. Subsequent studies follow a similar path, combining different sources of information. Some explore the joint use of change and commit logs [322], while others investigate the combination of manually crafted features and class dependency networks (CDNs) [338]. Integrating these diverse data sources provides richer and more expressive semantic features, leading to further improvements in defect prediction capabilities.

9.6 Datasets

The field frequently relies on datasets like NASA, PROMISE, AEEEM, and Relink, valued for their extensive coverage of Java and C, the primary programming languages [323, 342]. These datasets predominantly focus on file-level granularity and typically encompass fewer than 1000 instances within a project.

• **Programming language.** The majority of datasets comprise open-source projects, prominently featuring programming languages such as Java, C, and C++ [323]. Typically, these studies encompass around 10 projects, with over 80% utilizing Apache projects.

• **Prediction granularity.** The datasets exhibit varying levels of prediction granularity, spanning from file-level, change-level, and function-level, to statement-/line-level. File-level granularity is the most prevalent (nearly 80%), whereas statement-/line-level granularity receives less emphasis [333, 345].

• **Training dataset size.** In the majority of studies, within-version defect prediction is the norm, leading to training sets typically comprising fewer than 1000 instances. Notably, only a handful of change-level defect prediction studies leverage notably larger datasets, with instance counts reaching tens of thousands [329].

In summary, existing studies primarily concentrate on projects involving a small number of programming languages, mainly focusing on coarse-grained defect prediction. In particular, the training set sizes for deep learning are often limited.

9.7 Challenges and opportunities

While numerous studies have put forward different defect prediction methodologies, several significant challenges hinder the development of accurate and cost-effective approaches for defect prediction. These

challenges encompass issues like model interpretability, thorough assessment, and the replication of experiments. Simultaneously, there are promising opportunities on the horizon that could potentially address these challenges in future studies on defect prediction.

9.7.1 *Challenges*

The application of deep learning to defect prediction presents the following main challenges that need to be addressed.

• **Limited interpretability of control- and data-flow in relation to defect-proneness.** Practitioners highly value understanding the root causes within code that contribute to defect-proneness, particularly concerning control- and data-flow dynamics. Such understanding is pivotal in comprehending the occurrence of defects and subsequently addressing them. However, the opacity of deep learning models presents a challenge, as they are often perceived as black boxes, impeding the ability to interpret the rationale behind their predictions. This lack of transparency restricts the interpretability of the models, posing a barrier to the adoption in defect prediction, an area where interpretability holds critical importance.

• **Insufficient assessment of the effectiveness in comparison to traditional models.** Many studies evaluating new deep learning models tend to solely compare them against earlier deep learning approaches, often neglecting comprehensive comparisons with established traditional models. This oversight results in a significant gap in understanding the true advancements brought about by deep learning techniques in defect prediction. Consequently, it becomes challenging to gauge the tangible progress and assess how deeply these adopted deep learning methodologies truly drive the field forward compared to well-established traditional models.

• **Substantial challenges in replicating the reported findings externally.** The absence of standardized practices for distributing replicated packages, encompassing datasets and their corresponding scripts, poses a significant challenge. In practice, it is well known that even minor variations in the reimplementation of prior models can lead to substantial differences in defect prediction performance. This lack of transparency and accessibility hinders external validation of insights derived from past studies. As a result, ensuring the robustness and applicability of reported conclusions becomes progressively more challenging.

Future studies should aim to address the aforementioned challenges to accurately gauge the progress made in the field of defect prediction. Failing to do so will hamper our understanding of the extent to which advancements have been made. Furthermore, there is a risk of unintentionally drawing misleading conclusions regarding the advancement of the state-of-the-art, potentially leading to missed opportunities for further progress in defect prediction.

9.7.2 *Opportunities*

There are several valuable opportunities available to address the challenges associated with deep learning-based defect prediction.

• **Development of interpretable techniques.** One promising opportunity lies in the development of novel techniques that enhance the interpretability of deep learning models in defect prediction. Researchers can explore methods such as feature attribution, saliency mapping, and attention mechanisms to gain insights into the factors contributing to defect-proneness. These techniques would enable a better understanding of model behavior and facilitate the identification of critical features and patterns related to defects.

• **Comparative evaluation frameworks.** A key opportunity is the establishment of comprehensive evaluation frameworks for deep learning-based defect prediction. This involves designing rigorous comparative studies that systematically compare the performance of deep learning models with traditional approaches on diverse datasets. By incorporating various evaluation metrics and statistical analysis, researchers can provide robust evidence on the effectiveness and advantages of deep learning models.

• **Replication and external validation studies.** Another important opportunity is to conduct replication and external validation studies to verify the reported findings in deep learning-based defect prediction. Collaboration among researchers, sharing of datasets and code, and adopting open science practices can facilitate the replication of experiments across different research groups. External validation studies conducted on independent datasets can provide further validation of the generalizability and reliability of the reported results.

By embracing these opportunities, researchers can address the challenges faced in deep learning-based defect prediction and pave the way for advancements in model interpretability, performance evaluation, and the overall reliability of defect prediction systems.

10 Bug finding

The term "bug" originates from a literal bug (i.e., moth) stuck in the relay of the Mark II computer, and has been used to denote any defect that violates its specification: unexpected crashes, incorrect results, and information leakages. Bug finding is a process that involves examining software artifacts (source code repositories, documentation, existing test inputs, etc.) and generating a list of potential bugs with explanations.

The process of bug finding goes beyond mere "wild guesses". Unlike defect prediction (Section 9), which identifies loose correlations between software metrics and bugs, bug finding techniques should provide concrete bug certificates. Certificates can aid developers in accurately identifying the actual presence and root cause of a bug. Such certificates could be a hint, like a bug anti-pattern, or a specific test case that triggers a program error. This section discusses the three mainstream approaches to bug finding.

(1) Static analysis. Bugs reside within source code, and bugs can be identified by "reading" the source code and its associated artifacts, without the need to execute the program in a real environment. Static bug does not require resources (such as hardware platform, computational power, environment, and dependencies) to bootstrap the software, making it accessible to any party involved in a software's life cycle. From Lint tools [347] to bug finders [348,349], static analysis serves as a fundamental gate-keeping procedure in the software development process.

(2) Dynamic testing. More reliable certificates provide undeniable evidence of a bug's existence, with test cases [350] being the most commonly used such certificates. Developers create unit and system test cases, and software is also extensively validated using machine-generated test cases. Passing test cases enhances the confidence that the software functions under various circumstances.

(3) Formal verification. In theory, one can exhaustively test a (finite-state) system to prove its bug-freedom. To make this procedure practical, one can leverage path-based abstraction and employ a constraint solver to manage the search space [351,352]. Alternatively, one can provide a machine-checkable proof to a proof assistant. There have been successful reports on the verification of compilers [353], operating system kernels [354], and other software systems [355].

The three mainstream approaches have undergone decades of evolution, particularly the classical (formal) algorithmic bug-finding techniques. These techniques rely on algorithms, logical procedures that can be mechanically implemented over simple axioms, to identify bugs [356]. With the advent of deep learning, probabilistic techniques based on machine learning have gained popularity, because learned models can effectively digest inputs from the chaotic and complex human world.

The algorithmic and probabilistic paradigms complement each other in the deep-learning era. This is because software, which projects the requirements of the probabilistic human world onto the algorithmic computing world, intersects both of these realms [357,358]. To provide a bug certificate, one must possess not only a thorough understanding of the requirements but also the ability to engage in complex logical reasoning.

Therefore, we argue that significant research efforts should be invested to answer how to leverage learned (albeit imperfect) domain-knowledge to facilitate an effective algorithmic bug-finding procedure. This approach builds upon the success of AlphaGo [359], harnessing the potential of "AI-in-the-(algorithmic)-loop", akin to having "experienced human experts" consistently offering insights into software artifacts and intermediate results whenever decisions are required in static checking, dynamic testing, and formal verification. On the other hand, it would also be equally interesting to explore whether bug-finding can be driven by an autonomous agent, like the chain of thought [360, 361], by automatically exploiting the existing algorithmic bug-finding techniques.

In addition to the classical papers that shed light on the fundamentals of bug finding, this section also comprehensively surveys recent papers by conducting Google Scholar search using the combinations of the following two sets of keywords.

- S_1: Testing, validation, verification, fuzzing, program analysis, static analysis, dynamic analysis, symbolic execution, formal methods.

• S_2: Neural network (NN), DL, machine learning (ML), reinforcement learning (RL), LLM, transformer.

We collect recent (2018–2023) papers and narrow our selection to papers from top-tier conferences and journals in the fields of software engineering, programming languages, and computer systems. Additionally, we include widely-cited papers with 30 or more citations. In total, we select 72 papers for this section (and more in the case study on vulnerability detection in Subsection 10.4). These papers are categorized and discussed as follows.

10.1 Static bug-finding: program analysis

10.1.1 *Code proofreading*

Even if we do not check against any specification on application behavior, we expect that source code reads like natural language texts, following the famous quote from a student's first programming class [362]:

> Programs are meant to be read by humans and only incidentally for computers to execute.

Readability implies that the code is easier to maintain, and researchers do find that programs resemble natural-language texts to some extent [363]. This "naturalness" of software serves as the foundation of code proofreading (by either a human or a checker of probabilistic distribution) by identifying anomalies that correlate to bugs.

Such "proof reading" can be algorithmic, i.e., checking against formal best-practice specifications [364, 365] or lint rules [347]. Adding a bit of probability to linters yields interesting chemical reactions: any "odd" (infrequent) pattern may suggest a bug [366]. This "may belive" approach paves a way for harnessing the power of both formal templates and code-level counter-coincidental couplings, leading to the family of bug-finding techniques that mine frequent items and consider minor items as bugs, e.g., CP-miner [367].

Today's deep learning models, which are fine-tuned over billion lines of code (e.g., llama2-code), are far more capable than finding these counter-coincidental couplings between lexical tokens. Large language models exhibit strong usefulness in finding (and even fixing) the "unnatural" bugs, even for semantics bugs that violate application specifications. For example, GitHub Copilot is an AI pair programmer which facilitates the finding and repairing of bugs through its context-sensitive recommendations[3)]. BUGLAB is a self-supervised approach that trains bug detectors by co-training a bug selector, which produces bugs that are elusive to detect [368].

Seemingly surprising, the intuition behind these papers is straightforward: a small code snippet is "almost completely determined" by its surrounding context. Based on the fact that most of the bugs are not "new" and there are similar bug samples in the training data, a language model can well predict the existence of them—code proofreading is very useful as a gate-keeping procedure for improving software quality. On the other hand, we still cannot fully rely on it, either algorithmically or probabilistically, to find bugs. Probability distributions like ChatGPT are always happy to provide proofreading results and explanations on the bug, but many times "talks nonsense with a straight face". Algorithmic lint tools produce an excessive number of false alarms. These fundamental limitations can be alleviated, but are not expected to be resolved in the near future [357, 369].

10.1.2 *Semantic analysis*

To catch subtle semantic bugs that appear to be correct and escape proofreading, it is necessary to understand precisely what a piece of program can and cannot do (i.e., conducting a semantic analysis) [370]. Observing that each program statement's behavior has been rigorously defined as a part of the language specification, it would be theoretically possible to predict a program's all possible behaviors by an algorithmic procedure (see Subsection 10.3; however, the general problem is undecidable [371]). On the other hand, while today's language models can somehow find semantic bugs, their probabilistic nature makes them less predictable (and thus less reliable) than algorithms.

Classical semantic analysis strives to over-approximate the program semantics to obtain sound predictions of program outcomes. In practice, static analysis techniques have to trade off between usefulness, thoroughness, and engineering efforts. For decades, the data-flow and abstract interpretation framework

3) OpenAI and GitHub. Github copilot: Your AI pair programmer, 2021.

dominates the field, and numerous properties about programs can be automatically proven under rigorous logical reasoning upon the formal operational semantics [372], and scaling to millions of lines of code [349].

Still, static analyzers are limited to finding bugs of limited "general" types of semantic bugs like pointer errors and data races, which are neutral to the application logic. It seems still a long way for today's analyzers to automatically prove arbitrary developer's assertions about program states. Certificates from state analysis are also imperfect for bug-finding. Static analyzers have to draw indefinite conclusion like "two variables may have the same value" (but actually not) to achieve scalability, resulting in false bug certificates.

The paths for algorithmic and probabilistic static analysis have long been diverged, and the progress to unite them is still exploratory. Probabilistic distributions are useful in providing heuristic hints, e.g., control of the degree of analysis sensitivity to meet resource constraints [373]. Ref. [374] proposes a technique to learn effective state-selection heuristics from data, in order to keep track of a small number of beneficial program states in Infer [349]. LLift [375] complements semantic analysis by providing speculative conclusions on undecided (timeout) symbolic-execution paths. Static analyses can also be complemented by learned features over reduced programs [376], or leveraging anomaly detection techniques to learn a balance between precision and scalability [377].

Machine learning can also be incorporated with specific static analysis techniques for better effectiveness and precision. Refs. [378, 379] addressed the challenge of manually developing cost-effective analysis heuristics for pointer analysis using an algorithm for heuristic learning. LAIT [380] utilizes an iterative learning algorithm that develops a neural policy to identify and eliminate redundant constraints throughout the sequence in order to produce a faster and more scalable numerical analysis.

Today's deep learning models are fundamentally limited in predicting the execution results of even a small code snippet [381], and we believe that static analysis is a field to be revolutionized by deep learning. The potential lies in the naturalness of software, which "informally" reflects program semantics, like the following piece of code, which is difficult to fully (rigorously) analyze. On the other hand, a language model can speculate variable types or even possible values at a specific program point [382,383], even if the static analyzer cannot prove it. These results may further benefit analyses on other parts of the program, which may be useful in developing efficient analyses. Unfortunately, there still lacks a framework that can simultaneously exploit the power of rigorous and long logical reasoning and the power of understanding the human-world semantics of programs from LLMs [384, 385].

Another promising direction is approximating program semantics via deep-learning models. For example, neural program smoothing [386, 387] approximates programs as differentiable functions of inputs and accordingly generates new test inputs using gradient descent. Ref. [388] constructed a benchmark suite with 28 open-source projects and proposed PreFuzz, which guides fuzzing by a resource-efficient edge selection mechanism and a probabilistic byte selection mechanism. However, the efficacy of neural program smoothing remains an area for further exploration according to extensive evaluation [389].

Enabling AI-in-the-loop for static analysis brings another challenge: machine learning models can be as large as billions of parameters, and it is impractical for static analyzers that frequently query the model. Today's AI-aided static analysis is still limited to manual feature engineering, in which simple classifiers like gradient boosting [374] are used. Deep learning model inference is considerably more expensive, and the problem remains open.

10.2 Dynamic bug-finding: software testing

10.2.1 *Test oracle*

Before one can test a program, a fundamental question should be answered first: what do we mean by a program to be "correct"? Some correctness criteria are obvious: a program should not crash, should be race-free, memory-safe, assertion-pass, and, most importantly, functional. A somehow desperate fact is that as long as software reflects a physical-world procedure (i.e., requirements), being functional becomes a myth[4)], and this is referred to as the "specification crisis" [390].

In the context of testing, such a specification, which decides whether each test case passes, is called a test oracle. General programming errors, e.g., semantic errors discussed in Subsection 10.1, can be a part of a test oracle and can be effectively checked at runtime. Sanitizers [391, 392] are famous for being

4) For example, more types of genders are recognized by the society over time, but software systems often fall behind.

effective in bug-finding for practical systems. However, finding a generic test oracle that decides whether a program is functional remains as an open problem.

Despite that the research community has a strong focus on modeling and lightweight formal methods [393], we argue that the test oracle problem may be one of the first problems to be effectively solved by deep learning. A key observation is that, while programs are generally hard for neural networks to understand, an execution trace really looks like a (maybe long) story consisting of events. The implication of this observation is twofold.

(1) Deep learning models can mimic human developers that write test cases or directly act as a human test operator, and simultaneously generate expected program behavior (test oracle) [394], following that a test case describes a "natural" procedure in the human world, e.g., generating natural faults for mutation testing [395].

(2) Deep learning models can digest software's execution traces. Traces can be projected to the human domain, and a language model can find "common sense violation" cases [396].

Test oracles for unit testing are a promising direction to be solved by machine learning [397, 398]. AthenaTest [399] is a method designed to generate unit test cases through a sequence-to-sequence learning model, trained first on a large unsupervised Java corpus for denoising and fine-tuned on real-world methods and developer-written test cases. Atlas [400] is an NMT-based approach designed to predict a meaningful assert statement to assess the correctness of the focal method. Ref. [401] leveraged a transformer model, first pre-trained on an English textual corpus and then semi-supervised trained on a substantial source code corpus, culminating in finetuning for generating assert statements in unit tests. CallMeMaybe [402] is a technique employing NLP to analyze Javadoc comments for identifying temporal constraints, thereby aiding test case generators in executing method call sequences that adhere to these constraints. TOGA [403] is a unified transformer-based neural method designed to deduce exceptional and assertion test oracles from the context of the focal method, effectively addressing units with unclear, absent documentation, or even missing implementations. ChatUniTest [404] is a ChatGPT-based automated unit test generation tool which creates adaptive focal contexts from projects, uses them in prompts for ChatGPT, and then validates and repairs the generated tests using rule-based and ChatGPT-based approaches. A3Test [405] is a DL-based approach that addresses the limitations of AthenaTest [399] by incorporating assertion knowledge with a mechanism to verify naming consistency and test signatures. CodaMosa [394] leverages an LLM to generate unit tests (oracles) to reach a designated program part. To generate test oracles for games, Neatest [406] navigates through a program's statements, constructing neural networks that manipulate the program for dependable statement coverage, essentially learning to strategically explore various code segments. One can also model the test generation problem as a completion problem [114].

Existing studies show that the quality of LLM-generated test cases (including oracles) still has significant room for improvement [407–410]. Ref. [411] pointed out that unit tests generated by ChatGPT still suffer from correctness issues, including diverse compilation errors and execution failures.

The challenge of implementing a "neural test oracle" is that program traces are too verbose for today's transformer architecture to digest. Neural networks fall short on finding long-range logical dependencies in the trace. The probabilistic nature of deep learning models also implies that they are imperfect test oracles. Nevertheless, we can always generate more test cases, and deep neural networks are not likely (though still possible) to make false-negative predictions on all test cases that trigger the same bug.

10.2.2 *Test input generation*

Even if the specification crisis is resolved, bug-finding is still a challenge. The number of test inputs is astronomically high, and we do not have the resources to examine all of them. This issue, which we refer to as the "search-space curse," is akin to finding needles in a haystack. Consequently, we only afford sampling useful test inputs that exercise diverse program behaviors (in hope of revealing bugs). A natural idea is to decompose the input space over its structure.

(1) Connecting the search space, based on the observation that useful (generally available, e.g., regression tests [412, 413]) inputs can be slightly mutated (modified) to obtain another useful input, which may manifest different program outcomes. Then, the input space can be decomposed into a graph, where vertices (inputs) are connected by mutation operators. The exploration procedure can be guided, e.g., by leaning towards test inputs that cover new code [414–416]. Machine learning is also useful in creating mutants [387, 395, 417].

(2) Partitioning the search space, and only sample representative test input(s) are selected in the equivalent classes. To alleviate the search-space curse, symbolic execution essentially merges all test inputs that share the same control-flow path. Then, for every path, we only care about whether it is reachable or not. The search is postponed, and a clever constraint solver may quickly draw a conclusion.

We see solid progress in improving how we decompose (and explore) the input space. Learned probability distributions over test inputs, traces, and execution logs are undoubtedly useful in whenever decisions should be made, e.g., in ranking the mutants [418, 419] or providing useful "golden" seeds [420, 421].

Deep learning-based generative models, particularly LLMs, exhibit exceptional performance in code generation tasks and are thus frequently employed in generating seeds for testing compilers or deep learning libraries. Deng et al. [420] presented TitanFuzz as the first technique to directly leveraging LLMs to generate seeds for fuzzing DL libraries. FuzzGPT [421] follows up the work by using LLMs to synthesize unusual programs for DL fuzzing. WhiteFox [422] uses LLMs to produce test programs with source-code information to fuzz compilers. Fuzz4All [423] is the first universal fuzzer capable of targeting a wide range of input languages and their various features. It capitalizes on LLMs for input generation and mutation, generating diverse and realistic inputs for any language. COMFORT [424] is another compiler fuzzing framework designed to identify bugs in JavaScript engines, utilizing advancements in deep learning-based language models for automatic JS test code generation. DeepSmith [425] uses generative models to generate tens of thousands of realistic programs, thereby accelerating the fuzzing process of compilers.

Machine-learning can also provide heuristic decisions for accelerating fuzzing. Specific work includes RegFuzz, a directed fuzzing approach that employs a linear regression model to predict seed effectiveness, thereby allocating more energy and fuzzing opportunities to efficient seeds [426]. HATAFL [427] utilizes pre-trained LLMs to construct grammars for protocol message types and assist in mutating messages for protocol fuzzing. SEAMFUZZ [417] learns effective mutation strategies by capturing the characteristics of individual seed inputs. Ref. [418] proposed a reinforcement learning-based hierarchical seed scheduling strategy for greybox fuzzing. RLF [428] models the fuzzing process of smart contracts as a Markov decision process and uses a specially designed reward system that considers both vulnerability and code coverage. Ref. [429] leveraged the Monte Carlo tree search (MCTS) to drive DL model generation, thus improved the quality of DL inference engines. Ref. [430] employed machine learning models and meta-heuristic search algorithms to strategically guide the fuzzing of actuators, aiming to maneuver a cyber-physical system (CPS) into various unsafe physical states. NeuFuzz [419] utilizes deep neural networks for intelligent seed selection in graybox fuzzing which learns vulnerability patterns in program paths. Some studies also utilize machine learning models to integrate fuzzing with symbolic execution. For example, Ref. [431] predicted the timing for switching between concrete and symbolic executions. Ref. [432] trained a neural network-based fuzzing policy on the dataset generated by symbolic execution, enabling the application of the learned policy to fuzz new programs. JOpFuzzer [433] learns the relationships between code features and optimization choices to direct seed mutation for JIT compiler fuzzing.

The challenge here is similar to incorporating deep learning within static analysis: an excessive amount of expensive queries may outweigh simply exercising more test cases.

Machine learning models generally perform better for domain-specific test input generation. For example, WebExplor [434] leverages a curiosity-driven reinforcement learning to generate high-quality action sequences (test cases) for web testing. FIGCPS [435] adopts deep reinforcement learning to interact with the CPS under test and effectively searches for failure-inducing input guided by rewards. Mobile applications provide a natural human interface, which can be effectively understood by machine-learning models. QTypist [436] utilizes a pre-trained LLM for generating text inputs based on the context of a mobile application's GUI. Ref. [437] proposed Deep GUI, which enhances black-box testing by utilizing deep learning to generate intelligent GUI inputs. AdaT [438] is a lightweight image-based approach that uses a deep learning model to dynamically adjust inter-event times in automated GUI testing based on the GUI's rendering state. Badge [439] is an approach for automated UI testing which uses a hierarchical multi-armed bandit model to prioritize UI events based on their exploration value and exploration diversity. Q-testing [440] employs a curiosity-driven strategy to focus on unexplored functionalities and uses a neural network as a state comparison module to efficiently differentiate between functional scenarios. Ref. [441] introduced Avgust, a system that automates the creation of usage-based tests for mobile apps by using neural models for image understanding.

Machine learning models are also capable of understanding "stories"—API call sequences. APICAD

[442] and NLPtoREST [443] are tools that enhance REST API testing by applying NLP techniques from API documents and specifications. Ref. [444] described an adaptive REST API testing technique that employs reinforcement learning to prioritize API operations and parameters, using dynamic analysis of request and response data and a sampling-based strategy to efficiently process API feedback.

10.3 Proving bug-freedom: formal verification

10.3.1 *Searching for needles in the haystack*

To the extreme end of testing, one can theoretically test over all possible inputs to verify that a program is bug-free (or to find all bugs). Exhaustive enumeration is the ultimate victim of the search-space curse. While we cannot leverage the small explanation hypothesis in verification (we cannot leave any corner case unchecked), the idea of search space decomposition still applies[5)]. For example, one can separately verify each program path, in which each verification is essentially a smaller search problem that can be solved by a constraint solver. Search spaces may have their own structures and pruning opportunities [445], which can be accelerated by machine learning [446].

On the other hand, path-based verification is not a silver bullet. Loops, even nested with simple control flow, post significant scalability challenges to a symbolic verifier. For example, a loop-based popcount implementation, which sequentially checks each bit of a 32-bit integer and increments a counter when the bit is set, consists of 2^{32} distinct program paths, and off-the-shelf symbolic execution engines fail to verify it. Sometimes we may rewrite the above function into one formula that can be recognized by a constraint solver [447], e.g.,

$$\text{popcount}(x) = \sum_{0 \leqslant i < 32} \texttt{AND}(\texttt{SHR}(x, i), 1),$$

to "offload" the 2^{32} paths to the constraint solver. Generally, we do not have this luck for most of the practical cases, and symbolic program verification is still limited to small programs. The complexity issues raised by control flows, pointers, memory allocation, libraries, and environments, are all challenges to verify practical programs [448].

Dynamic symbolic execution is a path-based program verification technique, and many learning-based techniques have emerged to ease the search-space curse in symbolic execution. Most studies focus on employing machine learning techniques to devise an optimized search strategy, thereby reducing the time and space overhead of path enumeration. Learch [449] utilizes a machine learning model to predict the potential of a program state, specifically its capability to maximize code coverage within a given time budget. Refs. [450–452] dynamically adapted search heuristics through a learning algorithm that develops new heuristics based on knowledge garnered during previous testing. There are also techniques to prune the search space. Homi [453] identifies promising states by a learning algorithm that continuously updates the probabilistic pruning strategy based on data accumulated during the testing process. Others optimize symbolic execution from different perspectives, including the prediction and optimization of path constraints [454, 455], as well as the fine-tuning of search parameters [456], and transformation of target code [457].

How can probabilistic techniques be useful in software verification, i.e., an exhaustive search over the input space? The use of a machine learning model must be sound, i.e., the checking results remain correct even under prediction errors. This problem remains open today.

10.3.2 *Providing a checkable proof*

Proving bug-freedom does not really require an exhaustive enumeration. Programs are rigorous mathematical objects: programs can be regarded as a function taking an input and producing an execution. One would always provide such a program with a logical proof that asserts all produced executions satisfying the specification, like we prove the correctness of any algorithm, e.g., bubble sort indeed gives a sorted array after $n-1$ iterations. The validity of such proofs can be checked by a proof assistant like Coq [458] or Isabelle/HOL [459, 460], to provide a certificate that the proof is correct [353, 354].

Fully automatic theorem proving is hard, even for short mathematical proofs. We could also search for the proof, carrying the search-space curse, and exploiting deep neural networks for heuristics. In contrast to programs that implement a human-world requirement, mechanical proofs are quite "unnatural", and

5) Decomposition is not limited to input spaces. One can also decompose the search space consisting of program states for model checking.

understanding a proof usually requires a careful examination of the proof stack, while code in mainstream programming languages reads much more like a natural language text. This implies that training deep learning models for creating proofs is considerably more challenging [461, 462], and learning to accelerate the search for proof tactics is still in the preliminary stage [463].

We argue that proofs for verifying software systems have a considerably different structure compared with proofs for mathematical theorems (the focus of today's research [464]), and the research community may have a paradigm shift: the core of a proof is invariants, which "summarizes" what happens in the intermediates of program execution, to form inductive hypotheses for machine-checkable proofs, and we identify strong patterns for program invariants. They can be done by humans, and we see opportunities that human work can be replaced by deep learning, e.g., machine-learning models can rank generated invariants [465].

An interesting observation is that we are trying to prove that a program satisfies a specification regardless of the input space size and the program execution length. However, the input space can be huge (or even infinite), and the program execution can be lengthy! Both the program and the proof seem much more concise compared with the set of all possible program execution traces, and proof checking can be done reasonably efficient. This phenomenon, which connects to the small explanation hypothesis, suggests that the program's execution space (inputs and their corresponding traces) follows a somehow simple structure that can be described algorithmically, and we might avoid a costly exhaustive enumeration. We speculate that practical software implementations are of the same magnitude of the minimum specification-satisfying implementation, like the "Kolmogorov complexity" of software. The implications of this phenomenon also remain open.

10.4 Case study: vulnerability detection

Following the above framework, this section further describes how deep learning techniques can improve the effectiveness and efficiency of finding a specific kind of bugs-security vulnerabilities, which can be exploited by attackers to gain unauthorized access, perform malicious activities, or steal sensitive data.

10.4.1 *Static vulnerability detection*

In modern times, the dominant model for software development revolves around library-based programming. The primary objective is to enhance development efficiency, minimize program complexity, and streamline operations such as development and maintenance. Program documentation plays a crucial role in providing a natural language description of the program, aiding users in comprehending and utilizing it effectively. Within a code base, an API serves as an interface that enables users to access its various functions. These APIs are subject to certain security constraints, such as manually releasing function return pointers, among others. These security constraints, known as security protocols, are documented by the code base developers within the program documentation. By documenting these protocols, developers offer users of the code base a valuable point of reference and guidance. During a call to the API, the developers must comply with the constraints of API calls. Otherwise, API misuse can occur, leading to serious software security issues, such as NULL pointer dereference, pointer use after free, and logical bugs.

In recent years, many researchers have used text analysis to find various security problems automatically, including access control configuration errors, wrong access requests, and logic flaws. For example, application developers provide privacy policies and notify users, but users cannot tell whether the application's natural behavior is consistent with their privacy policies. In response to this problem, Zimmeck et al. [466] proposed a systematic solution to automate the analysis of privacy policies to detect inconsistencies between them and application-specific behavior. Tools such as WHYPER [467] and AUTOCOG [468] examine whether Android applications correctly describe usage permissions in application descriptions. Similarly, Liu et al. [469] used text categorization and rule-based analysis to test for consistency between standard EU data protection regulations and applicable privacy policies.

The approaches above operate under the assumption that the documentation is accurate and does not contain errors. Consequently, if the code contains defects related to an API that lacks documentation, these defects cannot be detected. Rubio-González et al. [470] examined 52 file systems and discovered discrepancies between the error codes returned by functions and those recorded in the documentation. This investigation revealed over 1700 undocumented error codes. Tan et al. [471] utilized a series of rule

templates and a pre-trained decision tree to filter out comments from code that described API usage specifications. The program was then analyzed with user-provided function names (e.g., lock/unlock function names) to identify inconsistencies between the comments and the code. TCOMMENT [472] focuses on parameter values in Java comments and verifies the consistency of exceptions thrown under those values with the actual types of exceptions thrown by the code. Wen et al. [473] performed data mining on 1500 software code submissions and manually analyzed 500 to classify inconsistencies between code and comments. They also discussed the degree to which a code submission necessitates concurrent modification of comments, guiding for identifying and resolving inconsistencies between code and comments. Pandita et al. [474] employed machine learning models to filter out sentences in documents that describe API usage timing. They subsequently employed traditional natural language processing techniques to transform these sentences into first-order logical expressions. They further identified code defects that deviate from these specifications by constructing a semantic diagram of the document statements and inferring API usage timing specifications. Ren et al. [475] extracted an API declaration graph from semi-structured API declarations and derived usage specifications from the natural language descriptions within API documents. They then used this information to generate a knowledge graph encompassing API usage constraints, facilitating the detection of API misuse. Lv et al. [476] introduced advance, the first comprehensive API misuse detection tool, employing document analysis and natural language processing techniques.

Several studies summarize security protocols from many code usage examples for vulnerability detection. One intuitive way to do this is to automate the analysis of a large amount of code, then take a majority vote and use the most frequent code used as a reference for API usage. For example, APISan [477] extracts usage patterns from a large amount of code through parameter semantics, and causality and then extracts API usage references from usage patterns based on a majority vote. Thus, APISan no longer needs to define defect patterns manually. APEx [478] finds criteria for the API return value range on this branch based on fewer observations of code branch statements that handle errors and then infers the API's error specification based on the principle that most people are right and diagnosing defects for handling code snippets that do not follow the error definition for API return values. Like APEx, Ares [479] uses heuristic rules to identify error-handling blocks of code. A majority vote on the entry criteria for these blocks and a range merge results in an API error definition and diagnosing a defect by checking the return value of the API against a check that violates the error definition. APISan, APEx, and Ares all rely on the majority vote, but the majority vote is only sometimes right, which leads to the fallacy of the inferred specification itself.

Deep learning-based LLMs, represented by the transformer structure, are being applied to vulnerability detection tasks, mainly for static code analysis. Given a piece of code snippet, LLMs are asked through a question-answering dialogue whether it detects any vulnerabilities in the code and provides an explanation. However, LLMs still cannot handle various types of sensitive detection (including flow-sensitive, domain-sensitive, and context-sensitive). Therefore, it is necessary to fine-tune the large model or introduce additional knowledge through prompts to guide LLMs to gradually correct their analysis results. For specific field program vulnerability detection, such as smart contracts and shell scripts, due to their short length and low complexity, LLMs usually have a more accurate performance.

10.4.2 *Dynamic vulnerability detection*

In the field of vulnerability detection, fuzzy testing is an efficient dynamic detection technique. It explores and detects vulnerabilities in programs by continuously constructing unexpected abnormal data and providing them to the target program for execution while monitoring program execution anomalies [480]. During security testing, a large amount of data can be produced and further employed, such as test cases, execution traces, system states, software implementation specifications, and vulnerability descriptions. This information can be analyzed with deep learning techniques to improve fuzzy testing. For example, natural language processing can be used to understand text descriptions related to vulnerabilities, which can assist in generating test cases [481]. Through the strong fitting ability of deep learning models, mappings between program inputs and states can be accurately established, which can guide test case mutation [386]. Models trained and generated from existing test cases can automatically learn some input specifications to facilitate input generation [482]. At the same time, with the trained model, guidance and reasoning can be performed at a relatively low cost, which helps use deep learning techniques in real-time during fuzzy testing.

As for the objectives, the application of deep learning in fuzzy testing can be divided into two categories: reducing human processing overhead and increasing decision intelligence. The former includes reducing preparation work before testing, such as input model inference and mutation operation customization; the latter includes tasks such as seed file scheduling, mutation operation scheduling, and test case filtering.

Researchers have proposed to use deep learning algorithms to learn a generation model from existing test cases for input model inference or to enhance existing input models by automatically understanding auxiliary information such as input specifications using machine learning algorithms [483, 484]. On the other hand, researchers have used machine learning techniques to customize mutation operations for different programs. Angora [485] first uses taint analysis technology to obtain input positions that affect specific branches in a program. By converting branch conditions into input functions, Angora uses gradient descent to mutate corresponding input positions to generate test cases covering specified branches. This adaptive gradient-based mutation operation does not require manual setting and outperforms randomly using existing mutation operations in experiments.

Subsequently, researchers [386–388] further propose to use a single function to fit the input to the corresponding branch coverage of a program, followed by selecting input positions affecting specified branches based on the gradient information of the function and performing targeted mutations accordingly. According to the universal approximation theorem [486], neural networks have strong function fitting capabilities and can approximate any function. Additionally, they have good generalization ability and easy calculation of gradients. Therefore, researchers propose to use neural networks as a function to fit program behaviors, with the principle of larger gradients indicating greater impact on the corresponding edge as the basis for the automatic selection of mutation positions and adaptive mutation based on the size and direction of gradients. These techniques alleviate the cost of expert-designed mutation operations to some extent while also having adaptability for different programs.

Due to the inherent randomness of mutation operations, generated test cases may not meet specific test input generation standards for fuzzy testing. The main cost of a mutation-based fuzzy testing tool lies in the execution of test cases [487, 488]. If deep learning can be used to filter inputs before execution, it can reduce unnecessary running costs of target programs and improve the efficiency of fuzzy testing. Deep learning-based directed fuzzy testing, such as NEUZZ [386] and FuzzGuard [489], provides a novel approach to filtering redundant test cases. This approach collects a large number of test samples and uses whether they are reachable on a sensitive path as the classification standard to train deep learning models to classify and predict the reachability of future test samples. However, one limitation of this approach is whether the model can accurately understand the code logic. To overcome this limitation, we need to combine the semantics of the code itself so that the model can correctly understand the logic of the code and make accurate judgments about test cases.

10.5 Datasets

Considering the naturalness and complexity of modern software systems, it is not likely that anyone can train neural network models from scratch. Therefore, a mainstream approach to bug-finding is embracing pre-trained models for static analysis, dynamic testing, and formal verification [394, 420]. To train domain-specific machine-learning models, e.g., for GUI trace understanding or heuristic decision-making, datasets are needed [386, 400]. Alternatively, one may randomly select a subset of program execution results to serve as training datasets [449]. A unified, large-scale dataset has not been identified, hence it is not explored in this discussion.

10.6 Challenges and opportunities

10.6.1 *Challenges*

In pursuing effective bug-finding techniques, the challenges for either static analysis, dynamic testing, or formal verification, all point to the specification crisis and the search-space curse. Interestingly, both the crisis and the curse arise from the formal aspect of programs and the algorithmic nature of the bug-finding techniques.

Machine learning, particularly deep learning techniques, serves as a bridge between the algorithmic realm and the human realm. In the short term, even if today's deep learning models are still superficial and fall short on the long chain of logical inference, they are extremely good at digesting software artifacts. Whenever there is a need for heuristics, deep learning models have the potential to perform

significantly better than hand-crafted heuristics. The effectiveness of heuristics is multiplicative: among a huge number of decisions, even small improvements may result in magnitudes of significant efficiency improvements.

• To the probabilistic end, while deep learning models are generally replacing human beings in conducting simple, fast jobs like writing unit tests, the context-length of today's transformers makes it fundamentally limited in understanding large-scale systems—we still need an effective mechanism to simplify or reduce large systems such that neural networks can handle various analysis tasks on these systems.

• To the algorithmic end, we dream one day, a compiler[6)] is sufficiently powerful to automatically generate a proof for arbitrary assertions, even in natural language, or provide a counter-example. All programs that compile will automatically be "provably correct" to some extent. However, providing a proof, particularly for large-scale programs, is far beyond the capability of today's verifiers (model checkers) and automatic theorem provers.

10.6.2 *Opportunities*

Looking back at the academia's main theme of bug finding in the past decades, we see the thriving of both fully automatic bug-finding algorithms and end-to-end models. This is partly because for both parts we have available benchmarks for the push-button, reproducible evaluation. To go even further, perhaps we have overlooked the fact that we have developers, and Q/A teams, who are also "probabilistic" and whose performance varies day by day in the loop of software development. We could be more open to semi-automated techniques that invoke humans, exploit humans, and tolerate biases and errors. Such "humans" will eventually be replaced by a deep learning model on the availability of data.

• To the probabilistic end, it remains open and interesting whether there is an effective chain-of-thought to draw useful conclusions on software with the ability to understand both the software artifacts and analysis results from algorithmic tools.

• To the algorithmic end, we may hit a balance in the middle: in case the compiler is not powerful enough to prove an assertion, compilation will fail and the program should improve the code to make the program easier to verify. The Rust programming language is an early-stage attempt following this pathway [490].

11 Fault localization

Fault localization (FL) in software engineering is the process of identifying the specific code elements (i.e., statements or functions) in a program where defects occur [491,492]. Currently, due to the requirement of oracles, FL techniques mainly focus on functional bugs, which could be found by correctness specifications such as unit tests.

Traditionally, FL techniques involve manual debugging techniques [493], such as print statements, code inspection, and step-wise program execution. While these techniques have been widely used, they can be time-consuming, error-prone, and inefficient, particularly in large and complex software systems. To overcome these limitations, researchers from the software engineering community have explored automated FL techniques. These techniques aim to leverage various information sources, such as program execution traces, test cases, and code coverage information, to identify the locations in the code base that are most likely responsible for the observed failures.

Automated FL techniques can be broadly categorized into heuristic FL and statistical FL approaches. Heuristic FL techniques rely on pre-defined heuristic rules to locate the bugs that share similar buggy behavior. For example, some utilize dynamic dependencies (i.e., program slices) [494], some utilize stack traces [495], and some mutate the crisis values during test execution [496]. On the other hand, statistical FL techniques [497–499] build statistical models of buggy programs to analyze the relationships between code elements and failures. Most statistical FL techniques utilize program spetra [500–502], a kind of coverage information collected from test execution, to learn a model or a formula and then use it to rank the code elements. Some use other information sources, such as mutation analysis [503,504]. The former family is called spectrum-based FL (SBFL), while the latter is called mutation-based FL (MBFL).

6) The term "compiler" may no longer be appropriate at that time. Better to call it a "terminator" that kills programmers.

Recently, with the rapid development of deep learning, deep-learning-based fault localization (DLFL) techniques have shown the potential to automate and improve the accuracy of fault localization. By utilizing neural networks and sophisticated learning algorithms, these approaches can effectively identify fault-prone regions of code, prioritize debugging efforts, and accelerate the resolution of software defects. In the following, we divide deep learning-based fault localization into two categories: techniques for directly enhancing fault localization and techniques for augmenting input data for fault localization.

11.1 Fault localization approaches

Before the rise of deep learning, researchers had already attempted to establish a connection between the coverage information of test executions and the test results, in order to predict the location of faulty code. As early as 2011, Wong et al. [505] proposed back-propagation neural network models for defect localization. Subsequently, Wong et al. [506] made improvements by using a more complex radial basis function network.

To the best of our knowledge, Zheng et al. [507] and Zhang et al. [508] first proposed to adopt deep learning approaches into FL, in 2016 and 2017, respectively. They used a simple full-connection DNN that is trained against the same input of the SBFL. The primary evaluation results show the potential of DNN, which significantly outperforms the DStar approach, which was recognized as one of the most effective SBFL techniques at that time.

DeepFL, the milestone of DLFL, gained huge attention in the field of software engineering [509]. To address the limitations of traditional fault localization techniques, DeepFL utilizes various deep learning architectures, such as MLP and RNN, to capture different aspects of the software system and learn intricate patterns and correlations. By training on labeled data consisting of program execution traces, test cases, and associated fault information, DeepFL is able to make accurate predictions on the likelihood of specific code locations being responsible for failures. The complex DNN models enable DeepFL to handle multiple kinds of information (including SBFL features, MBFL features, and code-complexity features) and complex run-time traces.

Zhang et al. [510] proposed CNN-FL, which utilizes a tailored CNN model to process data for FL. First, CNNs are capable of effectively learning local features of the code, leading to more accurate fault localization. Second, CNN-FL can handle large-scale software systems and exhibits good scalability on extensive code repositories. Finally, this approach does not rely on specific feature extraction techniques but rather automatically learns the most relevant features through the network.

Li et al. [511] proposed DEEPRL4FL, a fault localization approach for buggy statements/methods. DEEPRL4FL exploits the image classification and pattern recognition capability of the CNN to apply to the code coverage matrix. CNNs are capable of learning the relationships among nearby cells via a small filter and can recognize the visual characteristic features to discriminate faulty and non-faulty statements/methods. To date, DEEPRL4FL still holds the best performance in terms of the Top-1 metric.

Lou et al. [512] proposed GRACE, a method-level FL approach based on the GNN. GRACE represents a program by a graph, where nodes represent code elements or tests, and edges represent coverage relationships or code structures. By leveraging the power of graph representations and learning latent features, the approach enhances fault localization accuracy and helps identify faulty code locations more effectively.

Qian et al. [513] utilized GCN to improve localization accuracy and proposed AGFL. AGFL represents abstract syntax trees by adjacent matrix and program tokens by word2vec, and then combines these features to further train GCN models. AGFL applies attention and GCN to classify whether an AST node is buggy.

Qian et al. [514] proposed GNet4FL, which is based on the GCN. To improve the performance, GNet4FL collects both static features based on code structure and dynamic features based on test results. It utilizes GraphSAGE to generate node representations for source codes and performs feature fusion for entities consisting of multiple nodes, preserving the graph's topological information. The entity representations are then fed into a multi-layer perceptron for training and ranking.

Zhang et al. [515] proposed CAN, a context-aware FL approach based on GNN. CAN represents the failure context by constructing a program dependency graph, which shows how a set of statements interact with each other. Then, CAN utilizes GNNs to analyze and incorporate the context (e.g., the dependencies among the statements) into suspiciousness evaluation.

Li et al. [516] proposed FixLocator, a DLFL approach that can locate faulty statements in one or multiple methods that need to be modified accordingly in the same fix. FixLocator utilizes dual-task learning with a method-level model and a statement-level model. Similarly, Dutta et al. [517] designed a hierarchical FL approach that uses two three-layer DNNs to first localize a function and then localize a statement.

Yu et al. [518] proposed CBCFL, a context-based cluster approach that aims to alleviate the influence of coincidental correctness (CC) tests. CBCFL uses the failure context, which includes statements that affect the output of failing tests, as input for cluster analysis to improve CC test identification. By changing the labels of CC tests to failing tests, CBCFL incorporates this context into fault localization techniques.

Li et al. [519] proposed a two-phase FL approach based on GNN. It extracts information from both the control flow graph and the data flow graph via GNN. The localization process is divided into two phases: (1) computing the suspiciousness score of each method and generating a ranking list, and (2) highlighting potential faulty locations inside a method using a fine-grained GNN.

Yosofvand et al. [520] proposed to treat the FL problem as a node classification problem, where the Gumtree algorithm is used to label nodes in graphs comparing buggy and fixed code. This paper uses GraphSAGE, a GNN model that handles big graphs with big neighborhoods well.

Wu et al. [521] proposed GMBFL which improves MBFL via GNN. Existing GNN-based approaches mainly focus on SBFL, while GMBFL first represents mutants and tests by a graph. The nodes of a graph are code elements, mutants, and tests. The edges are the mutation relationship between a code element and a mutant, the killed relationship between a mutant and a test, and the code structural relationship between two code elements of different hierarchies. Then GMBFL trains a gated graph attention neural network model to learn useful features from the graph.

In addition to test-based fault localization methods, there are also approaches that localize the bugs from change sets. BugPecker [522] is the first to encode the commits and bug reports into revision graphs. Ciborowska et al. [523] proposed to fine-tune the BERT model for locating the buggy change set.

To evaluate and compare the performance of different DNN models, Zhang et al. [524] processed a large-scale empirical study, which involves CNN, RNN, and multi-layer perceptron. The evaluation results show that CNNs perform the best in terms of identifying real faults.

11.2 Data augmentation and data processing approaches for fault localization

Data augmentation refers to the technique of artificially increasing the size and diversity of a dataset by applying various transformations or modifications to the original data. It plays a crucial role in improving the performance and robustness of deep learning models. In the context of fault localization, data augmentation can be used to enhance the effectiveness of deep learning-based approaches to fault localization.

Zhong and Mei [525] proposed CLAFA, which employs word embedding techniques to process the names within code. Then it compares program dependency graphs from buggy and fixed code to locate buggy nodes and extracts various graph features for training a classifier.

Zhang et al. [526] addressed the data imbalance problem in FL, which is caused by the fact that the number of failing test cases is much smaller than that of passing test cases. This paper employs test case resampling to representative localization models using deep learning, and improves the accuracy of DLFL.

Xie et al. [527] proposed Aeneas, which employs a revised principal component analysis (PCA) to generate a reduced feature space, resulting in a more concise representation of the original coverage matrix. This reduction in dimensionality not only improves the efficiency of data synthesis but also addresses the issue of imbalanced data. Aeneas tackles the imbalanced data problem by generating synthesized failing test cases using a conditional variational autoencoder (CVAE) from the reduced feature space.

Hu et al. [528] proposed Lamont, which uses revised linear discriminant analysis to reduce the dimensionality of the original coverage matrix and leverages synthetic minority over-sampling (SMOTE) to generate the synthesized failing tests.

Lei et al. [529] proposed BCL-FL, a data augmentation approach based on between-class learning. By leveraging the characteristics of real failed test cases, BCL-FL produces synthesized samples that closely resemble real test cases. The mixing ratio of original labels is used to assign a continuous value between 0 and 1 to the synthesized samples. This ensures a balanced input dataset for FL techniques.

Table 15 Frequently used evaluation metrics.

Metric name	Used times	Ref.
Top-N	12	[509, 511–514, 516, 518, 521–523, 530, 531]
MAR (mean average rank)	8	[509, 511–514, 521, 530, 531]
MFR (mean first rank)	8	[509, 511, 512, 518, 519, 521, 530, 531]
EXAM	6	[505–508, 510, 517, 519]
RImp (relative improvement)	6	[508, 510, 515, 518, 530, 531]
MAP (mean average precision)	2	[522, 523]
MRR (mean reciprocal rank)	2	[522, 523]
Hit-N (multi-defects)	1	[516]

Lei et al. [530] proposed CGAN4FL, a data augmentation approach to address the data imbalance problem in FL. CGAN4FL employs program dependencies to create a context that exposes the root causes of failures. Subsequently, CGAN4FL harnesses a generative adversarial network (GAN) to examine this failure-inducing context and generate test cases that belong to the minority class (i.e., failing test cases). Ultimately, CGAN4FL integrates the synthesized data into the existing test cases to obtain a balanced dataset suitable for FL.

Zhang et al. [531] proposed UNITE, which utilizes context information of trace data. UNITE combines three sources of information (i.e., a statement, a test case, and all test cases of a test suite), and then computes the influence relationship of the statements by program dependencies. In evaluation, UNITE significantly improves the state-of-the-art DLFL approaches.

Also, researchers try to improve DLFL from other aspects. Tian et al. [532] proposed to use DNNs to extract deep semantic changes to construct better mutants to improve FL performance. Zhang et al. [533] proposed to synthesize failing tests to improve the performance of DLFL.

11.3 Evaluation metrics

Evaluation metrics play a crucial role in assessing the performance of FL techniques. Similar to traditional FL approaches, DLFL studies adopt the commonly used metrics, shown as follows.

(1) Top-N. This metric measures the number of the model in identifying the correct faulty code element within the top-N ranked elements.

(2) MAR (mean average rank). MAR is the mean of the average rank.

(3) MFR (mean first rank). MFR is the mean of the first faulty statement's rank of all faults using a localization approach.

(4) EXAM. EXAM stands for expected maximum fault localization, which measures the expected rank of the first correct fault location in a ranked list of code elements.

(5) RImp (relative improvement). RImp is to compare the total number of statements that need to be examined to find all faults using one FL approach versus the number that need to be examined without using the FL approach.

(6) MAP (mean average rank). MAP first computes the average precision for each fault, and then calculates the mean of the average precision.

(7) MRR (mean reciprocal rank). MRR metric calculates the mean of the reciprocal position at which the first relevant method is found.

(8) Hit-N. Hit-N is a metric designed for multi-defects, which measures the number of bugs that the predicted set contains at least N faulty statements.

Table 15 [505–519,521–523,530,531] summarizes the commonly used metrics. The most popular metrics are Top-N, which is used 12 times. The second most common metrics are MAR and MFR, which are used 8 times. MAP and MRR are used in only two studies, while the Hit-N metric is used only once to measure the multi-defect FL approaches.

11.4 Datasets

The datasets used in the DLFL field are largely similar to those in the program repair and other related fields. Current approaches primarily utilize Java programs for evaluation, with a smaller portion using C language programs.

In Java, the Defects4J [534] benchmark is the most extensively used. Defects4J is a widely recognized benchmark dataset in the FL field. This dataset is well-maintained and continues to be updated. Early FL

techniques use Defects4J v1.2, primarily evaluating against six projects within it. Recent work employs Defects4J v2.0, which includes more open-source projects. Additionally, the BEARS [535] benchmark and nanoxml from the SIR [536] dataset are also employed.

In C, some C programs from the SIR dataset, as well as the ManyBugs [537] dataset, are more frequently used. Existing DLFL research has utilized programs like space from the SIR dataset, as well as python, gzip, libtiff, and others from the ManyBugs dataset.

11.5 Challenges and opportunities

This subsection summarizes the challenges and highlights opportunities for future work in fault localization.

11.5.1 *Challenges*

Deep learning-based fault localization techniques have shown great potential in improving the accuracy and effectiveness of fault localization. However, they also face the following challenges that need to be addressed to fully exploit their capabilities.

• **The risk of overfitting.** The current existing DLFL approaches are mainly evaluated on the popular benchmark Defects4J [534], which consists of Java projects and hundreds of bugs and is used as an important dataset in debugging-related fields. Most papers use Defects4J v1.2, which only contains 395 bugs from six Java projects. Worse still, some approaches discard the Closure project and only use 224 of the bugs. This leads to the risk that the conclusion of most existing methods might be overfitting to this particular dataset or the Java programming language. Currently, there exist a few studies that target Python or JavaScript programs and we suggest evaluating the novel approaches across multiple languages and benchmarks.

• **Inadequate availability of high-quality labeled data.** The current research is limited to the Defects4J dataset due to a shortage of high-quality labeled data. This scarcity not only affects the evaluation of DLFL approaches but also impacts the training of deep learning models. In addition, existing FL data suffer from imbalance, i.e., the data from passing tests overwhelms the data from failing tests. This characteristic poses challenges for many learning-based approaches. Deep learning models require a large amount of accurately labeled data for training, which can be difficult to obtain, especially when dealing with real-world bugs across different languages.

• **Interpretability of deep learning models.** The lack of interpretability has long been a challenge for deep learning and similarly affects fault localization based on deep learning, which makes it difficult to analyze the results of fault localization, specifically the relationship between failing tests, the code elements, and the traces.

• **Occasionally low efficiency.** The current fault localization systems struggle to achieve real-time fault localization. This is partly due to their reliance on test runs to collect trace information and partly because large-scale deep-learning models could slow training and prediction.

11.5.2 *Opportunities*

Despite these challenges, deep learning-based fault localization techniques present the following promising opportunities.

• **Data augmentation.** As discussed in current challenges, existing DLFL approaches are suffering from low-quality data that are imbalanced or of limited scale. Thus data augmentation methods offer avenues for addressing the challenge.

• **Enhancing multiple FL.** Existing DLFL approaches generally assume that there is only one bug in the target project. However, a real-world buggy project often contains multiple bugs. The interaction between these bugs makes it more difficult to train and predict using traditional learning approaches, highlighting potential opportunities for improvement in this direction.

• **The integration of domain-specific knowledge.** FL relies on code syntax structures, code semantics, and execution traces, which require a deep understanding of the underlying programming languages and software engineering principles. The integration of domain-specific knowledge, such as code smells, development experience, and debugging heuristics, can significantly improve the effectiveness of FL systems. Especially, representing and learning context information of the buggy code is promising,

because it enables the models to better understand the specific scenarios in which faults occur and how they relate to the text execution.

12 Program repair

Program repair refers to the process of identifying and fixing software defects in programs. Program repair requires a large number of time costs and human resources from the project development team [538]. Due to the growing demand for efficient software maintenance, automatic program repair techniques have emerged as a solution [539].

Automatic program repair allows developers to automate (or nearly automate) defect detection and correction. This makes program repair efficient, reliable, and cost-effective [540]. In recent years, the development and implementation of automatic program repair techniques have been widely recognized by the software development community [541]. The success of deep learning in recent years makes it a promising approach for locating and repairing buggy programs. The family of deep learning approaches to program repair develops and applies deep learning techniques to identify software defects and generate patches [542]. Deep learning models have been widely applied to repair a wide range of program errors. We divide existing work into three categories, including compilation error repair, runtime error repair, and specific domain error repair.

12.1 Compilation error repair

A compilation error is an error that can be detected during compilation. Programs with compilation errors fail in program compilation or linking; meanwhile, programs with compilation errors cannot be directly handled by program analysis tools [543–545].

Compilation errors prevent source code from being transformed into executable machine code during compilation. These compilation errors contain issues like type errors, undefined variables, syntax errors, and other errors that violate programming language standards. Studies in the early stage have proposed error-correcting parsers to achieve program repair of faulty code [546–549]. Recently, researchers have explored the advantage of deep learning techniques for automatically fixing compilation errors. These techniques use deep neural networks to analyze extensive repair datasets and recommend precise code fixes [541].

To avoid misleading messages returned by compilers, Gupta et al. [550] trained a deep neural network, named DeepFix, to identify incorrect locations in source code and provide the corresponding repaired statements. They collected 6971 erroneous C programs from code written by students for 93 programming tasks and found that DeepFix is able to completely repair 27% and partially repair 19% of these erroneous programs. Bhatia et al. [551] presented RLAssist, a technique that combines RNNs with constraint-based reasoning to fix programming assignments with syntax errors. Ahmed et al. [552] proposed an end-to-end system, called Tracer, for fixing code with multiple errors. In the same year of 2018, Santos et al. [553] proposed an approach to correcting syntax errors using n-gram and LSTM models. They evaluated the approach on the BlackBox dataset [554].

Mesbah et al. [555] proposed DeepDelta, which converts an AST into a domain-specific language before feeding the converted tree into an NMT network. DeepDelta achieves a success rate of 50% in generating correct repairs for missing symbols and mismatched method signatures. Gupta et al. [556] proposed a programming language correction framework that uses reinforcement learning to assist novice programmers with syntactic errors. This framework outperforms their previous work DeepFix [550] in 2017. Wu et al. [557] trained a deep learning model on the DeepFix dataset. The model, called graph-based grammar fix, combines token sequences and graph structures based on ASTs to predict the error position and generate correct tokens. To integrate program-feedback graphs and a self-supervised learning framework, Yasunaga et al. [558] proposed DrRepair, a program repair technique based on graph attention networks.

With the rapid development of deep learning, the effectiveness of automatic program repair has been further improved. Hajipour et al. [559] proposed a generative model for fixing compilation errors in C programs. Their model learned a distribution over potential fixes and encouraged diversity over a semantic embedding space. Allamanis et al. [368] proposed to train a selector concurrently with the model that locates and fixes errors in source code. The selector was utilized to automatically generate faulty code, which is used to enrich the training set for the original model. To make the generated

faulty code closely resemble real-world error scenarios, Yasunaga et al. [560] proposed BIFI, an iterative training approach for fixing syntax errors. They trained two models, including the breaker and the fixer, in an iterative manner. The breaker creates faulty code that closely resembles real-world errors while the fixer converts the faulty code into the correct version. Ahmed et al. [561] presented a lenient parser for imperfect code (i.e., the union of fragmentary code, incomplete code, and ill-formed code) and proposed an indirectly supervised approach for training the parser. To fix the parser errors in programming languages, Sakkas et al. [562] proposed a language-agnostic neurosymbolic technique in 2022. Their technique, called Seq2Parse, combined symbolic error correcting parsers and neural networks. Seq2Parse was evaluated on 1.1 million Python programs. Li et al. [563] proposed TransRepair, which utilizes a Transformer-based neural network via considering both the context of erroneous code and the feedback by compilers to fix compilation errors in C programs. Ahmed et al. [564] introduced SynShine, a three-stage approach that combines the feedback of Java compiler with models based on a robustly optimized BERT (Roberta) [565]. They indicated that SynShine achieves 75% of effectiveness in fixing the single-line errors of the code in the Blackbox dataset [554].

12.2 Runtime error repair

Runtime errors, also known as dynamic errors, occur when the program executes. Runtime errors result in crashes or incorrect behaviors during program execution. Automatic program repair techniques generate patches for faulty code based on test cases, crashes, references, contracts, etc. [566].

Approaches to runtime error repair can help save time costs and effort in software development. These repair approaches can be roughly divided into search-based repair [567,568], constraint-based repair [569–571], template-based repair [572–574], and learning-based repair [575–577]. With the support of deep neural networks, learning-based repair approaches can generate high-quality code patches. The framework of repair approaches based on deep learning for runtime errors generally consists of five steps: fault location, data pre-processing, feature extraction, patch generation, and patch selection.

Long et al. [578] proposed a hybrid repair system, called Prophet. Prophet integrates a probabilistic model trained on the benchmark presented by Goues et al. [579] and ranks code patches to fix runtime errors. Tufano et al. [580] investigated the possibility of learning patches through the translation model based on neural networks. Their model achieves a prediction rate of 9% via training on the GitHub repositories. Sun et al. [581] developed a sequence-to-sequence service based on the attention techniques. White et al. [575] presented a deep learning model, DeepRepair, to produce patches that cannot be searched by redundancy-based techniques. Tufano et al. [234] explored the potential of an NMT model to create code changes made by developers while adding pull requests.

Researchers have been exploring new ways of tackling issues in program repair. Ding et al. [582] conducted a study that investigates the differences between sequence-to-sequence models and translation models for program repair. They proposed a strategy based on the empirical findings and development knowledge in patch generation. Yang et al. [583] proposed an automatic model to locate faults and generate patches. They scored the ranks between bug reports and source code based on CNNs and auto-encoder. Then, they created patches through the Seq-GAN algorithm [584]. Lutellier et al. [585] proposed CoCoNuT, which leverages the strength of CNNs and NMTs to achieve multi-language repair for Java, C, Python, and JavaScript programs. Li et al. [21] proposed DLFix, a dual-level deep learning model designed to address the limitations of learning-based program repair. The first layer of DLFix is a tree-based RNN model that captures the context of fixed code while the second layer utilizes the output from the first layer to learn and apply code patches. The validation experiments are conducted on the datasets of Defects4J[7)] [586] and Bugs.jar[8)] [587]. Tian et al. [588] explored different deep learning-based approaches. Their experimental results show that embeddings from pre-trained and re-trained neural networks are beneficial to reason and generate correct patches. Dinella et al. [589] proposed Hoppity, a Javascript-targeted automatic repair model. This learning-based model focuses on graph structures of faulty code.

Tang et al. [590] proposed a grammar-based approach to syntax correct patch generation. Huang et al. [591] discussed the use of a pyramid encoder in seq2seq models to reduce computational and memory costs while achieving a similar repair rate to their non-pyramid counterparts. Their study focuses on the automatic correction of logic errors. Jiang et al. [577] presented a three-stage based NMT model, called

7) https://github.com/rjust/defects4j.

8) https://github.com/bugs-dot-jar/bugs-dot-jar.

CURE. They first pre-trained a programming language model to incorporate the real-world coding style and then proposed a technique to expand the search space. They integrated subword tokenization, a technique used in natural language processing that splits words into smaller units called subword tokens, to generate precise patches. Based on the LSTM network and the bidirectional recurrent neural network, Rahman et al. [592] presented BiLSTM to identify and classify the faulty code and generate possible fixes. Chen et al. [576] proposed SequenceR, a sequence-to-sequence-based deep learning system. The model adapts a copy mechanism to deal with large-scale code instances. Berabi et al. [593] proposed TFix, a pre-trained and fine-tuned language model that generates improved patches for JavaScript programs. Tang et al. [594] proposed a graph-to-sequence learning model called GrasP, based on code structure. Szalontai et al. [595] focused on the uncommon parts of Python code. They constructed a neural network model to classify and generate replacement code snippets.

Li et al. [596] proposed DEAR, a deep learning-based repair approach, which attempts to locate multi-bugs and generate patches to fix multiple errors in a single program. Xu et al. [597] presented M3V, an approach that integrates LSTM and GNN models for fault location and patch generation. Their approach primarily focuses on null-pointer exceptions and out-of-bounds exceptions in Java programs. Meng et al. [598] introduced Transfer, a technique that mines deep semantic information. Their model integrates multiple features to rank patches, including semantic-based features, spectrum-based features, and mutation-based features. Kim et al. [599] focused on locations of software defects. They improved the accuracy of deep learning approaches in program repair via using genetic algorithms to obtain the precise defective code. Wardat et al. [600] proposed DeepDiagnosis, which focuses on defects in deep learning applications. Using well-trained models, DeepDiagnosis classifies faults into eight categories and provides feasible patches for each category of faults. Yao et al. [601] proposed Bug-Transformer, a context-based deep learning model. Bug-Transformer retains the contextual information of the faulty code and uses a transformer model for training. Yan et al. [602] proposed Crex, a transfer-learning-based technique [603] for validating C program patch correctness. Their model aims to learn the code semantic similarity to improve the accuracy validation. Chakraborty et al. [604] proposed CODIT, a tree-based deep learning model for generating code change suggestions. Ye et al. [605] proposed RewardRepair, an NMT-based model with fine-tuned loss functions. RewardRepair incorporates compilation information and test cases into the calculation of the loss functions. Ye et al. [606] also proposed ODS, a deep learning system for predicting patch correctness. They extracted code features in ASTs from patches and buggy code. Then, they employed supervised learning to determine the correctness of patches. Another technique proposed by Ye et al. [607] is SelfAPR, a self-supervised model based on employing perturbation-generated data for patch generation. Xia et al. [608] proposed AlphaRepair, a CodeBERT-based model for code generation. AlphaRepair uses zero-shot learning techniques rather than training models with historical erroneous and corrected code. Kim et al. [609] investigated the efficacy of deep learning-based repair techniques for Java-to-Kotlin conversion programs. Their technique enhances the defect fixing performance by applying transfer learning techniques. Tian et al. [610] proposed BATS, a learning-based model designed to predict the correctness of patches. BATS is an unsupervised model that repairs faulty programs by detecting program behavior during failed test cases. Yuan et al. [611] proposed CIRCLE, a T5-based model for patch generation. CIRCLE employs continual learning techniques and is able to work on multiple programming languages. They also designed the Prompt feature to make it capable of understanding natural language commands. Chen et al. [612] trained an iterative model, which aims to learn from generated patches and test execution.

12.3 Specific domain error repair

Deep learning has also been applied to domain-specific tasks related to automated program repair. The specific domain repair refers to the application of specialized knowledge or techniques from a specific domain to improve the effectiveness and efficiency of program repair.

Test repair aims at fixing errors in test cases that are unusable due to software updates. Stocco et al. [613] fixed web test cases by analyzing visual features based on an image processing approach. Pan et al. [614] presented Meter, a computer vision-based technique for fixing test cases in the graphical user interface (GUI) of mobile applications.

Build scripts are crucial components in the automatic building of software systems. Program repair for build scripts refers to the process of automatically detecting and fixing faults in build scripts [615].

Table 16 Evaluation datasets for deep learning-based program repair tools.

Year	Dataset	Language	Type	Size	URL
2014	Defect4J	Java	Runtime error	835	https://github.com/rjust/defecrrorts4j
2014	Blackbox	Java	Hybrid error	Over 2000000000	http://www.cs.kent.ac.uk/~nccb/blackbox
2016	IntroClassJava	Java	Runtime error	297	https://github.com/Spirals-Team/IntroClassJava
2017	DroixBench	Java	Hybrid error	24	https://droix2017.github.io
2018	Bugs.jar	Java	Runtime error	1158	https://github.com/bugs-dot-jar/bugs-dot-jar
2018	Santos et al.	Java	Compilation error	1715313	https://archive.org/details/sensibility-saner2018
2019	BugSwarm	Java	Build Failure	3091	https://github.com/BugSwarm/bugswarm
2019	Ponta	Java	Vulnerability	1068	https://github.com/SAP/project-kb
2022	Vul4J	Java	Vulnerability	79	https://github.com/tuhh-softsec/vul4j
2015	ManyBugs	C	Hybrid error	185	https://repairbenchmarks.cs.umass.edu
2015	IntroClass	C	Runtime error	998	https://repairbenchmarks.cs.umass.edu
2016	Prutor	C	Compilation error	6971	https://www.cse.iitk.ac.in/users/karkare/prutor
2017	DBGBENCH	C	Runtime error	27	https://dbgbench.github.io
2017	CodeFlaws	C	Hybrid error	3902	https://codeflaws.github.io
2018	TRACER	C	Compilation error	16985	https://github.com/umairzahmed/tracer
2019	TEGCER	C	Compilation error	15579	https://github.com/umairzahmed/tegcer
2020	Big-Vul	C/C++	Vulnerability	3754	https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset
2021	CVEfixes	C/C++	Vulnerability	5495	https://github.com/secureIT-project/CVEfixes
2021	BugsCpp	C/C++	Runtime error	215	https://github.com/Suresoft-GLaDOS/bugscp
2017	QuixBugs	Java/Python	Runtime error	40	https://github.com/jkoppel/Quixbugs
2017	HireBuild	Build Script	Build failure	175	https://sites.google.com/site/buildfix2017
2019	BugsJS	JavaScript	Runtime error	453	https://bugsjs.github.io
2019	Defexts	Kotlin/Groovy/etc.	Runtime error	654	https://github.com/ProdigyXable/defexts
2019	Refactory	Python	Hybrid error	1783	https://github.com/githubhuyang/refactory
2020	TANDEM	Java/C/SQL/etc.	Hybrid error	125	https://github.com/belene/tandem
2022	Ring	6 Languages	Hybrid error	1200	https://github.com/microsoft/prose-benchmarks
2022	CrossVul	40 Languages	Vulnerability	5131	https://zenodo.org/record/4734050

Hassan et al. [616] built a benchmark of 175 build failures and the relevant remedies for Gradle[9]. They presented HireBuild as the first model of patch generation for fixing build scripts in Gradle. Based on the previous work of HireBuild, Lou et al. [617] extended the benchmark from Top-1000 GitHub projects and proposed an improved search model, which generates patches according to current test projects and external resources. Recently, Loriot et al. [618] proposed Styler, an approach based on the LSTM neural network to generate patches for code violations against format rules.

Software vulnerability refers to security weaknesses that can compromise the integrity and availability of software systems. Ma et al. [619] developed a tool called VuRLE that automatically locates and fixes vulnerabilities in source code. Their tool generates repair templates and selects patches based on the ASTs of source code. Harer et al. [620] proposed a technique based on GANs that generates corrupted data and uses correct-incorrect pairs to train an NMT model. Zhou et al. [621] proposed SPVF, which combines ASTs, security properties, and the attention mechanism into an integrated neural network for both C/C++ and Python programs. Huang et al. [622] attempted to fix vulnerabilities by leveraging pre-trained large language models. They reported an accuracy rate of 95.47% for single-line errors and 90.06% for multiple-line errors. Chen et al. [623] hypothesized that there could be a correlation between program repairing and vulnerability fixing. They proposed VRepair, a transfer learning model designed to solve security vulnerabilities in C programs with limited data. Due to the increasing number of reported vulnerabilities, Chi et al. [624] developed SeqTrans, an NMT-based tool that automatically fixed vulnerabilities. Their approach involves learning from historical patches and contextual features of source code.

12.4 Datasets

The diverse range of programming languages leads to the creation of various types of datasets for program repair. In light of existing research, there are numerous available datasets that are specifically tailored for the application of automatic program repair tools.

The evaluation datasets are listed in Table 16. We divide the datasets into three categories according to the programming languages: in Java, in C/C++, and in other programming languages. We briefly introduce typical datasets as follows. Prutor [625] is a tutorial system, which helps students solve programming problems. Prutor is used in the introductory programming course in IIT Kanpur. Thus, Prutor collects many pieces of C code, including the buggy code and the correct code.

Blackbox [554] is a project since 2013. It collects data from the BlueJ IDE[10], a tutorial environment for JAVA learners. After five years of collection, the Blackbox dataset has amassed over two terabytes

9) Gradle, https://gradle.org/.

10) BlueJ IDE, https://www.bluej.org/.

of data [626]. In 2018, Santos et al. [553] refined the Blackbox dataset to evaluate their sensibility approach. They selected 1715312 program pairs of previous and current versions from the Blackbox dataset, including 57.39% pairs with one syntax error and 14.48% with two syntax errors.

Defects4J [534] is one of the most widely-used benchmark in program repair [585, 586, 605, 627]. The latest version of Defects4J is a collection of 835 reproducible bugs from 17 open-source Java projects. Each bug in Defects4J corresponds to a set of test cases that can trigger the bugs. GrowingBugs[11)] is highly similar to Defects4J in that bug-irrelevant changes in bug-fixing commits have been excluded from the patches. The current version of GrowingBugs contains 1008 real-world bugs collected from open-source applications. The only difference between GrowingBugs and Defects4J is that the latter excludes bug-irrelevant changes from bug-fixing commits manually, whereas the former does it automatically by BugBuilder [628, 629].

Vul4J [630] is a dataset of reproducible Java vulnerabilities. All vulnerabilities in Vul4J correspond to human patches and proof-of-vulnerability (POV) test cases. CrossVul [631] contains vulnerabilities over 40 programming languages. Each file corresponds to an ID of common vulnerability exposures (CVEs) and its source repository. This dataset also contains commit messages, which may serve as human-written patches.

12.5 Challenges and opportunities

We present challenges and opportunities in deep learning for program repair in this section.

12.5.1 *Challenges*

There are several challenges in deep learning for program repair. These challenges reveal that there is still a long way to go in applying deep learning techniques to automatic program repair.

Training data. Deep learning requires a considerable number of data to train learnable models. In program repair, labeled data of buggy code and patches are limited. Obtaining high-quality erroneous and patched code is a severe challenge due to the limited datasets of program repair. Another challenge is to determine how to use buggy code and patches in model training. A deep learning model can involve the location of buggy code, its specific type, its context, its syntax, and its semantics. The mapping between buggy code and patched code is a key step in training models in program repair. To date, there is no theoretical analysis for such a challenge.

Model interpretability. The lack of interpretability for deep learning models makes it difficult to ensure the correctness of the generated patches. The original goal of program repair is to assist the real-world developers. Thus, it may be difficult to persuade developers to use deep learning-based program repair in real-world development.

Self-validation of data. Models of deep learning are built on training data. If the training data contains errors, these errors may propagate to the generated patches. This hurts the effectiveness of automatic program repair. Developers are unable to confirm the reliability of patches generated by deep learning.

12.5.2 *Opportunities*

Despite the challenges, we identify the following opportunities in the field of deep learning for program repair.

Data collection and generation. Automatic data collection and generation can benefit program repair. Researchers can train deep learning models to handle a wide range of programming scenarios based on high-quality data. Additionally, data generation methods such as program synthesis may be able to enrich the existing data.

Hybrid approaches. Hybrid approaches that combine deep learning with other existing techniques, like symbolic reasoning, template-based repair, and rule-based search, have shown promising results in program repair. These hybrid approaches can improve the effectiveness and efficiency of current program repair tools by integrating the strength of multiple techniques.

Model optimization. Researchers are able to explore ways to optimize deep learning models for program repair. Such optimization contains model re-sizing knowledge distillation, neural architecture

11) Growingbugs, 2024. https://github.com/jiangyanjie/GrowingBugs.

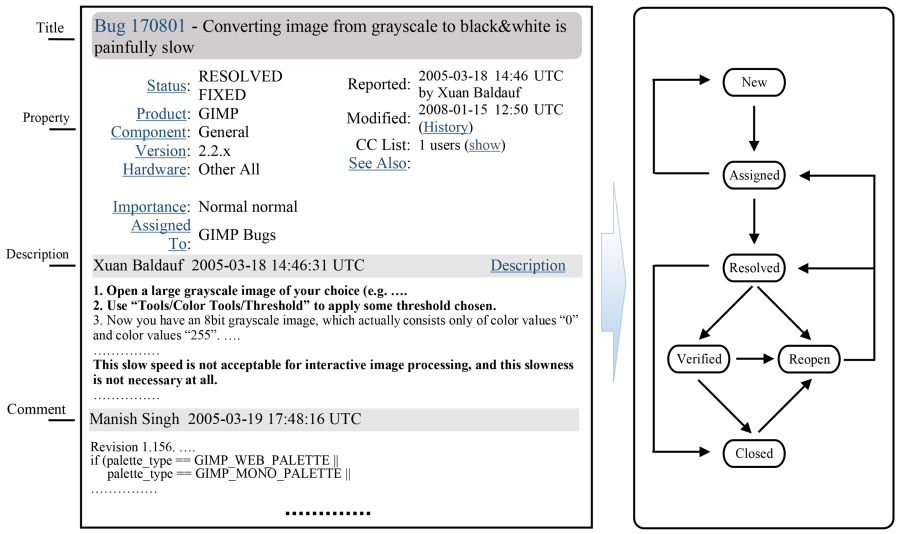

Figure 6 (Color online) Example of a bug report and its lifecycle.

search, and so on. Model optimization can help researchers transfer available knowledge from a large and complex model to new-coming or unknown scenarios.

In conclusion, the opportunities in deep learning for program repair are vast and exciting. Continuous research and development in this field may lead to more advanced and effective approaches for program repair.

13 Bug report management

Given the intricate nature of software systems, bugs are unavoidable. A previous study shows that a collection of 606 software failures reported in 2017 has affected approximately 3.7 billion users and caused financial losses of $1.7 trillion. Therefore, efficiently fixing bugs becomes a fundamental step for software projects [632].

To fix bugs, most software projects use bug reports to manage bugs. A bug report is used to record specific details of the bug such as the title, the descriptions (e.g., reproducible steps and stack traces), the property fields, and the comments, which assist developers in identifying and rectifying buggy code [633]. Figure 6 is an example of a bug report from the Gnome project with bug report ID 170801[12]. In this example, Xuan Baldauf issued a bug report titled "converting image from grayscale to black&white is painfully slow", and provided detailed information on how to reproduce the bug in the description field (e.g., "Open a large $\cdots$ at all"). Additionally, the reporter specified the property of the bug, such as the product and component containing the bug. After the submission, other participants contributed comments in the comment field. The bug report and its associated comments are typically utilized by software developers to facilitate subsequent software activities.

The general activities to manage bug reports for bug fixing involve six steps [634] (as shown in Figure 6). Upon the submission of a new bug report, its initial status is labeled as new. Subsequently, the bug report undergoes manual scrutiny to verify its validity and prevent duplication. Once the bug is confirmed, the bug report is assigned to a developer. The developer then proceeds to fix the bug and sets its status to resolved after fixing the bug. For resolved bugs, additional developers conduct code review to verify the bug resolution. If the bug resolution is verified, developers can close this bug; otherwise, they label the bug report as reopen, which means that the bug is not successfully fixed. In a typical bug report management (BRM) cycle, these processes are mainly carried out manually by software developers.

However, with the exponential growth in the number of bug reports submitted to many large-scale software projects, the size and complexity of bug report repositories increase significantly. It becomes a laborious task to manually manage bug reports, which consumes a significant amount of time for developers [635]. For instance, the Eclipse project received a total of 552334 bug reports in recent 20 years (from January 2003 to December 2022), averaging around 76 new bug reports daily. Additionally, given

12) https://bugzilla.gnome.org/show_bug.cgi?id=170801.

Table 17 BRM tasks addressed by deep learning techniques.

#	Task	Description	Total
1	Bug report refinement	Generate a high-quality bug report by enriching/modifying an existing one.	1
2	Duplicate bug detection	Detect duplicate or similar bug reports in bug repositories.	8
3	Bug assignment	Recommend the most appropriate developers to fix the bug.	5
4	Bug severity/priority prediction	Predict the severity/priority of a bug report before fixing.	2
5	Bug fixing time prediction	Predict how long it will take to fix the bug.	1
6	Bug report summarization	Summarize a bug report into a much shorter form.	6
7	Bug localization	Locate relevant source code files or methods that possibly contain the bug based on the bug report.	16
8	Bug-Commit linking	Link bug reports with bug fixing commits or bug inducing commits.	3

the varied reporting experience of bug reporters [633], not all bug reports provide sufficient information to assist developers with bug fixing. Developers have to spend tremendous time understanding and managing these bug reports.

To tackle this challenge, many studies propose to utilize text mining and ML [632,636,637] to automate BRM. These studies employ classical ML techniques such as Naïve Bayes, random forest, support vector machine, and k-nearest neighbors to automatically detect duplicate bug reports, triager bug reports, and identify reopened bug reports.

However, the effectiveness of these classical ML techniques is limited [523, 638]. Therefore, recent studies have utilized DL techniques to enhance the automation of BRM. DL uses its powerful feature engineering capability to deeply analyze bug reports. The exponential growth number of bug reports also becomes an important source to effectively train DL models for different classification and regression tasks. For example, Li et al. [639] introduced an autoencoder architecture that extracts multiple features from bug reports to generate a bug report summary. Fang et al. [640] proposed the use of RNN to capture sequential information in bug reports and source code for bug localization. These studies provide compelling evidence of the effectiveness of DL in improving automated BRM. In this section, we survey the BRM tasks improved by DL techniques and discuss the challenges and opportunities in this area.

Table 17 summarizes the main BRM tasks addressed by DL techniques, including the task name, the task description, and the number of related papers for each task. In total, researchers have applied DL techniques for eight tasks in BRM.

13.1 Bug report refinement

When a bug report is submitted, bug report refinement aims to refine the bug report and improve its quality for better understanding. For example, we can enrich the bug report with more information collected from similar bug reports or reformulate the content of the bug report. DL can be used to improve the refinement process. Zhou et al. [641] reformulated a bug report as a query representation by leveraging multi-level embeddings through CNNs with the self-attention mechanism. The reformulation is used to help developers understand the bug report and retrieve similar bug reports in the repository.

13.2 Duplicate bug detection

Duplicate bug report detection aims to identify whether a given bug report is a duplicate of an existing one. The detection not only eliminates redundant data but also provides additional insights into the reported issues. To automate this task, different types of deep neural networks have been used to train duplicate bug report detection models, such as simple DNN, CNN, RNN, and LSTM [642–644]. For example, Deshmukh et al. [645] used CNN and LSTM to retrieve duplicate bug reports, achieving a precision value of 90% and a recall value of 80%.

The main goal of these DL techniques is to represent the content of bug reports for distinguishing them. Evaluation demonstrates that DL techniques such as DNN-based 2D embedding and BERT show higher performance in detecting duplicate bug reports than traditional language models (e.g., n-gram-based model) [646–648]. Despite the promising performance, a recent study [638] analyzed the effectiveness of state-of-the-art duplicate bug report detection approaches in a realistic setting using industrial datasets. The results of the study show that a simple technique already adopted in practice can achieve comparable results as a recently proposed research tool.

13.3 Bug assignment

A submitted bug report should be assigned to a developer for resolution. This assignment process is known as bug assignment (or triaging), which can be automated to reduce the manual effort. An

existing empirical study shows that DL techniques outperform the other traditional ML techniques for this task [649]. For bug assignment, DL models are employed to extract word sequences, semantic and syntactic features from bug report textual contents [650]. The features contain discriminative information that indicates who should fix the bug.

Lee et al. [651] proposed an automatic bug triager using CNN and word embedding. The results demonstrate benefits in both industrial and open source projects. Mani et al. [652] proposed a novel bug report representation algorithm using an attention-based deep bidirectional recurrent neural network (DBRNN-A), which addresses the problem that existing bag-of-words models fail to consider syntactical and sequential word information in unstructured texts. The empirical results show that DBRNN-A provides a higher rank-10 average accuracy. Liu et al. [653] proposed the BT-RL model, which uses deep reinforcement learning for bug triaging. It utilizes deep multi-semantic feature fusion for high-quality feature representation, and an online dynamic matching model employing reinforcement learning to recommend developers for bug reports.

13.4 Bug severity/priority prediction

The severity/priority of bug reports indicates the importance of fixing a bug, which is often manually decided by developers. Bug severity/priority prediction aims to automatically assign the severity/priority property of a bug report based on the knowledge of historical bug reports. Han et al. [654] used word embeddings and a one-layer shallow CNN to automatically capture discriminative word and sentence features in bug report descriptions for predicting the severity. Gomes et al. [655] conducted a survey on bug severity/priority prediction, which also confirms the effectiveness of DL techniques on this task.

13.5 Bug fixing time prediction

This task predicts the time to be taken to fix a bug, which helps software team better allocate the work of developers. Previously, many popular ML methods such as KNN and decision trees have been used to predict the fixing time of bug reports. In recent years, DL has also been employed. For example, Noyori et al. [656] adopted a CNN and gradient-based visualization approach for extracting bug report features related to bug fixing time from comments of bug reports. These features can significantly improve the effectiveness of bug fixing time prediction models.

13.6 Bug report summarization

The common practice for software developers to fix newly reported bugs is to read the bug report and similar historical bug reports. Statistics show that nearly 600 sentences have to be read on average if a developer refers to only 10 historical bug reports during bug fixing [639]. Bug report summarization automatically identifies important sentences in a bug report or directly generates a short, high-level summary of the bug report [657]. Regarding DL techniques, Huang et al. [28] proposed a CNN-based approach to analyze the intention of each sentence in bug report for the ease of reading the content. Li et al. [639] and Liu et al. [658] proposed auto-encoder networks with different structures for informative sentence extraction in bug reports. In recent studies, DL-based language models are also employed to automatically generate titles of bug reports [659, 660].

13.7 Bug localization

Bug localization is the main BRM task facilitated by DL techniques, where 16 related papers are found. For this task, DL techniques are used to bridge the gap between the natural language in bug reports and the source code [661, 662]. For example, DL networks such as CNNs can be used to extract semantic correlations between bug reports and source files. Then, these features are merged and passed to a classification layer to compute the relevancy scores between bug reports and source files [663]. The relevancy scores can be combined with the scores computed from other ML or information retrieval techniques [664–666] (such as learning to rank [667]) to better associate source code with a bug report.

In recent years, advanced DL techniques such as BERT [523], adversarial transfer learning [668], and attention mechanism [669] have been employed to improve the localization effectiveness. Using these powerful DL techniques, a bug report can be located to the source code on different levels (e.g., file level [670], component level [671], and method level [522]). These techniques also enable both within-project bug localization and cross-project bug localization [668, 672].

Table 18 Datasets used for BRM tasks.

#	Task	<1000	1000–10000	10001–30000	30001–100000	>100000
1	Bug report refinement	*	*	*	*	1
2	Duplicate bug detection	*	*	*	*	5
3	Bug assignment	*	*	*	1	2
4	Bug severity/priority prediction	*	*	*	1	*
5	Bug fixing time prediction	*	*	*	1	*
6	Bug report summarization	2	*	*	1	1
7	Bug localization	1	3	10	*	1
8	Bug-Commit linking	1	*	2	*	*

13.8 Bug-commit linking

Bug-commit linking associates bug reports with bug fixing commits or bug inducing commits. Consequently, developers can better understand which commit fixes the bug and why/how/when the bug is introduced [632]. A variant of this task is to link bug reports with reviews or comments in software engineering forums (e.g., APP reviews) [673]. For this task, DL techniques (e.g., word embedding and RNN) learn the semantic representation of natural language descriptions and code in bug reports and commits, as well as the semantic correlation between bug reports and commits [674, 675].

13.9 Datasets

The type of datasets used for BRM tasks is determined by the nature of each BRM task. For the majority of BRM tasks, the main datasets are the bug reports (also called issue reports). Such BRM tasks include bug report refinement, duplicate bug detection, bug assignment, bug severity/priority prediction, bug fixing time prediction, and bug report summarization. Regarding bug localization and bug-commit linking, datasets require bug reports, source code, and commit messages for analysis. Existing studies evaluate their DL techniques using datasets from different software repositories, such as Open Office, Eclipse, Mozilla, and Net Beans [645, 646, 651, 676]. Since many software projects nowadays are managed by distributed collaboration platforms such as GitHub, bug reports from GitHub are also important dataset sources for BRM tasks, such as TensorFlow, Docker, Bootstrap, and VS Code [28].

Table 18 shows the sizes of datasets used for different BRM tasks (notably, * means the dataset in this scale has not been used in the corresponding task). The number in each cell represents the number of datasets with the certain size used for a BRM task. All BRM tasks have datasets with more than 10000 BRM-related items (e.g., bug reports). For five out of eight BRM tasks, they have constructed large datasets with more than 100000 BRM-related items. The large datasets for BRM tasks not only improve the reliability of the evaluation, but also facilitate the training of DL techniques.

13.10 Challenges and opportunity

Based on the preceding analysis of deep learning-based BRM, we present the challenges and opportunities for future research.

13.10.1 *Challenges*

• Computation cost. Computation cost of deep learning is one of the major concerns in BRM. Deep learning works well for many tasks but generally costs a large amount of computation resources. Sometimes the training time lasts for hundreds of hours of CPU/GPU time, especially for complex network architectures. The less computation cost is important for using DL in real BRM scenario in industry, because the long computation time leads to power consumption and heat dissipation issues, which increase the total financial cost of software companies [677].

• Training datasets. The size of a training set is important for DNNs. Biswas et al. [678] found that there are sometimes no performance increases for domain-specific training of DL, due to the small-scale training set. Nizamani et al. [679] also observed a trend of performance improvement for deep learning when the training set size is increased. For deep learning applications in BRM, small training sets often lead performance decline and over-fitting.

• Interpretability. The interpretability is important in BRM. Results provided by machine learning algorithms are sometimes difficult to understand. DL techniques are even worse due to the complex network architectures. Therefore, more interpretable DL techniques are important to help developers get trustworthy prediction results.

13.10.2 *Opportunities*

• DL acceleration in the BRM context. SE studies accelerate the network training with optimization strategies in AI, e.g., batch gradient descent and RMSprop. For a given BRM task, some studies select faster neural networks as substitutes for the slow ones. For example, CBOW is a faster model than Skip-gram in Word2Vec. Li et al. [680] compared the two models and used CBOW for their task, as the two models achieved similar performance. Meanwhile, existing studies also reduce the computation cost by using the distributed model training [681] and dynamic GPU memory manager [682].

• BRM data enrichment. The challenge of training datasets can be alleviated as the exponential growth in the number of bug reports. BRM studies also adopt DL techniques such as fine-tuning [647] and transfer learning [668] to address this problem. In addition, we can automatically generate (relatively low-quality) artificial data to enlarge the training set. Typically, data generation is treated in a case-by-case manner. For instance, Moran et al. [683] used APP screenshots and the labeled GUI-component names in screenshots to train DNNs for GUI-component classification. They synthesized APP screenshots by placing GUI-components of specified types on a single screen with randomized sizes and values. They also performed color perturbation on the images to further enlarge the training set.

• Interpretable DL techniques for BRM tasks. Existing studies try to interpret the prediction results of DL by visualization, including t-SNE and heat map visualizations. The t-SNE (t-distributed stochastic neighbor embedding) technique projects high-dimensional vectors into two-dimensional spaces. It is useful to understand the embedding (vectors) generated by DNNs. With t-SNE, we can understand the semantically related APIs and SE terms [152] calculated by deep learning. A heat map is usually used to visualize network parameters. By visualizing parameters of a layer, we can understand which part of information on which the network focuses more. A heat map assumes the more important (in deeper color) a region is, the more weight the network assigns to features in that area. SE studies use heat maps to visualize the important part in SE images for classification and the attention of RNN [186].

14 Developer collaboration

Software development usually relies on highly collaborative efforts among developers and is widely known as a type of social-technical activity. Hence effective collaboration among software developers is one of the most important factors that greatly benefit productive software development. Brooks [684] highlights the collaboration cost in software development in his famous book, *The Mythical Man-Month*. On one hand, for large-scale software projects inside an organization, hundreds or thousands of developers could be involved, in which developers may not know each other well, and it is a great challenge to establish effective collaboration among developers. On the other hand, open source and crowdsourcing projects are becoming increasingly prevalent software development paradigms, and millions of developers are loosely organized in Internet-based development platforms, where both the development tasks and developers are characterized by variety and scale, and how to support collaborative development in such open environments is another big challenge. While developer collaboration may involve many aspects, development tasks are the pivotal points. Thus, the core issues for developer collaboration include (1) understanding developers, and (2) assigning a task to one or multiple proper developers. In this regard, thanks to the availability of the large volume of software development data, machine learning (especially deep learning) has been employed to analyze the data and provide intelligent support for facilitating developer collaboration. In particular, we survey developer collaboration from the following three angles: developer expertise profiling, intelligent task assignment, and development team forming.

14.1 Developer expertise profiling

Developer expertise profiling mainly aims at realizing certain forms of representation of the skills that a developer has mastered, which is the basis for collaborative development. Profiling developer expertise has received much attention in the software engineering community [685–689]. The typical approach is to

mine developers' past experience data to measure their expertise with machine learning and data mining techniques.

We first review the research efforts with traditional machine learning techniques. For instance, Ref. [685] presented an approach, Expertise Browser, to measure the expertise of developers with the data in change management systems, and Ref. [688] studies how developers learn their expertise by quantifying the development fluency. Refs. [689–691] evaluated developers by graph-based algorithms. Ref. [692] presented a conceptual theory of software development expertise for programming mainly by a large-scale survey of real-world software developers. Ref. [693] evaluated developers' contributions by development values consisting of the effect of code reuse and the impact on development. Expertise profiling is also used for modeling developers in open source software community. For example, Venkataramani et al. [694] captured the expertise of developers by mining their activities from the open source code repositories. Saxena et al. [695] annotated GitHub code with tags in Stack Overflow and then created a detailed technology skill profile of a developer based on code repository contributions of the developer. Considering single-community data could be insufficient for accurately characterizing developers, several techniques have been proposed to connect users in different software communities [696–702]. Ref. [699] conducted an empirical study of user interests across GitHub and Stack Overflow and they found that developers do share common interests in the two communities. Huang et al. [700] proposed CPDScorer to model the programming ability of developers across CQA sites and OSS communities. They first analyzed the answers posted in CQA and the projects submitted in OSS to score developer expertise in the two communities, respectively. They then computed the final expertise by summing up the two scores. However, they did not consider the interactions among developers, which have implications for evaluating the expertise of developers. Furthermore, most approaches to developer expertise profiling ignore the fact that developer expertise evolves over time due to learning or forgetting. To fill this gap, Yan et al. [701] and Song et al. [703] proposed heterogeneous information network-based approaches to profiling developer expertise with GitHub and Stack Overflow data, where there are four types of nodes including developers, skills, questions, and projects, and nine types of edges in the network. As a result, the problem of profiling developers is formulated as estimating the distance of developer nodes and skill nodes. That work combines the historical contributions of developers, the dynamics of expertise due to forgetting, and collaborations among developers, which is particularly featured by incorporating the collaboration relationships into the estimation of developers' expertise. Montandon et al. [702] employed data from social coding platforms (i.e., Stack Overflow and GitHub), built three different machine-learning models to identify the technical roles of open source developers such as backend, frontend, and full-stack, and they simply leveraged the data from Stack Overflow to build a ground truth for evaluating the performance of their approach.

Apart from traditional machine learning, deep learning techniques are also introduced in profiling developer expertise as vectors. The vectors can be utilized for predicting or recommending downstream tasks to developers. Dey et al. [704] proposed a deep neural network-based approach to generating a vectorized representation of software developers based on the APIs they have used. Specifically, the authors [705] collected the open source data with WoC (World of Code), extracted the mappings of projects, developers, and the APIs, and then used Doc2Vec [233] to obtain the vectors representing APIs, developers, and projects. Similarly, Dakhel et al. [706] also used Doc2vec to generate vector embeddings of developers, but they incorporated more data including the textual data of repositories and issues beyond APIs. Javeed et al. [707] proposed to use LSTM and convolutional neural networks to train deep learning models to classify whether a developer is an expert or a novice according to six attributes of source code, including security, reliability, complexity, lines of code, maintainability, and duplication.

14.2 Intelligent task assignment

Given a software development task, one important issue is how to find appropriate developers to handle the task, which is also known as development task assignment. In the following, we summarize how machine learning and deep learning are used to support development task assignments in various scenarios including crowdsourcing software development, open source development, and bug triager.

14.2.1 *Crowdsourcing developer recommendation*

Crowdsourcing software development usually adopts the open-call mode to solicit workers (i.e., developers) for various tasks published online by requesters. Developers hope to find appropriate tasks by considering the task factors such as reward, difficulty, and needed efforts while requesters need to find capable

developers to complete the tasks with guaranteed quality. As a result, recommending suitable developers for a task, or vice versa, becomes an important research topic in crowdsourcing software development.

As prior studies often overlook the skill improvement of developers over time, Wang et al. [708] proposed a new technique for crowdsourcing developer recommendations. After conducting an empirical study of 74 developers on Topcoder and re-calculating developers' scores, they found that the skill improvement of developers fits well with the negative exponential learning curve. Based on the learning curve, a skill prediction technique is designed and a skill improvement-aware framework for recommending developers is proposed. Zhang et al. [709] proposed a meta-learning-based policy model to address the challenge of identifying developers who are most likely to win a given task in crowdsourcing software development. This model firstly filters out developers unlikely to participate or submit to a given task, then recommends the top k developers with the highest possibility of winning. Yu et al. [710] proposed a new deep model, which is a cross-domain developer recommendation algorithm using feature matching based on collaborative filtering, for T-shaped expert finding. Their recommendation model leverages the data from software crowdsourcing platforms (i.e., ZhuBaJie and Joint Force), solves the problem of data sparsity, and finally improves the recommendation performance to some extent. Wang et al. [711] presented PTRec, a context-aware task recommendation technique, capturing in-process progress-oriented information and crowdworkers' traits through a testing context model. Using random forest, it dynamically recommends tasks aligned with worker skills and interests. The evaluation shows its excellence in precision and recall, saving efforts, and enhancing bug detection. Furthermore, Wang et al. [712] presented iRec2.0, which integrates dynamic testing context modeling, learning-based ranking, and multi-objective optimization for crowdworker recommendations in crowdtesting. It aims to detect bugs earlier, shorten non-yielding windows, and alleviate recommendation unfairness, demonstrating the potential to improve cost-effectiveness.

14.2.2 *Reviewer recommendation*

Code review is one of the important tasks for ensuring code quality, which relies on professional developers, known as reviewers, to identify defects by reading source code. Thus finding appropriate reviewers is a core issue for achieving effective code reviews. To address this issue, researchers have conducted extensive studies on recommending reviewers for code review tasks, especially in the context of open source development.

Ying et al. [713] proposed a reviewer recommendation approach (EARec) for a given pull request, considering developer expertise and authority simultaneously. Jiang et al. [714] provided an approach to recommending developers to comment on a PullRequest in social-coding platforms like GitHub. Zhang et al. [715] presented CORAL, a reviewer recommendation technique using a socio-technical graph and a graph convolutional neural network. Trained on 332 Microsoft repositories, CORAL identifies qualified reviewers missed by traditional systems, excelling in large projects, while traditional systems perform better in smaller ones. Rebai et al. [716] framed code reviewer recommendation as a multi-objective search problem, balancing expertise, availability, and collaboration history. Validation on 9 open-source projects confirms its superiority over existing approaches. Zanjani et al. [717] introduced cHRev, an approach for automatic reviewer recommendation based on historical contributions of reviewers in their previous reviews. Due to leveraging specific previous review information, cHRev outperforms existing approaches on three open-source systems and a Microsoft code base. Hannebauer et al. [718] empirically compared six modification expertise-based algorithms and two reviews of expertise-based algorithms on four FLOSS projects. The study concludes that review expertise-based algorithms, particularly the weighted review count (WRC), are more effective. Rong et al. [719] introduced HGRec, a recommendation system utilizing hypergraph techniques to model complex relationships involving multiple reviewers per pull-request in code review. Evaluated on 12 OSS projects, HGRec demonstrates superior accuracy, emphasizing the potential of hypergraphs in this field. Kovalenko et al. [720] evaluated a reviewer recommendation system in a company setting, covering over 21000 code reviews. Despite the relevance of recommendations, it identifies no evidence of influence on user choices, highlighting the need for more user-centric design and evaluation in reviewer recommendation tools. Ahasanuzzaman et al. [721] proposed KUREC, a code reviewer recommender utilizing Java programming language knowledge units (KUs) to generate developer expertise profiles and select reviewers. Evaluated against baselines, KUREC is found to be equally effective but more stable. Besides, combining KUREC with baselines further enhances performance. Gonçalves et al. [722] identified 27 competencies vital for code review through expert validation and

ranked them using a survey of 105 reviewers. The findings reveal that technical skills are essential and commonly mastered, but improvements are needed in clear communication and constructive feedback.

14.2.3 *Other tasks*

Besides the aforementioned tasks, there are also a series of other development tasks that benefit from machine learning and deep learning technologies, e.g. bug assignment (Subsection 13.3), question answering tasks in online Q&A sites, and various development tasks for open source contributors.

Huang et al. [723] proposed an approach to recommending appropriate answers for questions posted to Q&A sites. Specifically, they leveraged graph attention networks to represent the interactive information between candidate answerers, and an LDA topic model to capture the text information. It was verified by experiments that the approach outperforms the state-of-the-art techniques of that time. Jin et al. [724] proposed CODER, a graph-based code recommendation framework for OSS developers, that models user-code and user-project interactions via a heterogeneous graph to predict developers' future contributions. CODER has shown superior performance in various experimental settings, including intra-project and cross-project recommendations. Xiao et al. [725] introduced RecGFI, an approach for recommending "Good First Issues" to newcomers in open-source projects. Utilizing features from content, background, and dynamics. Employing an XGBoost classifier, RecGFI achieves up to 0.853 AUC in evaluation, demonstrating superiority over alternative models.

Santos [726] proposed an automatic open issue labeling strategy to assist OSS contributors in selecting suitable tasks and helping OSS communities attract and retain more contributors. The technique uses API-domain tags to label issues and relies on qualitative studies to formulate recommendation strategies and quantitative investigations to analyze the relevance between API-domain labels and contributors. The results show that the predicted labels have an average precision of 75.5%, demonstrating the superiority of the technique. Costa et al. [727] presented TIPMerge, an approach for recommending participants for collaborative merge sessions within large development projects with multiple branches. TIPMerge builds a ranked list of developers appropriate to collaborate by considering their changes in previous history, branches, and dependencies, and recommends developers with complementary knowledge. The approach demonstrates a mean normalized improvement of 49.5% for joint knowledge coverage compared to selecting the top developers.

14.3 Development team formation

Complex development tasks often ask for a team of developers. Thus how to find a cohort of developers who can collaboratively handle a complex task is an important issue in software development.

In order to address the complexity of finding collaborators with shared interests in large open-source software, Constantino et al. [728] proposed a visual and interactive web application tool named CoopFinder. They further presented and evaluated two collaborator recommendation strategies based on co-changed files [729]. The strategies utilize TF-IDF scheme to estimate the importance of files modified by developers and measure developers' similarity using the Cosine metric. Through an extensive survey of 102 real-world developers, the strategies show up to an 81% acceptance rate, enhancing collaboration efficiency among developers. Surian et al. [730] introduced a technique to find compatible collaboration among developers. They first created a collaboration network using information of developers and projects from Sourceforge.Net, and then recommended collaborators for developers based on their programming skills and past projects through a random walk with restart procedures. Canfora et al. [731] introduced Yoda, a technique for recommending mentors to newcomers in software projects. By mining data from mailing lists and versioning systems, developers with experience meanwhile actively interacting with newcomers are selected as their mentors. Evaluation of five open-source projects and surveys with developers shows the potential usefulness of Yoda in supporting newcomers in a team and indicates that top committers are not always the best mentors. Ye et al. [732] introduced a personalized teammate recommendation approach for crowdsourcing developers. Through an empirical study on Kaggle, three factors influencing developers' teammate preferences are identified and a linear programming-based technique is proposed to compute developers' teammate preferences. Finally, a recommendation approach with an approximation algorithm to maximize collaboration willingness is designed.

14.4 Datasets

Here, we present some commonly used datasets in the three main kinds of research efforts toward intelligent developer collaboration. For developer expertise profiling, data primarily originate from collaborative software development platforms such as GitHub, encompassing version control data from various open-source projects with diverse developer information [705, 706, 733]. For intelligent task assignment, datasets are sourced from crowdsourcing platforms like Topcoder and Baidu CrowdTest, where historical data contain developer information and the corresponding task categories [708, 711]. In the case of development team formation, datasets are predominantly obtained from platforms like SourceForge and Kaggle, showcasing richer collaboration patterns among developers [730, 732]. These datasets provide abundant information for studying developer behavior, intelligent task assignment, and team formation.

• **Developer expertise profiling.** The WoC dataset [705] is a versioned and expansive repository of version control data from free/libre and open source software (FLOSS) projects using Git. The dataset, collected in March 2020, contains 7.9 billion blobs, 2 billion commits, 8.3 billion trees, 17.3 million tags, 123 million distinct repositories, and 42 million unique author IDs. WoC supports various research tasks, including developer expertise profiling. Using the WoC dataset, Fry et al. [733] proposed a technique to identify all author IDs belonging to a single developer in the entire dataset, revealing aliases. Using machine learning, Fry et al. processed around 38 million author IDs, identifying 14.8 million with aliases linked to 5.4 million developers. This dataset enhanced models of developer behavior at the global open-source software ecosystem level, facilitating rapid resolution of new author IDs. Meanwhile, Dakhel et al. [706] collected a dataset to determine the domain expertise of developers using information from GitHub. Their data collection process involved three main types of information: repositories that developers have contributed to, issue-resolving history, and API calls involved in a commit. This dataset contained information about 1272 developers with expertise labels in five job roles. The dataset consisted of textual information from 58000 repositories, issue-resolving history from 60000 issues, and API calls from 21 million commits across different GitHub repositories. The dataset aimed to provide comprehensive insights into developers' expertise by considering their contributions to repositories, issue resolution history, and API usage in commits across diverse projects on GitHub.

• **Intelligent task assignment.** Crowdsourcing platforms provide the feasibility of data collection for intelligent task assignments. For instance, Topcoder is a competition-based crowdsourcing software development platform. Topcoder offers various types of tasks, such as "Test Suites," "Assembly," and "Bug Hunt," each representing a category of challenges. Challenges are instances of task types, and developers choose whether to participate in these challenges. In this respect, Wang et al. [708] collected a dataset from Topcoder. Their dataset involved 32565 challenges, 7620 developers, and 59230 submissions spanning from 2006 to 2016. The dataset focused on 100 developers with over 100 submissions, containing the evolution of the skills of each developer. After filtering out submissions with a final score of 0 (indicating unfinished submissions), there were 74 developers left in the dataset. There has also been a lot of work on dataset building using other crowdsourcing platforms. For example, Wang et al. [711] collected a dataset from Baidu CrowdTest. The dataset involved 2404 crowdworkers and comprised 80200 submitted reports. For each testing task, comprehensive information was gathered, including task-related details and all submitted test reports with associated information, such as submitter and device. The dataset serves as a valuable resource for analyzing crowdtesting dynamics and outcomes.

• **Development team formation.** Work related to development team formation is often obtained from open-source software development platforms with collaborative and social nature, such as SourceForge and Kaggle. For example, Surian et al. [730] collected a dataset by analyzing the SourceForge database, which contained information about projects, project categories, and programming languages. There were 209009 developers associated with 151776 projects. The dataset included details such as 354 project categories and information about the usage of 90 different programming languages within the projects. Meanwhile, Ye et al. [732] collected a dataset by crawling data from Kaggle, including details from 275 competitions and 74354 developers. The data spanned from April 2010 to January 2018 and encompassed 191300 submissions. Additionally, developers' social data from Kaggle's communities were crawled, enhancing the dataset with information about interactions and discussions among developers.

14.5 Challenges and opportunities

In general, software is becoming more complex as the major enabling force for the infrastructure of various information systems. Consequently, more developers participate in software projects, and software

development is increasingly exhibiting a social-technical characteristic, which requires more effective collaboration among developers. Thus, more research efforts are needed to produce new theories, techniques, and tools to further improve collaborative development. To that end, we summarize the challenges and opportunities in this research area.

14.5.1 *Challenges*

The use of deep learning to improve developer collaboration faces the following research challenges in terms of collaborative tasks, software development data, and evaluation benchmarks.

• **Complex collaboration among multiple developers.** Existing research mainly considers the collaboration between two developers. In other words, one developer posts a task requirement, and another developer is required to fulfill the task. However, at higher levels (e.g., from a project's perspective), we must consider the collaboration among a group of developers, where global collaboration effects and constraints should be of greater concern.

• **Continuously growing data size and heterogeneity.** On the one hand, developers generate more data, which are often distributed across various platforms, including personal IDEs, proprietary enterprise environments, open-source platforms (e.g., GitHub), and public forums (e.g., Stack Overflow). On the other hand, software development data are essentially heterogeneous, involving natural language data, source code, AI models, and even graphical data. Dealing with highly dispersed, heterogeneous, and large-scale development data is a great challenge for using deep learning to achieve more effective and efficient collaborative development.

• **Lack of benchmarks for effective evaluation.** As the effectiveness of developer collaboration usually cannot be observed in a short time, evaluating a newly proposed technique is difficult. Therefore, having benchmark datasets on one or multiple collaborative development tasks is highly desirable.

14.5.2 *Opportunities*

Although deep learning has shown promising results in improving developer collaboration, there are still abundant research opportunities for advancing this direction.

• Application of deep learning to a wider spectrum of collaborative development tasks. As collaborative activities are pervasive in software development processes, deep learning can be introduced to deal with more tasks other than those in existing studies.

• Incorporating advanced deep learning technologies. Although deep learning has developed dramatically for nearly two decades, new technologies are still being put forward continually, such as graph neural networks and large language models. Applying these technologies to handle large-scale heterogeneous development data embraces more opportunities for intelligent collaborative development.

• Novel collaborative development activities enabled by deep learning. Large language models like ChatGPT are becoming more powerful in handling development tasks, such as code generation and program repair. It is possible to see novel collaborative development activities among human developers and deep learning-based AI models.

15 Conclusion

In this paper, we present the first task-oriented survey on deep learning-based software engineering. Our survey focuses on twelve of the important tasks in software development and maintenance: requirements engineering, code generation, code search, code summarization, software refactoring, code clone detection, software defect prediction, bug finding, fault localization, program repair, bug report management, and developer collaboration. For each of the selected tasks, we summarize the most recent advances concerning the application of deep learning for the given task, as well as relevant challenges and future opportunities. On the basis of the surveyed deep learning-based subareas of software engineering, we make the following observations.

• **Widespread applications and impressive results.** Deep learning techniques have been widely applied across various subareas of software engineering and have achieved impressive results. First, deep learning techniques typically improve the performance of most tasks. For example, deep learning-based code clone detection always achieves higher recall and better precision than the best classical approaches. An existing empirical study also shows that deep learning techniques outperform other traditional machine

learning techniques for bug assignments. Moreover, in requirements engineering, most studies report precision and recall exceeding 80%, and the F_1 score is often above 75%. Second, the powerful feature engineering capability of deep learning enables the capture of semantic information. Typically, some tasks (e.g., software defect prediction and software refactoring) involve complex feature engineering. Deep learning helps release researchers and practitioners from tedious feature engineering. Third, deep learning can further contribute to the automation of software engineering processes. Traditional techniques may rely on complex preprocessing and/or postprocessing techniques to automate the whole process. Deep learning can turn the whole process in an end-to-end manner. For example, deep learning techniques can directly deal with bug reports and help automate the process of bug report management.

• **Challenges in high-quality training data acquisition.** The primary challenge faced by most subareas of software engineering is obtaining high-quality training data. First, the sizes of datasets are often limited. For example, publicly accessible data for requirements typically are small in volume and lack some details, making it difficult to train effective deep learning models. Second, the quality of datasets is often unguaranteed. For example, in code generation, it is unclear whether datasets contain vulnerable code snippets that may result in unsafe codes. Third, some existing datasets are the results of specially designed data production activities and may not well align with real-world scenarios. Although this can lead to strong model performance during evaluation, the performance may not be effectively transferred into practical use.

• **Interpretability challenge.** The interpretability of deep learning models is also a common challenge across various subareas of software engineering. First, the lack of interpretability for deep learning models makes ensuring correctness difficult. For example, ensuring whether generated patches are correct is difficult. Second, deep learning models are known for their "black-box" nature, making it challenging for developers to understand their rationale. Therefore, it is hard for developers to work with deep learning models.

• **Generalization across languages.** Developing deep learning models that generalize well across different languages is a considerable challenge. Different programming languages have their own syntax and semantics, making it difficult to develop a universal model that performs well across diverse languages. Therefore, the current practice is to deal with each programming language individually. For example, for different programming languages, researchers often need to train different models for the same code summarization task. Furthermore, to incorporate additional code structure information in deep learning models, researchers must parse code snippets with their corresponding parsers, thus resulting in significant differences for different programming languages.

Our survey demonstrates that deep learning-based software engineering has achieved significant advances recently and has the potential for further improvement. However, some critical challenges should be resolved before deep-learning-based software engineering can reach its maximal potential. We believe that future research should focus on resolving these challenges.

Acknowledgements We thank the following persons for their prior contributions to the manuscript preparation (in alphabetical order): Yuze GUO (Beihang University), Ruiqi HONG (Beihang University), Mingwei LIU (Fudan University), Xiaofan LIU (Wuhan University), Di WU (Beihang University), Hongjun YANG (Beihang University), Yanming YANG (Zhejiang University), Binquan ZHANG (Beihang University), and Zhuang ZHAO (Wuhan University).

References

1 Hinton G E, Salakhutdinov R R. Reducing the dimensionality of data with neural networks. Science, 2006, 313: 504–507
2 Liu L, Ouyang W, Wang X, et al. Deep learning for generic object detection: a survey. Int J Comput Vis, 2020, 128: 261–318
3 Hinton G E, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets. Neural Comput, 2006, 18: 1527–1554
4 Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. Commun ACM, 2017, 60: 84–90
5 Lecun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition. Proc IEEE, 1998, 86: 2278–2324
6 Elman J L. Finding structure in time. Cogn Sci, 1990, 14: 179–211
7 Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput, 1997, 9: 1735–1780
8 Schuster M, Paliwal K K. Bidirectional recurrent neural networks. IEEE Trans Signal Process, 1997, 45: 2673–2681
9 Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of Advances in Neural Information Processing Systems, 2017. 30

10 Yang Y M, Xia X, Lo D, et al. A survey on deep learning for software engineering. ACM Comput Surv, 2022, 54: 1–73
11 Nguyen G, Dlugolinsky S, Bobák M, et al. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. Artif Intell Rev, 2019, 52: 77–124
12 Wang J, Ma Y, Zhang L, et al. Deep learning for smart manufacturing: Methods and applications. J Manuf Syst, 2018, 48: 144–156
13 Shen D, Wu G, Suk H I. Deep learning in medical image analysis. Annu Rev Biomed Eng, 2017, 19: 221–248
14 Berman D S, Buczak A L, Chavis J S, et al. A survey of deep learning methods for cyber security. Information, 2019, 10: 122
15 Le T H, Chen H, Babar M A. Deep learning for source code modeling and generation: models, applications, and challenges. ACM Comput Surv, 2021, 53: 1–38
16 Svyatkovskiy A, Zhao Y, Fu S, et al. Pythia: AI-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019. 2727–2735
17 Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016
18 Aniche M, Maziero E, Durelli R, et al. The effectiveness of supervised machine learning algorithms in predicting software refactoring. IEEE Trans Software Eng, 2020, 48: 1432–1450
19 Gu X, Zhang H, Kim S. Deep code search. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 933–944
20 Wardat M, Le W, Rajan H. Deeplocalize: fault localization for deep neural networks. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering, 2021. 251–262
21 Li Y, Wang S, Nguyen T N. DLFix: context-based code transformation learning for automated program repair. In: Proceedings of the 42nd International Conference on Software Engineering, Seoul, 2020. 602–614
22 Zou D, Wang S, Xu S, et al. μVulDeePecker: a deep learning-based system for multiclass vulnerability detection. IEEE Trans Dependable Secure Comput, 2019, 18: 2224–2236
23 Humbatova N, Jahangirova G, Tonella P. DeepCrime: mutation testing of deep learning systems based on real faults. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021. 67–78
24 Watson C, Cooper N, Palacio D N, et al. A systematic literature review on the use of deep learning in software engineering research. ACM Trans Softw Eng Methodol, 2022, 31: 1–58
25 Niu C, Li C, Luo B, et al. Deep learning meets software engineering: a survey on pre-trained models of source code. 2022. ArXiv:2205.11739
26 Zhang Q, Fang C, Xie Y, et al. A survey on large language models for software engineering. 2023. ArXiv:2312.15223
27 Jin Z. Environment Modeling-Based Requirements Engineering for Software Intensive Systems. San Francisco: Morgan Kaufmann Publishers Inc., 2017
28 Huang Q, Xia X, Lo D, et al. Automating intention mining. IEEE Trans Software Eng, 2020, 46: 1098–1119
29 Pudlitz F, Brokhausen F, Vogelsang A. Extraction of system states from natural language requirements. In: Proceedings of the IEEE 27th International Requirements Engineering Conference (RE), 2019. 211–222
30 Li M, Shi L, Yang Y, et al. A deep multitask learning approach for requirements discovery and annotation from open forum. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2021. 336–348
31 Guo H, Singh M P. Caspar: extracting and synthesizing user stories of problems from app reviews. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 628–640
32 Mekala R R, Irfan A, Groen E C, et al. Classifying user requirements from online feedback in small dataset environments using deep learning. In: Proceedings of the IEEE 29th International Requirements Engineering Conference (RE), 2021. 139–149
33 Tizard J, Devine P, Wang H, et al. A software requirements ecosystem: linking forum, issue tracker, and faqs for requirements management. IEEE Trans Software Eng, 2023, 49: 2381–2393
34 Shi L, Xing M, Li M, et al. Detection of hidden feature requests from massive chat messages via deep Siamese network. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 641–653
35 Pan S, Bao L, Ren X, et al. Automating developer chat mining. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021. 854-866
36 Türetken O, Su O, Demirörs O. Automating software requirements generation from business process models. In: Proceedings of the 1st Conference on the Principles of Software Engineering (PRISE'04), 2004
37 Cox K, Phalp K T, Bleistein S J, et al. Deriving requirements from process models via the problem frames approach. Inf Software Tech, 2005, 47: 319–337
38 Maiden N A M, Manning S, Jones S, et al. Generating requirements from systems models using patterns: a case study. Requir Eng, 2005, 10: 276–288
39 Yu E S K, Bois P D, Dubois E, et al. From organization models to system requirements: a 'cooperating agents' approach. In: Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS-95), 1995. 194–204
40 Letier E, van Lamsweerde A. Deriving operational software specifications from system goals. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2002. 119–128
41 Landtsheer R D, Letier E, van Lamsweerde A. Deriving tabular event-based specifications from goal-oriented requirements models. Requir Eng, 2004, 9: 104–120
42 van Lamsweerde A. Goal-oriented requirements enginering: a roundtrip from research to practice [enginering read engineering]. In: Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004. 4–7
43 van Lamsweerde A, Willemet L. Inferring declarative requirements specifications from operational scenarios. IEEE Trans Software Eng, 1998, 24: 1089–1114
44 Meziane F, Athanasakis N, Ananiadou S. Generating natural language specifications from UML class diagrams. Requir Eng, 2008, 13: 1–18
45 Berenbach B. The automated extraction of requirements from UML models. In: Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE 2003), 2003. 287
46 Souag A, Mazo R, Salinesi C, et al. Using the AMAN-DA method to generate security requirements: a case study in the maritime domain. Requir Eng, 2018, 23: 557–580
47 Zhao Z, Zhang L, Lian X, et al. ReqGen: keywords-driven software requirements generation. Mathematics, 2023, 11: 332
48 Koscinski V, Hashemi S, Mirakhorli M. On-demand security requirements synthesis with relational generative adversarial networks. In: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023. 1613–1625
49 Li M, Yang Y, Shi L, et al. Automated extraction of requirement entities by leveraging LSTM-CRF and transfer learning. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020. 208–219
50 Casillo F, Deufemia V, Gravino C. Detecting privacy requirements from user stories with NLP transfer learning models. Inf

Software Tech, 2022, 146: 106853
51 Ezzini S, Abualhaija S, Arora C, et al. Automated handling of anaphoric ambiguity in requirements: a multi-solution study. In: Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022. 187–199
52 Wang Y, Shi L, Li M, et al. Detecting coreferent entities in natural language requirements. Requir Eng, 2022, 27: 351–373
53 Wang Y, Shi L, Li M, et al. A deep context-wise method for coreference detection in natural language requirements. In: Proceedings of the IEEE 28th International Requirements Engineering Conference (RE), 2020. 180–191
54 Ezzini S, Abualhaija S, Arora C, et al. AI-based question answering assistance for analyzing natural-language requirements. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
55 Baker C, Deng L, Chakraborty S, et al. Automatic multi-class non-functional software requirements classification using neural networks. In: Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), 2019. 610–615
56 Hey T, Keim J, Koziolek A, et al. NoRBERT: transfer learning for requirements classification. In: Proceedings of the IEEE 28th International Requirements Engineering Conference (RE), 2020. 169–179
57 Luo X, Xue Y, Xing Z, et al. PRCBERT: prompt learning for requirement classification using BERT-based pretrained language models. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2023
58 Winkler J P, Grönberg J, Vogelsang A. Predicting how to test requirements: an automated approach. In: Proceedings of the IEEE 27th International Requirements Engineering Conference (RE), 2019. 120–130
59 AlDhafer O, Ahmad I, Mahmood S. An end-to-end deep learning system for requirements classification using recurrent neural networks. Inf Software Tech, 2022, 147: 106877
60 Guo J, Cheng J, Cleland-Huang J. Semantically enhanced software traceability using deep learning techniques. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017. 3–14
61 Jahan M S, Khan H U, Akbar S, et al. Bidirectional language modeling: a systematic literature review. Sci Program, 2021. doi: 10.1155/2021/6641832
62 Lee J, Yoon W, Kim S, et al. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. Bioinformatics, 2019, 36: 1234–1240
63 Feng Z, Guo D, Tang D, et al. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of Findings of the Association for Computational Linguistics, 2020. 1536–1547
64 Lin J, Liu Y, Zeng Q, et al. Traceability transformed: generating more accurate links with pre-trained BERT models. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. 324–335
65 Tian J, Zhang L, Lian X. A cross-level requirement trace link update model based on bidirectional encoder representations from transformers. Mathematics, 2023, 11: 623
66 Lin J, Liu Y, Cleland-Huang J. Information retrieval versus deep learning approaches for generating traceability links in bilingual projects. Empir Software Eng, 2022, 27: 5
67 ISO/IEC/IEEE International Standard. Systems and software engineering — life cycle processes — requirements engineering. ISO/IEC/IEEE 29148:2018(E), 2018. 1–104. https://www.iso.org/standard/72089.html.
68 Mavin A, Wilkinson P, Harwood A, et al. Easy approach to requirements syntax (EARS). In: Proceedings of the 17th IEEE International Requirements Engineering Conference, 2009. 317–322
69 Franch X, Glinz M, Mendez D, et al. A study about the knowledge and use of requirements engineering standards in industry. IEEE Trans Software Eng, 2022, 48: 3310–3325
70 Liang J T, Yang C, Myers B A. A large-scale survey on the usability of AI programming assistants: successes and challenges. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2023
71 Kelly S, Tolvanen J P. Domain-Specific Modeling: Enabling Full Code Generation. Hoboken: John Wiley & Sons, 2008
72 Allamanis M, Barr E T, Devanbu P, et al. A survey of machine learning for big code and naturalness. ACM Comput Surv, 2018, 51: 1–37
73 Murphy G C, Kersten M, Findlater L. How are Java software developers using the Eclipse IDE? IEEE Softw, 2006, 23: 76–83
74 Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009. 213–222
75 Gvero T, Kuncak V, Kuraj I, et al. Complete completion using types and weights. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013. 27–38
76 Zheng Q, Xia X, Zou X, et al. CodeGeeX: a pre-trained model for code generation with multilingual evaluations on HumanEval-X. 2023. ArXiv:2303.17568
77 Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, 2017. 1139–1149
78 Iyer S, Cheung A, Zettlemoyer L. Learning programmatic idioms for scalable semantic parsing. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, 2019. 5425–5434
79 Yin P, Neubig G. A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, 2017. 440–450
80 Yin P, Neubig G. TRANX: a transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2018. 7–12
81 Jiang H, Zhou C, Meng F, et al. Exploring dynamic selection of branch expansion orders for code generation. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, 2021. 5076–5085
82 Dong L, Lapata M. Language to logical form with neural attention. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016
83 Yu T, Zhang R, Yang K, et al. Spider: a large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, Brussels, 2018. 3911–3921
84 Sethi A, Sankaran A, Panwar N, et al. DLPaper2Code: auto-generation of code from deep learning research papers. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2018
85 Yang G, Zhou Y, Chen X, et al. ExploitGen: template-augmented exploit code generation based on CodeBERT. J Syst Software, 2023, 197: 111577
86 Ling W, Blunsom P, Grefenstette E, et al. Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016
87 Lyu C, Wang R, Zhang H, et al. Embedding API dependency graph for neural code generation. Empir Software Eng, 2021, 26: 61

88 Clement C B, Drain D, Timcheck J, et al. PyMT5: multi-mode translation of natural language and Python code with transformers. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2020. 9052–9065
89 Le H, Wang Y, Gotmare A D, et al. CodeRL: mastering code generation through pretrained models and deep reinforcement learning. In: Proceedings of Advances in Neural Information Processing Systems, 2022. 35: 21314–21328
90 Wang Y, Wang W, Joty S R, et al. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2021. 8696–8708
91 Sun Y, Tang D, Duan N, et al. Semantic parsing with syntax- and table-aware SQL generation. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, 2018. 361–372
92 Wang X, Wang Y, Wan Y, et al. Compilable neural code generation with compiler feedback. In: Proceedings of Findings of the Association for Computational Linguistics, 2022. 9–19
93 Poesia G, Polozov A, Le V, et al. Synchromesh: reliable code generation from pre-trained language models. In: Proceedings of the 10th International Conference on Learning Representations, 2022
94 Wei B, Li G, Xia X, et al. Code generation as a dual task of code summarization. In: Proceedings of Advances in Neural Information Processing Systems, 2019. 32
95 Ahmad W U, Chakraborty S, Ray B, et al. Unified pre-training for program understanding and generation. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021. 2655–2668
96 Ye W, Xie R, Zhang J, et al. Leveraging code generation to improve code retrieval and summarization via dual learning. In: Proceedings of the Web Conference 2020, 2020. 2309–2319
97 Hashimoto T B, Guu K, Oren Y, et al. A retrieve-and-edit framework for predicting structured outputs. In: Proceedings of Advances in Neural Information Processing Systems, 2018. 31
98 Kulal S, Pasupat P, Chandra K, et al. SPoC: search-based pseudocode to code. In: Proceedings of Advances in Neural Information Processing Systems, 2019. 32
99 Parvez M R, Ahmad W U, Chakraborty S, et al. Retrieval augmented code generation and summarization. In: Proceedings of Findings of the Association for Computational Linguistics, 2021. 2719–2734
100 Iyer S, Konstas I, Cheung A, et al. Mapping language to code in programmatic context. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2018. 1643–1652
101 Guo D, Tang D, Duan N, et al. Coupling retrieval and meta-learning for context-dependent semantic parsing. In: Proceedings of the 57th Conference of the Association for Computational Linguistics, 2019. 855–866
102 Li J, Li Y, Li G, et al. SkCoder: a sketch-based approach for automatic code generation. 2023. ArXiv:2302.06144
103 Dong L, Lapata M. Coarse-to-fine decoding for neural semantic parsing. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, 2018. 731–742
104 Shen S, Zhu X, Dong Y, et al. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022. 1533–1543
105 Sun Z, Zhu Q, Mou L, et al. A grammar-based structural CNN decoder for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2019. 7055–7062
106 Sun Z, Zhu Q, Xiong Y, et al. TreeGen: a tree-based transformer architecture for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2020. 8984–8991
107 Xie B, Su J, Ge Y, et al. Improving tree-structured decoder training for code generation via mutual learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2021. 14121–14128
108 Chung J, Gulcehre C, Cho K, et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. 2014. ArXiv:1412.3555
109 Liu F, Li G, Zhao Y, et al. Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2021. 473–485
110 Izadi M, Gismondi R, Gousios G. CodeFill: multi-token code completion by jointly learning from structure and naming sequences. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 401–412
111 Tang Z, Ge J, Liu S, et al. Domain adaptive code completion via language models and decoupled domain databases. 2023. ArXiv:2308.09313
112 Sun Z, Du X, Song F, et al. CodeMark: imperceptible watermarking for code datasets against neural code completion models. 2023. ArXiv:2308.14401
113 Wang C, Hu J, Gao C, et al. Practitioners' expectations on code completion. 2023. ArXiv:2301.03846
114 Nie P, Banerjee R, Li J J, et al. Learning deep semantics for test completion. 2023. ArXiv:2302.10166
115 Dahal S, Maharana A, Bansal M. Analysis of tree-structured architectures for code generation. In: Proceedings of Findings of the Association for Computational Linguistics, 2021. 4382–4391
116 Norouzi S, Tang K, Cao Y. Code generation from natural language with less prior knowledge and more monolingual data. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, 2021. 776–785
117 Mastropaolo A, Pascarella L, Guglielmi E, et al. On the robustness of code generation techniques: an empirical study on GitHub copilot. 2023. ArXiv:2302.00438
118 Xu F F, Vasilescu B, Neubig G. In-IDE code generation from natural language: promise and challenges. ACM Trans Softw Eng Methodol, 2022, 31: 1–47
119 Liang Q, Sun Z, Zhu Q, et al. Lyra: a benchmark for turducken-style code generation. In: Proceedings of the 31st International Joint Conference on Artificial Intelligence, 2022. 4238–4244
120 Hendrycks D, Basart S, Kadavath S, et al. Measuring coding challenge competence with APPS. In: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, 2021
121 Lu S, Guo D, Ren S, et al. CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. In: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, 2021
122 Shen X, Chen Z, Backes M, et al. In ChatGPT we trust? Measuring and characterizing the reliability of ChatGPT. 2023. ArXiv:2304.08979
123 Lukins S K, Kraft N A, Etzkorn L H. Source code retrieval for bug localization using latent Dirichlet allocation. In: Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, 2008. 155–164
124 Chatterjee S, Juvekar S, Sen K. SNIFF: a search engine for Java using free-form queries. In: Fundamental Approaches to Software Engineering. Berlin: Springer, 2009. 385–400
125 Hill E, Roldan-Vega M, Fails J A, et al. NL-based query refinement and contextualized code search results: a user study. In: Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, 2014. 34–43
126 McMillan C, Grechanik M, Poshyvanyk D, et al. Portfolio: finding relevant functions and their usage. In: Proceedings of

the 33rd International Conference on Software Engineering, 2011. 111–120
127 Li X, Wang Z, Wang Q, et al. Relationship-aware code search for JavaScript frameworks. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 690–701
128 Sachdev S, Li H, Luan S, et al. Retrieval on source code: a neural code search. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2018. 31–41
129 Zou Y, Ling C, Lin Z, et al. Graph embedding based code search in software project. In: Proceedings of the 10th Asia-Pacific Symposium on Internetware, 2018. 1–10
130 Gu W, Li Z, Gao C, et al. Cradle: deep code retrieval based on semantic dependency learning. Neural Networks, 2021, 141: 385–394
131 Wan Y, Shu J, Sui Y, et al. Multi-modal attention network learning for semantic source code retrieval. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, 2019. 13–25
132 Ling X, Wu L, Wang S, et al. Deep graph matching and searching for semantic code retrieval. ACM Trans Knowledge Discov Data, 2021, 15: 1–21
133 Liu S, Xie X, Ma L, et al. GraphSearchNET: enhancing GNNs via capturing global dependency for semantic code search. 2021. ArXiv:2111.02671
134 Li X, Gong Y, Shen Y, et al. CodeRetriever: unimodal and bimodal contrastive learning. 2022. ArXiv:2201.10866
135 Jiang H, Nie L, Sun Z, et al. ROSF: leveraging Information Retrieval and Supervised Learning for Recommending Code Snippets. IEEE Trans Serv Comput, 2019, 12: 34–46
136 Guo D, Ren S, Lu S, et al. GraphCodeBERT: pre-training code representations with data flow. In: Proceedings of the 9th International Conference on Learning Representations, 2021
137 Guo D, Lu S, Duan N, et al. UniXcoder: unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022. 7212–7225
138 Shi Z, Xiong Y, Zhang X, et al. Cross-modal contrastive learning for code search. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022. 94–105
139 Bui N D Q, Yu Y, Jiang L. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021. 511–521
140 Shi E, Wang Y, Gu W, et al. CoCoSoDa: effective contrastive learning for code search. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023. 2198–2210
141 Bajracharya S K, Ngo T C, Linstead E, et al. Sourcerer: a search engine for open source code supporting structure-based search. In: Proceedings of Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2006. 681–682
142 Lu M, Sun X, Wang S, et al. Query expansion via WordNet for effective code search. In: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2015. 545–549
143 Lv F, Zhang H, Lou J, et al. CodeHow: effective code search based on API understanding and extended Boolean model (E). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, 2015. 260–270
144 Rahman M M. Supporting code search with context-aware, analytics-driven, effective query reformulation. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, 2019. 226–229
145 Hill E, Pollock L L, Vijay-Shanker K. Improving source code search with natural language phrasal representations of method signatures. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, 2011. 524–527
146 Liu J, Kim S, Murali V, et al. Neural query expansion for code search. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2019. 29–37
147 Cao K, Chen C, Baltes S, et al. Automated query reformulation for efficient search based on query logs from stack overflow. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. 1273–1285
148 Li D, Shen Y, Jin R, et al. Generation-augmented query expansion for code retrieval. 2022. arXiv:2212.10692
149 Luan S, Yang D, Barnaby C, et al. Aroma: code recommendation via structural code search. Proc ACM Program Lang, 2019, 3: 1–28
150 Mathew G, Stolee K T. Cross-language code search using static and dynamic analyses. In: Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, 2021. 205–217
151 Perez D, Chiba S. Cross-language clone detection by learning over abstract syntax trees. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019. 518–528
152 Nguyen T D, Nguyen A T, Phan H D, et al. Exploring API embedding for API usages and applications. In: Proceedings of the 39th International Conference on Software Engineering, 2017. 438–449
153 Chen B, Abedjan Z. Interactive cross-language code retrieval with auto-encoders. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021. 167–178
154 Huang J, Tang D, Shou L, et al. CoSQA: 20,000+ web queries for code search and question answering. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, 2021. 5690–5700
155 Khan M A M, Bari M S, Do X L, et al. xCodeEval: a large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. 2023. ArXiv:2303.03004
156 Wang C, Peng X, Xing Z C, et al. XCoS: explainable code search based on query scoping and knowledge graph. ACM Trans Softw Eng Methodol, 2023, 32: 1–28
157 Sun Z, Li L, Liu Y, et al. On the importance of building high-quality training datasets for neural code search. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering, 2022. 1609–1620
158 Gotmare A D, Li J, Joty S R, et al. Cascaded fast and slow models for efficient semantic code search. 2021. ArXiv:2110.07811
159 Gu W, Wang Y, Du L, et al. Accelerating code search with deep hashing and code classification. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022. 2534–2544
160 Rush A M, Chopra S, Weston J. A neural attention model for abstractive sentence summarization. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2015. 379–389
161 Alon U, Brody S, Levy O, et al. code2seq: generating sequences from structured representations of code. In: Proceedings of the 7th International Conference on Learning Representations, 2019
162 Xu K, Wu L, Wang Z, et al. Graph2Seq: graph to sequence learning with attention-based neural networks. 2018. ArXiv:1804.00823
163 Sridhara G, Hill E, Muppaneni D, et al. Towards automatically generating summary comments for Java methods. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, 2010. 43–52
164 Abid N J, Dragan N, Collard M L, et al. Using stereotypes in the automatic generation of natural language summaries for

C++ methods. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2015. 561–565
165 Haiduc S, Aponte J, Moreno L, et al. On the use of automated text summarization techniques for summarizing source code. In: Proceedings of the 17th Working Conference on Reverse Engineering, 2010. 35–44
166 Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010. 223–226
167 Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with neural networks. In: Proceedings of Advances in Neural Information Processing Systems, 2014. 3104–3112
168 Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33rd International Conference on Machine Learning, 2016. 2091–2100
169 Ahmad W U, Chakraborty S, Ray B, et al. A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020. 4998–5007
170 Wang R, Zhang H, Lu G, et al. Fret: functional reinforced transformer with BERT for code summarization. IEEE Access, 2020, 8: 135591
171 Zhang J, Wang X, Zhang H, et al. Retrieval-based neural source code summarization. In: Proceedings of the 42nd International Conference on Software Engineering, Seoul, 2020. 1385–1397
172 LeClair A, Bansal A, McMillan C. Ensemble models for neural source code summarization of subroutines. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, 2021. 286–297
173 Gong Z, Gao C, Wang Y, et al. Source code summarization with structural relative position guided transformer. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering, 2022. 13–24
174 Chen Q, Zhou M. A neural framework for retrieval and summarization of source code. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 826–831
175 Jiang S, Armaly A, McMillan C. Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017. 135–146
176 Jiang S, McMillan C. Towards automatic generation of short summaries of commits. In: Proceedings of the 25th International Conference on Program Comprehension, 2017. 320–323
177 Jiang S. Boosting neural commit message generation with code semantic analysis. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, 2019. 1280–1282
178 Liu Z, Xia X, Treude C, et al. Automatic generation of pull request descriptions. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, 2019. 176–188
179 Bansal A, Haque S, McMillan C. Project-level encoding for neural source code summarization of subroutines. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension, 2021. 253–264
180 Xie R, Ye W, Sun J, et al. Exploiting method names to improve code summarization: a deliberation multi-task learning approach. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension, 2021. 138–148
181 Hu X, Li G, Xia X, et al. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, 2018. 200–210
182 Hu X, Li G, Xia X, et al. Deep code comment generation with hybrid lexical and syntactical information. Empir Software Eng, 2020, 25: 2179–2217
183 Huang Y, Huang S, Chen H, et al. Towards automatically generating block comments for code snippets. Inf Software Tech, 2020, 127: 106373
184 Tang Z, Shen X, Li C, et al. AST-Trans: code summarization with efficient tree-structured attention. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering, 2022. 150–162
185 Liu S, Gao C, Chen S, et al. ATOM: commit message generation based on abstract syntax tree and hybrid ranking. IEEE Trans Software Eng, 2022, 48: 1800–1817
186 Wan Y, Zhao Z, Yang M, et al. Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 397–407
187 LeClair A, Jiang S, McMillan C. A neural model for generating natural language summaries of program subroutines. In: Proceedings of the 41st International Conference on Software Engineering, 2019. 795–806
188 Xu S, Yao Y, Xu F, et al. Commit message generation for source code changes. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence, 2019. 3975–3981
189 Zhou Y, Shen J, Zhang X, et al. Automatic source code summarization with graph attention networks. J Syst Softw, 2022, 188: 111257
190 Liang Y, Zhu K. Automatic generation of text descriptive comments for code blocks. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2018
191 Wang W, Zhang Y, Zeng Z, et al. TranS^3: a transformer-based framework for unifying code summarization and code search. 2020. ArXiv:2003.03238
192 Lin C, Ouyang Z, Zhuang J, et al. Improving code summarization with block-wise abstract syntax tree splitting. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension, 2021. 184–195
193 Shi E, Wang Y, Du L, et al. CAST: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2021. 4053–4062
194 Fernandes P, Allamanis M, Brockschmidt M. Structured neural summarization. In: Proceedings of the 7th International Conference on Learning Representations, 2019
195 LeClair A, Haque S, Wu L, et al. Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, Seoul, 2020. 184–195
196 Liu S, Chen Y, Xie X, et al. Retrieval-augmented generation for code summarization via hybrid GNN. In: Proceedings of the 9th International Conference on Learning Representations, 2021
197 Liu X, Wang D, Wang A Y, et al. HAConvGNN: hierarchical attention based convolutional graph neural network for code documentation generation in Jupyter notebooks. In: Proceedings of Findings of the Association for Computational Linguistics, 2021. 4473–4485
198 Cheng W, Hu P, Wei S, et al. Keyword-guided abstractive code summarization via incorporating structural and contextual information. Inf Software Tech, 2022, 150: 106987
199 Guo J, Liu J, Wan Y, et al. Modeling hierarchical syntax structure with triplet position for source code summarization. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022. 486–500
200 Ma Z, Gao Y, Lyu L, et al. MMF3: neural code summarization based on multi-modal fine-grained feature fusion. In: Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki Finland, 2022. 171–182
201 Wang Y, Dong Y, Lu X, et al. GypSum: learning hybrid representations for code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022. 12–23
202 Hu X, Li G, Xia X, et al. Summarizing source code with transferred API knowledge. In: Proceedings of the 27th International

Joint Conference on Artificial Intelligence, 2018. 2269–2275
203 Shahbazi R, Sharma R, Fard F H. API2Com: on the improvement of automatically generated code comments using API documentations. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension, 2021. 411–421
204 Gao X, Jiang X, Wu Q, et al. GT-SimNet: improving code automatic summarization via multi-modal similarity networks. J Syst Software, 2022, 194: 111495
205 Zhou Y, Yan X, Yang W, et al. Augmenting Java method comments generation with context information based on neural networks. J Syst Software, 2019, 156: 328–340
206 Wang W, Zhang Y, Sui Y, et al. Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Trans Software Eng, 2022, 48: 102–119
207 Wang Y, Du L, Shi E, et al. CoCoGUM: Contextual Code Summarization With Multi-Relational GNN on UMLs. Microsoft, Technical Report, MSR-TR-2020-16, 2020
208 Son J, Hahn J, Seo H, et al. Boosting code summarization by embedding code structures. In: Proceedings of the 29th International Conference on Computational Linguistics, 2022. 5966–5977
209 Zhang C, Zhou Q, Qiao M, et al. Re_Trans: combined retrieval and transformer model for source code summarization. Entropy, 2022, 24: 1372
210 Huang Y, Huang J, Chen X, et al. BCGen: a comment generation method for bytecode. Autom Softw Eng, 2023, 30: 5
211 Barone A V M, Sennrich R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In: Proceedings of the 8th International Joint Conference on Natural Language Processing, 2017. 314–319
212 Guo H Y, Chen X P, Huang Y, et al. Snippet comment generation based on code context expansion. ACM Trans Softw Eng Methodol, 2024, 33: 1–30
213 Fowler M, Beck K, Brant J, et al. Refactoring: Improving the Design of Existing Code. Redding: Addison-Wesley Professional, 1999
214 Tsantalis N, Chatzigeorgiou A. Identification of move method refactoring opportunities. IEEE Trans Software Eng, 2009, 35: 347–367
215 Terra R, Valente M T, Miranda S, et al. JMove: a novel heuristic and tool to detect move method refactoring opportunities. J Syst Software, 2018, 138: 19–36
216 Liu H, Xu Z, Zou Y. Deep learning based feature envy detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 385–396
217 Kurbatova Z, Veselov I, Golubev Y, et al. Recommendation of move method refactoring using path-based representation of code. In: Proceedings of the 4th International Workshop on Refactoring, 2020. 315–322
218 Sharma T, Efstathiou V, Louridas P, et al. Code smell detection by deep direct-learning and transfer-learning. J Syst Software, 2021, 176: 110936
219 Liu H, Jin J H, Xu Z F, et al. Deep learning based code smell detection. IEEE Trans Software Eng, 2021, 47: 1811–1837
220 LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521: 436–444
221 Wang X, Zhao Y, Pourpanah F. Recent advances in deep learning. Int J Mach Learn Cyber, 2020, 11: 747–750
222 Barbez A, Khomh F, Gúéhéneuc Y G. Deep learning anti-patterns from code metrics history. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, 2019. 114–124
223 Yu D, Xu Y, Weng L, et al. Detecting and refactoring feature envybased on graph neural network. In: Proceedings of the 33rd International Symposium on Software Reliability Engineering, 2022. 458–469
224 Alon U, Zilberstein M, Levy O, et al. Code2vec: learning distributed representations of code. In: Proceedings of the ACM on Programming Languages, 2019. 1–29
225 Cui D, Wang S, Luo Y, et al. RMove: recommending move method refactoring opportunities using structural and semantic representations of code. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, 2022. 281–292
226 Yedida R, Menzies T. On the value of oversampling for deep learning in software defect prediction. IEEE Trans Software Eng, 2022, 48: 3103–3116
227 Yedida R, Menzies T. How to improve deep learning for software analytics: (a case study with code smell detection). In: Proceedings of the 19th International Conference on Mining Software Repositories, 2022. 156–166
228 Liu H, Liu Q, Liu Y, et al. Identifying renaming opportunities by expanding conducted rename refactorings. IEEE Trans Software Eng, 2015, 41: 887–900
229 Liang J, Zou W, Zhang J, et al. A deep method renaming prediction and refinement approach for Java projects. In: Proceedings of the 21st International Conference on Software Quality, Reliability and Security), 2021. 404–413
230 Kenton J D M W C, Toutanova L K. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2019. 4171–4186
231 Rosenthal S, Farra N, Nakov P. SemEval-2017 task 4: sentiment analysis in Twitter. In: Proceedings of the 11th International Workshop on Semantic Evaluation, 2017. 502–518
232 Liu K, Kim D, Bissyandé T F, et al. Learning to spot and refactor inconsistent method names. In: Proceedings of the 41st International Conference on Software Engineering, 2019. 1–12
233 Le Q, Mikolov T. Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning, 2014. 1188–1196
234 Tufano M, Pantiuchina J, Watson C, et al. On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering, 2019. 25–36
235 Nyamawe A S, Liu H, Niu N, et al. Feature requests-based recommendation of software refactorings. Empir Software Eng, 2020, 25: 4315–4347
236 AlOmar E A, Ivanov A, Kurbatova Z, et al. Just-in-time code duplicates extraction. Inf Software Tech, 2023, 158: 107169
237 Chi X Y, Liu H, Li G J, et al. An automated approach to extracting local variables. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, 2023
238 Desai U, Bandyopadhyay S, Tamilselvam S. Graph neural network to dilute outliers for refactoring monolith application. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2021. 72–80
239 Madeyski L, Lewowski T. MLCQ: industry-relevant code smell data set. In: Proceedings of the 24th Evaluation and Assessment in Software Engineering, 2020. 342–347
240 Liu B, Liu H, Li G J, et al. Deep learning based feature envy detection boosted by real-world examples. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, 2023

241 Tsantalis N, Ketkar A, Dig D. RefactoringMiner 2.0. IEEE Trans Software Eng, 2022, 48: 930–950
242 Silva D, da Silva J P, Santos G, et al. RefDiff 2.0: a multi-language refactoring detection tool. IEEE Trans Software Eng, 2021, 47: 2786–2802
243 Kim M, Gee M, Loh A, et al. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, 2010. 371–372
244 Yin X, Shi C, Zhao S. Local and global feature based explainable feature envy detection. In: Proceedings of the IEEE 45th Annual Computers, Software, and Applications Conference, 2021. 942–951
245 Liu B, Liu H, Li G J, et al. Automated software entity matching between successive versions. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, 2023
246 Svajlenko J, Islam J F, Keivanloo I, et al. Towards a big data curated benchmark of inter-project code clones. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, 2014. 476–480
247 Chochlov M, Ahmed G A, Patten J V, et al. Using a nearest-neighbour, BERT-based approach for scalable clone detection. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022. 582–591
248 Sajnani H, Saini V, Svajlenko J, et al. SourcererCC: scaling code clone detection to big-code. In: Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016. 1157–1168
249 Arshad S, Abid S, Shamail S. CodeBERT for code clone detection: a replication study. In: Proceedings of the IEEE 16th International Workshop on Software Clones (IWSC), 2022. 39–45
250 Mehrotra N, Agarwal N, Gupta P, et al. Modeling functional similarity in source code with graph-based siamese networks. IEEE Trans Software Eng, 2022, 48: 3771–3789
251 Xue Z, Jiang Z, Huang C, et al. SEED: semantic graph based deep detection for Type-4 clone. In: Proceedings of Reuse and Software Quality, 2022. 120–137
252 Karthik S, Rajdeepa B. A collaborative method for code clone detection using a deep learning model. Adv Eng Software, 2022, 174: 103327
253 Li B, Ye C, Guan S, et al. Semantic code clone detection via event embedding tree and gat network. In: Proceedings of the IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020. 382–393
254 Zhang A, Liu K, Fang L, et al. Learn to align: a code alignment network for code clone detection. In: Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC), 2021. 1–11
255 Jo Y B, Lee J, Yoo C J. Two-pass technique for clone detection and type classification using tree-based convolution neural network. Appl Sci, 2021, 11: 6613
256 Kim D K. A deep neural network-based approach to finding similar code segments. IEICE Trans Inf Syst, 2020, E103.D: 874–878
257 Wu Y, Zou D, Dou S, et al. SCDetector: software functional clone detection based on semantic tokens analysis. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020. 821–833
258 Feng C, Wang T, Yu Y, et al. Sia-RAE: a siamese network based on recursive AutoEncoder for effective clone detection. In: Proceedings of the 27th Asia-Pacific Software Engineering Conference (APSEC), 2020. 238–246
259 Yuan Y, Kong W, Hou G, et al. From local to global semantic clone detection. In: Proceedings of the 6th International Conference on Dependable Systems and Their Applications (DSA), 2020. 13–24
260 Hua W, Sui Y, Wan Y, et al. FCCA: hybrid code representation for functional clone detection using attention networks. IEEE Trans Rel, 2021, 70: 304–318
261 Wang W, Li G, Ma B, et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020. 261–271
262 Fang C, Liu Z, Shi Y, et al. Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. 516–527
263 Guo C, Yang H, Huang D, et al. Review sharing via deep semi-supervised code clone detection. IEEE Access, 2020, 8: 24948–24965
264 Meng Y, Liu L. A deep learning approach for a source code detection model using self-attention. Complexity, 2020, 2020: 1–15
265 Zeng J, Ben K, Li X, et al. Fast code clone detection based on weighted recursive autoencoders. IEEE Access, 2019, 7: 125062
266 Zhang Y Y, Li M. Find me if you can: deep software clone detection by exploiting the contest between the plagiarist and the detector. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2019. 33: 5813–5820
267 Büch L, Andrzejak A. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019. 95–104
268 Yu H, Lam W, Chen L, et al. Neural detection of semantic code clones via tree-based convolution. In: Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019. 70–80
269 Wang C, Gao J, Jiang Y, et al. Go-clone: graph-embedding based clone detector for Golang. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019. 374–377
270 Shi H, Wang R, Fu Y, et al. Vulnerable code clone detection for operating system through correlation-induced learning. IEEE Trans Ind Inf, 2019, 15: 6551–6559
271 Saini V, Farmahinifarahani F, Lu Y, et al. Oreo: detection of clones in the twilight zone. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. 354–365
272 Zhao G, Huang J. DeepSim: deep learning code functional similarity. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. 141–151
273 Sheneamer A. CCDLC detection framework-combining clustering with deep learning classification for semantic clones. In: Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 2018. 701–706
274 Wei H H, Li M. Positive and unlabeled learning for detecting software functional clones with adversarial training. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, 2018. 2840–2846
275 Wei H H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, 2017. 3034–3040
276 White M, Tufano M, Vendome C, et al. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 87–98
277 Sheneamer A, Kalita J. Semantic clone detection using machine learning. In: Proceedings of the 15th IEEE International Conference on Machine Learning and Applications (ICMLA), 2016. 1024–1028
278 Zhang J, Wang X, Zhang H, et al. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019. 783–794
279 Wu M, Wang P, Yin K, et al. LVMapper: a large-variance clone detector using sequencing alignment approach. IEEE

Access, 2020, 8: 27986–27997
280 Li L, Feng H, Zhuang W, et al. CCLearner: a deep learning-based clone detection approach. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017. 249–260
281 Jiang L, Misherghi G, Su Z, et al. DECKARD: scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering, 2007. 96–105
282 Svajlenko J, Roy C K. Fast and flexible large-scale clone detection with cloneworks. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017. 27–30
283 Roy C K, Cordy J R. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008. 172–181
284 Kim S, Woo S, Lee H, et al. VUDDY: a scalable approach for vulnerable code clone discovery. In: Proceedings of the IEEE Symposium on Security and Privacy (SP), 2017. 595–614
285 Wang D, Jia Z, Li S, et al. Bridging pre-trained models and downstream tasks for source code understanding. In: Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022. 287–298
286 Siow J K, Liu S, Xie X, et al. Learning program semantics with code representations: an empirical study. In: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022. 554–565
287 Karakatič S, Miloševič A, Heričko T. Software system comparison with semantic source code embeddings. Empir Software Eng, 2022, 27: 70
288 Bui N D Q, Yu Y, Jiang L. InferCode: self-supervised learning of code representations by predicting subtrees. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. 1186–1197
289 Wu Q, Jiang X, Zheng Z, et al. Code representation based on hybrid graph modelling. In: Proceedings of Neural Information Processing. Cham: Springer International Publishing, 2021. 298–306
290 Chen L, Ye W, Zhang S. Capturing source code semantics via tree-based convolution over API-enhanced AST. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, 2019. 174–182
291 Gao Y, Wang Z, Liu S, et al. TECCD: a tree embedding approach for code clone detection. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019. 145–156
292 Tufano M, Watson C, Bavota G, et al. Deep learning similarities from different representations of source code. In: Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), 2018. 542–553
293 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. 1287–1293
294 Wang P, Svajlenko J, Wu Y, et al. CCAligner: a token based large-gap clone detector. In: Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018. 1066–1077
295 Terra R, Miranda L F, Valente M T, et al. Qualitas.class corpus: a compiled version of the qualitas corpus. SIGSOFT Softw Eng Notes, 2013, 38: 1–4
296 Yahya M A, Kim D K. CLCD-I: cross-language clone detection by using deep learning with InferCode. Computers, 2023, 12: 12
297 Wang K, Yan M, Zhang H, et al. Unified abstract syntax tree representation learning for cross-language program classification. In: Proceedings of the IEEE/ACM 30th International Conference on Program Comprehension (ICPC), 2022. 390–400
298 Bui N D Q, Yu Y, Jiang L. Bilateral dependency neural networks for cross-language algorithm classification. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019. 422–433
299 Nafi K W, Kar T S, Roy B, et al. CLCDSA: cross language code clone detection using syntactical features and API documentation. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019. 1026–1037
300 Bromley J, Guyon I, LeCun Y, et al. Signature verification using a "Siamese" time delay neural network. In: Proceedings of the 6th International Conference on Neural Information Processing Systems, San Francisco, 1993. 737–744
301 Vislavski T, Rakić G, Cardozo N, et al. LICCA: a tool for cross-language clone detection. In: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018. 512–516
302 Cheng X, Peng Z, Jiang L, et al. Mining revision histories to detect cross-language clones without intermediates. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016. 696–701
303 Marastoni N, Giacobazzi R, Preda M D. A deep learning approach to program similarity. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, 2018. 26–35
304 Xue H, Venkataramani G, Lan T. Clone-Slicer: detecting domain specific binary code clones through program slicing. In: Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation, 2018. 27–33
305 Xu X, Liu C, Feng Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2017. 363–376
306 Xue H, Venkataramani G, Lan T. Clone-hunter: accelerated bound checks elimination via binary code clone detection. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2018. 11–19
307 Feng Q, Zhou R, Xu C, et al. Scalable graph-based bug search for firmware images. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2016. 480–491
308 Mostaeen G, Svajlenko J, Roy B, et al. On the use of machine learning techniques towards the design of cloud based automatic code clone validation tools. In: Proceedings of the IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2018. 155–164
309 Saini V, Farmahinifarahani F, Lu Y, et al. Towards automating precision studies of clone detectors. In: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019. 49–59
310 Liu C, Lin Z, Lou J G, et al. Can neural clone detection generalize to unseen functionalities? In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021. 617–629
311 Yu H, Hu X, Li G, et al. Assessing and improving an evaluation dataset for detecting semantic code clones via deep learning. ACM Trans Softw Eng Methodol, 2022, 31: 1–25
312 Krinke J, Ragkhitwetsagul C. Bigclonebench considered harmful for machine learning. In: Proceedings of the IEEE 16th International Workshop on Software Clones (IWSC), 2022. 1–7
313 Al-Omari F, Roy C K, Chen T. SemanticCloneBench: a semantic code clone benchmark using crowd-source knowledge. In: Proceedings of the IEEE 14th International Workshop on Software Clones (IWSC), 2020. 57–63
314 Kamp M, Kreutzer P, Philippsen M. SeSaMe: a data set of semantically similar Java methods. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019. 529–533
315 Yang X, Lo D, Xia X, et al. Deep learning for just-in-time defect prediction. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security, 2015. 17–26
316 Phan A V, Nguyen M L, Bui L T. Convolutional neural networks over control flow graphs for software defect prediction.

In: Proceedings of the IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), 2017. 45–52
317 Li J, He P, Zhu J, et al. Software defect prediction via convolutional neural network. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017. 318–328
318 Huo X, Yang Y, Li M, et al. Learning semantic features for software defect prediction by code comments embedding. In: Proceedings of the IEEE International Conference on Data Mining (ICDM), 2018. 1049–1054
319 Liu Y, Li Y, Guo J, et al. Connecting software metrics across versions to predict defects. In: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018. 232–243
320 Tong H, Liu B, Wang S. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. Inf Software Tech, 2018, 96: 94–111
321 Qiu S, Lu L, Cai Z, et al. Cross-project defect prediction via transferable deep learning-generated and handcrafted features. In: Proceedings of International Conference on Software Engineering and Knowledge Engineering, 2019
322 Hoang T, Dam H K, Kamei Y, et al. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019. 34–45
323 Zhou T, Sun X, Xia X, et al. Improving defect prediction with deep forest. Inf Software Tech, 2019, 114: 204–216
324 Xu Z, Li S, Xu J, et al. LDFR: learning deep feature representation for software defect prediction. J Syst Software, 2019, 158: 110402
325 Turabieh H, Mafarja M, Li X. Iterated feature selection algorithms with layered recurrent neural network for software fault prediction. Expert Syst Appl, 2019, 122: 27–42
326 Dam H K, Pham T, Ng S W, et al. Lessons learned from using a deep tree-based model for software defect prediction in practice. In: Proceedings of the 16th International Conference on Mining Software Repositories, 2019. 46–57
327 Li H, Li X, Chen X, et al. Cross-project defect prediction via ASTToken2Vec and BLSTM-based neural network. In: Proceedings of the International Joint Conference on Neural Networks (IJCNN), 2019. 1–8
328 Chen J, Hu K, Yu Y, et al. Software visualization and deep transfer learning for effective software defect prediction. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 578–589
329 Zhu K, Zhang N, Ying S, et al. Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. IET Softw, 2020, 14: 185–195
330 Wang S, Liu T, Nam J, et al. Deep semantic feature learning for software defect prediction. IEEE Trans Software Eng, 2020, 46: 1267–1293
331 Deng J, Lu L, Qiu S. Software defect prediction via LSTM. IET softw, 2020, 14: 443–450
332 Shi K, Lu Y, Chang J, et al. PathPair2Vec: an AST path pair-based code representation method for defect prediction. J Comput Languages, 2020, 59: 100979
333 Majd A, Vahidi-Asl M, Khalilian A, et al. SLDeep: statement-level software defect prediction using deep-learning model on static code features. Expert Syst Appl, 2020, 147: 113156
334 Wen M, Wu R, Cheung S C. How well do change sequences predict defects? Sequence learning from software changes. IEEE Trans Software Eng, 2018, 46: 1155–1175
335 Shi K, Lu Y, Liu G, et al. MPT-embedding: an unsupervised representation learning of code for software defect prediction. J Software Evolu Process, 2021, 33: e2330
336 Xu Z, Zhao K, Zhang T, et al. Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. IEEE Trans Rel, 2022, 71: 204–220
337 Xu J, Wang F, Ai J. Defect prediction with semantics and context features of codes based on graph representation learning. IEEE Trans Rel, 2020, 70: 613–625
338 Zeng C, Zhou C Y, Lv S K, et al. GCN2defect: graph convolutional networks for SMOTETomek-based software defect prediction. In: Proceedings of the IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), 2021. 69–79
339 Xu J, Ai J, Liu J, et al. ACGDP: an augmented code graph-based system for software defect prediction. IEEE Trans Rel, 2022, 71: 850–864
340 Wang H, Zhuang W, Zhang X. Software defect prediction based on gated hierarchical LSTMs. IEEE Trans Rel, 2021, 70: 711–727
341 Zou Q, Lu L, Yang Z, et al. Joint feature representation learning and progressive distribution matching for cross-project defect prediction. Inf Software Tech, 2021, 137: 106588
342 Zhang N, Ying S, Zhu K, et al. Software defect prediction based on stacked sparse denoising autoencoders and enhanced extreme learning machine. IET Software, 2022, 16: 29–47
343 Uddin M N, Li B, Ali Z, et al. Software defect prediction employing BiLSTM and BERT-based semantic feature. Soft Comput, 2022, 26: 7877–7891
344 Ardimento P, Aversano L, Bernardi M L, et al. Just-in-time software defect prediction using deep temporal convolutional networks. Neural Comput Applic, 2022, 34: 3981–4001
345 Pornprasit C, Tantithamthavorn C K. DeepLineDP: towards a deep learning approach for line-level defect prediction. IEEE Trans Software Eng, 2023, 49: 84–98
346 Qiu S, Huang H, Jiang W, et al. Defect prediction via tree-based encoding with hybrid granularity for software sustainability. IEEE Trans Sustain Comput, 2024, 9: 249–260
347 Johnson S C. Lint, a C program checker. 1977. oai:CiteSeerX.psu:10.1.1.56.1841
348 Hovemeyer D, Pugh W. Finding bugs is easy. ACM SIGPLAN Not, 2004, 39: 92–106
349 Facebook. Infer: a tool to detect bugs in Java and C/C++/objective-C code before it ships, 2015. https://fbinfer.com/
350 Orso A, Rothermel G. Software testing: a research travelogue (2000–2014). In: Proceedings of Future of Software Engineering Proceedings, 2014
351 Cadar C, Dunbar D, Engler D R, et al. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008
352 Nelson L, Sigurbjarnarson H, Zhang K, et al. Hyperkernel: push-button verification of an OS kernel. In: Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP), 2017
353 Leroy X. Formal verification of a realistic compiler. Commun ACM, 2009, 52: 107–115
354 Klein G, Andronick J, Elphinstone K, et al. seL4: formal verification of an OS kernel. Commun ACM, 2010, 53: 107–115
355 D'Silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification. IEEE Trans Comput-Aided Des Integr Circ Syst, 2008, 27: 1165–1178
356 Knuth D E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd ed. Redding: Addison-Wesley Professional, 1997
357 Hou X, Zhao Y, Liu Y, et al. Large language models for software engineering: a systematic literature review. 2023. ArXiv:2308.10620
358 Fan A, Gokkaya B, Harman M, et al. Large language models for software engineering: survey and open problems. 2023.

ArXiv:2310.03533
359 Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529: 484–489
360 Qiao S, Ou Y, Zhang N, et al. Reasoning with language model prompting: a survey. 2022. ArXiv:2212.09597
361 Huang J, Chang K C C. Towards reasoning in large language models: a survey. 2022. ArXiv:2212.10403
362 Abelson H, Sussman G J. Structure and Interpretation of Computer Programs. 2nd ed. Cambridge: The MIT Press, 1996
363 Hindle A, Barr E T, Gabel M, et al. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2016
364 van Rossum G, Warsaw B, Coghlan N. PEP 8–style guide for python code. 2001. https://peps.python.org/pep-0008/
365 Reddy A. Java coding style guide, 2000
366 Engler D, Chen D Y, Hallem S, et al. Bugs as deviant behavior: a general approach to inferring errors in systems code. SIGOPS Oper Syst Rev, 2001, 35: 57–72
367 Li Z, Lu S, Myagmar S, et al. CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE Trans Software Eng, 2006, 32: 176–192
368 Allamanis M, Jackson-Flux H, Brockschmidt M. Self-supervised bug detection and repair. In: Proceedings of Advances in Neural Information Processing Systems, 2021. 27865–27876
369 Sharma T, Kechagia M, Georgiou S, et al. A survey on machine learning techniques for source code analysis. 2021. ArXiv:2110.09610v2
370 Jiang Y, Liu H, Zhang Y, et al. Do bugs lead to unnaturalness of source code? In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022
371 Rice H G. Classes of recursively enumerable sets and their decision problems. Trans Amer Math Soc, 1953, 74: 358–366
372 Livshits B, Sridharan M, Smaragdakis Y, et al. In defense of soundiness: a manifesto. Commun ACM, 2015, 58: 44–46
373 Heo K, Oh H, Yang H. Resource-aware program analysis via online abstraction coarsening. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019
374 Ko Y, Oh H. Learning to boost disjunctive static bug-finders. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
375 Li H, Hao Y, Zhai Y, et al. The hitchhiker's guide to program analysis: a journey with large language models. 2023. ArXiv:2308.00245
376 Chae K, Oh H, Heo K, et al. Automatically generating features for learning program analysis heuristics for C-like languages. In: Proceedings of the ACM on Programming Languages, 2017
377 Heo K, Oh H, Yi K. Machine-learning-guided selectively unsound static analysis. In: Proceedings of IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017
378 Jeon M, Lee M, Oh H. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. In: Proceedings of the ACM on Programming Languages, 2020
379 Jeong S, Jeon M, Cha S, et al. Data-driven context-sensitivity for points-to analysis. In: Proceedings of the ACM on Programming Languages, 2017
380 He J, Singh G, Püschel M, et al. Learning fast and precise numerical analysis. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020
381 Zaremba W, Sutskever I. Learning to execute. 2014. ArXiv:1410.4615
382 Malik R S, Patra J, Pradel M. NL2Type: inferring JavaScript function types from natural language information. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019
383 Jesse K, Devanbu P T, Ahmed T. Learning type annotation: is big data enough? In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021
384 Yu D, Yang B, Liu D, et al. A survey on neural-symbolic learning systems. Neural Netws, 2023, 166: 105–126
385 Wang W, Yang Y, Wu F. Towards data-and knowledge-driven AI: a survey on neuro-symbolic computing. IEEE Trans Pattern Anal Mach Intell, 2024. doi: 10.1109/TPAMI.2024.3483273
386 She D, Pei K, Epstein D, et al. NEUZZ: efficient fuzzing with neural program smoothing. In: Proceedings of the IEEE Symposium on Security and Privacy (SP), 2019. 803–817
387 She D, Krishna R, Yan L, et al. MTFuzz: fuzzing with a multi-task neural network. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 737–749
388 Wu M, Jiang L, Xiang J, et al. Evaluating and improving neural program-smoothing-based fuzzing. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 847–858
389 Nicolae M I, Eisele M, Zeller A. Revisiting neural program smoothing for fuzzing. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023
390 Zeller A. Mining specifications: a roadmap. In: The Future of Software Engineering. Berlin: Springer, 2011
391 Serebryany K, Bruening D, Potapenko A, et al. AddressSanitizer: a fast address sanity checker. In: Proceedings of USENIX Annual Technical Conference, 2012
392 Serebryany K, Iskhodzhanov T. ThreadSanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. 62–71
393 Jackson D. Software Abstractions: Logic, Language, and Analysis. Cambridge: The MIT Press, 2012
394 Lemieux C, Inala J P, Lahiri S K, et al. CODAMOSA: escaping coverage plateaus in test generation with pre-trained large language models. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
395 Khanfir A, Degiovanni R, Papadakis M, et al. Efficient mutation testing via pre-trained language models. 2023. ArXiv:2301.03543v1
396 Chen Z, Liu J, Gu W, et al. Experience report: deep learning-based system log analysis for anomaly detection. 2021. ArXiv:2107.05908
397 Wang J, Huang Y, Chen C, et al. Software testing with large language model: survey, landscape, and vision. 2023. ArXiv:2307.07221
398 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
399 Tufano M, Drain D, Svyatkovskiy A, et al. Unit test case generation with transformers and focal context. 2020. ArXiv:2009.05617v2
400 Watson C, Tufano M, Moran K, et al. On learning meaningful assert statements for unit test cases. In: Proceedings of IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020
401 Tufano M, Drain D, Svyatkovskiy A, et al. Generating accurate assert statements for unit test cases using pretrained transformers. 2022. ArXiv:2009.05634
402 Blasi A, Gorla A, Ernst M D, et al. Call Me Maybe: using NLP to automatically generate unit test cases respecting temporal

constraints. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022
403 Dinella E, Ryan G, Mytkowicz T, et al. TOGA: a neural method for test Oracle generation. 2022. ArXiv:2109.09262
404 Xie Z, Chen Y, Zhi C, et al. ChatUniTest: a ChatGPT-based automated unit test generation tool. 2023. ArXiv:2305.04764
405 Alagarsamy S, Tantithamthavorn C, Aleti A. A3Test: assertion-augmented automated test case generation. 2023. ArXiv:2302.10352
406 Feldmeier P, Fraser G. Neuroevolution-based generation of tests and oracles for games. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022
407 Schäfer M, Nadi S, Eghbali A, et al. Adaptive test generation using a large language model. 2023. ArXiv:2302.06527
408 Siddiq M L, Santos J, Tanvir R H, et al. Exploring the effectiveness of large language models in generating unit tests. 2023. ArXiv:2305.00418v1
409 Hossain S B, Filieri A, Dwyer M B, et al. Neural-based test oracle generation: a large-scale evaluation and lessons learned. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023
410 Liu Z, Liu K, Xia X, et al. Towards more realistic evaluation for neural test oracle generation. 2023. ArXiv:2305.17047
411 Yuan Z, Lou Y, Liu M, et al. No more manual tests? Evaluating and improving ChatGPT for unit test generation. 2023. ArXiv:2305.04207
412 Wong W E, Horgan J R, London S, et al. A study of effective regression testing in practice. In: Proceedings of the 8th International Symposium on Software Reliability Engineering, 1997
413 Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. Softw Test Verif Reliab, 2012, 22: 67–120
414 Manes V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: a survey. IEEE Trans Software Eng, 2021, 47: 2312–2331
415 Zhu X, Wen S, Camtepe S, et al. Fuzzing: a survey for roadmap. ACM Comput Surv, 2022, 54: 1–36
416 Li J, Zhao B, Zhang C. Fuzzing: a survey. Cybersecurity, 2018, 1: 6
417 Lee M, Cha S, Oh H. Learning seed-adaptive mutation strategies for greybox fuzzing. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
418 Wang J, Song C, Yin H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In: Proceedings of Network and Distributed Systems Security (NDSS) Symposium, 2021
419 Wang Y, Wu Z, Wei Q, et al. NeuFuzz: efficient fuzzing with deep neural network. IEEE Access, 2019, 7: 36340–36352
420 Deng Y, Xia C S, Peng H, et al. Large language models are zero-shot fuzzers: fuzzing deep-learning libraries via large language models. 2023. ArXiv:2212.14834
421 Deng Y, Xia C S, Yang C, et al. Large language models are edge-case fuzzers: testing deep learning libraries via FuzzGPT. 2023. ArXiv:2304.02014
422 Yang C, Deng Y, Lu R, et al. White-box compiler fuzzing empowered by large language models. 2023. ArXiv:2310.15991
423 Xia C S, Paltenghi M, Tian J L, et al. Universal fuzzing via large language models. 2023. ArXiv:2308.04748v1
424 Ye G, Tang Z, Tan S H, et al. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021. 435–450
425 Cummins C, Petoumenos P, Murray A, et al. Compiler fuzzing through deep learning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018
426 Lin M, Zeng Y, Li Y. RegFuzz: a linear regression-based approach for seed scheduling in directed fuzzing. In: Proceedings of the 4th Information Communication Technologies Conference (ICTC), 2023
427 Meng R, Mirchev M, Böhme M, et al. Large language model guided protocol fuzzing. In: Proceedings of Network and Distributed System Security (NDSS) Symposium, 2024
428 Su J, Dai H N, Zhao L, et al. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022
429 Luo W, Chai D, Ruan X, et al. Graph-based fuzz testing for deep learning inference engines. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021
430 Chen Y, Poskitt C M, Sun J, et al. Learning-guided network fuzzing for testing cyber-physical system defences. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019
431 Jiang L, Yuan H, Wu M, et al. Evaluating and improving hybrid fuzzing. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
432 He J, Balunović M, Ambroladze N, et al. Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2019. 531–548
433 Jia H, Wen M, Xie Z, et al. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
434 Zheng Y, Liu Y, Xie X, et al. Automatic web testing using curiosity-driven reinforcement learning. In: Proceedings of the 43rd International Conference on Software Engineering, 2021. 423–435
435 Zhang S, Liu S, Sun J, et al. FIGCPS: effective failure-inducing input generation for cyber-physical systems with deep reinforcement learning. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021
436 Liu Z, Chen C, Wang J, et al. Fill in the blank: context-aware automated text input generation for mobile GUI testing. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
437 YazdaniBanafsheDaragh F, Malek S. Deep GUI: black-box GUI input generation with deep learning. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021
438 Feng S, Xie M, Chen C. Efficiency matters: speeding up automated testing with GUI rendering inference. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
439 Ran D, Wang H, Wang W, et al. Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
440 Pan M, Huang A, Wang G, et al. Reinforcement learning based curiosity-driven testing of Android applications. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. 153–164
441 Zhao Y, Talebipour S, Baral K, et al. Avgust: automating usage-based test generation from videos of app executions. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022
442 Wang X, Zhao L. APICAD: augmenting API misuse detection through specifications from code and documents. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023
443 Kim M, Corradini D, Sinha S, et al. Enhancing REST API testing with NLP techniques. In: Proceedings of the 32nd ACM

SIGSOFT International Symposium on Software Testing and Analysis, 2023
444 Kim M, Sinha S, Orso A. Adaptive REST API testing with reinforcement learning. 2023. ArXiv:2309.04583
445 Alyahya T N, Menai M E B, Mathkour H. On the structure of the boolean satisfiability problem: a survey. ACM Comput Surv, 2023, 55: 1–34
446 Guo W, Zhen H L, Li X, et al. Machine learning methods in solving the Boolean satisfiability problem. Mach Intell Res, 2023, 20: 640–655
447 Avgerinos T, Rebert A, Cha S K, et al. Enhancing symbolic execution with veritesting. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 1083–1094
448 Baldoni R, Coppa E, D'elia D C, et al. A survey of symbolic execution techniques. ACM Comput Surv, 2019, 51: 1–39
449 He J, Sivanrupan G, Tsankov P, et al. Learning to explore paths for symbolic execution. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2021
450 Cha S, Oh H. Concolic testing with adaptively changing search heuristics. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019
451 Cha S, Hong S, Lee J, et al. Automatically generating search heuristics for concolic testing. In: Proceedings of IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018
452 Zhang T, Zhang Y, Chen Z, et al. Efficient multiplex symbolic execution with adaptive search strategy. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020
453 Cha S, Oh H. Making symbolic execution promising by learning aggressive state-pruning strategy. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020
454 Chen Z, Chen Z, Shuai Z, et al. Synthesize solving strategy for symbolic execution. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021. 348–360
455 Luo S, Xu H, Bi Y, et al. Boosting symbolic execution via constraint solving time prediction (experience paper). In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021. 336–347
456 Cha S, Lee M, Lee S, et al. SYMTUNER: maximizing the power of symbolic execution by adaptively tuning external parameters. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022
457 Chen J, Hu W, Zhang L, et al. Learning to accelerate symbolic execution via code transformation. In: Proceedings of the 32nd European Conference on Object-Oriented Programming, 2018
458 Development team T C. The Coq proof assistant. 1984. https://coq.inria.fr/coq-84
459 Development team T I. Isabelle. 1986. https://isabelle.in.tum.de/index.html
460 Paulson L C. Natural deduction as higher-order resolution. 1986. ArXiv:cs/9301104
461 Lample G, Lachaux M A, Lavril T, et al. HyperTree proof search for neural theorem proving. 2022. ArXiv:2205.11491
462 Wu Y, Jiang A Q, Li W, et al. Autoformalization with large language models. In: Proceedings of Advances in Neural Information Processing Systems, 2022
463 First E, Brun Y. Diversity-driven automated formal verification. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 749–761
464 Yang K, Swope A M, Gu A, et al. LeanDojo: theorem proving with retrieval-augmented language models. 2023. ArXiv:2306.15626
465 Chakraborty S, Lahiri S K, Fakhoury S, et al. Ranking LLM-generated loop invariants for program verification. 2023. ArXiv:2310.09342
466 Zimmeck S, Wang Z, Zou L, et al. Automated analysis of privacy requirements for mobile apps. In: Proceedings of the AAAI Fall Symposium Series, 2016
467 Mahanipour A, Nezamabadi-pour H. GSP: an automatic programming technique with gravitational search algorithm. Appl Intell, 2019, 49: 1502–1516
468 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. 26
469 Liu S, Zhao B, Guo R, et al. Have you been properly notified? Automatic compliance analysis of privacy policy text with GDPR article 13. In: Proceedings of the Web Conference 2021, 2021. 2154–2164
470 Rubio-González C, Liblit B. Expect the unexpected: error code mismatches between documentation and the real world. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2010. 73–80
471 Tan L, Yuan D, Krishna G, et al. /*icomment: bugs or bad comments?*/. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, 2007. 145–158
472 Tan S H, Marinov D, Tan L, et al. @tComment: testing Javadoc comments to detect comment-code inconsistencies. In: Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation, 2012. 260–269
473 Wen F, Nagy C, Bavota G, et al. A large-scale empirical study on code-comment inconsistencies. In: Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019. 53–64
474 Pandita R, Taneja K, Williams L, et al. ICON: inferring temporal constraints from natural language API descriptions. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016. 378–388
475 Ren X, Ye X, Xing Z, et al. API-misuse detection driven by fine-grained API-constraint knowledge graph. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 461–472
476 Lv T, Li R, Yang Y, et al. RTFM! automatic assumption discovery and verification derivation from library document for API misuse detection. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2020. 1837–1852
477 Yun I, Min C, Si X, et al. APISan: sanitizing API usages through semantic cross-checking. In: Proceedings of Usenix Security Symposium, 2016. 363–378
478 Kang Y, Ray B, Jana S. APEx: automated inference of error specifications for C APIs. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 472–482
479 Li C, Zhou M, Gu Z, et al. Ares: inferring error specifications through static analysis. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019. 1174–1177
480 Takanen A, Demott J D, Miller C, et al. Fuzzing for Software Security Testing and Quality Assurance. Norwood: Artech House, Inc. 2018
481 You W, Zong P, Chen K, et al. SemFuzz: semantics-based automatic generation of proof-of-concept exploits. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2017. 2139–2154
482 Godefroid P, Peleg H, Singh R. Learn&Fuzz: machine learning for input fuzzing. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017. 50–59
483 Liu X, Li X, Prajapati R, et al. DeepFuzz: automatic generation of syntax valid C programs for fuzz testing. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2019. 1044–1051

484 Lee S, Han H, Cha S K, et al. Montage: a neural network language model-guided JavaScript engine fuzzer. In: Proceedings of the 29th USENIX Conference on Security Symposium, 2020. 2613–2630
485 Chen P, Chen H. Angora: efficient fuzzing by principled search. In: Proceedings of the IEEE Symposium on Security and Privacy (SP), 2018. 711–725
486 Funahashi K I. On the approximate realization of continuous mappings by neural networks. Neural Netws, 1989, 2: 183–192
487 Nagy S, Hicks M. Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing. In: Proceedings of the IEEE Symposium on Security and Privacy (SP), 2019. 787–802
488 Zhou C, Wang M, Liang J, et al. Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 858–870
489 Zong P, Lv T, Wang D, et al. FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: Proceedings of the 29th USENIX Conference on Security Symposium, 2020. 2255–2269
490 Jung R, Jourdan J H, Krebbers R, et al. Safe systems programming in Rust. Commun ACM, 2021, 64: 144–152
491 Wong W E, Gao R, Li Y, et al. A survey on software fault localization. IEEE Trans Software Eng, 2016, 42: 707–740
492 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
493 Xie X, Liu Z, Song S, et al. Revisit of automatic debugging via human focus-tracking analysis. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 808–819
494 Agrawal H, Horgan J, London S, et al. Fault localization using execution slices and dataflow tests. In: Proceedings of the 6th International Symposium on Software Reliability Engineering, 1995. 143–151
495 Wong C P, Xiong Y, Zhang H, et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, 2014. 181–190
496 Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proceedings of the 28th International Conference on Software Engineering, New York, 2006. 272–281
497 Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, 2002. 467–477
498 Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation. ACM SIGPLAN Not, 2005, 40: 15–26
499 Abreu R, Zoeteweij P, Golsteijn R, et al. A practical evaluation of spectrum-based fault localization. J Syst Software, 2009, 82: 1780–1792
500 Xie X Y, 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
501 Zou D, Liang J, Xiong Y, et al. An empirical study of fault localization families and their combinations. IEEE Trans Software Eng, 2019, 47: 332–347
502 Widyasari R, Prana G A A, Haryono S A, et al. XAI4FL: enhancing spectrum-based fault localization with explainable artificial intelligence. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022. 499–510
503 Moon S, Kim Y, Kim M, et al. Ask the mutants: mutating faulty programs for fault localization. In: Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation, 2014. 153–162
504 Papadakis M, Traon Y L. Metallaxis-FL: mutation-based fault localization. Software Testing Verif Rel, 2015, 25: 605–628
505 Wong W E, Qi Y U. Bp neural network-based effective fault localization. Int J Soft Eng Knowl Eng, 2009, 19: 573–597
506 Wong W E, Debroy V, Golden R, et al. Effective software fault localization using an RBF neural network. IEEE Trans Rel, 2012, 61: 149–169
507 Zheng W, Hu D, Wang J. Fault localization analysis based on deep neural network. Math Problems Eng, 2016, 2016: 1–11
508 Zhang Z, Lei Y, Tan Q, et al. Deep learning-based fault localization with contextual information. IEICE Trans Inf Syst, 2017, E100.D: 3027–3031
509 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
510 Zhang Z, Lei Y, Mao X G, et al. CNN-FL: an effective approach for localizing faults using convolutional neural networks. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019
511 Li Y, Wang S, Nguyen T. Fault localization with code coverage representation learning. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. 661–673
512 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
513 Qian J, Ju X, Chen X, et al. AGFL: a graph convolutional neural network-based method for fault localization. In: Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021. 672–680
514 Qian J, Ju X, Chen X. GNet4FL: effective fault localization via graph convolutional neural network. Autom Softw Eng, 2023, 30: 16
515 Zhang Z, Lei Y, Mao X, et al. Context-aware neural fault localization. IEEE Trans Software Eng, 2023, 49: 3939–3954
516 Li Y, Wang S, Nguyen T N. Fault localization to detect co-change fixing locations. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, 2022. 659–671
517 Dutta A, Manral R, Mitra P, et al. Hierarchically localizing software faults using DNN. IEEE Trans Rel, 2020, 69: 1267–1292
518 Yu J, Lei Y, Xie H, et al. Context-based cluster fault localization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, New York, 2022. 482–493
519 Li Z, Tang E, Chen X, et al. Graph neural network based two-phase fault localization approach. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware, 2022. 85–95
520 Yousofvand L, Soleimani S, Rafe V. Automatic bug localization using a combination of deep learning and model transformation through node classification. Software Qual J, 2023, 31: 1045–1063
521 Wu S, Li Z, Liu Y, et al. GMBFL: optimizing mutation-based fault localization via graph representation. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2023. 245–257
522 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 (ASE), 2020. 1214–1218
523 Ciborowska A, Damevski K. Fast changeset-based bug localization with BERT. In: Proceedings of the 44th International Conference on Software Engineering, New York, 2022. 946–957
524 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
525 Zhong H, Mei H. Learning a graph-based classifier for fault localization. Sci China Inf Sci, 2020, 63: 162101

526 Zhang Z, Lei Y, Mao X, et al. Improving deep-learning-based fault localization with resampling. J Software Evolu Process, 2021, 33: e2312
527 Xie H, Lei Y, Yan M, et al. A universal data augmentation approach for fault localization. In: Proceedings of the 44th International Conference on Software Engineering, New York, 2022. 48–60
528 Hu J, Xie H, Lei Y, et al. A light-weight data augmentation method for fault localization. Inf Software Tech, 2023, 157: 107148
529 Lei Y, Liu C, Xie H, et al. BCL-FL: a data augmentation approach with between-class learning for fault localization. In: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022. 289–300
530 Lei Y, Wen T, Xie H, et al. Mitigating the effect of class imbalance in fault localization using context-aware generative adversarial network. In: Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension, 2023
531 Zhang Z, Lei Y, Su T, et al. Influential global and local contexts guided trace representation for fault localization. ACM Trans Softw Eng Methodol, 2023, 32: 1–27
532 Tian Z, Chen J, Zhu Q, et al. Learning to construct better mutation faults. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. 1–13
533 Zhang Z, Lei Y, Mao X, et al. Improving fault localization using model-domain synthesized failing test generation. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022. 199–210
534 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
535 Madeiral F, Urli S, Maia M, et al. BEARS: an extensible Java bug benchmark for automatic program repair studies. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019. 468–478
536 Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir Software Eng, 2005, 10: 405–435
537 Goues C L, Holtschulte N, Smith E K, et al. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans Software Eng, 2015, 41: 1236–1256
538 Weiß C, Premraj R, Zimmermann T, et al. How long will it take to fix this bug? In: Proceedings of the 4th International Workshop on Mining Software Repositories, 2007
539 Gazzola L, Micucci D, Mariani L. Automatic software repair: a survey. IEEE Trans Software Eng, 2019, 45: 34–67
540 Xuan J, Ren Z, Wang Z, et al. Progress on approaches to automatic program repair (in Chinese). J Software, 2016, 27: 771–784
541 Monperrus M. The Living Review on Automated Program Repair. Research Report hal-01956501, HAL Archives Ouvertes, 2018. Version: 5
542 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
543 Kern C, Esparza J. Automatic error correction of Java programs. In: Proceedings of the 15th International Workshop on Formal Methods for Industrial Critical Systems, 2010. 67–81
544 Tian Y, Ray B. Automatically diagnosing and repairing error handling bugs in C. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, 2017. 752–762
545 Carvalho A, Luz W P, Marcilio D, et al. C-3PR: a bot for fixing static analysis violations via pull requests. In: Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, 2020. 161–171
546 Aho A V, Peterson T G. A minimum distance error-correcting parser for context-free languages. SIAM J Comput, 1972, 1: 305–312
547 Graham S L, Rhodes S P. Practical syntactic error recovery. In: Proceedings of Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, 1973. 52–58
548 Anderson S O, Backhouse R C. Locally least-cost error recovery in Earley's algorithm. ACM Trans Program Lang Syst, 1981, 3: 318–347
549 Burke M G, Fisher G A. A practical method for LR and LL syntactic error diagnosis and recovery. ACM Trans Program Lang Syst, 1987, 9: 164–197
550 Gupta R, Pal S, Kanade A, et al. DeepFix: fixing common C language errors by deep learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence, San Francisco, 2017. 1345–1351
551 Bhatia S, Kohli P, Singh R. Neuro-symbolic program corrector for introductory programming assignments. In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 60–70
552 Ahmed U Z, Kumar P, Karkare A, et al. Compilation error repair: for the student programs, from the student programs. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, 2018. 78–87
553 Santos E A, Campbell J C, Patel D, et al. Syntax and sensibility: using language models to detect and correct syntax errors. In: Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, 2018. 311–322
554 Brown N C C, Kölling M, McCall D, et al. Blackbox: a large scale repository of novice programmers' activity. In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education, Atlanta, 2014. 223–228
555 Mesbah A, Rice A, Johnston E, et al. DeepDelta: learning to repair compilation errors. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, 2019. 925–936
556 Gupta R, Kanade A, Shevade S K. Deep reinforcement learning for syntactic error repair in student programs. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence, the 31st Innovative Applications of Artificial Intelligence Conference, and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, 2019. 930–937
557 Wu L, Li F, Wu Y, et al. GGF: a graph-based method for programming language syntax error correction. In: Proceedings of the 28th International Conference on Program Comprehension, Seoul, 2020. 139–148
558 Yasunaga M, Liang P. Graph-based, self-supervised program repair from diagnostic feedback. In: Proceedings of the 37th International Conference on Machine Learning, 2020. 10799–10808
559 Hajipour H, Bhattacharyya A, Staicu C, et al. SampleFix: learning to generate functionally diverse fixes. In: Proceedings of Machine Learning and Principles and Practice of Knowledge Discovery in Databases, 2021. 119–133
560 Yasunaga M, Liang P. Break-it-fix-it: unsupervised learning for program repair. In: Proceedings of the 38th International Conference on Machine Learning, 2021. 11941–11952
561 Ahmed T, Devanbu P, Hellendoorn V J. Learning lenient parsing & typing via indirect supervision. Empir Software Eng, 2021, 26: 29
562 Sakkas G, Endres M, Guo P J, et al. Seq2Parse: neurosymbolic parse error repair. Proc ACM Program Lang, 2022, 6: 1180–1206

563 Li X, Liu S, Feng R, et al. TransRepair: context-aware program repair for compilation errors. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, 2022. 1–13
564 Ahmed T, Ledesma N R, Devanbu P. SynShine: improved fixing of syntax errors. IEEE Trans Software Eng, 2023, 49: 2169–2181
565 Liu Z, Lin W, Shi Y, et al. A robustly optimized BERT pre-training approach with post-training. In: Proceedings of the 20th China National Conference on Chinese Computational Linguistics, Hohhot, 2021. 471–484
566 Gu Y F, Ma P, Jia X Y, et al. Progress on software crash research (in Chinese). Sci Sin Inform, 2019, 49: 1383–1398
567 Goues C L, Nguyen T V, Forrest S, et al. GenProg: a generic method for automatic software repair. IEEE Trans Software Eng, 2012, 38: 54–72
568 Wong C, Santiesteban P, Kästner C, et al. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In: Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, 2021. 354–366
569 Nguyen H D T, Qi D, Roychoudhury A, et al. SemFix: program repair via semantic analysis. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 772–781
570 Mechtaev S, Yi J, Roychoudhury A. Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, Austin, 2016. 691–701
571 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
572 Tan S H, Roychoudhury A. relifix: automated repair of software regressions. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, Florence, 2015. 471–482
573 Saha S, Saha R K, Prasad M R. Harnessing evolution for multi-hunk program repair. In: Proceedings of the 41st International Conference on Software Engineering, Montreal, 2019. 13–24
574 Liu K, Koyuncu A, Kim D, et al. TBar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, 2019. 31–42
575 White M, Tufano M, Martinez M, et al. Sorting and transforming program repair ingredients via deep learning code similarities. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, Hangzhou, 2019. 479–490
576 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, 47: 1943–1959
577 Jiang N, Lutellier T, Tan L. CURE: code-aware neural machine translation for automatic program repair. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, Madrid, 2021. 1161–1173
578 Long F, Rinard M C. 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
579 Goues C L, 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
580 Tufano M, Watson C, Bavota G, et al. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, 2018. 832–837
581 Sun Z, Xin C, Sun Y. An automatic semantic code repair service based on deep learning for programs with single error. In: Proceedings of the IEEE World Congress on Services, Milan, 2019. 360–361
582 Ding Y, Ray B, Devanbu P T, et al. Patching as translation: the data and the metaphor. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, 2020. 275–286
583 Yang G, Min K, Lee B. Applying deep learning algorithm to automatic bug localization and repair. In: Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing, 2020. 1634–1641
584 Yu L, Zhang W, Wang J, et al. SeqGAN: sequence generative adversarial nets with policy gradient. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence, San Francisco, 2017. 2852–2858
585 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
586 Martinez M, Durieux T, Sommerard R, et al. Automatic repair of real bugs in Java: a large-scale experiment on the defects4j dataset. Empir Software Eng, 2017, 22: 1936–1964
587 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, Gothenburg, 2018. 10–13
588 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, Melbourne, 2020. 981–992
589 Dinella E, Dai H, Li Z, et al. Hoppity: learning graph transformations to detect and fix bugs in programs. In: Proceedings of the 8th International Conference on Learning Representations, Addis Ababa, 2020
590 Tang Y, Zhou L, Blanco A, et al. Grammar-based patches generation for automated program repair. In: Proceedings of Findings of the Association for Computational Linguistics, 2021. 1300–1305
591 Huang S, Zhou X, Chin S. Application of Seq2Seq models on code correction. Front Artif Intell, 2021, 4: 590215
592 Rahman M M, Watanobe Y, Nakamura K. A bidirectional LSTM language model for code evaluation and repair. Symmetry, 2021, 13: 247
593 Berabi B, He J, Raychev V, et al. TFix: learning to fix coding errors with a text-to-text transformer. In: Proceedings of the 38th International Conference on Machine Learning, 2021. 780–791
594 Tang B, Li B, Bo L, et al. GrasP: graph-to-sequence learning for automated program repair. In: Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security, Hainan, 2021. 819–828
595 Szalontai B, Vadász A, Borsi Z R, et al. Detecting and fixing nonidiomatic snippets in Python source code with deep learning. In: Proceedings of Intelligent Systems and Applications, Amsterdam, 2021. 129–147
596 Li Y, Wang S, Nguyen T N. DEAR: a novel deep learning-based approach for automated program repair. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering, Pittsburgh, 2022. 511–523
597 Xu X, Wang X, Xue J. M3V: multi-modal multi-view context embedding for repair operator prediction. In: Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization, Seoul, 2022. 266–277
598 Meng X, Wang X, Zhang H, et al. Improving fault localization and program repair with deep semantic features and transferred knowledge. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering, Pittsburgh, 2022. 1169–1180
599 Kim M, Kim Y, Heo J, et al. Impact of defect instances for successful deep learning-based automatic program repair. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution, Limassol, 2022. 419–423

600 Wardat M, Cruz B D, Le W, et al. DeepDiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering, Pittsburgh, 2022. 561–572

601 Yao J, Rao B, Xing W, et al. Bug-Transformer: automated program repair using attention-based deep neural network. J Circuit Syst Comp, 2022, 31: 2250210

602 Yan D, Liu K, Niu Y, et al. Crex: predicting patch correctness in automated repair of C programs through transfer learning of execution semantics. Inf Software Tech, 2022, 152: 107043

603 Pei K, Xuan Z, Yang J, et al. Learning approximate execution semantics from traces for binary function similarity. IEEE Trans Software Eng, 2023, 49: 2776–2790

604 Chakraborty S, Ding Y, Allamanis M, et al. CODIT: code editing with tree-based neural models. IEEE Trans Software Eng, 2022, 48: 1385–1399

605 Ye H, Martinez M, Monperrus M. Neural program repair with execution-based backpropagation. In: Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, 2022. 1506–1518

606 Ye H, Gu J, Martinez M, et al. Automated classification of overfitting patches with statically extracted code features. IEEE Trans Software Eng, 2022, 48: 2920–2938

607 Ye H, Martinez M, Luo X, et al. SelfAPR: self-supervised program repair with test execution diagnostics. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, 2022. 1–13

608 Xia C S, Zhang L. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 2022. 959–971

609 Kim M, Kim Y, Jeong H, et al. An empirical study of deep transfer learning-based program repair for Kotlin projects. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 2022. 1441–1452

610 Tian H, Li Y, Pian W, et al. Predicting patch correctness based on the similarity of failing test cases. ACM Trans Softw Eng Methodol, 2022, 31: 1–30

611 Yuan W, Zhang Q, He T, et al. CIRCLE: continual repair across programming languages. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022. 678–690

612 Chen L, Pei Y, Pan M, et al. Program repair with repeated learning. IEEE Trans Software Eng, 2023, 49: 831–848

613 Stocco A, Yandrapally R, Mesbah A. Visual web test repair. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, 2018. 503–514

614 Pan M, Xu T, Pei Y, et al. GUI-guided test script repair for mobile apps. IEEE Trans Software Eng, 2022, 48: 910–929

615 Ren Z, Sun S, Xuan J, et al. Automated patching for unreproducible builds. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering, Pittsburgh, 2022. 200–211

616 Hassan F, Wang X. HireBuild: an automatic approach to history-driven repair of build scripts. In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 1078–1089

617 Lou Y, Chen J, Zhang L, et al. History-driven build failure fixing: how far are we? In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019. 43–54

618 Loriot B, Madeiral F, Monperrus M. Styler: learning formatting conventions to repair Checkstyle violations. Empir Software Eng, 2022, 27: 149

619 Ma S, Thung F, Lo D, et al. VuRLE: automatic vulnerability detection and repair by learning from examples. In: Proceedings of the 22nd European Symposium on Research in Computer Security, Oslo, 2017. 229–246

620 Harer J, Ozdemir O, Lazovich T, et al. Learning to repair software vulnerabilities with generative adversarial networks. In: Proceedings of Advances in Neural Information Processing Systems, 2018. 7944–7954

621 Zhou Z, Bo L, Wu X, et al. SPVF: security property assisted vulnerability fixing via attention-based models. Empir Software Eng, 2022, 27: 171

622 Huang K, Yang S, Sun H, et al. Repairing security vulnerabilities using pre-trained programming language models. In: Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2022. 111–116

623 Chen Z, Kommrusch S, Monperrus M. Neural transfer learning for repairing security vulnerabilities in C code. IEEE Trans Software Eng, 2023, 49: 147–165

624 Chi J, Qu Y, Liu T, et al. SeqTrans: automatic vulnerability fix via sequence to sequence learning. IEEE Trans Software Eng, 2023, 49: 564–585

625 Das R, Ahmed U Z, Karkare A, et al. Prutor: a system for tutoring CS1 and collecting student programs for analysis. 2016. ArXiv:1608.03828

626 Brown N C C, Altadmri A, Sentance S, et al. Blackbox, five years on: an evaluation of a large-scale programming data collection project. In: Proceedings of the ACM Conference on International Computing Education Research, New York, 2018. 196–204

627 Motwani M, Sankaranarayanan S, Just R, et al. Do automated program repair techniques repair hard and important bugs? In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 25

628 Jiang Y, Liu H, Niu N, et al. Extracting concise bug-fixing patches from human-written patches in version control systems. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE'21), 2021

629 Jiang Y, Liu H, Luo X, et al. BugBuilder: an automated approach to building bug repository. IEEE Trans Software Eng, 2023, 49: 1443–1463

630 Bui Q C, Scandariato R, Ferreyra N E D. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In: Proceedings of the IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), 2022. 464–468

631 Nikitopoulos G, Dritsa K, Louridas P, et al. CrossVul: a cross-language vulnerability dataset with commit data. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021. 1565–1569

632 Zou W, Lo D, Chen Z, et al. How practitioners perceive automated bug report management techniques. IEEE Trans Software Eng, 2018, 46: 836–862

633 Bettenburg N, Just S, Schröter A, et al. What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008. 308–318

634 Lee D G, Seo Y S. Systematic review of bug report processing techniques to improve software management performance. J Inf Process Syst, 2019, 15: 967–985

635 Anvik J. Automating bug report assignment. In: Proceedings of the 28th International Conference on Software Engineering, 2006. 937–940

636 Jiang H, Li X, Ren Z, et al. Toward better summarizing bug reports with crowdsourcing elicited attributes. IEEE Trans Rel, 2018, 68: 2–22

637 Tan Y, Xu S, Wang Z, et al. Bug severity prediction using question-and-answer pairs from Stack Overflow. J Syst Software, 2020, 165: 110567
638 Zhang T, Han D, Vinayakarao V, et al. Duplicate bug report detection: how far are we? ACM Trans Softw Eng Methodol, 2023, 32: 1–32
639 Li X, Jiang H, Liu D, et al. Unsupervised deep bug report summarization. In: Proceedings of the 26th Conference on Program Comprehension, 2018. 144–155
640 Fang F, Wu J, Li Y, et al. On the classification of bug reports to improve bug localization. Soft Comput, 2021, 25: 7307–7323
641 Zhou C, Li B, Sun X, et al. Leveraging multi-level embeddings for knowledge-aware bug report reformulation. J Syst Software, 2023, 198: 111617
642 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
643 Xiao G, Du X, Sui Y, et al. HINDBR: heterogeneous information network based duplicate bug report prediction. In: Proceedings of the IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), 2020. 195–206
644 Xie Q, Wen Z, Zhu J, et al. Detecting duplicate bug reports with convolutional neural networks. In: Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC), 2018. 416–425
645 Deshmukh J, Annervaz K, Podder S, et al. Towards accurate duplicate bug retrieval using deep learning techniques. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017. 115–124
646 Budhiraja A, Dutta K, Reddy R, et al. DWEN: deep word embedding network for duplicate bug report detection in software repositories. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, 2018. 193–194
647 Isotani H, Washizaki H, Fukazawa Y, et al. Duplicate bug report detection by using sentence embedding and fine-tuning. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021. 535–544
648 Jiang Y, Su X, Treude C, et al. Does deep learning improve the performance of duplicate bug report detection? An empirical study. J Syst Software, 2023, 198: 111607
649 Koc U, Wei S, Foster J S, et al. An empirical assessment of machine learning approaches for triaging reports of a Java static analysis tool. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019. 288–299
650 Florea A C, Anvik J, Andonie R. Parallel implementation of a bug report assignment recommender using deep learning. In: Proceedings of the 26th International Conference on Artificial Neural Networks and Machine Learning, 2017. 64–71
651 Lee S R, Heo M J, Lee C G, et al. Applying deep learning based automatic bug triager to industrial projects. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, 2017
652 Mani S, Sankaran A, Aralikatte R. DeepTriage: exploring the effectiveness of deep learning for bug triaging. In: Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, 2019. 171–179
653 Liu Y, Qi X, Zhang J, et al. Automatic bug triaging via deep reinforcement learning. Appl Sci, 2022, 12: 3565
654 Han Z, Li X, Xing Z, et al. Learning to predict severity of software vulnerability using only vulnerability description. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017. 125–136
655 Gomes L A F, Torres R S, Côrtes M L. Bug report severity level prediction in open source software: a survey and research opportunities. Inf Software Tech, 2019, 115: 58–78
656 Noyori Y, Washizaki H, Fukazawa Y, et al. Deep learning and gradient-based extraction of bug report features related to bug fixing time. Front Comput Sci, 2023, 5: 1032440
657 Liu H, Yu Y, Li S, et al. How to cherry pick the bug report for better summarization? Empir Software Eng, 2021, 26: 119
658 Liu H, Yu Y, Li S, et al. BugSum: deep context understanding for bug report summarization. In: Proceedings of the 28th International Conference on Program Comprehension, 2020. 94–105
659 Chen S, Xie X, Yin B, et al. Stay professional and efficient: automatically generate titles for your bug reports. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 385–397
660 Lin H, Chen X, Chen X, et al. TitleGen-FL: quality prediction-based filter for automated issue title generation. J Syst Software, 2023, 195: 111513
661 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
662 Xiao Y, Keung J, Mi Q, et al. Improving bug localization with an enhanced convolutional neural network. In: Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC), 2017. 338–347
663 Wang B, Xu L, Yan M, et al. Multi-dimension convolutional neural network for bug localization. IEEE Trans Serv Comput, 2020, 15: 1649–1663
664 Lam A N, Nguyen A T, Nguyen H A, et al. Bug localization with combination of deep learning and information retrieval. In: Proceedings of the IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017. 218–229
665 Cheng S, Yan X, Khan A A. A similarity integration method based information retrieval and word embedding in bug localization. In: Proceedings of the IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020. 180–187
666 Lam A N, Nguyen A T, Nguyen H A, et al. Combining deep learning with information retrieval to localize buggy files for bug reports (N). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 476–481
667 Loyola P, Gajananan K, Satoh F. Bug localization by learning to rank and represent bug inducing changes. In: Proceedings of the 27th ACM International Conference on Information and Knowledge Management, 2018. 657–665
668 Zhu Z, Li Y, Tong H H, et al. CooBa: cross-project bug localization via adversarial transfer learning. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence, 2020
669 Han J, Huang C, Sun S, et al. bjXnet: an improved bug localization model based on code property graph and attention mechanism. Autom Softw Eng, 2023, 30: 12
670 Liang H, Hang D, Li X. Modeling function-level interactions for file-level bug localization. Empir Software Eng, 2022, 27: 186
671 Choetkiertikul M, Dam H K, Tran T, et al. Automatically recommending components for issue reports using deep learning. Empir Software Eng, 2021, 26: 1–39
672 Huo X, Thung F, Li M, et al. Deep transfer bug localization. IEEE Trans Software Eng, 2019, 47: 1368–1380
673 Haering M, Stanik C, Maalej W. Automatically matching bug reports with related app reviews. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. 970–981
674 Ruan H, Chen B, Peng X, et al. DeepLink: recovering issue-commit links based on deep learning. J Syst Software, 2019, 158: 110406
675 Xie R, Chen L, Ye W, et al. DeepLink: a code knowledge graph based deep learning approach for issue-commit link recovery. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER),

2019. 434–444
676 Xi S, Yao Y, Xiao X, et al. An effective approach for routing the bug reports to the right fixers. In: Proceedings of the 10th Asia-Pacific Symposium on Internetware, 2018. 1–10
677 Fu W, Menzies T. Easy over hard: a case study on deep learning. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, New York, 2017. 49–60
678 Biswas E, Vijay-Shanker K, Pollock L. Exploring word embedding techniques to improve sentiment analysis of software engineering texts. In: Proceedings of IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019. 68–78
679 Nizamani Z A, Liu H, Chen D M, et al. Automatic approval prediction for software enhancement requests. Autom Softw Eng, 2018, 25: 347–381
680 Li X, Jiang H, Kamei Y, et al. Bridging semantic gaps between natural languages and APIs with word embedding. IEEE Trans Software Eng, 2018, 46: 1081–1097
681 Rhu M, Gimelshein N, Clemons J, et al. VDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016
682 Wang L, Ye J, Zhao Y, et al. Superneurons: dynamic GPU memory management for training deep neural networks. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, 2018. 41–53
683 Moran K, Bernal-Cardenas C, Curcio M, et al. Machine learning-based prototyping of graphical user interfaces for mobile apps. IEEE Trans Software Eng, 2018, 46: 196–221
684 Brooks F P. The Mythical Man-Month: Essays on Software Engineering. Reading: Addison-Wesley, 1975
685 Mockus A, Herbsleb J D. Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of the 24th International Conference on Software Engineering, New York, 2002. 503–512
686 Anvik J, Hiew L, Murphy G C. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, New York, 2006. 361–370
687 Ma D, Schuler D, Zimmermann T, et al. Expert recommendation with usage expertise. In: Proceedings of the IEEE International Conference on Software Maintenance, 2009. 535–538
688 Zhou M, Mockus A. Developer fluency: achieving true mastery in software projects. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, 2010. 137–146
689 Fritz T, Murphy G C, Murphy-Hill E, et al. Degree-of-knowledge: modeling a developer's knowledge of code. ACM Trans Softw Eng Methodol, 2014, 23: 1–42
690 Joblin M, Mauerer W, Apel S, et al. From developer networks to verified communities: a fine-grained approach. In: Proceedings of the 37th International Conference on Software Engineering, 2015. 563–573
691 Meng X, Miller B P, Williams W R, et al. Mining software repositories for accurate authorship. In: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), 2013. 250–259
692 Baltes S, Diehl S. Towards a theory of software development expertise. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018
693 Ren J, Yin H, Hu Q, et al. Towards quantifying the development value of code contributions. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. 775–779
694 Venkataramani R, Gupta A, Asadullah A, et al. Discovery of technical expertise from open source code repositories. In: Proceedings of the 22nd International Conference on World Wide Web, 2013. 97–98
695 Saxena R, Pedanekar N. I know what you coded last summer: mining candidate expertise from GitHub repositories. In: Proceedings of Companion of the ACM Conference on Computer Supported Cooperative Work and Social Computing, 2017. 299–302
696 Liu S, Wang S, Zhu F, et al. HYDRA: large-scale social identity linkage via heterogeneous behavior modeling. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014. 51–62
697 Kouters E, Vasilescu B, Serebrenik A, et al. Who's who in Gnome: using LSA to merge software repository identities. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012. 592–595
698 Mo W, Shen B, Chen Y, et al. TbIL: a tagging-based approach to identity linkage across software communities. In: Proceedings of Software Engineering Conference (APSEC), 2015. 56–63
699 Lee R K, Lo D. GitHub and stack overflow: analyzing developer interests across multiple social collaborative platforms. In: Proceedings of the 9th International Conference on Social Informatics, 2017. 245–256
700 Huang W, Mo W, Shen B, et al. CPDScorer: modeling and evaluating developer programming ability across software communities. In: Proceedings of SEKE, 2016. 87–92
701 Yan J, Sun H, Wang X, et al. Profiling developer expertise across software communities with heterogeneous information network analysis. In: Proceedings of the 10th Asia-Pacific Symposium on Internetware, Beijing, 2018. 1–9
702 Montandon J E, Valente M T, Silva L L. Mining the technical roles of GitHub users. Inf Software Tech, 2021, 131: 106485
703 Song X, Yan J, Huang Y, et al. A collaboration-aware approach to profiling developer expertise with cross-community data. In: Proceedings of IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), 2022. 344–355
704 Dey T, Karnauch A, Mockus A. Representation of developer expertise in open source software. In: Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2020. 995–1007
705 Ma Y, Bogart C, Amreen S, et al. World of Code: an infrastructure for mining the universe of open source VCS data. In: Proceedings of IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019. 143–154
706 Dakhel A M, Desmarais M C, Khomh F. Dev2vec: representing domain expertise of developers in an embedding space. Inf Software Tech, 2022, 159: 107218
707 Javeed F, Siddique A, Munir A, et al. Discovering software developer's coding expertise through deep learning. IET softw, 2020, 14: 213–220
708 Wang Z, Sun H, Fu Y, et al. Recommending crowdsourced software developers in consideration of skill improvement. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017. 717–722
709 Zhang Z, Sun H, Zhang H. Developer recommendation for Topcoder through a meta-learning based policy model. Empir Software Eng, 2019, 25: 859–889
710 Yu X, He Y, Fu Y, et al. Cross-domain developer recommendation algorithm based on feature matching. In: Proceedings of CCF Conference on Computer Supported Cooperative Work and Social Computing, 2019. 443–457
711 Wang J J, Yang Y, Wang S, et al. Context-aware personalized crowdtesting task recommendation. IEEE Trans Software Eng, 2021, 48: 3131–3144
712 Wang J, Yang Y, Wang S, et al. Context- and fairness-aware in-process crowdworker recommendation. ACM Trans Softw Eng Methodol, 2022, 31: 1–31

713 Ying H, Chen L, Liang T, et al. EARec: leveraging expertise and authority for pull-request reviewer recommendation in GitHub. In: Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering, 2016. 29–35

714 Jiang J, Yang Y, He J, et al. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. Inf Software Tech, 2017, 84: 48–62

715 Zhang J, Maddila C S, Bairi R, et al. Using large-scale heterogeneous graph representation learning for code review recommendations at Microsoft. In: Proceedings of IEEE/ACM 45th International Conference on Software Engineering, 2022. 162–172

716 Rebai S, Amich A, Molaei S, et al. Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations. Autom Softw Eng, 2020, 27: 301–328

717 Zanjani M B, Kagdi H, Bird C. Automatically recommending peer reviewers in modern code review. IEEE Trans Software Eng, 2016, 42: 530–543

718 Hannebauer C, Patalas M, Stünkel S, et al. Automatically recommending code reviewers based on their expertise: an empirical comparison. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016. 99–110

719 Rong G, Zhang Y, Yang L, et al. Modeling review history for reviewer recommendation: a hypergraph approach. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022. 1381–1392

720 Kovalenko V, Tintarev N, Pasynkov E, et al. Does reviewer recommendation help developers? IEEE Trans Software Eng, 2020, 46: 710–731

721 Ahasanuzzaman M, Oliva G A, Hassan A E. Using knowledge units of programming languages to recommend reviewers for pull requests: an empirical study. Empir Software Eng, 2024, 29: 33

722 Gonçalves P W, Calikli G, Serebrenik A, et al. Competencies for code review. In: Proceedings of the ACM on Human-Computer Interaction, 2023. 1–33

723 Huang Y, Sun H. Best answerers prediction with topic based GAT in Q&A sites. In: Proceedings of the 12th Asia-Pacific Symposium on Internetware, 2020. 156–164

724 Jin Y, Bai Y, Zhu Y, et al. Code recommendation for open source software developers. In: Proceedings of the ACM Web Conference, 2023

725 Xiao W, He H, Xu W, et al. Recommending good first issues in GitHub OSS projects. In: Proceedings of IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022. 1830–1842

726 Santos F. Supporting the task-driven skill identification in open source project issue tracking systems. ACM SIGSOFT Softw Eng Notes, 2023, 48: 54–58

727 Costa C, Figueiredo J, Pimentel J F, et al. Recommending participants for collaborative merge sessions. IEEE Trans Software Eng, 2021, 47: 1198–1210

728 Constantino K, Figueiredo E. CoopFinder: finding collaborators based on co-changed files. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2022. 1–3

729 Constantino K, Belém F, Figueiredo E. Dual analysis for helping developers to find collaborators based on co-changed files: an empirical study. Softw Pract Exp, 2023, 53: 1438–1464

730 Surian D, Liu N, Lo D, et al. Recommending people in developers' collaboration network. In: Proceedings of the 18th Working Conference on Reverse Engineering, 2011. 379–388

731 Canfora G, Penta M D, Oliveto R, et al. Who is going to mentor newcomers in open source projects? In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012

732 Ye L, Sun H, Wang X, et al. Personalized teammate recommendation for crowdsourced software developers. In: Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2018. 808–813

733 Fry T, Dey T, Karnauch A, et al. A dataset and an approach for identity resolution of 38 million author IDs extracted from 2B Git commits. In: Proceedings of IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), 2020