Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit

YAO WAN, Huazhong University of Science and Technology, Wuhan, China
ZHANGQIAN BI, Huazhong University of Science and Technology, Wuhan, China
YANG HE, Simon Fraser University, Burnaby, Canada
JIANGUO ZHANG, Salesforce Inc, San Francisco, United States
HONGYU ZHANG, Chongqing University, Chongqing, China
YULEI SUI, University of New South Wales, Sydney, Australia
GUANDONG XU, University of Technology Sydney, Sydney, Australia
HAI JIN, Huazhong University of Science and Technology, Wuhan, China
PHILIP YU, University of Illinois at Chicago, Chicago, United States

Code intelligence leverages machine learning techniques to extract knowledge from extensive code corpora, with the aim of developing intelligent tools to improve the quality and productivity of computer programming. Currently, there is already a thriving research community focusing on code intelligence, with efforts ranging from software engineering, machine learning, data mining, natural language processing, and programming languages. In this paper, we conduct a comprehensive literature review on deep learning for code intelligence, from the aspects of code representation learning, deep learning techniques, and application tasks. We also benchmark several state-of-the-art neural models for code intelligence, and provide an open-source toolkit tailored for the rapid prototyping of deep-learning-based code intelligence models. In particular, we inspect the existing code intelligence models under the basis of code representation learning, and provide a comprehensive overview to enhance comprehension of the present state of code intelligence. Furthermore, we publicly release the source code and data resources to provide the community with a ready-to-use benchmark, which can facilitate the evaluation and comparison of existing and future code intelligence models

This work is supported by the National Natural Science Foundation of China under grant No. 62102157, and partially supported by NSF under grants 2106758 and 2346158. We would like to thank all the anonymous reviewers for their insightful comments.

Authors' Contact Information: Yao Wan, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; e-mail: wanyao@hust.edu.cn; Zhangqian Bi, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; e-mail: zqbi@hust.edu.cn; Yang He, Simon Fraser University, Burnaby, British Columbia, Canada; e-mail: yha244@sfu.ca; Jianguo Zhang, Salesforce Inc, San Francisco, California, United States; e-mail: jianguozhang@salesforce.com; Hongyu Zhang, Chongqing University, Chongqing, Sichuan, China; e-mail: hyzhang@cqu.edu.cn; Yulei Sui, University of New South Wales, Sydney, New South Wales, Australia; email: y.sui@unsw.edu.au; Guandong Xu, University of Technology Sydney, Sydney, New South Wales, Australia; e-mail: guandong.xu@uts.edu.au; Hai Jin, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: hjin@hust.edu.cn; Philip Yu, Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois, United States; e-mail: psyu@uic.edu.

ACM 0360-0300/2024/10-ART309
https://doi.org/10.1145/3664597

(https://xcodemind.github.io). At last, we also point out several challenging and promising directions for future research.

CCS Concepts: • **Software and its engineering → Documentation;**

Additional Key Words and Phrases: Code intelligence, code representation, deep learning, large language models, survey, benchmark, toolkit

ACM Reference Format:
Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. 2024. Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit. *ACM Comput. Surv.* 56, 12, Article 309 (October 2024), 41 pages. https://doi.org/10.1145/3664597

1 INTRODUCTION

Software is eating the world [15]. With the advancement of ***Artificial Intelligence (AI)***, it is time to expand that maxim: software ate the world, and AI is eating the software. As the software is primarily composed of code, we define the emerging concept of *code intelligence* as the application of AI techniques to extract knowledge from large-scale code repositories, with the aim of developing intelligent tools to improve the quality and productivity of computer programming [163]. This concept is fueled by the ever-expanding reservoir of source code, often referred to as "*Big Code*" [7], which is harvested from platforms such as GitHub [1] and StackOverflow [2]. In this paper, our research scope is confined to code intelligence, with a particular focus on the application of deep learning techniques.

Achieving code intelligence necessitates a collaborative synergy in research across the domains of software engineering, machine learning, ***Natural Language Processing (NLP)***, programming language, and security. From our investigation, precise and reliable code representation learning (or code embedding), which aims to efficiently and effectively encode the semantics of source code into distributed vector representations, is the foundation for code intelligence. Such embedding vectors are then used in various downstream tasks, such as code completion [121, 153, 203, 229], code search [83, 111, 240], code summarization [11, 108, 112, 243, 293], type inference [8, 104, 193, 260], and program synthesis [17, 181, 183, 204] and so on. In terms of code representation learning, significant progress has been made by utilizing deep learning and NLP techniques to encode code.

Analogous to word2vec [170] in NLP, Alon et al. [14] proposed code2vec, a distributed representation of code, based on a collection of paths extracted from the ***Abstract Syntax Tree (AST)*** of code. Recently, a multitude of neural networks tailored for specific tasks have been proposed and trained using supervised methods. As pre-trained language models (e.g., BERT [64] and GPT-3 [28]) have been widely applied to NLP, many pre-trained language models for code have been proposed [72, 89, 119] to better represent the semantics of code. More recently, the emergence of ***Large Language Models (LLMs)***, exemplified by ChatGPT, has illuminated the pathway for further advancement of pre-trained language models, with a notable trend of increasing model sizes. This trend has extended to the domain of code intelligence, resulting in the development of various LLMs tailored for code, including but not limited to CodeT5 [253], StarCoder [134], and Code Llama [204]. In this paper, we examine code intelligence through the lenses of code representation learning, deep learning methods, and their applications.

Related Surveys and Differences. Within our literature review, we identified several surveys related to ours. Notably, Allamanis et al. [7] conducted an exhaustive examination of machine

Table 1. Comparison of our Work with Previous Related Surveys

Paper	Artifact	Technique	Survey	Benchmark	Toolkit
Allamanis et al. [7]	Software	Machine Learning	✓	×	×
Watson et al. [256] Wang et al. [248] Yang et al. [276] Devanbu et al. [63]	Software	Deep Learning	✓	×	×
Lu et al. [163]	Software	Deep Learning	×	✓	×
Ours	Code	Deep Learning	✓	✓	✓

learning approaches for modeling the naturalness of programming language. They primarily emphasize machine learning algorithms, with a specific focus on probabilistic models, as opposed to those based on deep learning. Recently, Watson et al. [256], Wang et al. [248] and Yang et al. [276] conducted a thorough review of the literature on applications of deep learning in software engineering research. They investigated mostly software engineering and AI conferences and journals, focusing on various software engineering tasks (not limited to the code) that are based on deep learning. [63] is a report that summarizes the current status of research on the subject of the intersection between deep learning and software engineering, as well as suggests several future directions. In [163], the authors introduced CodeXGLUE, a benchmark dataset for code representation and generation. They also presented benchmark results, notably leveraging pre-trained language models like CodeBERT.

Table 1 summarizes the differences between our paper when compared with several related surveys in code intelligence. In contrast to [7] that focuses on traditional machine learning approaches, this paper places greater emphasis on leveraging deep learning techniques for code intelligence. In contrast to [256], [248], [276], and [63] that cover various tasks in broad software engineering, our study narrows its focus to tasks associated with source code, examining them specifically from the perspective of deep learning. In addition, we survey papers from various fields including software engineering, programming languages, machine learning, NLP, and security. Furthermore, existing surveys do not provide comprehensive benchmark evaluation results, nor do they develop an open-source toolkit to facilitate further research. This paper addresses this gap by presenting an open-source toolkit, referred to as **NaturalCC** (standards for **Natural Code Comprehension)** [239]. The toolkit is designed to streamline the prototyping of code intelligence models and to serve as a benchmarking platform for evaluating various state-of-the-art models. In complement to CodeXGLUE [163], our focus lies in the building of infrastructures that support diverse model implementations and provide users with the ability to conduct rapid prototyping. Compared to CodeXGLUE, our toolkit contains a more extensive array of tools designed for the entire pipeline involved in constructing code intelligence models, offering heightened flexibility.

Our Contributions. This paper is targeted at researchers and practitioners intrigued by the convergence of code intelligence and deep learning, with a specific emphasis on intelligent software engineering, NLP, and programming languages. In this paper, we begin by providing a thorough review of existing research on deep learning for code intelligence. Subsequently, we advance our contribution by developing an open-source toolkit, referred to as NaturalCC, that incorporates state-of-the-art models across various downstream tasks. Employing NaturalCC, we conduct a comprehensive performance benchmark of each model across five downstream tasks, including code summarization, code search, code completion, and type inference. The major contributions of this paper are summarized as follows.

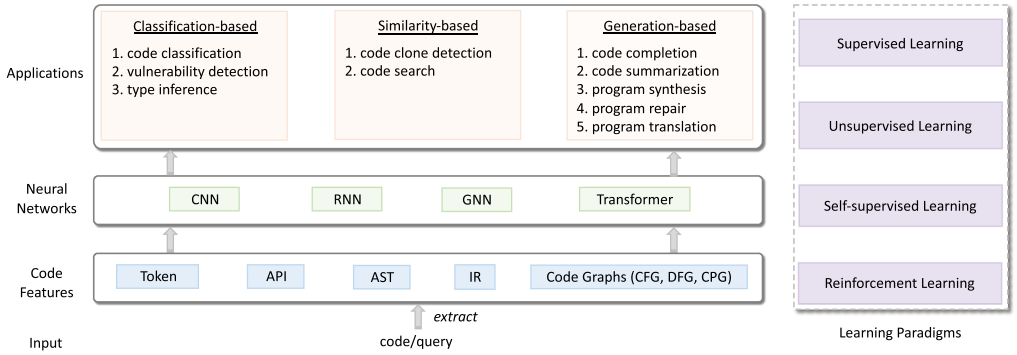


Fig. 1. Code intelligence tasks based on code representation learning.

— We conduct a comprehensive review of deep learning for code intelligence. Specifically, we have collected 276 papers from various top-tier venues and arXiv, covering multiple domains including software engineering, artificial intelligence, NLP, programming languages, and security.
— We benchmark the performance of 18 leading models across five different tasks (i.e., code summarization, code search, code completion, program synthesis, and type inference). All the resources, datasets and source code are publicly available.[1]
— We introduce NaturalCC, an open-source toolkit featuring integrated state-of-the-art baselines across various tasks, aimed at streamlining research in code intelligence. Researchers in software engineering, NLP, and related domains can leverage this toolkit for rapid prototyping.

2 SURVEY METHODOLOGY

2.1 A Unified View from Code Representation Learning

We propose to summarize existing deep-learning-based approaches to code intelligence from the lens of code representation learning in this paper. As shown in Figure 1, for code representation learning, researchers first extract features that potentially describe the semantics of code, and then design various neural networks to encode them into distributed vectors. Code representation learning can be viewed as the foundation for different downstream applications. Based on the characteristics of each application, the downstream applications can be divided into three groups: (1) *Classification-based.* In these tasks (e.g., code classification, vulnerability detection, and type inference), a classifier layer (e.g., softmax) is used to map the code embeddings to labels/classes. (2) *Similarity-based.* In these tasks (e.g., code search and code clone detection), Siamese neural network structure [51] is often adopted, where dual encoders are used to encode the source code and natural-language query into embedding vectors. Based on the two embeddings of code and query, a constraint (such as a triplet loss function) is always used to regularize the similarity between them. Note that, in several approaches to code search and code clone detection, the two embeddings of code and query are also concatenated, and the task is reformulated as a classification task to determine whether the code and query are related [72]. (3) *Generation-based.* In these tasks (e.g., code completion, code summarization, program translation, program synthesis, and program repair), the objective is to generate source code, natural-language descriptions, or

[1] https://xcodemind.github.io

programs in another programming language from a given code snippet. These tasks usually follow the encoder-decoder paradigm, where an encoder network is used to represent the semantics of code, and a decoder network (e.g., RNN) is designed to generate sequences, e.g., natural-language descriptions or source code. Additionally, we categorize the learning paradigms into four groups: supervised learning, unsupervised learning, self-supervised learning, and reinforcement learning.

2.2 Paper Selection

Deep learning for code intelligence has been studied in many related research communities. In this paper, we review high-quality papers selected from top-tier conferences and journals, ranging from software engineering, programming languages, NLP, and AI, to security. Overall, we have identified 32 publication venues (see the Supplementary Materials). We first manually check the publication list of the venues and obtain an initial collection of papers. Particularly, we systematically query the aforementioned venue names within the DBLP database[2] and examine the associated proceedings. Subsequently, two authors, both possessing over five years of expertise in deep learning for code intelligence, collaboratively undertake the task of manually refining the results. This involves meticulous scrutiny of titles and a brief review of abstracts to identify and filter out papers that are potentially relevant to code intelligence. For those large conferences (e.g., AAAI and IJCAI) that accept thousands of papers per year, we first filter out those papers whose titles contain the keywords of "code" or "program", and then manually check them.

Based on this initial collection of papers, we start to augment it through keyword searching. We systematically search DBLP and Google Scholar using the following keywords: "code representation", "program comprehension", "code embedding", "code classification", "vulnerability detection", "bug finding", "code completion", "type inference", "code search/retrieval", "code clone detection", "code summarization", "program translation", "program synthesis", and "program repair", with a combination of "deep", "learning", "neural", and "network".

It is worth noting that, in addition to accepted papers from the aforementioned venues, we also consider some recent publications from the pre-print archive, as they reflect the most current research outputs. We choose publications from arXiv based on two criteria: paper quality, and author reputation. The quality of a pre-printed paper can be assessed based on the number of citations it has garnered in recent months. The reputations of authors can be indicated by their Google Scholar citations. If a paper satisfies either of these selection criteria, we include it for consideration. Having obtained this collection of papers, we then filter out the irrelevant papers by manual checking. Finally, we obtained a collection of 276 papers. To ensure transparency and accessibility, a comprehensive table of the surveyed papers and the source of papers is maintained online.[3] Note that this survey is systematically conducted up to January 2023, as per the submission date. It is evident that the field of code intelligence has witnessed a significant shift towards LLMs since January 2023. Although we have included several papers on LLMs in code intelligence, research in this area is active and rapidly evolving. A comprehensive survey on LLMs for code intelligence is available in our online resource.

3 LITERATURE REVIEW

3.1 Taxonomy

Figure 2 illustrates the taxonomy of current studies on deep learning for code intelligence surveyed in this paper, categorized into three distinct aspects: code features, deep learning techniques, and applications. (1) *Code Features.* Code representation forms the cornerstone of

[2] https://dblp.uni-trier.de
[3] https://xcodemind.github.io

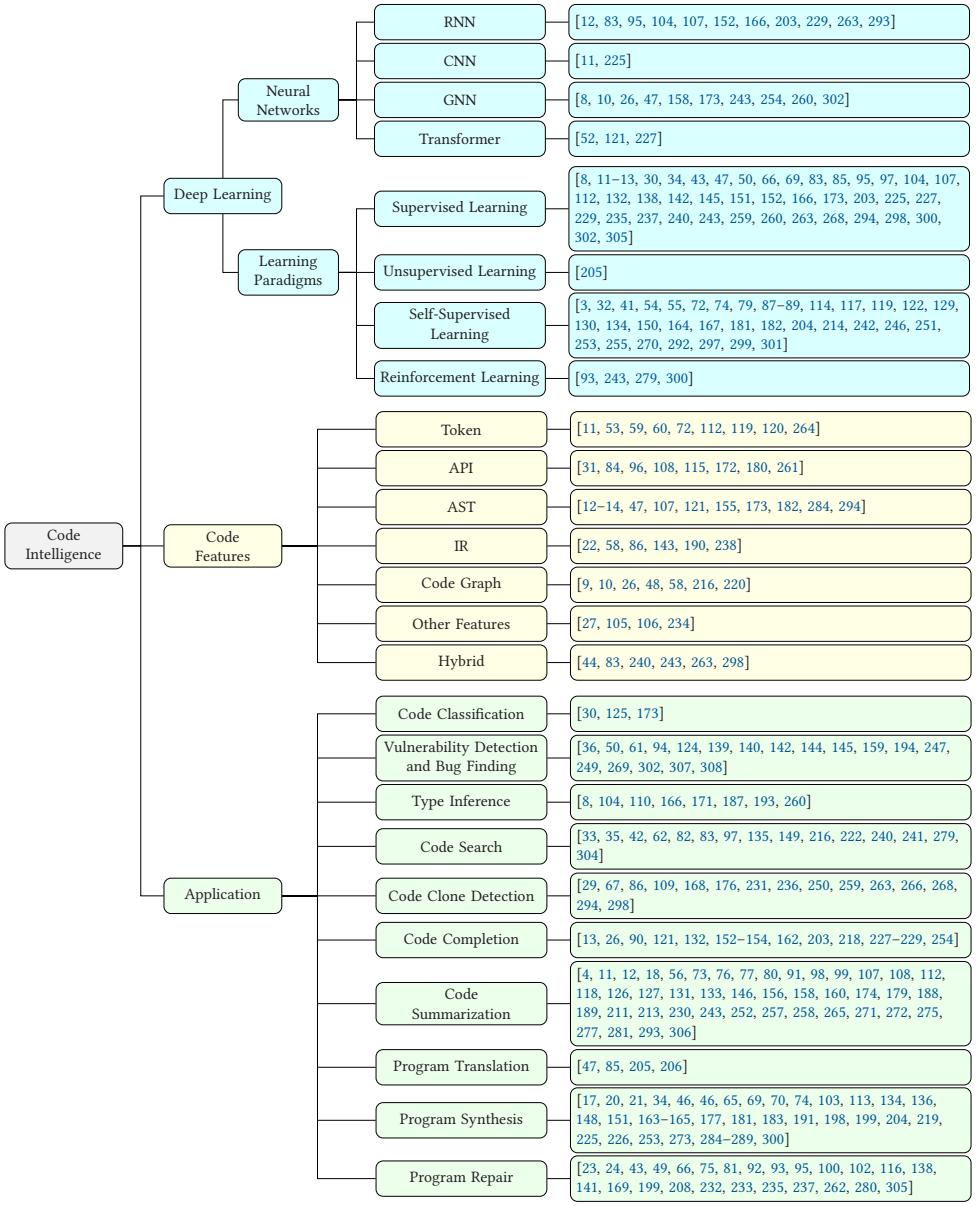


Fig. 2. The taxonomy of deep learning for code intelligence.

deep-learning-based code intelligence. We classify current approaches based on the features of input code they utilize, including code tokens, ***Intermediate Representations (IRs)***, ***Application Programming Interfaces (APIs)***, ***Abstract Syntax Trees (ASTs)***, and code graphs (e.g., control-flow and data-flow graphs). (2) Within the domain of deep learning, we begin by exploring a range of neural network architectures, i.e., RNNs, CNNs, Transformers, and GNNs. Subsequently, we examine various learning paradigms utilized for modeling source code, i.e., supervised learning, unsupervised learning, self-supervised learning, and reinforcement learning.

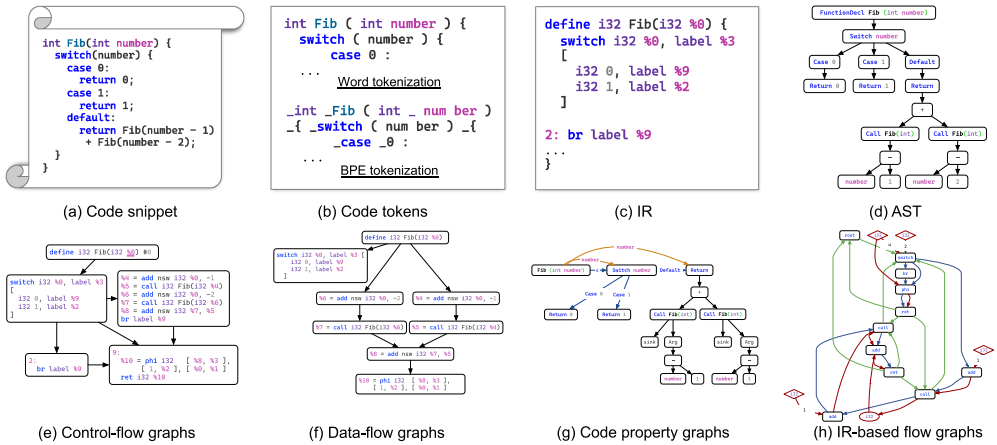


Fig. 3. A detailed C code snippet with its corresponding tokens, IR, AST, and IR-based flow graphs.

(3) We investigate multiple downstream applications that are based on code representation and deep learning techniques, including code classification, vulnerability detection and bug finding, type inference, code search, code clone detection, code completion, code summarization, program translation, program synthesis, and program repair.

3.2 Code Features

To represent source code effectively, the initial step is determining what aspects to capture. Various studies have proposed extracting code features from multiple angles, such as code tokens, IRs, ASTs, and different forms of code graphs. Figure 3 provides a comprehensive illustration of a C code snippet alongside its associated code tokens, IR, AST, control-flow graph, data-flow graph, code property graph, and IR-based flow graphs.

3.2.1 Code Tokens. Code tokens, shaping the textual appearance of source code, are composed of *function name*, *keywords*, and various *variable identifiers*. These tokens are simple yet effective in representing the semantics of programs. Just as sentences in natural language can be tokenized, source code can also undergo tokenization at various levels of granularity, such as character-level, word-level, and sub-word level. Cummins et al. [59] introduced a character-level LSTM network for program synthesis, which circumvents the issue of out-of-vocabulary due to the finite set of characters used in programming. However, tokenizing at the character level can obscure the original word meanings and inflate the code sequence length, potentially complicating comprehension of the program's overall semantics.

More coarsely, numerous word-level approaches have been developed to tokenize source code by utilizing separators. For instance, White et al. [264] and Iyer et al. [112] proposed tokenizing programs based on whitespace, employing RNNs to summarize and complete code. Additionally, Allamanis et al. [11] devised a CNN with an attention mechanism to capture the hierarchical code structure effectively by tokenizing via camel case subtokens, aiming to predict function names.

Out-of-Vocabulary (OOV) Issue. As the variables and function names are always defined by developers without constraints, the vocabulary size escalates dramatically with increasing training data, leading to a pronounced *out-of-vocabulary issue*, more severe than in NLP. To address this, Cvitkovic et al. [60] introduced a graph-structured cache, adding nodes for new words encountered

and connecting them based on code occurrences. Additionally, Chirkova and Troshin [53] proposed a simple yet effective solution for mitigating the OOV issue through identifier anonymization, yielding promising performance enhancements.

Another effective approach involves tokenizing the source code at a sub-word level, such as through techniques like ***Byte Pair Encoding (BPE)***, aimed at constructing a set of sub-words for representing the entire code corpus. In Figure 3(b), source tokens obtained via word tokenization and BPE tokenization are illustrated. For the input variable `number`, word tokenization maintains the original word as a rare token, while BPE tokenization splits it into two common sub-words: `num` and `ber`. Pre-trained language models for source code, such as CuBERT [119] and CodeBERT [72], commonly employ BPE to reduce vocabulary size. Karampatsis et al. [120] conducted an empirical study on word segmentation granularity, demonstrating that BPE tokenization can significantly reduce vocabulary size.

3.2.2 Application Programming Interfaces (API). Several methods have been proposed for analyzing API sequences in programs. One line of research focuses on mining API usage patterns from large code corpora to illustrate API usage. For instance, Moreno et al. [172] introduced Muse, a method for mining and ranking code examples to demonstrate API usage. Another line of work is API recommendation, which aims to suggest or generate API sequences for users. Jiang et al. [115] proposed a method for discovering relevant tutorial fragments for APIs by calculating correlation scores based on PageRank and topic relevance. Gu et al. [84] introduced DeepAPI, a language model employing sequence-to-sequence learning for generating API sequences from natural language descriptions. In contrast to DeepAPI, Nguyen et al. [180] proposed API2Vec to capture contextual information of API elements within sequences, along with a tool named API2API for migrating APIs across different programming languages, such as Java to C#. Ling et al. [147] introduced a method that integrates API call interactions and project structure into a single graph, leveraging it to design a graph-based collaborative filtering system for API usage recommendations. Bui et al. [31] proposed a cross-language API mapping approach from Java to C# using transfer learning across multiple domains. Hu et al. [108] suggested incorporating API information to enhance code summarization. For improving semantics representation in natural-language queries and API sequences, Wei et al. [261] proposed a contrastive learning approach for API recommendation, while Hadi et al. [96] investigated pre-trained models' effectiveness in generating API sequences from natural-language queries.

3.2.3 Abstract Syntax Tree (AST). AST is a tree-structured, intermediate representation of code, delineating the syntactic structure of a program. As illustrated in Figure 3(d), leaf nodes (e.g., `number`, `Fib`) typically correspond to variables and method names, while non-leaf nodes (e.g., `FuncName`, `SwitchStmt`) denote code structure elements like function definitions and branching. This representation facilitates the capture of both lexical information (e.g., variable `number`) and syntactic structure of code. Various open-source tools, including `ANTLR`[4] parser, `tree-sitter`[5] parser, and LLVM Clang,[6] can be utilized to extract ASTs. To represent the ASTs, Mou et al. [173] proposed a tree structure-based CNN, and verified it in a code classification task. Liu et al. [155] introduced an enhanced LSTM to address long-distance dependencies within AST nodes, applied in code completion, code classification, and code summarization. To better process an AST, Zhang et al. [294] partitioned ASTs into sentence-based subtrees and represented them using a two-way loop network. Recently, Kim et al. [121] proposed leveraging relative position embeddings in

[4] https://www.antlr.org
[5] https://tree-sitter.github.io/tree-sitter
[6] https://clang.llvm.org

Transformers for code completion tasks. Additionally, Niu et al. [182] integrated AST information into a pre-trained source code model.

Another line of work [12, 14, 107] involves indirectly representing ASTs through traversal or path sampling. Hu et al. [107] proposed traversing ASTs to transform them into linear sequences of nodes, employing RNNs to represent the sequences for code summarization tasks. Alon et al. [14] conducted path sampling on ASTs and utilized word2vec to represent program semantics. Similarly, Alon et al. [12] applied a comparable approach to code summarization. Additionally, Alon et al. [13] introduced a structured code language model for code completion by sampling paths from incomplete ASTs. In program synthesis, an AST is utilized to guide the synthesis of programs. Yin and Neubig [284] introduced an encoder-decoder framework for code generation. Here, the encoder processes natural language, the decoder generates an AST, and subsequently, the AST is transformed into source code. Additionally, Chen et al. [47] proposed a Tree2Tree model for program translation. This model employs a TreeLSTM to represent the source program and another TreeLSTM to produce the target program in a different programming language.

3.2.4 Intermediate Representation (IR). The IR is a well-formed structure that is independent of programming languages and machine architectures. It is used by compilers to accurately represent the source code during the translation process from the source code to low-level machine code. The IR can express the operations of the target machine. Leveraging IRs in enhancing code embeddings has been proposed [143], offering the advantage of a constrained vocabulary to mitigate the ***Out-of-Vocabulary (OOV)*** issue. In this study, we utilize LLVM-IR, integral to the LLVM infrastructure [123], as depicted in Figure 3(c). To represent IRs, Ben-Nun et al. [22] introduced inst2vec, which initially compiles a program using LLVM Clang to obtain the LLVM-IR, followed by the adoption of skip-gram to represent the instructions. VenkataKeerthy et al. [238] proposed IR2Vec, conceptualizing the intermediate code representation as triples within a knowledge graph, and subsequently exploring multiple knowledge graph representation methods. Cummins et al. [58] presented ProGraML, a novel graph-based code representation built upon IR. This code graph offers fresh avenues for representing source code semantics at a granular level using machine learning techniques such as ***Graph Neural Networks (GNNs)***, facilitating complex downstream tasks like program optimization and analysis. Peng et al. [190] suggested representing augmented IR of source code via pre-training and contrastive learning methods, guided by compiler optimization. Notably, Gui et al. [86] investigated a new problem of matching binary code and source code across different programming languages by transforming both into LLVM-IRs.

3.2.5 Code Graphs. Numerous approaches have been proposed to convert programs into graphs for enhanced representation of their rich structural information, such as ***Control-Flow Graph (CFG)***, ***Data-Flow Graph (DFG)***, and ***Code Property Graph (CPG)*** [58, 274]. The CFG, as depicted in Figure 3(e), illustrates program computation and control flow, where nodes represent basic blocks and edges denote control flow transitions. Meanwhile, the DFG, shown in Figure 3(f), visualizes data relationships among functions, with nodes featuring input and output data ports, and edges linking these ports. To represent multiple structural information of code using a joint data structure, Yamaguchi et al. [274] proposed an innovative CPG to merge the structural information of code, including AST, CFG and ***Program Dependence Graph (PDG)***, into a single graph, as shown in Figure 3(g). In practice, we can build CFGs and DFGs using LLVM Clang, and build CPGs using Plume.[7] Recently, Cummins et al. [58] built a unified graph, termed ProGraML, which includes the CFG, DFG and call-graph, as shown in Figure 3(h).

[7] https://plume-oss.github.io/plume-docs

To represent these code graphs, Allamanis et al. [10] incorporated the data flow into the ASTs and formed a code graph. Subsequently, a ***Gated Graph Neural Network (GGNN)*** [137] was developed to learn the data dependencies within this structure. Allamanis and Brockschmidt [9] incorporated contextual information of variables into data flow for automated pasting in programming. Brockschmidt et al. [26] extended incomplete code into a graph and proposed a GNN for code completion. Sui et al. [220] made the code representation more accurate by using the value-flow graph of a program. Furthermore, Shi et al. [216] converted code graphs (e.g., CFG and DFG) into sequences through traversal for code search, while Chen et al. [48] introduced a general method to transform a code graph into a sequence of tokens and pointers.

3.2.6 Other Features of Code. In addition to the aforementioned features of code that have already been widely explored, several kinds of features are also utilized in specific scenarios. For instance, Henkel et al. [105] introduced a novel feature for code representation learning based on abstractions of traces collected from the symbolic execution of a program, while Hoang et al. [106] proposed employing deep learning to learn distributed representations of code changes/edits for generating software patches. Regarding code changes, various related works have also been proposed for representation or prediction. For example, Tufano et al. [234] suggested automating code editing through sequence-to-sequence-based neural machine translation, while Brody et al. [27] proposed representing code edits first, followed by iteratively generating tree edits over the AST.

3.2.7 Hybrid Representation. To harness multiple code features, various approaches to hybrid source code representation have been developed. For example, Gu et al. [83] employed three separate RNNs to encode function names, code tokens, and API sequences, respectively, as evaluated in the code search task. Similarly, White et al. [263] utilized two distinct RNNs to represent code tokens and AST node sequences for code cloning detection. Additionally, Zhao and Huang [298] proposed incorporating code flow graphs into a semantic matrix to represent source code, along with a neural network model to assess the functional similarity between code snippet representations. Similarly, Wan et al. [240, 243] developed a hybrid network incorporating an LSTM for code tokens, a GGNN for code CFG, and a TreeLSTM for code AST, for tasks such as code summarization and code search. Additionally, Chakraborty and Ray [44] proposed utilizing three modalities of information –edit location, edit code context, and commit messages–to represent the context of programming and generate code patches automatically.

3.3 Deep Learning Techniques

We investigate the types of neural networks and classify the learning paradigms into four groups: supervised learning, unsupervised learning, self-supervised learning, and reinforcement learning.

3.3.1 Neural Networks. It is natural to model source code as sequential text and apply NLP techniques directly to represent it. Specifically, RNNs [12, 83, 95, 104, 107, 152, 166, 203, 229, 263, 293] and CNNs [11, 225] are commonly employed to capture the sequential structure of source code. To capture syntax structure, particularly the AST of source code, several tree-structured neural networks [47, 173, 243] have been designed. Moreover, to represent semantic structures (e.g., CFG and DFG) of source code, GNNs [8, 10, 26, 158, 254, 260, 302] have been introduced. Recently, Transformer architecture has also been utilized for source code representation [121, 227]. Chirkova and Troshin [52] conducted a comprehensive empirical study on how well Transformers can leverage syntactic information in source code for various tasks. The preliminaries about the mentioned neural networks can be found in Supplementary Materials.

3.3.2 Supervised Learning. Supervised learning aims to learn a function that maps an input to an output based on a set of input-output pairs as training data, constituting a widely used

paradigm in deep learning. Our investigation reveals that current deep learning approaches for code intelligence predominantly rely on supervised learning. Specifically, for tasks such as code classification [30, 173], vulnerability detection and bug finding [50, 142, 145, 302], code completion [13, 132, 152, 203, 227, 229], type inference [8, 104, 166, 260], code search [83, 97, 240], code clone detection [259, 263, 268, 294, 298], code summarization [11, 12, 107, 112, 243], program translation [47, 85], program synthesis [34, 69, 151, 225, 300], and program repair [43, 66, 95, 138, 235, 237, 305], paired input-output data is initially collected. Supervised learning for each task is guided by a specific loss function. However, a limitation of this approach is its dependency on a substantial amount of well-labeled input-output pairs, which can be costly to collect in certain scenarios.

3.3.3 Unsupervised Learning. As opposed to supervised learning, unsupervised learning aims to discern patterns from a dataset without labels. One notable work is TransCoder [205], which trains a fully unsupervised neural source-to-source translator via unsupervised machine translation. This learning paradigm poses challenges for code intelligence, warranting further research efforts.

3.3.4 Self-Supervised Learning. Self-supervised learning can be thought of as a blend of supervised learning and unsupervised learning. Unlike supervised learning, which relies on labeled data for training, self-supervised learning derives supervisory signals directly from the data itself, often leveraging its inherent structure. A common approach in self-supervised learning involves predicting unobserved or masked segments of input based on the segments that can be observed. As a representative technique of self-supervised learning, language model pre-training has garnered significant attention in the domain of source code [72, 89, 119]. Kanade et al. [119] introduced the training of CuBERT on a Python code corpus, validating its efficacy across various downstream tasks, including variable misuse, operator classification, and function-document matching. CodeBERT [72] is another pre-trained model designed for handling both source code and natural-language descriptions. It employs ***Masked Language Modeling (MLM)*** and has demonstrated promising performance in tasks such as code search and code completion. Building upon CodeBERT, GraphCodeBERT [89], SPT-Code [182], and TreeBERT [117] have been proposed to incorporate structural information from source code. Lachaux et al. [122] introduced a pre-training objective based on deobfuscation as an alternative criterion. Inspired by BART [128], a pre-trained model tailored for natural language understanding and generation, Ahmad et al. [3] developed PLBART for tasks related to code generation and understanding. Zhang et al. [292] trained CoditT5 on vast datasets comprising source code and natural-language comments, enabling tasks such as comment updating, bug fixing, and automated code review. Wang et al. [251] and Guo et al. [88] proposed contrastive learning methods to enhance the representation of source code semantics by integrating source code and natural language modalities. Mastropaolo et al. [167] and Wang et al. [255] explored building pre-trained models based on the **T5 (Text-To-Text Transfer Transformer)** architecture, which has attained state-of-the-art results in NLP tasks. Bui et al. [32] introduced InferCode, a self-supervised learning approach for predicting subtrees from AST contexts. Jain et al. [114] proposed a contrastive learning approach for task-agnostic code representation, leveraging program transformations in compilers. ERNIE-Code [41] is a unified pre-trained model spanning 116 natural languages and 6 programming languages, aiming to bridge the gap between multilingual natural and programming languages. Additionally, Wang et al. [246] explored fine-tuning pre-trained code models through curriculum learning to better adapt to downstream tasks, addressing dataset distribution discrepancies.

Instead of improving the capability of code embedding, Wan et al. [242] explored the explainability of pre-trained models for code intelligence, i.e., what kind of information these models capture, through structural analysis. Zhang et al. [297] and Shi et al. [214] proposed

compressing pre-trained code models to improve efficiency in practical applications. Zhou et al. [301] conducted an empirical study to evaluate the generalizability of CodeBERT across different datasets and downstream tasks.

Large Language Models (LLMs) of Code. The aforementioned pre-trained code models have shown promising capabilities in understanding the semantics of source code. More recently, fueled by the remarkable performance of LLMs in NLP, such as the success of ChatGPT, a diverse range of LLMs has been tailored for code intelligence tasks, particularly code generation. Nijkamp et al. [181] introduced a novel code generation task that enables users to progressively express their intentions through multi-turn interactions, and further trained a family of LLMs with up to 16.1 billion parameters, called CodeGen, for this task. PaLM [54] is a general-purpose LLM developed by Google, which is pre-trained on a dataset comprising both text and code corpora, boasting a vast parameter size of up to 540 billion. Derived from PaLM, PaLM-Coder is a model specifically fine-tuned for code-related tasks, such as code generation and program translation. InCoder [74] is a LLM developed by Meta, which employs a causal masking objective for the purpose of infilling code blocks based on arbitrary left and right contexts. PanGu-Coder [55], from Huawei, is an LLM designed for code generation. It follows a two-stage training approach: first, pre-training with ***Causal Language Modeling (CLM)*** on raw code corpora, then combining CLM and MLM objectives to focus on generating code from text. CodeGeeX [299] is a multilingual model with 13 billion parameters for code generation, pre-trained on 850 billion tokens of 23 programming languages. StarCoder [134] is an LLM for code, up to 15.5 billion parameters, which is trained on an extensive dataset consisting of 1 trillion tokens sourced from a vast collection of permissively licensed GitHub repositories. Code Llama [204] is a family of LLMs for code, released by Meta, built upon the foundation of Llama 2. These models are distinguished by their advanced infilling capabilities, extensive long-context fine-tuning, and precise instruction fine-tuning. CodeT5+ [253], released by Salesforce, represents a novel family of LLMs explicitly tailored for a broad spectrum of tasks related to both code comprehension and code generation. This model introduces innovative pre-training objectives, including text-code contrastive learning, matching, and CLM tasks on text-code data. phi-1 [87] is a comparatively smaller LLM for code, consisting of 1.3 billion parameters, achieved through data set refinement, while maintaining competitive performance in code generation.

Different from traditional pre-trained code models that are designed for specific tasks, the LLMs for code are distinguished by their strong capabilities in zero-shot learning. To unleash the zero-shot capabilities of LLMs, many techniques such as prompt tuning, in-context learning, chain-of-thought, and instruction tuning, have been developed. Recently, numerous studies have explored the potential of LLMs in tasks such as code generation [150], code summarization [79], and code repair [270], all achieved through the design of textual prompts. As a specific prompting, in-context learning seeks to bolster the capabilities of LLMs by furnishing them with contextual information or illustrative examples. Li et al. [130] explored in-context learning for better code generation based on LLMs. The chain-of-thought is designed to ensure the outputs of LLMs follow a logical chain. Li et al. [129] explored chain-of-thought for better code generation based on LLMs. The instruction tuning is initially designed to enhance the generalization capabilities of LLMs across different tasks. WizardCoder [164] is crafted to augment the capabilities of StarCoder by creating sophisticated code instruction data via the code-specific *Evol-Instruct* approach.

3.3.5 Reinforcement Learning. Reinforcement learning aims to learn an agent through interacting with the environment without input-output pairs. This kind of learning paradigm has been used in various domains such as code summarization [243], code search [279], program repair [93], and program synthesis [300].

3.4 Classification-based Applications

Classification-based applications, such as code classification, vulnerability detection, and type inference, seek to train a classifier with the objective of mapping the source code to specific labels or classes, such as vulnerability status or variable types.

3.4.1 Code Classification. Classifying source code into different classes (e.g., different functionalities and programming languages), is important for many tasks such as code categorization, programming language identification, code prediction, and vulnerability detection. Various studies have been conducted to classify code snippets into categories based on their functionalities. To represent programs in the form of ASTs, Mou et al. [173] developed a **Tree-Based *Convolutional Neural Network (TBCNN)***, which was then verified on code classification. In the wider realm of software categorization, LeClair et al. [125] devised a series of adaptations, incorporating techniques such as word embedding and neural architectures, to tailor NLP methods for text classification specifically to the domain of source code. Bui et al. [30] presented a bilateral neural network for the cross-language algorithm classification task, where each sub-network is used to encode the semantics of code in a specific language, and an additional classification module is designed to model the connection of those bilateral programs.

3.4.2 Vulnerability Detection and Bug Finding. Detecting vulnerabilities or bugs in programs is essential for assuring the quality of software, as well as saving much effort and time during software development. Although many tools have been developed for vulnerability detection, e.g., Clang Static Analyzer,[8] Coverity,[9] Fortify,[10] Flawfinder,[11] Infer,[12] and SVF [221], most of them are based on static analysis. Recently, an increasing number of works have utilized deep learning to uncover vulnerabilities. An early endeavor by Wang et al. [249] applied deep belief networks to predict software defects, leveraging semantic features learned from programs based on ASTs. Dam et al. [61] proposed an LSTM-based method to exploit both the syntactic and semantic aspects of source code, and apply the embeddings for both within-project and cross-project vulnerability detection. VulDeePecker [145], μVulDeePecker [307] and SySeVR [144] are a series of works that preserve the semantics of program by extracting API function calls and program slices for vulnerability detection. Le et al. [124] introduced a maximal divergence sequential auto-encoder network for identifying vulnerabilities in binary files. The model is designed to maximize the divergence between embeddings of vulnerable and invulnerable code. Zhou et al. [302] proposed Devign for vulnerability detection, which first represents a program by fusing its AST, CFG and DFG into a unified CPG, and subsequently designs a GNN to model the CPG of code. Similarly, Wang et al. [247] and Cao et al. [36] proposed a flow-sensitive framework for vulnerability detection, utilizing a GNN to represent the control, data, and call dependencies within a program. Cheng et al. [50] introduced DeepWukong, a GNN-based model for vulnerability detection of C/C++ programs, in which the flow information of programs is preserved. Liu et al. [159] introduced a GNN model with expert knowledge for detecting vulnerabilities in smart contracts, which incorporates the flow information of programs. Drawing inspiration from image processing, Wu et al. [269] proposed a method to enhance the scalability of vulnerability detection by transforming code into an image with semantics preserved, and implementing a CNN to capture them effectively.

Recently, several works have attempted to explain the results of deep learning models for vulnerability detection. Li et al. [140] introduced a GNN model for vulnerability detection that allows

[8] https://clang-analyzer.llvm.org/scan-build.html
[9] https://scan.coverity.com
[10] https://www.hpfod.com
[11] https://dwheeler.com/flawfinder
[12] https://fbinfer.com

for interpretability, by providing users with parts of the ***Program Dependency Graph (PDG)*** that may contain the vulnerability. Additionally, Zou et al. [308] proposed an interpretable deep-learning-based model based on heuristic searching for vulnerability detection.

In contrast to vulnerability detection which only classifies a program as vulnerable or non-vulnerable, another line of work is bug finding, which aims to pinpoint the buggy location. DeepBugs [194] is an approach for name-based bug detection, which trains a classifier to distinguish buggy or non-buggy code, based on deep learning. To enhance bug detection accuracy, Li et al. [142] proposed a fusion method leveraging both PDG and DFG for enhanced representation. The attention mechanism assigns higher weights to buggy paths to identify potential vulnerabilities. Gupta et al. [94] introduced a tree-structured CNN to pinpoint vulnerabilities or faults in erroneous programs concerning failed tests. Furthermore, Li et al. [139] framed fault localization as an image recognition problem and offered a deep learning approach integrating code coverage, data dependencies between statements, and code representations.

3.4.3 Type Inference. Programming languages with dynamic typing, such as Python and JavaScript, allow for rapid prototyping for developers and can save the time of software development dramatically. However, without the type information, unexpected run-time errors are prone to occur, which may introduce bugs and produce low-quality code. Current works on type inference, aimed at automatically inferring variable types, predominantly categorize into two types: static-analysis-based and learning-based approaches. Traditional static-analysis approaches [101, 207] are often imprecise since the behavior of programs is always over-approximated. Moreover, these approaches typically analyze the dependencies of entire programs, resulting in comparatively lower efficiency.

Recently, numerous deep learning techniques have been introduced for type inference. To the best of our knowledge, Hellendoorn et al. [104] was the first to employ deep learning for type inference. They proposed a neural network based on sequence-to-sequence architecture, named DeepTyper, which uses GRUs to represent the program context and predict the type annotations for TypeScript. Subsequently, Malik et al. [166] proposed NL2Type to predict type annotations by leveraging the natural-language information of programs. Based on NL2Type, Pradel et al. [193] further proposed TypeWriter, which utilizes both the natural-language information and programming context (e.g., arguments usage within a function). Wei et al. [260] introduced LambdaNet, a method for type inference using GNNs. LambdaNet begins by encoding code into a type dependency graph, preserving typed variables and their logical constraints. Subsequently, a GNN is employed to propagate and aggregate features across associated type variables, ultimately predicting type annotations. Pandi et al. [187] introduced OptTyper, which extracts relevant logical constraints and formulates type inference as an optimization problem. Allamanis et al. [8] proposed Typilus for Python type inference, expanding ASTs into a graph structure and predicting type annotations using GNNs over this graph. To address the challenge of large-scale type vocabulary, Mir et al. [171] introduced Type4Py, a similarity-based deep learning model with type clusters, enabling the inference of rare types and user-defined classes. Recently, Huang et al. [110] reformulated the type inference task as a cloze-style fill-in-the-blank problem, followed by training a CodeBERT model through prompt tuning.

3.5 Similarity-based Applications

Similarity-based applications, such as code search and code clone detection, aim to assess the likeness between a query (in either natural language or programming language) and a candidate code snippet. It is important to note that several approaches propose to reframe these tasks as a classification problem, where both the code and query are concatenated, and the goal is to determine their

relatedness [72]. In this paper, we differentiate between similarity-based and classification-based applications by the objects they address, namely, the query and candidate code snippet. Specifically, similarity-based applications center on tasks involving two objects.

3.5.1 Code Search. Code search aims to retrieve a code snippet by a natural-language query (*nl-to-code*) or code query (*code-to-code*). The *nl-to-code* search refers to searching code fragments that have similar semantics to the natural-language query from a codebase. As the first solution for code search using deep learning, Gu et al. [83] proposed DeepCS, which simultaneously learns the source code representation (e.g., function name, parameters and API usage) and the natural-language query in a shared feature vector space, with triplet criterion as the objective function. On the basis of DeepCS, Wan et al. [240] and Deng et al. [62] included more structural information of source code, including the ASTs and CFGs, under a multi-modal neural network equipped with an attention mechanism for better explainability. Ling et al. [149] first converted code fragments and natural-language descriptions into two different graphs, and presented a matching technique for better source code and natural-language description matching. Furthermore, Shi et al. [216] suggested an improved code search method by converting code graphs (e.g., CFGs and PDGs) into sequences through traversing. Haldar et al. [97] proposed a multi-perspective matching method to calculate the similarities among source code and natural-language query from multiple perspectives. Cambronero et al. [35] empirically evaluated the architectures and training techniques when applying deep learning to code search. Bui et al. [33] and Li et al. [135] leveraged contrastive learning with semantics-preserving code transformations for better code representation in code search.

Similar but different to the DeepCS framework, several more works have been proposed as complements for code search. Yao et al. [279] proposed using reinforcement learning to first generate the summary of code snippet and then use the summary for better code search. Sun et al. [222] suggested parsing source code to machine instructions, then mapping them into natural-language descriptions based on several predefined rules, followed by an LSTM-based code search model like DeepCS. Zhu et al. [304] considered the overlapped substrings between natural-language query and source code, and developed a neural network component to represent the overlap matrix for code search.

Recently, Chai et al. [42] suggested a transfer learning method for domain-specific code search, with the aim of transferring knowledge from Python to SQL. Wan et al. [241] examined the robustness of different neural code search models, and showed that some of them are vulnerable to data-poisoning-based backdoor attacks. Gu et al. [82] proposed to optimize code search by deep hashing techniques.

In contrast to *nl-to-code* search, the input of *code-to-code* search is source code, rather than natural-language description. The objective of the code-to-code search is to find code snippets that are semantically related to an input code from a codebase. The core technique of code-to-code search is to measure the similarity index between two code snippets, which is identical to the process of identifying code clones. More related work will be investigated in the code clone detection section.

3.5.2 Code Clone Detection. Numerous software engineering activities, including code reuse, vulnerability detection, and code search, rely on detecting similar code snippets (or code clones). There are basically four main types of code clones: Type-1 code clones are ones that are identical except for spaces, blanks, and comments. Type-2 code clones denote identical code snippets except for the variable, type, literal, and function names. Type-3 code clones denote two code snippets that are almost identical except for a few statements that have been added or removed. Type-4 code clones denote heterogeneous code snippets with similar functionality but differing code structures or syntax. To handle different types of code clones, various works have been proposed.

Recently, several deep-learning-based approaches have been designed for the semantics representation of a pair of code snippets for the task of clone detection. The core of these approaches lies in representing the source code as distributed vectors, in which the semantics are preserved. As an example, White et al. [263] proposed DLC, which comprehends the semantics of source code by considering its lexical and syntactic information, and then designs RNNs for representation. To improve the representation of the syntactic structure of code, Wei and Li [259] applied TreeLSTM to incorporate AST information of source code. Zhao and Huang [298] proposed encoding the CFG and DFG of code into a semantic matrix, and introduced a deep learning model to match similar code representations. Zhang et al. [294] and Büch and Andrzejak [29] designed approaches to better represent the ASTs of the program, and applied them for code clone detection task. Furthermore, Wang et al. [250], Nair et al. [176] and Mehrotra et al. [168] proposed to convert source code into graphs (e.g., CFG), represent the code graphs via GNN, and then measure the similarities between them. Instead of using GNN, Wu et al. [268] and Hu et al. [109] introduced a centrality analysis approach on the flow graph (e.g., CFG) of code for clone detection, inspired by social network analysis. Wu et al. [266] considered the nodes of an AST as distinct states and constructed a model based on a Markov chain to convert the tree structure into Markov state transitions. Then, for code clone detection, a classifier model is trained on the state transitions. Tufano et al. [236] empirically evaluated the effectiveness of learning representation from diverse perspectives for code clone detection, including identifiers, ASTs, CFGs, and bytecode. Recently, Ding et al. [67] and Tao et al. [231] utilized program transformation techniques to augment the training data, and then applied pre-training and contrastive learning techniques for clone detection. Gui et al. [86] studied a new problem of cross-language binary-source code matching by transforming both source and binary into LLVM-IRs.

3.6 Generation-based Applications

Generation-based applications, including code completion, code summarization, program translation, program synthesis, and program repair, are designed to produce source code, natural-language descriptions, or programs in an alternative programming language, in response to specific requirements presented in either natural language or (partial) code.

3.6.1 Code Completion. Code completion is a core feature of most modern IDEs. It offers the developers a list of possible code hints based on available information. Raychev et al. [203] made the first attempt to combine the program analysis with neural language models for better code completion. It first extracts the abstract histories of programs through program analysis, and then learns the probabilities of histories via an RNN-based neural language model. Similarly, various works [132, 152, 229] resort to inferring the next code token over the partial AST, by first traversing the AST in a depth-first order, and then introducing an RNN-based neural language model. To better represent the structure of code, Kim et al. [121] suggested predicting the missing partial code by feeding the ASTs to Transformers. Alon et al. [13] presented a structural model for code completion, which represents code by sampling paths from an incomplete AST. Furthermore, Wang and Li [254] suggested a GNN-based approach for code completion, which parses the flattened sequence of an AST into a graph, and represents it using a GGNN [137]. Guo et al. [90] modeled the problem of code completion as filling in a hole, and developed a Transformer model guided by the grammar file of a specified programming language. Brockschmidt et al. [26] expanded incomplete code into a graph representation, and then proposed a GNN for code completion. Svyatkovskiy et al. [227] proposed IntelliCode Compose, a pre-trained language model of code based on GPT-2, providing instant code completion across different programming languages. Liu et al. [153, 154] proposed a multi-task learning framework that unifies the code completion and

type inference tasks into one overall framework. Lu et al. [162] suggested a retrieval-augmented code completion method that retrieves similar code snippets from a code corpus and then uses them as external context. Since instant code completion is desired, several studies aim to improve the efficiency and flexibility of code completion. Svyatkovskiy et al. [228] suggested improving the efficiency of neural network models for code completion by reshaping the problem from generation to ranking the candidates from static analysis. Additionally, Shrivastava et al. [218] proposed a code completion approach that supports fast adaption to an unseen file based on meta-learning.

3.6.2 Code Summarization. Inspired by the text generation work in NLP, many approaches have been put forward to systematically generate a description or function name to summarize the semantics of source code. To the best of our knowledge, Allamanis et al. [11] was the first to use deep learning for code summarization. They designed a CNN to represent the code and applied a hybrid breath-first search and beam search to predict the tokens of the function name. Concurrently, Iyer et al. [112] proposed an LSTM-based sequence-to-sequence network with an attention mechanism for generating descriptions for source code. The sequence-to-sequence network inspired a line of works for code summarization, distinguished in code representation learning. To represent the AST information, Hu et al. [107], Alon et al. [12], and LeClair et al. [127] proposed to linearize the ASTs via traversing or path sampling, and used RNNs to represent the sequential AST traversals/paths for code summarization. Likewise, Fernandes et al. [73], LeClair et al. [126] and Jin et al. [118] investigated representing the structure of source code via a GNN, and verified it in code summarization. Guo et al. [91] designed the triplet position to model hierarchies in the syntax structure of source code for better code summarization. Recently, several works [4, 80, 230, 265] proposed to improve code summarization by designing enhanced Transformers to better capture the structural information of code (i.e., ASTs). Wan et al. [243], Shi et al. [213], Yang et al. [277], Gao and Lyu [76], and Wang et al. [252] proposed a hybrid representation approach by combining the embeddings of sequential code tokens and structured ASTs, and feeding them into a decoder network to generate summaries. As a complement, Haque et al. [99] and Bansal et al. [18] advanced the performance of code summarization by integrating the context of summarized code, which contains important hints for comprehending subroutines of code. Shahbazi et al. [211] leveraged the API documentation as a knowledge resource for better code summarization. Instead of generating a sequence of summary tokens at once, Ciurumelea et al. [56] resorted to suggesting code comment completions based on neural language modeling. Lin et al. [146] proposed to improve the code summarization by splitting the AST under the guidance of CFG, which can decrease the AST size and make model training easier.

Another line of work aims to utilize code search to enhance the quality of code summaries generated by deep learning models. For example, Zhang et al. [293], Wei et al. [258], Liu et al. [158] and Li et al. [131] suggested augmenting the provided code snippet by searching similar source code snippets together with their comments, for better code summarization. Instead of acquiring the retrieved samples in advance, Zhu et al. [306] suggested a simple retrieval-based method for the task of code summarization, which estimates a probability distribution for generating each token given the current translation context.

Apart from the above approaches, several works [108, 257, 271, 275, 281] are also worthy to be mentioned. Hu et al. [108] transferred the code API information as additional knowledge to the code summarization task. Xie et al. [271] investigated the task of project-specific code summarization with limited historical code summaries using meta-transfer learning. Wei et al. [257] and Yang et al. [275] framed the code generation task as a dual of code summarization, integrating dual learning to enhance summary generation. Similarly, Ye et al. [281] employed dual learning to leverage code generation for code search and code summarization. Mu et al. [174] introduced a multi-pass

deliberation framework for code summarization, inspired by human cognitive processes. Xie et al. [272] proposed a multi-task learning framework by leveraging method name suggestion as an auxiliary task to improve code summarization. Haque et al. [98] emphasized that predicting the action word (always the first word) is an important intermediate problem in order to generate improved code summaries. Recently, the consistency between source code and comments has also attracted much attention, which is critical to ensure the quality of software. Liu et al. [156], Panthaplackel et al. [188], and Nguyen et al. [179] trained a deep-learning-based classifier to determine whether or not the function body and function name are consistent. Panthaplackel et al. [189] and Liu et al. [160] proposed automatically updating an existing comment when the related code is modified, as revealed in the commit histories. Gao et al. [77] proposed to automate the removal of obsolete TODO comments by representing the semantic features of TODO comments, code changes, and commit messages using neural networks. Li et al. [133] proposed to generate review comments automatically based on pre-trained code models.

3.6.3 Program Translation. Translating programs from a deprecated programming language to a modern one is important for software maintenance. Many neural machine translation-based methods have been proposed for program translation. In order to utilize the AST structure of code, Chen et al. [47] proposed Tree2Tree, a neural network with structural information preserved. It first converts ASTs into binary trees following the left-child right-sibling rule, and then feeds them into an encoder-decoder model equipped with TreeLSTM. Gu et al. [85] presented DeepAM, which can extract API mappings among programming languages without the need of bilingual projects. Recently, Rozière et al. [205] proposed TransCoder, a neural program translator based on unsupervised machine translation. Furthermore, Rozière et al. [206] leveraged the automated unit tests to filter out invalid translations for unsupervised program translation.

3.6.4 Program Synthesis. Program synthesis is a task for generating source code using high-level specifications (e.g., program descriptions or input-output samples). Given the natural-language inputs, current approaches resort to generating programs through machine translation. For semantic parsing, Dong and Lapata [69] proposed an attention-based encoder-decoder model, which first encodes input natural language into a vector representation using an RNN, and then incorporates another tree-based RNN to generate programs. Liu et al. [151] proposed latent attention for the If-Then program synthesis, which can effectively learn the importance of words in natural-language descriptions. Beltagy and Quirk [21] modeled the generation of If-Then programs from natural-language descriptions as a structure prediction problem, and investigated both neural network and logistic regression models for this problem.

Unlike synthesizing simple If-Then programs, Yin and Neubig [284] proposed a syntax-preserving model for general-purpose programming languages, which generates Python code from pseudo code, powered by a grammar model that explicitly captures the compilation rules. Maddison and Tarlow [165] proposed a probabilistic model based on **probabilistic context-free grammars (PCFGs)** for capturing the structure of code for code generation. Ling et al. [148] collected two datasets (i.e., Hearthstone and Magic the Gathering) for code generation in trading card games, and proposed a probabilistic neural network with multiple predictors. On the basis of [148], Rabinovich et al. [198] proposed to incorporate the structural constraints on outputs into a decoder network for executable code generation. Similarly, Sun et al. [225] and Sun et al. [226] designed a tree-based CNN and Transformer, respectively, for code generation and semantic parsing tasks based on the sequence-to-sequence framework. Hayati et al. [103] suggested using a neural code generation model to retrieve action subtrees at test time.

Instead of synthesizing programs from natural-language descriptions, several works resort to generating programs from the (pseudo) program in another format or language. Iyer et al. [113]

proposed to synthesize the AST derivation of source code given descriptions as well as the programmatic contexts. The above approaches are driven by well-labeled training examples, while Nan et al. [177] proposed a novel approach to program synthesis without using any training example, inspired by how humans learn to program.

Recently, various pre-trained code models also achieved significant progress in code generation. CodeGPT [163] is a Transformer-based model that is trained using corpus for program synthesis, following the same architecture of GPT-2. CodeT5 [255] is a pre-trained code model in eight programming languages based on T5 [199], which incorporates an identifier-aware objective during its pre-training phase. Xu et al. [273] endeavored to integrate external knowledge into the pre-training phase to enhance code generation from natural-language input. Codex [46] is a GPT model trained on a code corpus sourced from GitHub. This model has played a pivotal role as the underpinning framework for Copilot.[13] Li et al. [136] introduced AlphaCode, a code generation system designed to produce distinctive solutions for complex problems that demand profound cognitive engagement. Poesia et al. [191] introduced a constrained semantic decoding mechanism into a pre-trained model, to constrain outputs of the model in a set of valid programs. More recently, the code generation has been dominated by the LLMs, including CodeGen [181], CodeT5+ [253], InCoder [74], GPT-3.5 [184], StarCoder [134], Code Llama [204], and WizardCoder [164]. These LLMs have achieved significant progress in generating independent functional code as demonstrated in the HumanEval benchmark [46]. Despite this, generating higher-level (i.e., class-level and repository-level) code that requires contextual information remains challenging. In response, various benchmarks, such as ClassEval [70] and CoderEval [285], have been introduced to address this need.

Programming by example is another flourishing direction for program synthesis. Shu and Zhang [219] proposed a ***Neural Programming By Example (NPBE)*** model, which learns to solve string manipulation problems through inducting from input-output strings. Balog et al. [17] proposed DeepCoder, which trains a model to predict possible functions useful in the program space, so as to guide the conventional search-based synthesizer. Devlin et al. [65] proposed RobustFill, which is an end-to-end neural network for synthesizing programs from input-output examples. Nye et al. [183] developed a neuro-symbolic program synthesis system called SketchAdapt, which can build programs from input-output samples and code descriptions by intermediate sketch. Bavishi et al. [20] proposed a program candidate generator, backed by GNNs, for program synthesis in large real-world API.

It is worth mentioning that there are many works on generating code from natural language for specific domain-specific programming languages, e.g., Bash and SQL. WikiSQL [300], Spider [288], SparC [289], and CoSQL [287] are four datasets with human annotations for the task of text-to-SQL. Based on these datasets, many works [286, 287, 289] have been proposed. For example, Seq2SQL [300] is a neural machine translation model to generate SQL queries from natural-language descriptions with reinforcement learning. Cai et al. [34] further proposed an encoder-decoder framework to translate natural language into SQL queries, which integrates the grammar structure of SQL for better generation. Yu et al. [286] proposed a neural network SyntaxSQLNet, with syntax tree preserved, for the task of text-to-SQL translation across different domains, which takes the syntax tree of SQL into account during generation.

3.6.5 Program Repair. Automatically localizing and repairing bugs in programs can save much manual effort in software development. One line of work is to learn the patterns of how programmers edit the source code, which can be used to check syntax errors while compiling. Bhatia

[13] https://github.com/features/copilot

and Singh [24] and Santos et al. [208] proposed RNN-based language models for correcting syntax errors in programs. DeepFix [95] and SequenceR [49] are two sequence-to-sequence models for syntax error correction, by translating the erroneous programs into fixed ones. Furthermore, Gupta et al. [93] improved program repair by reinforcement learning. Vasic et al. [237] proposed multi-headed pointer networks (one head each for localization and repair) for jointly localizing and repairing misused variables in code. Dinella et al. [66] presented Hoppity to jointly detect and fix bugs based on neural Turing machine [81], where a GNN-based memory unit is designed for buggy program representation, and an LSTM-based central controller is designed to predict the operations of bug fixing, e.g., patch generation and type prediction. Tarlow et al. [232] proposed Graph2Diff, which designs a GNN for representing the graph structure of programs, and a pointer network to localize the initial AST to be edited. Mesbah et al. [169] and Chakraborty et al. [43] proposed to model the modifications of ASTs, and designed a neural machine translation model to generate correct patches. Zhu et al. [305] presented a syntax-directed decoder network with placeholder generation for program repair, which aims to generate program modifications rather than the target code. Yasunaga and Liang [280] proposed DrRepair, which first builds a program-feedback graph to align the corresponding symbols and diagnostic feedback, and then designs a GNN to generate repaired code. Li et al. [141] introduced a novel deep learning-based method for fixing general bugs, which combines spectrum-based fault localization with deep learning and flow analysis.

Benefiting from the pre-training techniques in NLP, TFix [23] and VulRepair [75] directly posed program repair as a text-to-text problem and utilized a model named T5 [199]. Specifically, it digests the error message and directly outputs the correct code. Jiang et al. [116] proposed CURE for program repair, which is composed of a pre-trained language model, a code-aware search method, and a sub-word tokenization technique.

Another line of work is focusing on repairing programs by generating patches. Tufano et al. [235] carried out an empirical study to evaluate the viability of applying machine translation to generate patches for program repair in real-world scenarios. Different from [235] which targets at function-level small code snippets, Hata et al. [102] trained a neural machine translation model, targeting at statements, by learning from the corresponding pre- and post-correction code in previous commits. Harer et al. [100] proposed to generate the input buggy code via generative adversarial networks so that the correction model can be trained without labeled pairs. Gupta et al. [92] embedded execution traces in order to predict a sequence of edits for repairing Karel programs. Li et al. [138] treated the program repair as code transformation and introduced two neural networks, a tree-based RNN for learning the context of a bug patch, and another one designed to learn the code transformation of fixing bugs. White et al. [262] introduced a novel approach for selecting and transforming program repair patches using deep-learning-based code similarities. Empirically, Tian et al. [233] studied the practicality of patch generation through representation learning of code changes.

4 BENCHMARK

Even though significant progress has been made in code intelligence with deep learning, two limitations remain obstacles to the development of this field. (1) *Lack of standardized implementation for reproducing the results.* It has become a common issue that deep-learning-based models are difficult to reproduce due to the sensitivity to data and hyperparameter tuning. From our investigation, most of them are implemented independently using different toolkits (i.e., PyTorch and TensorFlow). There is a need for a unified framework that enables developers to easily evaluate their models by utilizing some shared components. Actually, in the artificial intelligence area (e.g., NLP and computer vision), many toolkits such as Fairseq [186], AllenNLP [78], Detectron2 [267]

have been developed, which significantly advance the progress of their corresponding research areas. (2) *Lack of benchmarks for fair comparisons.* Currently, many approaches have been proposed and each of them claims that the proposed approach has outperformed other ones. To identify where the performance improvements come from, it is essential to create a benchmark for fair comparisons.

Based on these motivations, we propose NaturalCC (standards for Natural Code Comprehension), a thorough platform for evaluating source code models using deep learning techniques. Under this platform, we also benchmark five specific application tasks, including code summarization, code search, code completion, program synthesis, and type inference. The implementation and usage of NaturalCC will be introduced in Section 5.

4.1 Code Summarization

4.1.1 Approaches. Currently, most deep-learning-based code summarization methods use the encoder-decoder architecture. An encoder network is used to convert the input source code into an embedding vector, and the decoder network is used to generate output summaries from the encoded vector. In this paper, we benchmark the following representative methods for code summarization, including three different encoder models (i.e., LSTM, TreeLSTM, and Transformer), an encoder-decoder model (i.e., CodeT5), as well as a large-scale decoder model (i.e., GPT-3.5-turbo).

- **Seq2Seq+Attn** [112, 243] is a vanilla model following sequence-to-sequence architecture with attention mechanism. It is a famous method for neural machine translation. Unlike works that only represent the source code as token embedding [112], we represent the source code via an LSTM network and generate the summary via another LSTM network.
- **Tree2Seq+Attn** [243] also follows the structure of Seq2Seq. The difference is that it uses TreeLSTM as the encoder network for syntax-aware modeling of code. Moreover, an attention module is also designed to attend to different nodes of the syntax tree of code.
- **Transformer** [4] is currently considered the leading approach for code summarization, which has also achieved significant improvement in neural machine translation. In Transformer, a relative position embedding, rather than absolute position embedding, is introduced for modeling the positions of code tokens.
- **PLBART** [3] is built on the top of BART [128], which is originally designed for text understanding and generation. PLBART can be seen as a specific BART model pre-trained on code corpus.
- **CodeT5** [255] is an encoder-decoder model that builds upon the T5 architecture [199]. Pre-trained on the CodeSearchNet dataset [111], it supports six programming languages and can perform both code understanding and generation tasks.
- **GPT-3.5-turbo** [184] is a large-scale decoder-only model based on Transformer architecture. It is pre-trained on a diverse array of data, encompassing both natural language and code, and can learn a specific task given an instruction and several demonstration examples.

4.1.2 Results. We evaluate the performance of each model on the Python-Doc [19, 243] dataset using the BLEU, METEOR, and ROUGE metrics as in [243]. GPT-3.5-turbo is invoked online via the OpenAI API, and we present five code-summary pairs before the test item for demonstration. The time cost is reported for generating each summary item rather than for each batch. The overall performance is summarized in Table 2. This table shows that GPT-3.5-Turbo, which utilizes the Transformer architecture and pre-training techniques, achieves the highest performance in BLEU and METEOR scores. However, it is out of our expectation that GPT-3.5-turbo exhibits a lower ROUGE-L score. To understand the underlying causes, we manually examined the output, finding that it tends to produce fluent yet longer sentences compared to the reference solution,

Table 2. Performance of our Model and Baseline Methods for Code Summarization over Python-Doc Dataset

	BLEU	METEOR	ROUGE-L	Time Cost
Seq2Seq+Attn	25.57	14.40	39.41	0.09s/Batch
Tree2Seq+Attn	23.35	12.59	36.49	0.48s/Batch
Transformer	30.64	17.65	44.59	0.26s/Batch
PLBART	32.71	18.13	46.05	0.26s/Batch
CodeT5	33.25	20.35	**50.06**	0.82s/Batch
GPT-3.5-turbo	**40.52**	**29.66**	42.88	1.23s/Item

incorporating additional words to preserve fluency, thereby diminishing the longest common subsequence length shared with the reference solution, which is an important factor in evaluating the ROUGE-L score. This tendency is likely due to their few-shot in-context learning approach [28], which does not adequately teach the model to mimic the stylistic nuances in ground truth outputs (i.e., generating concise summaries). It is interesting to see that the simple Seq2Seq+Attn outperforms the Tree2Seq+Attn that considers the AST of code. For Transformer, we find that the relative position embedding can indeed represent the relative relationships among code tokens.

4.2 Code Search

4.2.1 Approaches. CodeSearchNet Challenge [111] is an open challenge designed to assess the current state of code search. In [111], the authors have benchmarked four code search methods. The fundamental idea of [111] is to learn a joint embedding of code and natural-language query in a shared vector space. That is, two encoders are used for representing the source code and query, respectively. A loss function is then designed to maximize the weighted sum for paired embeddings of source code and natural-language query. Based on different encoder networks, we have implemented the following four variant models.

- **Neural Bag of Words (NBOW)** [111] is a naive approach by representing the input sequences by a bag of words. For a given code snippet or some specified query written in natural language, it represents tokens into a collection of word embeddings before feeding them into a max pooling layer for creating a sentence-level representation.
- **Bidirectional RNN models (biRNN)** [111] proposes to represent the semantics of source code and query via RNN models. In particular, we adopt the two-layer bidirectional LSTM network.
- **1D Convolutional Neural Network (1D-CNN)** [111] employs convolutional neural layers for code and query representation, and builds a residual connection at each layer.
- **Self-Attention (SelfAtt)** [111] adopts self-attention layers to capture the semantic information of sequential source code and query.

4.2.2 Implementation Details. We employ word-level BPE to tokenize both code snippets and natural-language descriptions in the considered methods. Subsequently, a shared vocabulary of size 50,000 is constructed based on the sorted token frequency. All models undergo training on a singular Nvidia RTX V100 GPU, utilizing a learning rate of 5×10^{-4}. The gradient norm is maintained at 1.0, and a batch size of 1,000 is specified to expedite training. The optimization process for all models is executed using the Adam optimizer.

4.2.3 Results. We evaluate the performance of each model on the CodeSearchNet corpus using the MRR metric, as described in [111]. The overall performance of each model is summarized in

Table 3. MRR of our Model and Baseline Methods for Code Search over CodeSearchNet Dataset

	Go	Java	JavaScript	PHP	Python	Ruby	Time Cost
NBOW	66.59	59.92	47.15	54.75	63.33	42.86	0.16s/Batch
1D-CNN	70.87	60.49	38.81	61.92	67.29	36.53	0.30s/Batch
biRNN	65.80	48.60	23.23	51.36	48.28	19.35	0.74s/Batch
SelfAtt	**78.45**	**66.55**	**50.38**	**65.78**	**79.09**	**47.96**	0.25s/Batch

Table 4. MRR of our Model and Baseline Methods for Code Completion over Py150 Dataset

	Attribute	Number	Identifier	Parameter	All Tokens	Time Cost
LSTM	51.67	47.45	46.52	66.06	73.73	0.31s/Batch
GPT-2	70.37	62.20	63.84	**73.54**	82.17	0.43s/Batch
TravTrans	**72.08**	**68.55**	**76.33**	71.08	**83.17**	0.43s/Batch

Table 3. As shown in the table, it is clear that the NBOW model with the simplest architecture achieves a comparable performance, at the lowest cost. Moreover, we can also observe that the performance of biRNN is poor, in both effectiveness and efficiency. The recurrent characteristic of RNN makes it time-consuming. The SelfAttn model obtains the best results, which may be attributed to its use of the self-attention mechanism.

4.3 Code Completion

4.3.1 Approaches. The code completion task aims to generate the completion text based on the given partial code. In this paper, we investigate three representative approaches.

- **LSTM** [121] denotes the model that represents the partial code by LSTM, and then predicts the missing token via a softmax layer.
- **GPT-2** [121] is a pre-trained language model based on Transformer. It refers to the Transformer model that is trained by iteratively predicting the next code token.
- **TravTrans** [121] is designed to preserve the syntax structure of source code while predicting the missing token. It first linearizes the code ASTs into a sequence of tokens using depth-first traversing, and afterward feeds the traversal into Transformer for representation. It also uses a softmax layer to predict the missing token.

4.3.2 Implementation Details. For acquiring high-quality code tokens, we perform preprocessing on the code snippets by parsing them into ASTs and extracting their leaf nodes as code tokens. We establish a unified vocabulary comprising 50, 000 tokens, organized based on token frequency. All models undergo training utilizing four Nvidia RTX V100 GPUs, employing a learning rate of 1×10^{-3}, and a batch size of 32. The optimization of all models is executed using the Adam optimizer.

4.3.3 Results. We evaluate each model on the Py150 [202] dataset using the MRR metric as used in [121]. We divide the prediction tokens into five categories, namely attributes, numeric constants, identifier names, function parameters, and all tokens. We summarize the performance of each model in Table 4. From this table, when comparing GPT-2 with LSTM, we can observe that the Transformer architecture outperforms other models in representing the semantics of code, thus, resulting in better performance for code completion. Furthermore, when comparing TravTrans with GPT-2, we can see that the TravTrans that incorporates the syntax structure information achieves better performance, showing that the syntax information is useful for code completion.

Table 5. Pass@k with $k = 1, 5$ of Code Generation Models over HumanEval and MBPP Dataset

	Pass@1	Pass@5	Time Cost	Pass@1	Pass@5	Time Cost
	HumanEval			MBPP		
StarCoder	59.80%	68.02%	5.73 s/Batch	45.20%	51.35%	11.25 s/Batch
GPT-3.5-Turbo	70.10%	80.56%	1.17 s/Item	49.71%	58.50%	2.58 s/Item
GPT-4	80.18%	91.71%	1.81 s/Item	59.70%	67.25%	3.10 s/Item

4.4 Program Synthesis

4.4.1 Approaches. The task of program synthesis is focused on generating explicit code based on high-level specifications, such as program descriptions or input-output samples. This task is inherently more challenging than code completion, as it necessitates an understanding of human intentions and generating complete code. With the development of large language models, the field of program synthesis has recently garnered widespread interest. In this paper, we investigate three representative approaches.

— **StarCoder** [204] is a decoder-only LLM pre-trained on 1 trillion tokens from 80+ programming languages from GitHub. It can be used for various code tasks including code completion, editing via instructions, and explaining code in natural language. We choose the 15 billion parameter version fine-tuned using an EVOL-Instruct objective on 20K instruction-following dataset named Code Alpaca [164].
— **GPT-3.5-Turbo** [184] and **GPT-4** [185] are two large-scale decoder-only language models pre-trained on a diverse dataset including code and natural language. Compared with GPT-3.5-Turbo, GPT-4 has a significantly larger number of parameters (approximately 1.5 trillion, whereas GPT-3.5 has 175 billion), and has a 32K context window. GPT-4 is optimized for handling complex reasoning and planning tasks [185].

4.4.2 Results. We evaluate each model on the HumanEval [46] and the ***Mostly Basic Python Problems (MBPP)*** [16] dataset, and using the Pass@1 and Pass@5 metric as in [46]. We evaluate in a zero-shot setting, only presenting the task requirement without demonstration examples as these models are capable of code generation tasks. The performance of different models is summarized in Table 5. As shown in the table, it is clear that the GPT-4 model with the most parameters achieves the best performance in both the HumanEval and MBPP datasets, at the cost of longer inference time. Also, deploying LLMs such as StarCoder locally for code generation services needs acceleration techniques due to their excessively long inference time.

4.5 Type Inference

4.5.1 Approaches. Similar to code completion, the type inference task aims to predict the types of variables based on contextual information. It first represents the contextual code into a vector, and then predicts the missing types by a softmax layer. In our work, we employ two state-of-the-art methods for this task.

— **DeepTyper** [104] proposes to represent the contextual code by a two-layer biGRU, and then predicts the missing variable types via a softmax layer.
— **Transformer** [4] proposes to represent the contextual code by a Transformer encoder network, and then predicts the missing variable types via a softmax layer.

4.5.2 Implementation Details. We initially tokenize both the code snippets and natural-language descriptions. Subsequently, we establish a common vocabulary comprising 40,000 tokens, determined by sorting them based on frequency. The hardware configuration for training

Table 6. Accuracy of our Model and Baseline Methods for Type Inference over Py150 Dataset

	Accuracy@1	Accuracy@5	Accuracy@1	Accuracy@5	Time Cost
	All types		Any types		
DeepTyper	**0.52**	**0.67**	**0.43**	0.67	0.42s/Batch
Transformer	0.34	0.64	0.37	**0.75**	0.85s/Batch

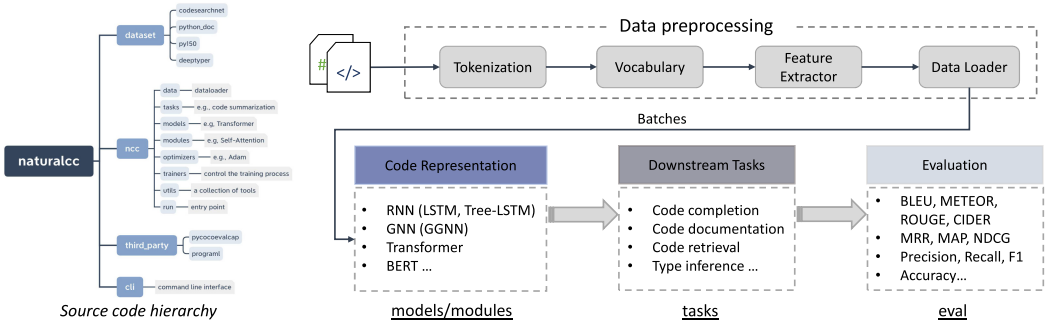


Fig. 4. The source code hierarchy and pipeline of NaturalCC.

and the optimizer employed remains consistent with the aforementioned specifications. A batch size of 16 and a learning rate of 1×10^{-4} are utilized.

4.5.3 Results. We evaluate each model on the Py150 [202], by using the Accuracy metric as in [114]. In particular, we measure the performance under the settings of *all types* and *any types*. The performance of different models is summarized in Table 6. From this table, it is interesting to see that the simple LSTM-based DeepTyper outperforms the Transformer-based approach, especially under the *all types* setting, at a lower time cost.

5 TOOLKIT AND DEMONSTRATION

This section introduces the design of NaturalCC and its user interface. Figure 4 (left) shows the code structure of NaturalCC. The `dataset` folder contains data preprocessing code. The `ncc` folder is the core module. The `third_party` folder holds model evaluation packages. The `gui` folder contains graphical user interface files and assets. As shown in Figure 4 (right), NaturalCC is composed of four components, i.e., data preprocessing, code representation, downstream tasks, and their corresponding evaluations. At the stage of data preprocessing, we process the source code with a series of steps, including word tokenization, building vocabulary, and feature extraction. Additionally, a data loader is used to iteratively yield batches of code samples with their features. The resulting batches are then sent into the code representation models, which facilitate a variety of downstream tasks, including code summarization, code search, code completion, and type inference. To evaluate the performance of each task, we also implement several corresponding metrics that have been widely adopted previously.

5.1 Data Preprocessing Module

In NaturalCC, we have collected and processed four datasets including CodeSearchNet [111], Python-Doc [243], Py150 [202], and DeepTyper [104]. First, we tokenize the input source code, and then build a vocabulary to map the code tokens into indexes. Currently, we support two types of tokenizations: space tokenizer and BPE tokenizer [120]. Along with code tokens, we also

explore different features of code, such as AST, IR, CFGs, and DFGs. All the related scripts for data preprocessing have been put in the `data` and `dataset` folders.

5.2 Code Representation Module

As the core component of NaturalCC, we have implemented several encoders that are widely used in state-of-the-art approaches for source code representation, including RNN, GNN, and Transformer. For example, we have implemented LSTM, TreeLSTM and Transformer networks for sequential tokens and (linearized) ASTs. We have also implemented a GNN, i.e., GGNN, to represent the control-flow graph of source code. It is worth mentioning that in NaturalCC, we have also incorporated the pre-training approaches for source code. We have implemented several state-of-the-art pre-trained code models, including CodeBERT [72], PLBART [3], and GPT-2 [163]. The `models` and `modules` folders contain all the implemented networks for code representation.

5.3 Tool Implementation

NaturalCC is mainly implemented by PyTorch, and builds upon other successful open-source toolkits in NLP, such as Fairseq, and AllenNLP.

Registry Mechanism. To be flexible, NaturalCC is expected to be easily extended to different tasks and model implementations, with minimum modification. Similar to Fairseq, we design a `register` decorator on instantiating a new task or model, the implementation of which is in the corresponding `__init__.py` in each folder. The registry mechanism is to create a global variable to store all the available tasks, models, and objects at the initialization stage, so that users can easily access them throughout the whole project.

Efficient Training. NaturalCC supports efficient training of models in a distributed way through `torch.distributed`. It can utilize multiple GPUs across different servers. Furthermore, NaturalCC can support calculation in mixed precision to further increase the training speed, including both FP32 and FP16 training. Typically, the gradients are updated in FP16 while the parameters are saved in FP32.

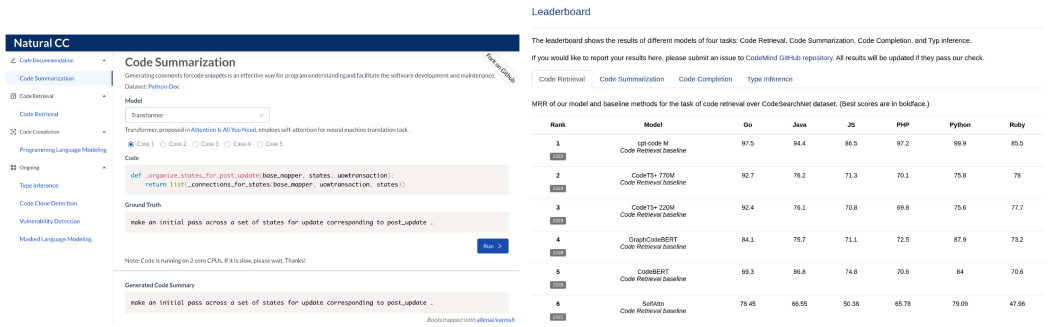
Flexible Configuration. Instead of employing `argparse` for managing command-line options within Fairseq, we advocate the adoption of individual `yaml` configuration files for each model's configuration. We contend that the flexibility offered by modifying these `yaml` configuration files is better suited for model exploration.

5.4 Graphical User Interface

We also develop a web-based graphical user interface to facilitate users in exploring the outcomes of trained models. The design is based on the open-source demonstration of AllenNLP [78]. Figure 5(a) displays a screenshot of our demonstration system, which currently features three tasks of code intelligence: code summarization, code search, and code completion. We leave the integration of other code intelligence tasks to our future work.

5.5 Leaderboard

We release a leaderboard so that researchers can report the results of their own models and compete with others, as shown in Figure 5(b). Currently, we only support researchers and developers who use NaturalCC to implement their approach and update the experimental results via pull requests in GitHub. In our future work, we will build a web-based service, which allows users to upload their predicted results and evaluate the model performance automatically using the ground-truth labels as a reference.



(a) Demonstration (b) Leaderboard

Fig. 5. Screenshots of GUI and leaderboard of NaturalCC.

6 CHALLENGES AND OPPORTUNITIES

Although much effort has been made into deep learning for code intelligence, this area of research is still in its infancy with many open challenges and opportunities. To inspire future research, this section suggests several potential directions that are worth pursuing.

Comprehensive Code Comprehension. Designing a representation approach to effectively and efficiently preserve the semantics of programs has always been a fundamental problem in code intelligence. Despite much effort on code representation, as mentioned in this paper, there are still three main obstacles to be overcome. *(a) Open Vocabulary.* Building a vocabulary to index the textual tokens of code is the first step toward applying deep learning models for code intelligence. Since the unambiguous characteristic of code, the vocabulary in code is much more open and complicated than the vocabulary in natural languages. The vocabulary of programming languages often consists of keywords, identifiers, customized method names, and variable names. The large vocabulary contains much "noise", making it difficult to comprehend the code. Although many attempts [53, 60, 120] have been made towards mitigating the OOV issue, it remains a challenge to design a simple yet effective approach to map the source code into indexes while preserving the semantics. *(b) Complex Structure of Program.* Unlike natural language, code is written with strict grammar. The computations described by code can be executed in an order that is different from the order in which the code was written. This is often seen in operations such as loops, recursions, and pointer manipulation. Although many attempts to capture the structure of code from different modalities, as we surveyed in this paper, we believe that the structures of code are not sufficiently preserved, and more effort is needed here. Inspired by the GNNs, there is potential to design specific GNNs to better represent the structure of programs. For example, from our analysis, ASTs, CFGs, DFGs and CPGs all have high heterogeneity. It is desirable to design some heterogeneous-information-network-based approaches [223] to represent the heterogeneous code graph. *(c) How to Feed the Program Structures into LLMs?* Current LLM architectures predominantly rely on pre-training with sequential data, lacking support for structural inputs. However, source code exhibits rich structural features, including CFG, DFG, and PDG. Consequently, integrating these structural features into LLMs has emerged as a prominent research endeavor. Recent studies [6, 244] have explored the incorporation of such features into LLMs through designing prompts.

Data Hungry and Data Quality. Despite much progress achieved in deep-learning-based approaches for code intelligence, we argue that existing approaches still suffer from the data-hungry issue. In other words, the effectiveness of cutting-edge techniques significantly depends on the

availability of vast quantities of expensive and labor-intensive well-labeled training data. Training the model on a small qualified dataset will result in far less imprecise results, especially for new programming languages or languages with an inadequate number of labeled samples. Therefore, it is important to design approaches to reduce the reliance on a large quantity of labeled data. A similar problem exists in the field of machine learning. One promising solution for this dilemma is transfer learning, which has achieved great success in transferring knowledge to alleviate the data-hungry issue in computer vision and NLP. Similarly, to model an emerging programming language with limited data, it is desirable to mitigate the data-hungry issue by leveraging models trained in programming languages with sufficient labeled training data [42, 45, 57]. Data quality is also a crucial issue for code intelligence, which may exacerbate the data-hungry problem. From our analysis, the collected datasets from online resources, like GitHub and StackOverflow, are not quality ensured. Sun et al. [224] and Shi et al. [215] investigated the importance of data quality and verified it on the tasks of code search and code summarization, respectively. In the area of LLMs, the significance of data quality has surged. For instance, Gunasekar et al. [87] demonstrated the efficacy of training an LLM with a relatively modest parameter count of 1.3 billion using textbook data. Their work underscores the growing importance of meticulously selecting training data and harnessing synthetic data, a trend expected to intensify in the foreseeable future.

Multi-Lingual and Cross-Language. The codebase written in multiple programming languages can be considered a multi-lingual corpus, as in NLP. However, the multi-lingual problem in programming languages has not been well investigated. Different from the multi-lingual problems studied in NLP, the corpus of multiple programming languages will bring more opportunities and challenges to future research. Recently, several attempts have been made to learn the common knowledge shared among multiple programming languages, and transfer the knowledge across different programming languages. For example, Zhang et al. [291] proposed obtaining better interpretability and generalizability by disentangling the semantics of source code from multiple programming languages based on variational autoencoders. Zügner et al. [309] introduced a language-agnostic code representation based on the features directly extracted from the AST. Ahmed and Devanbu [5] conducted an exploratory study and revealed the evidence that multilingual property indeed exists in the source code corpora. For example, it is more likely that programs that solve the same problem in different languages make use of the same or similar identifier names. They also investigate the effect of multilingual (pre-)training for code summarization and code search. Nafi et al. [175] proposed CLCDSA, a cross-language clone detector with syntactical features and API documentation. Bui et al. [30] proposed a bilateral neural network for the task of cross-language algorithm classification. Bui et al. [31] proposed SAR, which can learn cross-language API mappings with minimal knowledge. Recently, Chai et al. [42] proposed a novel approach termed CDCS for domain-specific code search through transfer learning across programming languages. Gui et al. [86] proposed an approach that matches source code and binary code across different languages based on intermediate representation.

Model Interpretability. Lack of interpretability is a common challenge for most deep learning-based techniques for code intelligence, as deep learning is a black-box method. New methods and studies on interpreting the working mechanisms of deep neural networks should be a potential research direction. Recently, several efforts have been made toward increasing the interpretability of deep-learning-based models. As an example, Li et al. [140] presented a novel approach to explain predicted results for GNN-based vulnerability detection by extracting sub-graphs in the program dependency graph. In addition, Zou et al. [308] proposed interpreting a deep-learning-based model for vulnerability detection by identifying a limited number of tokens that play a significant role in the final prediction of the detectors. Zhang et al. [295] proposed interpretable program synthesis

that allows users to see the synthesis process and have control over the synthesizer. Pornprasit et al. [192] proposed a local rule-based model-agnostic approach, termed PyExplainer, to explain the predictions of just-in-time defect models. Rabin et al. [196] proposed a model-agnostic explainer based on program simplification, inspired by the delta debugging algorithms. Wan et al. [242], López et al. [161], and Sharma et al. [212] investigated the explainability of pre-trained code models through probing the code attention and hidden representations. We believe that it is essential to enhance the interpretability of current deep-learning-based approaches for code intelligence.

Effective and Efficient Use of Models. Recently, substantial advancements have been observed in the capabilities of LLMs concerning code intelligence. However, the optimal utilization of LLMs to harness their full potential in code intelligence poses a pressing challenge. To this end, several efforts have been undertaken, including the following three aspects. *(a) Prompt Engineering.* One prominent avenue of research revolves around crafting prompts aimed at enhancing interactions with LLMs through model inference. Numerous studies have delved into crafting prompts customized for tasks such as code generation [150], code summarization [79], program repair [270], and vulnerability detection [68], respectively. *(b) Parameter-Efficient Tuning.* As the size of LLMs increases, the expense associated with fully fine-tuning them escalates. Consequently, prompt tuning has emerged as a solution to mitigate this challenge. In [245], the authors have empirically explored the prompt tuning techniques in code intelligence tasks. *(c) Model Compression.* As current pre-trained code models continue to increase in size, the computational expense of pre-training on large-scale code corpora remains a significant challenge, resulting in high costs associated with both training and model inference. Zhang et al. [297] and Shi et al. [214] proposed to improve the efficiency of the training process by model compressing. It is a promising research direction to reduce the computational resources of pre-trained code models.

Robustness and Security. Despite significant progress being made in the training of accurate models for code intelligence, the robustness and security of these models have rarely been explored. As seen in the fields of NLP and CV, deep neural networks are frequently not robust [40]. Specifically, current deep learning models can be easily deceived by adversarial examples, which are created by making small changes to the inputs of the model that it would consider as benign. There are many different ways to produce adversarial samples in the computer vision and NLP communities, particularly for image classification [37, 40, 71] and sentiment classification [296]. Similarly, for source code models, the adversarial attack also exists. Recently, there have been several efforts to investigate the robustness and security of deep-learning-based models for code intelligence. For example, Ramakrishnan et al. [201] and Yefet et al. [282] investigated how to improve the robustness of source code models through adversarial training. Nguyen et al. [178] empirically investigated the use of adversarial learning techniques for API recommendation. Bielik and Vechev [25] introduced a novel method that incorporates adversarial training and representation refinement to create precise and robust models of source code. Zhou et al. [303], Yang et al. [278] and Zhang et al. [290] proposed a black-box attack for neural code models by generating adversarial examples while preserving the semantics of source code. Based on semantics-preserving code transformations, Quiring et al. [195] and Liu et al. [157] developed a novel attack against authorship attribution of source code. Ramakrishnan and Albarghouthi [200] investigated the possibility of injecting a number of common backdoors into deep-learning-based models, and developed a protection approach based on spectral signatures. Schuster et al. [209] and Wan et al. [241] proposed attacking the neural code models through data poisoning, and verified it in code completion and code search, respectively. Severi et al. [210] suggested an explanation-guided backdoor approach to attack the malware classifiers. Overall, exploring the robustness and security of code intelligence models is an interesting and important research direction.

Privacy. Despite the significant progress in code intelligence achieved through deep learning techniques, particularly LLMs, criticism has been raised regarding their strong propensity to memorize training data, especially concerning privacy concerns due to the inadvertent exposure of sensitive information. One line of work is on membership inference attacks, the aim of which is to infer whether a data sample has been used in training or not [217, 283]. To achieve this goal, one dominant approach is shadow model training [217], which aims to train a binary attack classifier by creating multiple shadow models to mimic the behavior of the target model. Another line of work is on training data extraction attacks [38, 39, 197]. In [39], the authors empirically revealed that LLMs resort to memorizing privacy-sensitive information that has been presented in training data, including personally identifiable information, URLs, code snippets, and UUIDs.

7 CONCLUSION

In this paper, we study deep learning for code intelligence by conducting a comprehensive survey, establishing a benchmark, as well as developing an open-source toolkit. We begin by providing a thorough literature review on deep learning for code intelligence, from the perspectives of code representations, deep learning techniques, application tasks, and public datasets. We then present an open-source toolkit for code intelligence, termed NaturalCC. On top of NaturalCC, we have benchmarked five popular application tasks about code intelligence, i.e., code summarization, code search, code completion, program synthesis, and type inference. We hope that our study contributes to a better understanding of the latest developments in code intelligence. We also hope that our toolkit and benchmark will contribute to the development of more accurate, robust, and trustworthy code intelligence models.

REFERENCES

[1] 2023. GitHub. Retrieved from https://www.github.com
[2] 2023. StackOverflow. Retrieved from https://www.stackoverflow.com
[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *NAACL*. 2655–2668.
[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *ACL*. 4998–5007.
[5] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *ICSE*.
[6] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *ICSE*. 1004–1004.
[7] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
[8] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *PLDI*. 91–105.
[9] Miltiadis Allamanis and Marc Brockschmidt. 2017. Smartpaste: Learning to adapt source code. *arXiv:1705.07867* (2017).
[10] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *ICLR*.
[11] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*. 2091–2100.
[12] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. In *ICLR*.
[13] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *ICML*. 245–256.
[14] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. x‘. *POPL* 3 (2019), 1–29.
[15] Marc Andreessen. 2011. Why software is eating the world. *Wall Street Journal* 20, 2011 (2011), C2.
[16] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. CoRR abs/2108.07732, (2021).
[17] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *ICLR*.

[18] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *ICPC*. IEEE, 253–264.
[19] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. In *IJCNLP*. 314–319.
[20] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-backed generators for program synthesis. *OOPSLA* 3 (2019), 1–27.
[21] Islam Beltagy and Chris Quirk. 2016. Improved semantic parsers for if-then statements. In *ACL*. 726–736.
[22] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *NeurIPS*. 3589–3601.
[23] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to fix coding errors with a text-to-text transformer. In *ICML*. 780–791.
[24] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv:1603.06129* (2016).
[25] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *ICML*. 896–907.
[26] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative code modeling with graphs. In *ICLR*.
[27] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *OOPSLA* 4 (2020), 1–28.
[28] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. CoRR abs/2005.14165, (2020).
[29] Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *SANER*. 95–104.
[30] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral dependency neural networks for cross-language algorithm classification. In *SANER*. 422–433.
[31] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: Learning cross-language API mappings with little knowledge. In *ESEC/FSE*. 796–806.
[32] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-supervised learning of code representations by predicting subtrees. In *ICSE*. 1186–1197.
[33] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *SIGIR*. ACM, 511–521.
[34] Ruichu Cai, Boyan Xu, Zhenjie Zhang, Xiaoyan Yang, Zijian Li, and Zhihao Liang. 2018. An encoder-decoder framework translating natural language to database queries. In *IJCAI*. 3977–3983.
[35] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *ESEC/FSE*. 964–974.
[36] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In *ICSE*. 1456–1468.
[37] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. 2019. On evaluating adversarial robustness. *arXiv:1902.06705* (2019).
[38] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2022. Quantifying memorization across neural language models. *arXiv preprint arXiv:2202.07646* (2022).
[39] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting training data from large language models. In *USENIX Security Symposium*. 2633–2650.
[40] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *S&P*. 39–57.
[41] Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2022. ERNIE-code: Beyond English-centric cross-lingual pretraining for programming languages. *arXiv preprint arXiv:2212.06742* (2022).
[42] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with meta learning. In *ICSE*. 487–498.
[43] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *TSE* (2020).
[44] Saikat Chakraborty and Baishakhi Ray. 2021. On multi-modal learning of editing source code. In *ASE*. IEEE, 443–455.
[45] Fuxiang Chen, Fatemeh H. Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages. In *ICPC*. ACM, 401–412.

[46] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR* abs/2107.03374, (2021).

[47] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *NeurIPS*. 2552–2562.

[48] Zimin Chen, Vincent J. Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. PLUR: A unifying, graph-based view of program learning, understanding, and repair. In *NeurIPS*. 23089–23101.

[49] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *TSE* (2019).

[50] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *TOSEM* 30, 3 (2021), 1–33.

[51] Davide Chicco. 2021. Siamese neural networks: An overview. *Artificial Neural Networks* (2021), 73–94.

[52] Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In *ESEC/FSE*. 703–715.

[53] Nadezhda Chirkova and Sergey Troshin. 2021. A simple approach for handling out-of-vocabulary identifiers in deep learning for source code. In *NAACL*. 278–288.

[54] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling language modeling with pathways. *CoRR* abs/2204.02311, (2022).

[55] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. PanGu-Coder: Program synthesis with function-level language modeling. *CoRR* abs/2207.11280, (2022).

[56] Adelina Ciurumelea, Sebastian Proksch, and Harald C. Gall. 2020. Suggesting comment completions for Python using neural language models. In *SANER*. 456–467.

[57] Nan Cui, Yuze Jiang, Xiaodong Gu, and Beijun Shen. 2022. Zero-shot program representation learning. In *ICPC*. ACM, 60–70.

[58] Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. 2021. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In *ICML*.

[59] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *CGO*. 86–99.

[60] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *ICML*. 1475–1485.

[61] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *TSE* (2018).

[62] Zhongyang Deng, Ling Xu, Chao Liu, Meng Yan, Zhou Xu, and Yan Lei. 2022. Fine-grained co-attentive representation learning for semantic code search. In *SANER*. 396–407.

[63] Prem Devanbu, Matthew B. Dwyer, Sebastian G. Elbaum, Michael Lowry, Kevin Moran, et al. 2020. Deep learning & software engineering: State of research and future directions. *CoRR* abs/2009.08525 (2020).

[64] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*. 4171–4186.

[65] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *ICML*. 990–998.

[66] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *ICLR*.
[67] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards learning (Dis)-similarity of source code from program contrasts. In *ACL*. 6300–6312.
[68] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).
[69] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *ACL*.
[70] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).
[71] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust physical-world attacks on deep learning visual classification. In *CVPR*. 1625–1634.
[72] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of EMNLP*. 1536–1547.
[73] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *ICLR*.
[74] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *ICLR*.
[75] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 2022. VulRepair: A T5-based automated software vulnerability repair. In *ESEC/FSE*. 935–947.
[76] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-scale multi-modal approach based on transformer for source code summarization. In *ICPC*. ACM, 24–35.
[77] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *ESEC/FSE*. 218–229.
[78] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, et al. 2018. AllenNLP: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*. 1–6.
[79] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *ICSE*. 39:1–39:13.
[80] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source code summarization with structural relative position guided transformer. In *SANER*. 13–24.
[81] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing machines. *arXiv:1410.5401* (2014).
[82] Wenchao Gu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Michael R. Lyu. 2022. Accelerating code search with deep hashing and code classification. In *ACL*. 2534–2544.
[83] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*. 933–944.
[84] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.
[85] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. In *IJCAI*. 3675–3681.
[86] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. 2022. Cross-language binary-source code matching with intermediate representations. In *SANER*.
[87] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644* (2023).
[88] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *ACL*. 7212–7225.
[89] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, et al. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *ICLR*.
[90] Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2022. Learning to complete code with sketches. In *ICLR*.
[91] Juncai Guo, Jin Liu, Yao Wan, Li Li, and Pingyi Zhou. 2022. Modeling hierarchical syntax structure with triplet position for source code summarization. In *ACL*. 486–500.
[92] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. In *NeurIPS*.
[93] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep reinforcement learning for programming language correction. *arXiv:1801.10467* (2018).

[94] R. Gupta, A. Kanade, and S. Shevade. 2019. Neural attribution for semantic bug-localization in student programs. *NeurIPS* 32 (2019).
[95] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *AAAI*.
[96] Mohammad Abdul Hadi, Imam Nur Bani Yusuf, Ferdian Thung, Kien Gia Luong, Lingxiao Jiang, Fatemeh H. Fard, and David Lo. 2022. On the effectiveness of pretrained models for API learning. In *ICPC*. ACM, 309–320.
[97] Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. In *ACL*. 8563–8568.
[98] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. 2021. Action word prediction for neural source code summarization. In *SANER*. IEEE, 330–341.
[99] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *MSR*. 300–310.
[100] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Sang Peter Chin. 2018. Learning to repair software vulnerabilities with generative adversarial networks. In *NeurIPS*. 7944–7954.
[101] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *International Conference on Computer Aided Verification*. 12–19.
[102] Hideaki Hata, Emad Shihab, and Graham Neubig. 2018. Learning to generate corrective patches using neural machine translation. *arXiv:1812.07170* (2018).
[103] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *EMNLP*. 925–930.
[104] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *ESEC/FSE*. 152–162.
[105] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *ESEC/FSE*. 163–174.
[106] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *ICSE*. 518–529.
[107] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*. 200–20010.
[108] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. (2018). In *IJCAI*, Vol. 19. 2269–2275.
[109] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. 2022. TreeCen: Building tree graph for scalable semantic code clone detection. In *ASE*. ACM, 109:1–109:12.
[110] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. In *ASE*. ACM, 79:1–79:13.
[111] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv:1909.09436* (2019).
[112] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*. 2073–2083.
[113] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *EMNLP*. 1643–1652.
[114] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *EMNLP*. 5954–5971.
[115] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *ICSE*. 38–48.
[116] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *ICSE*. 1161–1173.
[117] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*. 54–63.
[118] Dun Jin, Peiyu Liu, and Zhenfang Zhu. 2022. Automatically generating code comment using heterogeneous graph neural networks. In *SANER*. 1078–1088.
[119] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *ICML*. 5110–5121.
[120] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *ICSE*. 1073–1085.
[121] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *ICSE*. 150–162.

[122] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. In *NeurIPS*. 14967–14979.
[123] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. 75–86.
[124] Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, and Lizhen Qu. 2018. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *ICLR*.
[125] Alexander LeClair, Zachary Eberhart, and Collin McMillan. 2018. Adapting neural text classification for improved software categorization. In *ICSME*. 461–472.
[126] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC*. 184–195.
[127] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*. 795–806.
[128] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, et al. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*. 7871–7880.
[129] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Enabling programming thinking in large language models toward code generation. *arXiv preprint arXiv:2305.06599* (2023).
[130] Jia Li, Ge Li, Chongyang Tao, Huangzhao Zhang, Fang Liu, and Zhi Jin. 2023. Large language model-aware in-context learning for code generation. *arXiv preprint arXiv:2310.09748* (2023).
[131] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A retrieve-and-edit framework for source code summarization. In *ASE*. 155–166.
[132] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *IJCAI*. 4159–4165.
[133] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. AUGER: Automatically generating review comments with pre-training models. In *ESEC/FSE*. 1009–1021.
[134] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, et al. 2023. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[135] Xiaonan Li, Yeyun Gong, Yelong Shen, et al. 2022. CodeRetriever: Unimodal and bimodal contrastive learning. In *EMNLP*.
[136] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, RÃĪmi Leblond, Tom Eccles, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.
[137] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *ICLR*.
[138] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based code transformation learning for automated program repair. In *ICSE*. 602–614.
[139] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault localization with code coverage representation learning. In *ICSE*. 661–673.
[140] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE*. 292–303.
[141] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A novel deep learning-based approach for automated program repair. In *ICSE*. 511–523.
[142] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *OOPSLA* 3 (2019), 1–30.
[143] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *ICSE*. 2253–2265.
[144] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *TDSC* (2021).
[145] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *NDSS*.
[146] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *ICPC*. IEEE, 184–195.
[147] Chunyang Ling, Yanzhen Zou, and Bing Xie. 2021. Graph neural network based collaborative filtering for API usage recommendation. In *SANER*. IEEE, 36–47.
[148] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *ACL*. 599–609.
[149] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *TKDD* 15, 5 (2021), 88:1–88:21.

[150] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT prompt for code generation. *arXiv preprint arXiv:2305.08360* (2023).
[151] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent attention for if-then program synthesis. *NeurIPS* 29 (2016), 4574–4582.
[152] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez, and Dawn Song. 2016. Neural code completion. *OpenReview.net* (2016).
[153] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *ICPC*. 37–47.
[154] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *ASE*. 473–485.
[155] Fang Liu, Lu Zhang, and Zhi Jin. 2020. Modeling programs hierarchically with stack-augmented LSTM. *Journal of Systems and Software* 164 (2020), 110547.
[156] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *ICSE*. 1–12.
[157] Qianjun Liu, Shouling Ji, Changchang Liu, and Chunming Wu. 2021. A practical black-box attack on source code authorship identification classifiers. *TIFS* (2021).
[158] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid GNN. In *ICLR*.
[159] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *TKDE* (2021).
[160] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *ASE*. 585–597.
[161] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari A. Sahraoui. 2022. AST-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *ASE*.
[162] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *ACL*. 6227–6240.
[163] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*.
[164] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).
[165] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *ICML*. 649–657.
[166] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *ICSE*. 304–315.
[167] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, et al. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *ICSE*. 336–347.
[168] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2021. Modeling functional similarity in source code with graph-based Siamese networks. *TSE* (2021).
[169] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to repair compilation errors. In *ESEC/FSE*. 925–936.
[170] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *ICLR*.
[171] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical deep similarity learning-based type inference for Python. In *ICSE*. 2241–2252.
[172] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method?. In *ICSE*, Vol. 1. 880–890.
[173] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, Vol. 30.
[174] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2022. Automatic comment generation via multi-pass deliberation. In *ASE*. ACM, 14:1–14:12.
[175] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross language code clone detection using syntactical features and API documentation. In *ASE*. 1026–1037.
[176] Aravind Nair, Avijit Roy, and Karl Meinke. 2020. funcGNN: A graph neural network approach to program similarity. In *ESEM*. 1–11.
[177] Zifan Nan, Hui Guan, and Xipeng Shen. 2020. HISyn: Human learning-inspired natural language programming. In *ESEC/FSE*. 75–86.

[178] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial attacks to API recommender systems: Time to wake up and smell the coffee?. In *ASE*. 253–265.
[179] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *ICSE*. 1372–1384.
[180] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *ICSE*. 438–449.
[181] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *ICLR*.
[182] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-sequence pre-training for learning source code representations. In *ICSE*. 1–13.
[183] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *ICML*. 4861–4870.
[184] OpenAI. 2022. ChatGPT. https://openai.com/blog/chatgpt/. (2022).
[185] OpenAI. 2023. ChatGPT. https://openai.com/research/gpt-4. (2023).
[186] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *NAACL-HLT: Demonstrations*.
[187] Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2020. OptTyper: Probabilistic type inference by optimising logical and natural constraints. *arXiv:2004.00348* (2020).
[188] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *AAAI*, Vol. 35. 427–435.
[189] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to update natural language comments based on code changes. In *ACL*. 1853–1868.
[190] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could neural networks understand programs?. In *ICML*, Vol. 139. 8476–8486.
[191] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *ICLR*.
[192] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the predictions of just-in-time defect models. In *ASE*. 407–418.
[193] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *ESEC/FSE*. 209–220.
[194] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *OOPSLA* 2 (2018), 1–25.
[195] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *USENIX Security 19*. 479–496.
[196] Md. Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *ESEC/FSE*. ACM, 441–452.
[197] Md. Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. 2023. Memorization and generalization in neural code intelligence models. *Information and Software Technology* 153 (2023), 107066.
[198] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *ACL*. 1139–1149.
[199] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR* 21 (2020), 1–67.
[200] Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in neural models of source code. In *ICPR*. IEEE, 2892–2899.
[201] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic robustness of models of source code. *arXiv:2002.03043* (2020).
[202] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51, 10 (2016), 731–747.
[203] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ICPC*. 419–428.
[204] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[205] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *NeurIPS*.
[206] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *ICLR*.

[207] Michael Salib. 2004. Faster than C: Static type inference with Starkiller. *PyCon Proceedings, Washington DC* 3 (2004).
[208] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *SANER*. 311–322.
[209] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security*.
[210] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. 2021. Explanation-guided backdoor poisoning attacks against malware classifiers. In *USENIX Security*.
[211] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. 2021. API2Com: On the improvement of automatically generated code comments using API documentations. In *ICPC*. IEEE, 411–421.
[212] Rishab Sharma, Fuxiang Chen, Fatemeh H. Fard, and David Lo. 2022. An exploratory study on code attention in BERT. In *ICPC*. ACM, 437–448.
[213] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, et al. 2021. CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *EMNLP*. 4053–4062.
[214] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2022. Compressing pre-trained models of code into 3 MB. In *ASE*. ACM, 24:1–24:12.
[215] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? On the importance of data preprocessing for code summarization. In *ESEC/FSE*. ACM, 107–119.
[216] Yucen Shi, Ying Yin, Zhengkui Wang, David Lo, Tao Zhang, Xin Xia, Yuhai Zhao, and Bowen Xu. 2022. How to better utilize code graphs in semantic code search?. In *ESEC/FSE*. 722–733.
[217] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *S&P*. 3–18.
[218] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2020. On-the-fly adaptation of source code models using meta-learning. *arXiv:2003.11768* (2020).
[219] Chengxun Shu and Hongyu Zhang. 2017. Neural programming by example. In *AAAI*. 1539–1545.
[220] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-flow-based precise code embedding. *OOPSLA* 4 (2020), 1–27.
[221] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. 265–266.
[222] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code search based on context-aware code translation. In *ICSE*. 388–400.
[223] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. 2022. Heterogeneous information networks: The past, the present, and the future. *Proc. VLDB Endow.* 15, 12 (2022), 3807–3811.
[224] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *ICSE*. ACM, 1609–1620.
[225] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural CNN decoder for code generation. In *AAAI*, Vol. 33. 7055–7062.
[226] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. In *AAAI*. 8984–8991.
[227] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *ESEC/FSE*. 1433–1443.
[228] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *MSR*. 329–340.
[229] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *SIGKDD*. 2727–2735.
[230] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zheling Zhu, and Bin Luo. 2022. AST-Trans: Code summarization with efficient tree-structured attention. In *ICSE*.
[231] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: Contrastive cross-language code clone detection. In *ICPC*. ACM, 413–424.
[232] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *ICSE Workshops*. 19–20.
[233] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, et al. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *ASE*. 981–992.
[234] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *ICSE*. 25–36.
[235] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, et al. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *ASE*. 832–837.

[236] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *MSR*. 542–553.
[237] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2018. Neural program repair by jointly learning to localize and repair. In *ICLR*.
[238] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2Vec: LLVM IR based scalable program embeddings. *TACO* 17, 4 (2020), 1–27.
[239] Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Yulei Sui, Hongyu Zhang, et al. 2022. NaturalCC: An open-source toolkit for code intelligence. In *ICSE, Companion Volume*.
[240] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *ASE*. 13–25.
[241] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You see what I want you to see: Poisoning vulnerabilities in neural code search. In *ESEC/FSE*. 1233–1245.
[242] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? - A structural analysis of pre-trained language models for source code. In *ICSE*. 2377–2388.
[243] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *ASE*. 397–407.
[244] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2024. Grammar prompting for domain-specific language generation with large language models. *NeurIPS* 36 (2024).
[245] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence. In *ESEC/FSE*. 382–394.
[246] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *ICSE*. 287–298.
[247] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, et al. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *TIFS* 16 (2020), 1943–1958.
[248] Simin Wang, Liguo Huang, Jidong Ge, Tengfei Zhang, Haitao Feng, Ming Li, He Zhang, and Vincent Ng. 2020. Synergy between machine/deep learning and software engineering: How far are we? *arXiv:2008.05515* (2020).
[249] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE*. 297–308.
[250] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *SANER*. 261–271.
[251] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv:2108.04556* (2021).
[252] Yu Wang, Yu Dong, Xuesong Lu, and Aoying Zhou. 2022. GypSum: Learning hybrid representations for code summarization. In *ICPC*. ACM, 12–23.
[253] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *EMNLP*. 1069–1088.
[254] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *AAAI*, Vol. 35. 14015–14023.
[255] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*. 8696–8708.
[256] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2020. A systematic literature review on the use of deep learning in software engineering research. *arXiv:2009.06520* (2020).
[257] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *NeurIPS*. 6559–6569.
[258] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: Exemplar-based neural comment generation. In *ASE*. 349–360.
[259] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*. 3034–3040.
[260] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic type inference using graph neural networks. In *ICLR*.
[261] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. CLEAR: Contrastive learning for API recommendation. In *ICSE*. 376–387.
[262] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *SANER*. 479–490.
[263] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *ASE*. 87–98.

[264] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *MSR*. 334–345.
[265] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of ACL*. 1078–1090.
[266] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. 2022. Detecting semantic code clones by building AST-based Markov chains model. In *ASE*. ACM, 34:1–34:13.
[267] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. 2019. Detectron2. https://github.com/facebookresearch/detectron2. (2019).
[268] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software functional clone detection based on semantic tokens analysis. In *ASE*. 821–833.
[269] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *ICSE*. 2365–2376.
[270] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *ICSE*.
[271] Rui Xie, Tianxiang Hu, Wei Ye, and Shikun Zhang. 2022. Low-resources project-specific code summarization. In *ASE*. ACM, 68:1–68:12.
[272] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In *ICPC*. IEEE, 138–148.
[273] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *ACL*. 6045–6052.
[274] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *S&P*. 590–604.
[275] Guang Yang, Xiang Chen, Yanlin Zhou, and Chi Yu. 2022. DualSC: Automatic generation and summarization of shellcode via transformer and dual learning. In *SANER*. 361–372.
[276] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Comput. Surv.* 54, 10s, Article 206 (Sep. 2022), 73 pages.
[277] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A multi-modal transformer-based code summarization approach for smart contracts. In *ICPC*. IEEE, 1–12.
[278] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *ICSE*. ACM, 1482–1493.
[279] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*. 2203–2214.
[280] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*. 10799–10808.
[281] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*. 2309–2319.
[282] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *OOPSLA* 4 (2020), 1–30.
[283] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. 2018. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 268–282.
[284] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *ACL*. 440–450.
[285] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *ICSE*. 1–12.
[286] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir R. Radev. 2018. SyntaxSQL-Net: Syntax tree networks for complex and cross-domain text-to-SQL task. In *EMNLP*. 1653–1663.
[287] Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, et al. 2019. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *EMNLP*. 1962–1979.
[288] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *EMNLP*. 3911–3921.
[289] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, et al. 2019. SParC: Cross-domain semantic parsing in context. In *ACL*. 4511–4523.
[290] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *AAAI*, Vol. 34. 1169–1176.
[291] Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu, and Yulei Sui. 2021. Disentangled code representation learning for multiple programming languages. In *Findings of ACL*. 4454–4466.
[292] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for source code and natural language editing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 22:1–22:12.

[293] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE*. 1385–1397.
[294] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*. 783–794.
[295] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable program synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.
[296] Wei Emma Zhang, Quan Z. Sheng, Ahoud Alhazmi, and Chenliang Li. 2020. Adversarial attacks on deep-learning models in natural language processing: A survey. *TIST* 11, 3 (2020), 1–41.
[297] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *ESEC/FSE*. 1073–1084.
[298] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep learning code functional similarity. In *ESEC/FSE*. 141–151.
[299] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).
[300] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103* (2017).
[301] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of CodeBERT. In *ICSME*. IEEE, 425–436.
[302] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NeurIPS*. 10197–10207.
[303] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2022. Adversarial robustness of deep code comment generation. *TOSEM* 31, 4 (2022), 60:1–60:30.
[304] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An overlapping-aware code retriever. In *ASE*. 883–894.
[305] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE*. 341–353.
[306] Xiaoning Zhu, Chaofeng Sha, and Junyu Niu. 2022. A simple retrieval-based method for code comment generation. In *SANER*. 1089–1100.
[307] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *TDSC* (2019).
[308] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *TOSEM* 30, 2 (2021), 1–31.
[309] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *ICLR*.

Received 19 January 2023; revised 7 April 2024; accepted 22 April 2024