Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit

YAO WAN, Huazhong University of Science and Technology, China
YANG HE, Simon Fraser University, Canada
ZHANGQIAN BI, Huazhong University of Science and Technology, China
JIANGUO ZHANG, Salesforce Research, USA
HONGYU ZHANG, University of Newcastle, Australia
YULEI SUI, University of New South Wales, Australia
GUANDONG XU, University of Technology Sydney, Australia
HAI JIN, Huazhong University of Science and Technology, China
PHILIP S. YU, University of Illinois at Chicago, USA

Code intelligence leverages machine learning and data mining approaches to extract knowledge from large-scale 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 perspectives 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 for rapid prototyping 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 for understanding the current status 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 (https://xcodemind.github.io). At last, we also point out several challenging and promising directions for future research.

1 INTRODUCTION

Software development has been a complex and costly engineering task, which requires much human effort. To improve the software development process and developer productivity, many intelligent tools, e.g., code completion and code search, have been developed. Recently, significant progress has been made to automate various software engineering activities using machine learning techniques. As source code is the main artifact of software development, in this paper, we focus our study on *code intelligence*, which is about empowering software developers with intelligent tools through mining knowledge from large-scale code corpus.

With software becoming ubiquitous in our daily life, both open- and closed-source code repositories are growing to unprecedented sizes and complexity. For example, the platforms such as GitHub

Authors' addresses: Yao Wan, wanyao@hust.edu.cn, 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; Yang He, Simon Fraser University, Vancouver, Canada, yanghece96@gmail.com; Zhangqian Bi, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, zqbi@hust.edu.cn; Jianguo Zhang, Salesforce Research, Chicago, USA, jzhan51@uic.edu; Hongyu Zhang, University of Newcastle, NSW, Australia, hongyu.zhang@newcastle.edu.au; Yulei Sui, University of New South Wales, Australia, ysui@unsw.edu.au; Guandong Xu, University of Technology Sydney, Australia, guandong.xu@uts.edu.au; Hai Jin, hjin@hust.edu.cn, 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; Philip S. Yu, University of Illinois at Chicago, Chicago, USA, psyu@uic.edu.

and StackOverflow have collected a large corpus of source code, also termed "*Big Code*" [4]. Powered by this kind of data fuel and increasing computational power, artificial intelligence, especially deep learning can make code intelligence feasible, showing the potential to change the landscape of modern software development.

The realization of code intelligence requires synergy in the research among software engineering, machine learning, natural language processing (NLP), and programming languages. 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 many downstream tasks, such as code completion [108, 136, 181, 205], code search [69, 97, 216], code summarization [8, 94, 98, 219, 264], type inference [5, 89, 172, 234], etc.

In terms of code embedding, significant progress has been made to apply deep learning and NLP techniques to represent source code, in order to build intelligent tools to facilitate programming. For example, analogous to word2vec [152] in NLP, Alon et al. [11] proposed code2vec, a distributed representation of code, based on a collection of paths extracted from the Abstract Syntax Tree (AST) of code. Furthermore, VenkataKeerthy et al. [214] proposed IR2Vec to represent programs in the form of the LLVM-IR and capture the syntax and semantics of programs. Recently, as large pre-trained language models (e.g., BERT [54]and GPT-3 [23]) have been widely applied to NLP, many approaches [60, 74, 106] have been proposed to pre-train masked language models for source code. Feng et al. [60] pre-trained a CodeBERT model for the bimodal programming language and natural language, which has demonstrated positive results in multiple downstream tasks, such as code search and code completion. In this paper, we examine deep-learning-based code intelligence from the views of code representation learning, deep learning methods, and applications.

Related Surveys and Differences. From our literature review, there have been several related surveys to ours. Allamanis et al. [4] carried out a comprehensive review on machine learning approaches to modeling the naturalness of programming language. They mainly focus on machine learning algorithms, especially probabilistic models, rather than deep-learning-based models. Recently, Watson et al. [230], Wang et al. [223] and Yang et al. [249] conducted a thorough review of the literature on applications of deep learning in software engineering research. They investigated mostly software engineering and artificial intelligence conferences and journals, focusing on various software engineering tasks (not limited to the source code) that are based on deep learning. [53] 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 [146], the authors established a benchmark dataset called CodeXGLUE for code representation and generation. In addition, several benchmark results especially based on pre-trained language models (i.e., CodeBERT) are presented.

Different from [4] that focuses on traditional machine learning approaches, this paper puts more emphasis on deep learning techniques for code intelligence. Different from [230], [223], [249], and [53] that cover various tasks in broad software engineering, we narrow down our focus to source code related tasks from the perspective of deep learning. In addition, we survey papers from various fields including software engineering, programming languages, machine learning, NLP, and security. Note that, as code intelligence based on deep learning is an emerging and active research topic, we also include several high-quality unpublished papers that are released in arXiv. This is because these unpublished works in arXiv can be seen as an indicator of future research. Furthermore, existing surveys do not provide comprehensive benchmark evaluation results, nor do they develop an open-source toolkit to facilitate further research. In this paper, we introduce an open-source toolkit termed NaturalCC (standards for Natural Code Comprehension) [215] to ease the prototyping of code intelligence models, as well as benchmark several state-of-the-art models.

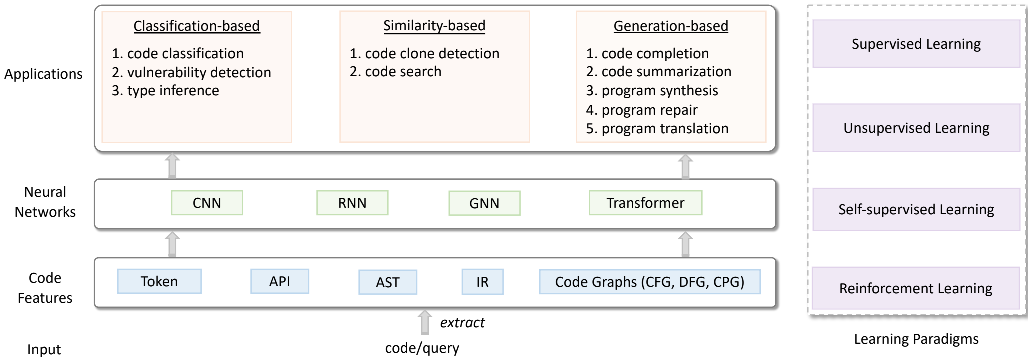


Fig. 1. Code intelligence tasks based on code representation learning.

As a complementary to CodeXGLUE [146] which intends to create a benchmark dataset for code understanding and generation especially based on pre-trained code models, we place an emphasis on developing the infrastructures for various model implementations and providing users with the ability to conduct rapid prototyping. Compared with CodeXGLUE, our toolkit contains more tools that may be used in the pipeline of building code intelligence models, with higher flexibility.

Our Contributions. This paper is for researchers and practitioners who are interested in the intersection between code intelligence and deep learning, especially in intelligent software engineering, NLP, and programming languages. In this paper, we first present a comprehensive review of the research efforts on deep learning for code intelligence. We then move a step forward to building an open-source toolkit NaturalCC for code intelligence, which implements many stat-of-the-art models over different downstream tasks. In addition, NaturalCC is well-modularized and is simple to adapt to new tasks and models. Using NaturalCC, we also benchmark the performance of each model across 4 downstream tasks, e.g., code summarization, code search, code completion, and type inference. The major contributions of this paper are summarized as follows.

- We conduct a comprehensive review on deep learning for code intelligence. Specifically, we have collected 257 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 13 leading models across four different tasks (i.e., code summarization, code search, code completion, and type inference). All the resources, datasets and source code are publicly available at http://xcodemind.github.io.
- We introduce NaturalCC, an open-source toolkit that has integrated many state-of-the-art baselines on different tasks, in order to facilitate research on code intelligence. Researchers in the fields of software engineering, natural language processing, and other fields can benefit from the toolkit for quick prototyping and replication.

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 characteristic 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 clone detection and code search), Siamese neural network structure [43] 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 triplet loss function) is always used to regularize the similarity between them. (3) *Generation-based.* In these tasks (e.g., code completion, code summarization, program translation, program synthesis, and program repair), source code, natural-language descriptions or programs written in another programming language are desired to be generated, given a 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 artificial intelligence, to security. Overall, we have identified 32 publication venues, as shown in the Supplementary Materials. We first manually check the publication list of the venues and obtain an initial collection of papers. Particularly, we search the aforementioned venue names in DBLP[1] and their corresponding content of proceedings. Two authors of this paper who have more than five-year experience in deep learning for code intelligence then work collaboratively to manually filter out those papers that may be related to code intelligence by checking the titles or quickly going through the abstract. 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.

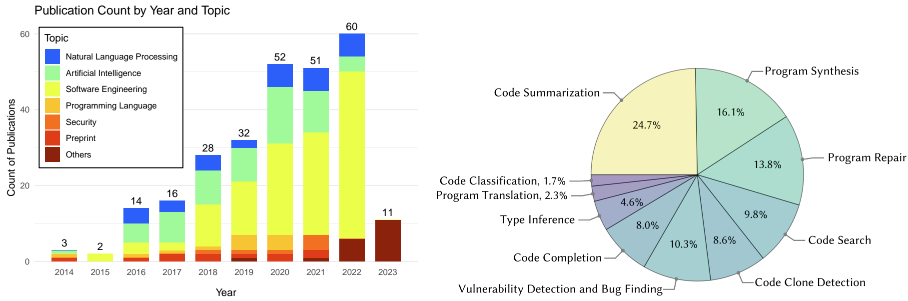
Based on this initial collection of papers, we start to augment it through keyword searching. We systematically search DBLP and Google Scholar using the following keywords: "code representation", "program comprehension", "code embedding", "code classification", "vulnerability detection", "bug finding", "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 e-Print archive, as they reflect the most current research outputs. We choose publications from arXiv based on three criteria: paper quality, author reputation, and technique innovation, which can be indicated by the number of citations. Having obtained this collection of papers, we then filter out the irrelevant papers by manual checking. We only consider full papers, while short papers are excluded. Finally, we obtained a collection of 257 papers. The complete list of studied papers can be found at https://github.com/CGCL-codes/awesome-code-intelligence.

2.3 Publication Trends of Code Intelligence

Figure 2 provides statistics of the surveyed papers to reveal the publication trend and research topic trend. Figure 2a shows the collected papers on deep learning for code intelligence, from January 2014 to December 2022. Although deep learning was first proposed in 2006 [91], it is initially used for source code modeling in 2014. From Figure 2a, we can see that the number of relevant

[1] https://dblp.uni-trier.de



(a) Number of publications in different years (b) Publication in each application

Fig. 2. Statistics of the surveyed papers to reveal the publication trend and research topic trend.

papers for code intelligence has increased significantly since 2018, indicating that deep learning has significantly advanced code intelligence research since then. This development can be attributed to the widespread use of deep learning in NLP since 2018, which has sparked a lot of studies on using NLP methods for tasks involving source code.

Figure 2b shows the distribution of papers across applications, including code classification, vulnerability detection, type inference, code search, code clone detection, code completion, code summarization, program translation, program synthesis, and program repair. This figure shows that the topics of code summarization, program synthesis, program repair, vulnerability detection, and code search, are hot research topics in recent years.

3 LITERATURE REVIEW

3.1 Taxonomy

Figure 3 illustrates the taxonomy of current studies on deep learning for code intelligence that we have surveyed in this paper. From our observation, the research in this field can be broken down into three distinct aspects: i.e., code features, deep learning techniques, and applications. (1) *Code Features.* As the foundation of deep-learning-based code intelligence, code representation seeks to represent source code as distributed vectors. We categorize the current code representation approaches by the features of input code that they use, such as code tokens, IR, APIs, ASTs and code graphs (e.g., graphs that illustrate control flow and data flow). (2) As for the deep learning techniques, we first explore the types of neural networks (i.e., RNNs, CNNs, Transformers, and GNNs), and then investigate the learning paradigms (i.e., supervised learning, unsupervised learning, self-supervised learning, and reinforcement learning) that have been used for modeling source code. (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, we need to first determine what to represent. Various work has proposed to extract code features from multiple perspectives, including code tokens, intermediate representation (IR), abstract syntax tree (AST) as well as many kinds of flow graphs. Figure 4 shows a detailed code snippet written in C, with its corresponding 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

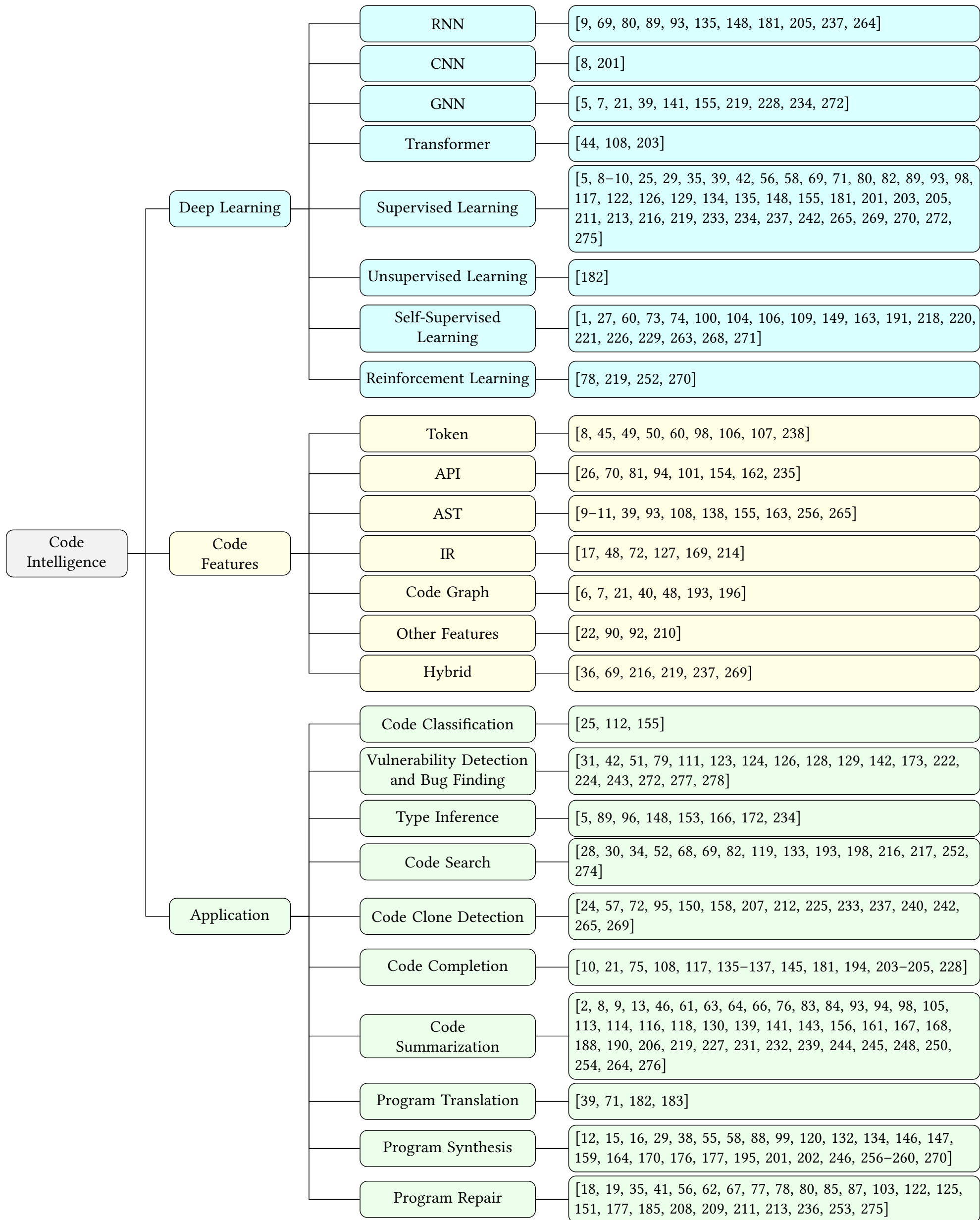


Fig. 3. The taxonomy of deep learning for code intelligence.

to represent the semantics of programs. The majority of approaches for processing code involve breaking the program down into a sequence of tokens based on specific delimiters, such as spaces or the capitalization patterns in identifiers (for identifiers like `SortList` and `intArray`). Cummins et al. [49] introduced a character-level LSTM network to represent the sequence of code characters for program synthesis. Since the set of characters to form a program is always in a limited size, the character-level code representation does not have the problem of out-of-vocabulary. However, this tokenization process at the character level breaks down the meaning of the original words and also

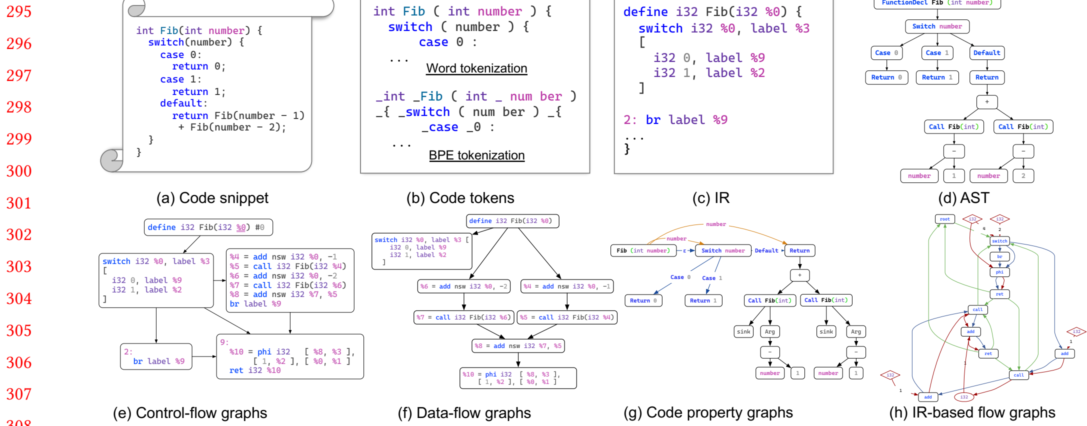


Fig. 4. A detailed C code snippet with its corresponding tokens, IR, AST, IR-based flow graphs.

increases the length of the code sequence, which can make it challenging to understand the overall semantics of the program.

More coarsely, many word-level approaches are proposed to tokenize source code into words by separators. For example, White et al. [238] and Iyer et al. [98] proposed to tokenize the program into words by whitespace, and designed RNNs to represent them for code summarization and code completion. Allamanis et al. [8] designed a CNN with an attention mechanism to better represent the hierarchical structure of code over the subtokens that are simply tokenized by Camel cases, to predict the function name.

Out-of-Vocabulary (OOV) Issue. Since the variables and function names are always defined by developers without constraints, the size of vocabulary will explosively increase with the increasing training data, resulting in the *out-of-vocabulary issue*, which is more severe than that in NLP. To mitigate this issue, Cvitkovic et al. [50] proposed a graph–structured cache, which introduces additional nodes for the encountered new words, and connects those nodes with edges based on where they occur in the code. Recently, Chirkova and Troshin [45] offered a straightforward yet effective solution to mitigate the OOV issue by using identifier anonymization, and observed promising performance improvement.

Another effective approach is to tokenize the source code at a sub-word level, such as using techniques like Byte Pair Encoding (BPE), which aims to construct a set of sub-words that can be combined to represent the entire code corpus. Figure 4 (b) shows the source tokens obtained by the strategy of word tokenization and BPE tokenization. For the input variable `number`, the word tokenization will maintain the original word and consider it as a rare word, while the BPE tokenization will split it into two common sub-words, i.e., `num` and `ber`. In the recent pre-trained language models of source code, e.g., CuBERT [106] and CodeBERT [60], BPE has commonly been adopted for reducing the vocabulary size. Karampatsis et al. [107] conducted an empirical study on the granularity of word segmentation, and showed that tokenizing code by BPE can significantly reduce the vocabulary size.

3.2.2 *Application Programming Interfaces (API).*

There have been multiple methods proposed to analyze the API sequences in programs. One line of work is about mining API usage patterns from a large code corpus to demonstrate how to use an API. For example, Moreno et al. [154] proposed a novel approach, named Muse, to demonstrate API usage by mining and ranking the code examples in usage. Another line of work is API recommendation, which aims to recommend or generate a sequence of APIs for users. Jiang et al. [101] proposed to discover relevant tutorial fragments for APIs by calculating the correlation score based on PageRank and topic relevance. Gu et al.

[70] proposed a language model named DeepAPI, under the framework of sequence-to-sequence learning, to produce API sequences in response to a given natural language description. Different from DeepAPI, Nguyen et al. [162] proposed API2Vec to represent the contextual information of API elements within an API sequence. Likewise, they also developed a tool called API2API based on API2Vec to migrate the APIs across different programming languages, i.e., from Java to C#, to validate the learned API embedding. Ling et al. [131] introduced a method that integrated API call interactions and project structure into a single graph, and used this graph to design a graph-based collaborative filtering for making API usage recommendations. Bui et al. [26] proposed a cross-language API mapping approach to map APIs from Java to C# with much less prior knowledge, through transfer learning across multiple domains. Hu et al. [94] suggested that incorporating API information as supplementary knowledge could improve code summarization. To improve the representation of semantics in natural-language queries and API sequences, Wei et al. [235] proposed a contrastive learning approach for API recommendation, and Hadi et al. [81] investigated the effectiveness of pre-trained models for generating API sequences from natural language queries.

3.2.3 *Abstract Syntax Tree (AST).*

The AST is a tree-structured intermediate representation of code that describes the syntactic structure of a program. As shown in Figure 4 (d), in an AST, the leaf nodes (e.g., `number`, `Fib`) typically correspond to the tokens of variables and method names in the source code, while the non-leaf nodes (e.g., `FuncName`, `SwitchStmt`) represent the syntactic structure of code, like function definition, branch functions. As a result, this representation allows ASTs to be useful for both capturing the lexical information (e.g., variable `number`) and the syntactic structure of the source code. In practice, we can extract ASTs using several open source tools, e.g., `tree-sitter`[2] parser, and LLVM Clang[3]. To represent the ASTs, Mou et al. [155] proposed a tree structure-based CNN, and verified it in a code classification task. In order to handle long-distance dependencies between nodes in an AST, Liu et al. [138] proposed an improved LSTM by introducing operations such as PUSH and POP, and verified it in the tasks of code completion, code classification, and code summarization. To better process an AST, Zhang et al. [265] divided an AST into sentence-based subtrees and represented them using a two-way loop network. Recently, Kim et al. [108] proposed using a relative position embedding for code completion to feed the AST to Transformers. Niu et al. [163] introduced a pre-trained model of source code by incorporating AST information.

Another line of work [9, 11, 93] is to represent ASTs indirectly by traversing or path sampling. Hu et al. [93] suggested traversing an AST to transform it into a linear series of nodes, and then using RNNs to represent the AST sequences for the task of code summarization. Alon et al. [11] performed path sampling on the ASTs, and then used word2vec to represent the semantics of a program. Furthermore, Alon et al. [9] also applied a similar idea to the task of code summarization. Similarly, Alon et al. [10] proposed a structured code language model for code completion, by sampling paths from an incomplete AST.

In program synthesis, an AST is also incorporated to guide the synthesis of programs. Yin and Neubig [256] proposed an encoder-decoder framework for code generation, in which the encoder first encodes the natural language, then the decoder generates an AST of code, and finally, the AST is converted into source code. Chen et al. [39] proposed a Tree2Tree model for program translation, which first uses a TreeLSTM to represent the source program, and another TreeLSTM to generate the target program written in another 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

[2]https://tree-sitter.github.io/tree-sitter
[3]https://clang.llvm.org

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. It is natural to enhance the code embeddings via utilizing IRs [127], with the benefit of limited vocabulary to significantly alleviate the OOV issue. In this paper, we employ LLVM-IR, which is used in the LLVM infrastructure [110], as shown in Figure 4 (c). To represent IRs, Ben-Nun et al. [17] proposed inst2vec, which first compiles a program using LLVM Clang to obtain the LLVM intermediate representation, and then adopts skip-gram to represent the instructions. VenkataKeerthy et al. [214] proposed IR2Vec, which regards the intermediate code representation as triples in knowledge graph, and then explores several knowledge graph representation methods. Cummins et al. [48] introduced ProGraML, a novel graph-based code representation based on IR. This code graph provides new opportunities to represent the semantics of source code in a low-level using machine learning techniques (e.g., GNNs), for complex downstream tasks such as program optimization and analysis. Peng et al. [169] proposed to represent the augmented IR of source code based on pre-training and contrastive learning techniques, guided by compiler optimization. Interestingly, Gui et al. [72] studied a new problem of matching binary code and source code across languages by transforming both of them into LLVM-IRs.

3.2.5 Code Graphs. Currently, many approaches have been proposed to convert programs into graphs to better represent the rich structural information within the programs , including control-flow graph (CFG), data-flow graph (DFG) and code property graph (CPG). As shown in Figure 4 (e), the CFG represents the computation and control flow of a program. In this representation, each node represents a basic block and each edge represents the transitions of control flow in the program. As shown in Figure 4 (f), the DFG is a directed graph that illustrates data relationships among various functions. Each node in the DFG has input and output data ports, and each edge links an output port to an input port on another node. To represent multiple structural information of code using a joint data structure, Yamaguchi et al. [247] 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 4 (g). In practice, we can build CFGs and DFGs using LLVM Clang, and build CPGs using Plume[4]. Recently, Cummins et al. [48] built a unified graph, termed ProGraML, which includes the CFG, DFG and call-graph, as shown in Figure 4 (h).

To represent these code graphs, Allamanis et al. [7] introduced the data flow on the top of ASTs and formed a code graph. Then, a Gated Graph Neural Network (GGNN) [121] was developed to learn the data dependencies among this code graph. Allamanis and Brockschmidt [6] built the data flow among variables and considered the contextual information of variables for the task of automated pasting in programming. Brockschmidt et al. [21] expanded the incomplete code into a graph, and then proposed a graph neural network for code completion. Sui et al. [196] made the code representation more accurate by using the value-flow graph of a program. Shi et al. [193] resorted to converting the code graphs (e.g., CFG and DFG) into sequences through traversing for the task of code search. Chen et al. [40] introduced a general method for transforming 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, there also exist several kinds of features that are used in some specific scenarios. For example, Henkel et al. [90] introduced a novel feature for code representation learning based on abstractions of traces collected from the symbolic execution of a program. Hoang et al. [92] proposed using deep learning to learn distributed representations of code changes/edits that may be used to generate software patches. In terms of code changes, several related works are also proposed to represent or predict them. Tufano et al. [210] proposed to automate code editing

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

through sequence-to-sequence-based neural machine translation. Brody et al. [22] proposed to represent the code edits first, and then iteratively generate tree edits over the AST.

3.2.7 Hybrid Representation. To leverage multiple code features, several approaches to representing source code in a hybrid fashion have been developed. For instance, Gu et al. [69] explored using three separate RNNs for representing function names, code tokens, as well as API sequences of code, respectively. It has also been evaluated in the code search task. White et al. [237] considered both the code tokens and AST node sequences, and used two different RNNs to represent these two sequences respectively, for the task of code cloning detection. Zhao and Huang [269] proposed to represent the source code by incorporating the flow graphs of code into a semantic matrix. They also developed a neural network model to assess the functional similarity between the representations of two code snippets. Similarly, Wan et al. [219] and Wan et al. [216] developed a hybrid network consisting of an LSTM representing the code tokens, a GGNN representing the CFG of code, and a TreeLSTM representing the AST of code, for the task of code summarization and code search. Chakraborty and Ray [36] suggested leveraging three modalities of information (e.g., 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 directly apply NLP techniques to represent it. Simply, RNN [9, 69, 80, 89, 93, 135, 148, 181, 205, 237, 264] and CNN [8, 201] neural networks can be easily applied to represent the sequential structure of source code. In order to capture the syntax structure, especially the AST of source code, many tree-structured neural networks [39, 155, 219] have also been designed. Furthermore, to represent the semantic structures (e.g., CFG and DFG) of source code, GNNs [5, 7, 21, 141, 228, 234, 272] have been introduced to represent the source code. Recently, the Transformer architecture has been utilized to represent the source code [108, 203]. Chirkova and Troshin [44] conducted a comprehensive empirical study of how well Transformers can leverage syntactic information in source code for various tasks. As the fundamental blocks for code representation, the neural networks will also be surveyed in Section 3.6 with respect to different code intelligence applications. More preliminaries about the mentioned neural networks are referred to the 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. It is a widely used learning paradigm in deep learning. From our investigation, current deep learning approaches for code intelligence are mainly based on supervised learning. For each specific code intelligence task, such as code classification [25, 155], vulnerability detection and bug finding [42, 126, 129, 272], code completion [10, 117, 135, 181, 203, 205], type inference [5, 89, 148, 234], code search [69, 82, 216], code clone detection [233, 237, 242, 265, 269], code summarization [8, 9, 93, 98, 219], program translation [39, 71], program synthesis [29, 58, 134, 201, 270], and program repair [35, 56, 80, 122, 211, 213, 275], a set of paired input-output data is collected first. For each task, supervised learning is guided by a specific loss function. One limitation of this kind of approach is that it relies on lots of well-labeled input-output pairs, which are always expensive to collect in some scenarios.

3.3.3 Unsupervised Learning. As opposed to supervised learning, unsupervised learning seeks to identify patterns from a dataset without labels. One representative work is TransCoder [182], in which a fully unsupervised neural source-to-source translator is trained based on unsupervised

machine translation. This kind of learning paradigm is challenging for code intelligence and more research work is still required.

3.3.4 Self-Supervised Learning. Self-supervised learning can be thought of as a blend of supervised learning and unsupervised learning. Different from supervised learning where data labels are available for training, self-supervised learning obtains the supervisory signals directly from the data itself, usually the underlying structure in the data. One common practice used by self-supervised learning is to predict any unobserved (or masked) part of input from the part that can be observed. As a representative technique of self-supervised learning, language model pre-training has been widely studied in source code [60, 74, 106]. Kanade et al. [106] proposed to train a CuBERT on the Python code corpus, and verified the pre-trained model on multiple downstream tasks such as variable misuse, operator classification, and function-document matching. CodeBERT [60] is yet another pre-trained model that deals with the two different modalities of source code and natural language descriptions. It is based on masked language modeling, and has achieved promising results in tasks such as code search and code completion. Based on CodeBERT, GraphCodeBERT [74], SPT-Code [163], and TreeBERT [104] are proposed with the aim of digesting the structural information from source code. Lachaux et al. [109] presented a pre-training objective based on deobfuscation as an alternative criterion. Inspired by BART [115] which is a pre-trained deep model especially designed towards natural language understanding and generation, Ahmad et al. [1] trained a similar pre-trained model PLBART for tasks that are related to code generation as well as code understanding. Zhang et al. [263] trained a model named CoditT5 on large amounts of source code and natural-language comments, for software-related editing tasks, e.g., comment updating, bug fixing, and automated code review. Wang et al. [226] and Guo et al. [73] proposed to train a model by unifying the modality of source code and natural language with contrastive learning, to improve the representation of the semantics of source code. Mastropaolo et al. [149] and Wang et al. [229] 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. [27] proposed InferCode, a self-supervised learning method through predicting subtrees that are identified from the context of ASTs. Jain et al. [100] proposed a contrastive learning approach for task-agnostic code representation based on program transformations in compiler.

Instead of improving the capability of code embedding, Wan et al. [218] investigated the explainability of pre-trained models for code intelligence, i.e., what kind of information do these models capture, through structural analysis. Zhang et al. [268] and Shi et al. [191] suggested compressing pre-trained models of code, as to accelerate their efficiency in practice. Zhou et al. [271] carried out an empirical study to assess the generalizability of CodeBERT when applied to various datasets and downstream tasks. Orthogonally, Wang et al. [221] and Wang et al. [220] investigated how to fine-tune pre-trained code models via curriculum learning and prompt tuning.

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 code summarization [219], code search [252], program repair [78], and program synthesis [270].

3.4 Classification-based Applications

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. [155] developed a tree-based convolutional neural network (TBCNN), which was then verified on code classification. On the broader topic of software

categorization, LeClair et al. [112] designed a set of adaptations (including word embedding and neural architectures) to adapt NLP techniques for text classification to the domain of source code. Bui et al. [25] 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 saves much effort and time for software development. Although many tools have been developed for vulnerability detection, e.g., Clang Static Analyzer[5], Coverity[6], Fortify[7], Flawfinder[8], Infer[9], and SVF [197], most of them are based on static analysis. Recently, a growing number of works employ deep learning to discover vulnerabilities. Wang et al. [224] made an early attempt at applying deep learning, specifically deep belief network, to predict the defects of software, which learns the semantic features of programs based on AST. Dam et al. [51] 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 [129], μVulDeePecker [277] and SySeVR [128] 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. [111] presented a maximal divergence sequential auto-encoder network to find vulnerabilities in binary files. The network is designed so that the embeddings of vulnerable code and invulnerable code are encouraged to be maximally divergent. Zhou et al. [272] proposed Devign for vulnerability detection, which first represents a program by fusing its AST, CFG and DFG into a unified CPG, and then designs a graph neural network to represent the CPG of code. Similarly, Wang et al. [222] and Cao et al. [31] proposed a flow-sensitive framework for vulnerability detection, which leverages a GNN to represent the control, data, and call dependencies of a program. Cheng et al. [42] introduced DeepWukong, a GNN-based model for vulnerability detection of C/C++ programs, in which the flow information of program are preserved. Liu et al. [142] introduced a GNN model with expert knowledge for detecting vulnerabilities in smart contracts, which incorporates the flow information of programs. Inspired by image processing, Wu et al. [243] 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. [124] introduced a GNN model for vulnerability detection that allows for interpretability, by providing users with parts of program dependency graph (PDG) that may contain the vulnerability. Additionally, Zou et al. [278] 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 [173] 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 the accuracy of bug detection, Li et al. [126] suggested a fusion method by exploiting both the PDG and DFG for better representation. Larger weights are assigned to the buggy paths using the attention mechanism to identify the possible vulnerability. Gupta et al. [79] developed a tree-structured CNN to identify the vulnerabilities or faults in a flawed program with respect to a failed test. Li et al. [123] defined the fault localization

[5]https://clang-analyzer.llvm.org/scan-build.html
[6]https://scan.coverity.com
[7]https://www.hpfod.com/
[8]https://dwheeler.com/flawfinder
[9]https://fbinfer.com

589 problem as image recognition, and provided a deep-learning-based approach that integrates code
590 coverage, data dependencies between statements, and source code representations.

591 *3.4.3 Type Inference.* Programming languages with dynamic typing, like Python and JavaScript,
592 allow for rapid prototyping for developers and can save the time of software development dramati-
593 cally. However, without the type information, unexpected run-time errors are prone to occur, which
594 may introduce bugs and produce low-quality code. Current works on type inference, with the aim
595 of automatically inferring variable types, mainly fall into two categories: the static-analysis-based
596 and learning-based. Traditional static-analysis approaches [86, 184] are often imprecise since the
597 behavior of programs is always over-approximated. In addition, static-analysis-based approaches
598 typically analyze the dependencies of an entire program, resulting in the relatively low efficiency.

599 Recently, many deep learning techniques have been introduced for type inference. To the best of
600 our knowledge, Hellendoorn et al. [89] was the first to employ deep learning for type inference.
601 They proposed a neural network based on sequence-to-sequence architecture, named DeepTyper,
602 which uses GRUs to represent the program context and predict the type annotations for TypeScript.
603 Furthermore, Malik et al. [148] proposed NL2Type to predict type annotations by leveraging the
604 natural-language information of programs. Based on NL2Type, Pradel et al. [172] further proposed
605 TypeWriter, which utilizes both the natural-language information and programming context (e.g.,
606 arguments usage a function). Wei et al. [234] proposed LambdaNet for type inference based on
607 GNNs, which first represents the code in the form of a type dependency graph, where typed
608 variables and logical constraints among them are preserved. Then a GNN is proposed to propagate
609 and aggregate features along related type variables, and eventually, predict the type annotations.
610 Pandi et al. [166] presented OptTyper, which first extracts relevant logical constraints, and shapes
611 type inference as an optimization problem. Allamanis et al. [5] proposed Typilus for type inference
612 in Python, which expands ASTs into a graph structure and predicts type annotations over this
613 graph using GNNs. To cope with large-scale type vocabulary, Mir et al. [153] presented Type4Py, a
614 similarity-based deep learning model with type clusters, which can support the inference of rare
615 types and user-defined classes. Recently, Huang et al. [96] formulated the type inference task as a
616 cloze-style fill-in-blank problem and then trained a CodeBERT model based on prompt tuning.
617
618
619 ### 3.5 Similarity-based Applications

620 *3.5.1 Code Search.* Code search aims to retrieve a code snippet by a natural-language query (*nl-to-*
621 *code*) or code query (*code-to-code*). The *nl-to-code* search refers to searching code fragments that
622 have similar semantics to the natural-language query from a codebase. As the first solution for code
623 search using deep learning, Gu et al. [69] proposed DeepCS, which simultaneously learns the source
624 code representation (e.g., function name, parameters and API usage) and the natural-language query
625 in a shared feature vector space, with triplet criterion as the objective function. On the basis of
626 DeepCS, Wan et al. [216] and Deng et al. [52] included more structural information of source code,
627 including the ASTs and CFGs, under a multi-modal neural network equipped with an attention
628 mechanism for better explainability. Ling et al. [133] first converted code fragments and natural-
629 language descriptions into two different graphs, and presented a matching technique for better
630 source code and natural-language description matching. Furthermore, Shi et al. [193] suggested an
631 improved code search method by converting code graphs (e.g., CFGs and PDGs) into sequences
632 through traversing. Haldar et al. [82] proposed a multi-perspective matching method to calculate the
633 similarities among source code and natural-language query from multiple perspectives. Cambronero
634 et al. [30] empirically evaluated the architectures and training techniques when applying deep
635 learning to code search. Bui et al. [28] and Li et al. [119] leveraged contrastive learning with
636 semantics-preserving code transformations for better code representation in code search.
637

Similar but different to the DeepCS framework, several more works have been proposed as complements for code search. Yao et al. [252] proposed using reinforcement learning to first generate the summary of code snippet and then use the summary for better code search. Sun et al. [198] 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. [274] 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. [34] suggested a transfer learning method for domain-specific code search, with the aim of transferring knowledge from Python to SQL. Wan et al. [217] 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. [68] 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 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. [237] proposed DLC, which comprehends semantics of source code by considering its lexical and syntactic information, and then designs RNNs for representation. To improve the representation of syntactic structure of code, Wei and Li [233] applied TreeLSTM to incorporate AST information of source code. Zhao and Huang [269] proposed encoding the CFG and DFG of code into a semantic matrix, and introduced a deep learning model to match the similar code representations. Zhang et al. [265] and Büch and Andrzejak [24] designed approaches to better represent the ASTs of the program, and applied them for code clone detection task. Furthermore, Wang et al. [225], Nair et al. [158] and Mehrotra et al. [150] 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. [242]and Hu et al. [95] 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. [240] considered the nodes of an AST as distinct states and constructed a model based on 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. [212] empirically evaluated the effectiveness of learning representation from diverse perspectives for code clone detection, including identifiers, ASTs, CFGs, and bytecode. Recently, Ding et al. [57] and Tao et al. [207] utilized program transformation techniques to augment the training data, and then applied pre-training and contrastive learning techniques for clone detection. Gui et al. [72] 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

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. [181] 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 [117, 135, 205] 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. [108] suggested predicting the missing partial code by feeding the ASTs to Transformers. Alon et al. [10] presented a structural model for code completion, which represents code by sampling paths from an incomplete AST. Furthermore, Wang and Li [228] suggested a GNN-based approach for code completion, which parses the flattened sequence of an AST into a graph, and represents it using Gated Graph Neural Networks (GGNNs) [121]. Guo et al. [75] 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. [21] expanded incomplete code into a graph representation, and then proposed a GNN for code completion. Svyatkovskiy et al. [203] 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. [136, 137] proposed a multi-task learning framework that unifies the code completion and type inference tasks into one overall framework. Lu et al. [145] 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. [204] suggested improving the efficiency of neural network model for code completion by reshaping the problem from generation to ranking the candidates from static analysis. Additionally, Shrivastava et al. [194] 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. [8] were 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 function name. Concurrently, Iyer et al. [98] proposed an LSTM-based sequence-to-sequence network with an attention mechanism for generating descriptions for source code. The sequence-to-sequence network [98] inspired a line of works for code summarization, distinguished in code representation learning. To represent the AST information, Hu et al. [93], Alon et al. [9], and LeClair et al. [114] 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. [61], LeClair et al. [113] and Jin et al. [105] investigated representing the structure of source code via a GNN, and verified it in code summarization. Guo et al. [76] designed the triplet position to model hierarchies in syntax structure of source code for better code summarization. Recently, several works [2, 66, 206, 239] proposed to improve code summarization by designing enhanced Transformers to better capture the structural information of code (i.e., ASTs). Wan et al. [219], Shi et al. [190], Yang et al. [250], Gao and Lyu [63], and Wang et al. [227] 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. [84] and Bansal et al. [13] 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. [188] 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. [46] resorted to suggesting code comment completions based on neural language modeling. Lin et al. [130] 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. [264], Wei et al. [232], Liu et al. [141] and Li et al. [116] 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. [276] 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 [94, 231, 244, 248, 254] are also worthy to be mentioned. Hu et al. [94] transferred the code API information as additional knowledge to code summarization task. Xie et al. [244] studied a new task of project-specific code summarization with limited historical code summaries via meta-transfer learning. Wei et al. [231] and Yang et al. [248] viewed the code generation task as a dual of code summarization, and incorporated dual learning for a better summary generation. Similarly, Ye et al. [254] leveraged code generation for code search and code summarization through dual learning as well. Mu et al. [156] introduced a multi-pass deliberation framework for code summarization, inspired by human cognitive processes. Xie et al. [245] proposed a multi-task learning framework by leveraging method name suggestion as an auxiliary task to improve code summarization. Haque et al. [83] emphasized that predicting the action word (always 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. [139], Panthaplackel et al. [167], and Nguyen et al. [161] trained a deep-learning-based classifier to determine whether or not the function body and function name are consistent. Panthaplackel et al. [168] and Liu et al. [143] proposed automatically updating an existing comment when the related code is modified, as revealed in the commit histories. Gao et al. [64] 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. [118] 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 AST structure of code, Chen et al. [39] 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. [71] presented DeepAM, which can extract API mappings among programming languages without the need of bilingual projects. Recently, Rozière et al. [182] proposed TransCoder, a neural program translator based on unsupervised machine translation. Furthermore, Rozière et al. [183] 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

785 inputs, current approaches resort to generating programs through machine translation. For semantic
786 parsing, Dong and Lapata [58] proposed an attention-based encoder-decoder model, which first
787 encodes input natural language into a vector representation using an RNN, and then incorporates
788 another tree-based RNN to generate programs. Liu et al. [134] proposed latent attention for the
789 If-Then program synthesis, which can effectively learn the importance of words in natural-language
790 descriptions. Beltagy and Quirk [16] modeled the generation of If-Then programs from natural-
791 language descriptions as a structure prediction problem, and investigated both neural network and
792 logistic regression models for this problem.

793 Unlike synthesizing simple If-Then programs, Yin and Neubig [256] proposed a syntax-preserving
794 model for general-purpose programming languages, which generates Python code from pseudo
795 code, powered by a grammar model that explicitly captures the compilation rules. Maddison and
796 Tarlow [147] proposed a probabilistic model based on probabilistic context-free grammars (PCFGs)
797 for capturing the structure of code for code generation. Ling et al. [132] collected two datasets (i.e.,
798 Hearthstone and Magic the Gathering) for code generation in trading card games, and proposed
799 a probabilistic neural network with multiple predictors. On the basis of [132], Rabinovich et al.
800 [176] proposed to incorporate the structural constraints on outputs into a decoder network for
801 executable code generation. Similarly, Sun et al. [201] and Sun et al. [202] designed a tree-based
802 CNN and Transformer, respectively, for code generation and semantic parsing tasks based on the
803 sequence-to-sequence framework. Hayati et al. [88] suggested using a neural code generation
804 model to retrieve action subtrees at test time.

805 Instead of synthesizing programs from natural-language descriptions, several works resort
806 to generating programs from the (pseudo) program in another format or language. Iyer et al.
807 [99] proposed to synthesize the AST derivation of source code given descriptions as well as the
808 programmatic contexts. The above approaches are driven by well-labeled training examples, while
809 Nan et al. [159] proposed a novel approach to program synthesis without using any training
810 example, inspired by how humans learn to program.

811 Recently, various pre-trained code models also achieved significant progress in code generation.
812 CodeGPT [146] is a Transformer-based model which is trained using corpus for program synthesis,
813 following the same architecture of GPT-2. CodeT5 is a pre-trained code model in eight programming
814 languages based on T5 [177], which includes an identifier-aware objective in pre-training. Xu et al.
815 [246] aimed to incorporate external knowledge during the pre-training process for code generation
816 from natural-language input. Codex [38] is a GPT model trained using a code corpus collected from
817 GitHub. It has served as the foundation of Copilot[10]. Remarkably, Li et al. [120] recently released
818 AlphaCode, a code generation system that may generate unique solutions to these challenging
819 problems requiring deeper thinking. Poesia et al. [170] introduced a constrained semantic decoding
820 mechanism into a pre-trained model, as to constrain outputs of the model in a set of valid programs.

821 Programming by example is another flourishing direction for program synthesis. Shu and Zhang
822 [195] proposed a Neural Programming By Example (NPBE) model, which learns to solve string
823 manipulation problems through inducting from input-output strings. Balog et al. [12] proposed
824 DeepCoder, which trains a model to predict possible functions useful in the program space, as to
825 guide the conventional search-based synthesizer. Devlin et al. [55] proposed RobustFill, which is
826 an end-to-end neural network for synthesising programs from input-output examples. Nye et al.
827 [164] developed a neuro-symbolic program synthesis system called SketchAdapt, which can build
828 programs from input-output samples and code descriptions by intermediate sketch. Bavishi et al.
829 [15] proposed a program candidate generator, backed by GNNs, for program synthesis in large
830 real-world API.
831

832 [10]https://github.com/features/copilot
833

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 [270], Spider [259], SparC [260], and CoSQL [258] are four datasets with human annotations for the task of text-to-SQL. Based on these datasets, many works [257, 258, 260] have been proposed. For example, Seq2SQL [270] is a neural machine translation model to generate SQL queries from natural-language descriptions with reinforcement learning. Cai et al. [29] 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. [257] 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 [102]. 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 and Singh [19] and Santos et al. [185] proposed RNN-based language models for correcting syntax errors in programs. DeepFix [80] and SequenceR [41] are two sequence-to-sequence models for syntax error correction, by translating the erroneous programs into fixed ones. Furthermore, Gupta et al. [78] improved program repair by reinforcement learning. Vasic et al. [213] proposed multi-headed pointer networks (one head each for localization and repair) for jointly localizing and repairing misused variables in code. Dinella et al. [56] presented Hoppity to jointly detect and fix bugs based on neural Turing machine [67], 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. [208] 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. [151] and Chakraborty et al. [35] proposed to model the modifications of ASTs, and designed a neural machine translation model to generate correct patches. Zhu et al. [275] 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 [253] 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. [125] 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 [18] and VulRepair [62] directly posed program repair as a text-to-text problem and utilized a model named T5 [177]. Specifically, it digests the error message and directly outputs the correct code. Jiang et al. [103] 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. [211] 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 [211] which targets at function-level small code snippets, Hata et al. [87] 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. [85] 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. [77] embedded execution traces in order to predict a sequence of edits for repairing Karel programs. Li et al. [122] 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. [236] introduced a novel approach for selecting and transforming program

Table 1. 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

repair patches using deep-learning-based code similarities. Empirically, Tian et al. [209] 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 [165], AllenNLP [65], Detectron2 [241] 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 four specific application tasks, including code summarization, code search, code completion, 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 encoders (i.e., LSTM, TreeLSTM, and Transformer) as well as a pre-training-based model.

- **Seq2Seq+Attn** [98, 219] 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 [98], we represent the source code via an LSTM network and generate the summary via another LSTM network.
- **Tree2Seq+Attn** [219] 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 over different nodes of the syntax tree of code.
- **Transformer** [2] 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** [1] is built on the top of BART [115], which is originally designed for text understanding and generation. PLBART can be seen as a specific BART model pre-trained on code corpus.

932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980

Table 2. 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

4.1.2 Results. We evaluate the performance of each model on the Python-Doc [14, 219] dataset using the BLEU, METEOR, and ROUGE metrics as in [219]. The overall performance is summarized in Table 1. This table shows that PLBART, which utilizes the Transformer architecture and pre-training techniques, achieves the highest performance. 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 [97] is an open challenge designed to assess the current state of code search. In [97], the authors have benchmarked four code search methods. The fundamental idea of [97] 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)** [97] 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)** [97] proposes to represent the semantics of source code and query via RNN models. Specially, we adopt the two-layer bidirectional LSTM network.
- **1D Convolutional Neural Network (1D-CNN)** [97] employs convolutional neural layers for code and query representation, and builds a residual connection at each layer.
- **Self-Attention (SelfAtt)** [97] adopts self-attention layers to capture the semantic information of sequential source code and query.

4.2.2 Implementation Details. For these methods, we tokenize the code snippets and natural-language descriptions by word-level BPE, and build a shared vocabulary of size $50,000$, according to the sorted token frequency. All the models are trained on a single Nvidia RTX V100 GPU with a learning rate of $5e$-4, and the gradient norm is set to 1.0. A batch size of $1,000$ is set for training acceleration. The Adam optimizer is used to optimize all the models.

4.2.3 Results. We evaluate the performance of each model on the CodeSearchNet corpus using the MRR metric, as described in [97]. The overall performance of each model is summarized in Table 2. 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.

Table 3. 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 4. 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

- **LSTM** [108] denotes the model that represents the partial code by LSTM, and then predicts the missing token via a softmax layer.
- **GPT-2** [108] 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** [108] 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. To obtain code tokens with high quality, we preprocess the code snippets by parsing them into ASTs, and collect their leaf nodes as code tokens. We build a shared vocabulary of size 50, 000, according to the sorted token frequency. All models are trained with four Nvidia RTX V100 GPUs, with the learning rate set to $1e$-3, and batch size to 32. The Adam optimizer is used to optimize all the models.

4.3.3 Results. We evaluate each model on the Py150 [180] dataset using the MRR metric as used in [108]. 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 3. 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 achieve better performance, showing that the syntax information is useful for code completion.

4.4 Type Inference

4.4.1 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** [89] proposes to represent the contextual code by a two-layer biGRU, and then predicts the missing variable types via a softmax layer.
- **Transformer** [2] proposes to represent the contextual code by a Transformer encoder network, and then predicts the missing variable types via a softmax layer.

4.4.2 Implementation Details. For these methods, we first tokenize the code snippets and natural-language descriptions, and then construct a shared vocabulary of size 40, 000, according to the sorted token frequency. The hardware for training and the optimizer is the same as above. We use a batch size of 16 and a learning rate of $1e$-4.

4.4.3 Results. We evaluate each model on the Py150 [180], by using the Accuracy metric as in [100]. In particular, we measure the performance under the settings of *all types* and *any types*. The performance of different models is summarized in Table 4. From this table, it is interesting to see

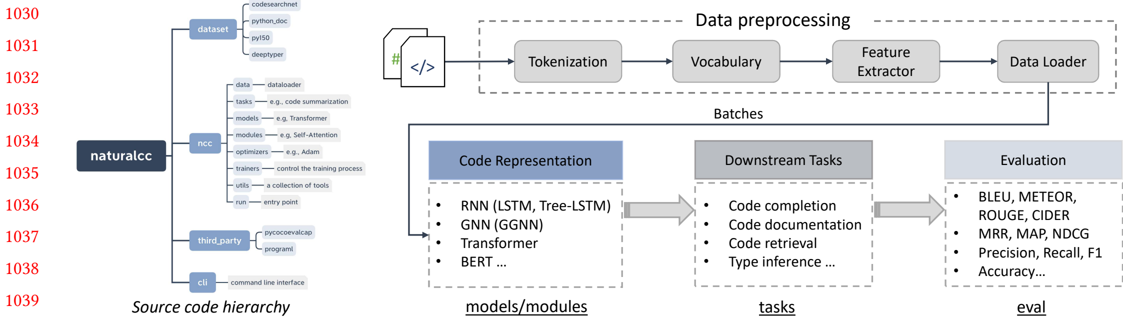


Fig. 5. The source code hierarchy and pipeline of NaturalCC.

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

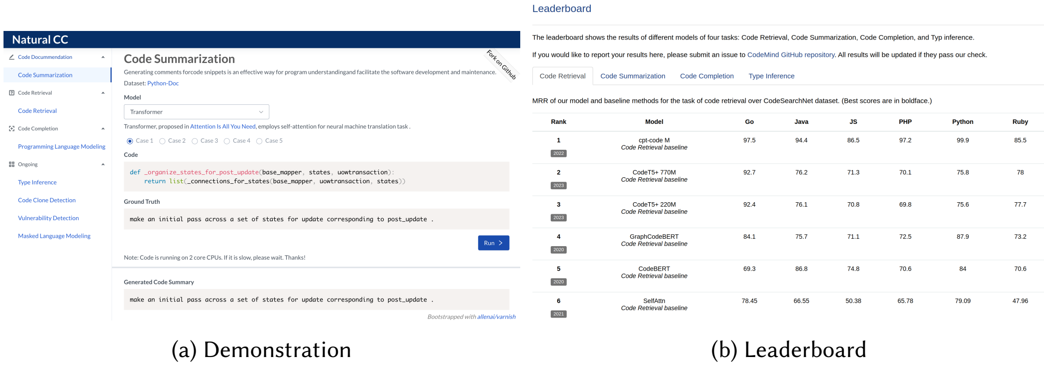
This section introduces the design of NaturalCC and its user interface. Figure 5 (left) shows the code structure of NaturalCC. The `dataset` folder contains data preprocessing code. The `ncc` folder is the core module. The `third_party` folder holds model evaluation packages. The `gui` folder contains graphical user interface files and assets. As shown in Figure 5 (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 [97], Python-Doc [219], Py150 [180], and DeepTyper [89]. 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 [107]. 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 [60], PLBART [1], and GPT-2 [146]. The `models` and `modules` folders contain all the implemented networks for code representation.



Rank	Model	Go	Java	JS	PHP	Python	Ruby
1 (2022)	cpt-code M *Code Retrieval baseline*	97.5	94.4	86.5	97.2	99.9	85.5
2 (2023)	CodeT5+ 770M *Code Retrieval baseline*	92.7	76.2	71.3	70.1	75.8	78
3 (2023)	CodeT5+ 220M *Code Retrieval baseline*	92.4	76.1	70.8	69.8	75.6	77.7
4 (2020)	GraphCodeBERT *Code Retrieval baseline*	84.1	75.7	71.1	72.5	87.9	73.2
5 (2020)	CodeBERT *Code Retrieval baseline*	69.3	86.8	74.8	70.6	84	70.6
6 (2021)	SelfAttn *Code Retrieval baseline*	78.45	66.55	50.38	65.78	79.09	47.96

(a) Demonstration (b) Leaderboard

Fig. 6. Screenshots of GUI and leaderboard of NaturalCC.

5.3 Tool Implementation

NaturalCC is mainly implemented by PyTorch, and many designs are borrowed from 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 using `argparse` for command-line options in Fairseq, we propose creating a `yaml` configuration file for each model for configuration. We believe that modifying the `yaml` configuration files is more flexible for model exploration.

5.4 Graphical User Interface

We also design a Web system as a graphical user interface to help users explore the results of trained models. The design is based on the open-source demonstration of AllenNLP [65]. Figure 6a shows the screenshot of our demonstration system. Currently, we have implemented four tasks that are related to code intelligence, i.e., code summarization, code search, and code completion. We leave the integration of other related tasks to our future work.

5.5 Leaderboard

We also develop a leaderboard so that researchers can report the results of their own models and compete with others, as shown in Figure 6b. 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.

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 Representation. 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-size vocabulary contains much "noise", making it difficult to comprehend the code. Although many attempts [45, 50, 107] have been made towards mitigating the OOV issue, it still 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 [199] to represent the heterogeneous code graph. *(c) Big Models of Code.* Despite the significant progress made by pre-trained code models in code intelligence, pre-training on a large-scale code corpus is still computationally expensive and very costly. Recently, Zhang et al. [268] and Shi et al. [191] proposed to improve the efficiency of training process by model compressing. It is a promising research direction to reduce the computational resource of pre-trained code models.

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 [34, 37, 47]. 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. [200] and Shi et al. [192] investigated the importance of data quality and verify it on the tasks of code search and code summarization, respectively.

Multi-Lingual and Cross-Language. The codebase written in multiple programming languages is 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. [262] proposed obtaining better interpretability and generalizability by disentangling the semantics of source code from multiple programming languages based on variational autoencoders. Zügner et al. [279] introduced a language-agnostic code representation based on the features directly extracted from the AST. Ahmed and Devanbu [3] conducted an exploratory study and reveal 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. [157] proposed CLCDSA, a cross-language clone detector with syntactical features and API documentation. Bui et al. [25] proposed a bilateral neural network for the task of cross-language algorithm classification. Bui et al. [26] proposed SAR, which can learn cross-language API mappings with minimal knowledge. Recently, Chai et al. [34] proposed a novel approach termed CDCS for domain-specific code search through transfer learning across programming languages. Gui et al. [72] 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. [124] 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. [278] 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. [266] proposed interpretable program synthesis that allows users to see the synthesis process and have control over the synthesizer. Pornprasit et al. [171] proposed a local rule-based model-agnostic approach, termed PyExplainer, to explain the predictions of just-in-time defect models. Rabin et al. [175] proposed a model-agnostic explainer based on program simplification, inspired by the delta debugging algorithms. Wan et al. [218], López et al. [144], and Sharma et al. [189] 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.

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 [33]. 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 [32, 33, 59] and sentiment classification [267]. 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. [179] and Yefet et al. [255] investigated how to improve the robustness of source code models through adversarial training. Nguyen et al. [160] empirically investigated the use of adversarial learning techniques for API recommendation. Bielik and Vechev [20] introduced a novel method that incorporates adversarial training and representation refinement to create

precise and robust models of source code. Zhou et al. [273], Yang et al. [251] and Zhang et al. [261] 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. [174] and Liu et al. [140] developed a novel attack against authorship attribution of source code. Ramakrishnan and Albarghouthi [178] 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. [186] and Wan et al. [217] proposed attacking the neural code models through data poisoning, and verified it in code completion and code search, respectively. Severi et al. [187] 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.

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 four popular application tasks about code intelligence, i.e., code summarization, code search, code completion, and type inference. We hope that our study contributes to a better understanding of the current status of code intelligence. We also hope that our toolkit and benchmark will contribute to the development of better code intelligence models.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *NAACL*. 2655–2668.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *ACL*. 4998–5007.

[3] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for Software Engineering. In *ICSE*.

[4] 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.

[5] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *PLDI*. 91–105.

[6] Miltiadis Allamanis and Marc Brockschmidt. 2017. Smartpaste: Learning to adapt source code. *arXiv:1705.07867* (2017).

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

[8] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*. 2091–2100.

[9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR*.

[10] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *ICML*. 245–256.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *POPL* 3 (2019), 1–29.

[12] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *ICLR*.

[13] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-Level Encoding for Neural Source Code Summarization of Subroutines. In *ICPC*. IEEE, 253–264.

[14] 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.

[15] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *OOPSLA* 3 (2019), 1–27.

[16] Islam Beltagy and Chris Quirk. 2016. Improved semantic parsers for if-then statements. In *ACL*. 726–736.

[17] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *NeurIPS*. 3589–3601.

[18] 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.

[19] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv:1603.06129* (2016).

[20] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *ICML*. 896–907.

[21] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *ICLR*.

[22] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *OOPSLA* 4 (2020), 1–28.

[23] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, et al. 2020. Language models are few-shot learners. *NeurIPS* 33 (2020), 1877–1901.

[24] Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *SANER*. 95–104.

[25] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral dependency neural networks for cross-language algorithm classification. In *SANER*. 422–433.

[26] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *ESEC/FSE*. 796–806.

[27] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *ICSE*. 1186–1197.

[28] 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.

[29] 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.

[30] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *ESEC/FSE*. 964–974.

[31] 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.

[32] 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).

[33] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *S&P*. 39–57.

[34] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-Domain Deep Code Search with Meta Learning. In *ICSE*. 487–498.

[35] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *TSE* (2020).

[36] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *ASE*. IEEE, 443–455.

[37] 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.

[38] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[39] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *NeurIPS*. 2552–2562.

[40] Zimin Chen, Vincent Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, et al. 2021. PLUR: A Unifying, Graph-Based View of Program Learning, Understanding, and Repair. *NeurIPS* 34 (2021).

[41] 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).

[42] 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.

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

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

[45] 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.

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

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

1324 [48] 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*.
1325
1326 [49] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *CGO*. 86–99.
1327 [50] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *ICML*. 1475–1485.
1328
1329 [51] 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).
1330
1331 [52] 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.
1332 [53] 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).
1333
1334 [54] 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.
1335
1336 [55] 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.
1337 [56] 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*.
1338
1339 [57] 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.
1340
1341 [58] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *ACL*.
1342 [59] 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.
1343 [60] 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.
1344
1345 [61] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *ICLR*.
1346 [62] 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.
1347 [63] Yuexiu Gao and Chen Lyu. 2022. M2TS: multi-scale multi-modal approach based on transformer for source code summarization. In *ICPC*. ACM, 24–35.
1348
1349 [64] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *ESEC/FSE*. 218–229.
1350
1351 [65] 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.
1352 [66] 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.
1353
1354 [67] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv:1410.5401* (2014).
1355 [68] 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.
1356
1357 [69] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*. 933–944.
1358 [70] 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.
1359 [71] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-Modal Sequence to Sequence Learning. In *IJCAI*. 3675–3681.
1360
1361 [72] 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*.
1362 [73] 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.
1363
1364 [74] 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*.
1365
1366 [75] Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2022. Learning to Complete Code with Sketches. In *ICLR*.
1367 [76] 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.
1368
1369 [77] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis. In *NeurIPS*.
1370
1371
1372

[78] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep reinforcement learning for programming language correction. *arXiv:1801.10467* (2018).

[79] R Gupta, A Kanade, and S Shevade. 2019. Neural attribution for semantic bug-localization in student programs. *NeurIPS* 32 (2019).

[80] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *AAAI*.

[81] 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.

[82] Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. 2020. A Multi-Perspective Architecture for Semantic Code Search. In *ACL*. 8563–8568.

[83] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. 2021. Action Word Prediction for Neural Source Code Summarization. In *SANER*. IEEE, 330–341.

[84] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *MSR*. 300–310.

[85] 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.

[86] 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.

[87] Hideaki Hata, Emad Shihab, and Graham Neubig. 2018. Learning to generate corrective patches using neural machine translation. *arXiv:1812.07170* (2018).

[88] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *EMNLP*. 925–930.

[89] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *ESEC/FSE*. 152–162.

[90] 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.

[91] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18, 7 (2006), 1527–1554.

[92] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *ICSE*. 518–529.

[93] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*. 200–20010.

[94] 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.

[95] 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.

[96] 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.

[97] 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).

[98] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*. 2073–2083.

[99] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *EMNLP*. 1643–1652.

[100] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *EMNLP*. 5954–5971.

[101] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *ICSE*. 38–48.

[102] Jiajun Jiang, Yingfei Xiong, and Xin Xia. 2019. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Sci. China Inf. Sci.* 62, 10 (2019), 200102:1–200102:16.

[103] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *ICSE*. 1161–1173.

[104] 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.

[105] Dun Jin, Peiyu Liu, and Zhenfang Zhu. 2022. Automatically Generating Code Comment Using Heterogeneous Graph Neural Networks. In *SANER*. 1078–1088.

[106] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *ICML*. 5110–5121.

[107] 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.

[108] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *ICSE*. 150–162.

[109] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *NeurIPS*. 14967–14979.

[110] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. 75–86.

[111] 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*.

[112] Alexander LeClair, Zachary Eberhart, and Collin McMillan. 2018. Adapting neural text classification for improved software categorization. In *ICSME*. 461–472.

[113] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC*. 184–195.

[114] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*. 795–806.

[115] 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.

[116] 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.

[117] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *IJCAI*. 4159–4165.

[118] 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.

[119] Xiaonan Li, Yeyun Gong, Yelong Shen, et al. 2022. CodeRetriever: Unimodal and Bimodal Contrastive Learning. In *EMNLP*.

[120] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, et al. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[121] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *ICLR*.

[122] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *ICSE*. 602–614.

[123] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *ICSE*. 661–673.

[124] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE*. 292–303.

[125] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *ICSE*. 511–523.

[126] 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.

[127] 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.

[128] 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).

[129] 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*.

[130] 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.

[131] Chunyang Ling, Yanzhen Zou, and Bing Xie. 2021. Graph Neural Network Based Collaborative Filtering for API Usage Recommendation. In *SANER*. IEEE, 36–47.

[132] 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.

[133] 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.

[134] 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.
[135] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion. (2016).
[136] 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.
[137] 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.
[138] Fang Liu, Lu Zhang, and Zhi Jin. 2020. Modeling programs hierarchically with stack-augmented LSTM. *Journal of Systems and Software* 164 (2020), 110547.
[139] 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.
[140] Qianjun Liu, Shouling Ji, Changchang Liu, and Chunming Wu. 2021. A Practical Black-box Attack on Source Code Authorship Identification Classifiers. *TIFS* (2021).
[141] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *ICLR*.
[142] 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).
[143] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *ASE*. 585–597.
[144] 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*.
[145] 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.
[146] 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*.
[147] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *ICML*. 649–657.
[148] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *ICSE*. 304–315.
[149] 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.
[150] 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).
[151] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *ESEC/FSE*. 925–936.
[152] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.
[153] 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.
[154] 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.
[155] 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.
[156] 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.
[157] 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.
[158] Aravind Nair, Avijit Roy, and Karl Meinke. 2020. funcGNN: A Graph Neural Network Approach to Program Similarity. In *ESEM*. 1–11.
[159] Zifan Nan, Hui Guan, and Xipeng Shen. 2020. HISyn: human learning-inspired natural language programming. In *ESEC/FSE*. 75–86.
[160] 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.
[161] Son Nguyen, Hung Phan, Trinh Le, and Tien N Nguyen. 2020. Suggesting natural method names to check name consistencies. In *ICSE*. 1372–1384.
[162] 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.

[163] 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.

[164] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *ICML*. 4861–4870.

[165] 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*.

[166] 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).

[167] 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.

[168] 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.

[169] 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.

[170] 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*.

[171] 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.

[172] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *ESEC/FSE*. 209–220.

[173] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *OOPSLA* 2 (2018), 1–25.

[174] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *USENIX Security 19*. 479–496.

[175] 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.

[176] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL*. 1139–1149.

[177] 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.

[178] Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in Neural Models of Source Code. In *ICPR*. IEEE, 2892–2899.

[179] 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).

[180] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51, 10 (2016), 731–747.

[181] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ICPC*. 419–428.

[182] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *NeurIPS*.

[183] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *ICLR*.

[184] Michael Salib. 2004. Faster than C: Static type inference with Starkiller. *PyCon Proceedings, Washington DC* 3 (2004).

[185] 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.

[186] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security*.

[187] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. 2021. Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers. In *USENIX Security*.

[188] 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.

[189] Rishab Sharma, Fuxiang Chen, Fatemeh H. Fard, and David Lo. 2022. An exploratory study on code attention in BERT. In *ICPC*. ACM, 437–448.

[190] 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.

[191] 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.

[192] 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.

[193] 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.

[194] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2020. On-the-Fly Adaptation of Source Code Models using Meta-Learning. *arXiv:2003.11768* (2020).

[195] Chengxun Shu and Hongyu Zhang. 2017. Neural Programming by Example. In *AAAI*. 1539–1545.

[196] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *OOPSLA* 4 (2020), 1–27.

[197] 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.

[198] 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.

[199] 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.

[200] 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.

[201] 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.

[202] 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.

[203] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *ESEC/FSE*. 1433–1443.

[204] 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.

[205] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *SIGKDD*. 2727–2735.

[206] 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*.

[207] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: contrastive cross-language code clone detection. In *ICPC*. ACM, 413–424.

[208] 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.

[209] 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.

[210] 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.

[211] 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.

[212] 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.

[213] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2018. Neural Program Repair by Jointly Learning to Localize and Repair. In *ICLR*.

[214] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *TACO* 17, 4 (2020), 1–27.

[215] 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*.

[216] 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.

[217] 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.

[218] 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.

[219] 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.

[220] 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.

[221] 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.

[222] 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.

[223] 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).

[224] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE*. 297–308.

[225] 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.

[226] 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).

[227] Yu Wang, Yu Dong, Xuesong Lu, and Aoying Zhou. 2022. GypSum: learning hybrid representations for code summarization. In *ICPC*. ACM, 12–23.

[228] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *AAAI*, Vol. 35. 14015–14023.

[229] 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.

[230] Cody Watson, Ncthan 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).

[231] 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.

[232] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *ASE*. 349–360.

[233] 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.

[234] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *ICLR*.

[235] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. CLEAR: contrastive learning for API recommendation. In *ICSE*. 376–387.

[236] 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.

[237] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *ASE*. 87–98.

[238] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *MSR*. 334–345.

[239] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Findings of ACL*. 1078–1090.

[240] 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.

[241] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. 2019. Detectron2. https://github.com/facebookresearch/detectron2.

[242] 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.

[243] 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.

[244] Rui Xie, Tianxiang Hu, Wei Ye, and Shikun Zhang. 2022. Low-Resources Project-Specific Code Summarization. In *ASE*. ACM, 68:1–68:12.

[245] 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.

[246] 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.

[247] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *S&P*. 590–604.

[248] 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.

[249] 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.
[250] 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.
[251] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *ICSE*. ACM, 1482–1493.
[252] 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.
[253] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*. 10799–10808.
[254] 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.
[255] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *OOPSLA* 4 (2020), 1–30.
[256] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*. 440–450.
[257] 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.
[258] 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.
[259] 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.
[260] 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.
[261] 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.
[262] 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.
[263] 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.
[264] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE*. 1385–1397.
[265] 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.
[266] 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.
[267] 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.
[268] 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.
[269] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *ESEC/FSE*. 141–151.
[270] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103* (2017).
[271] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing Generalizability of CodeBERT. In *ICSME*. IEEE, 425–436.
[272] 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.
[273] 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.
[274] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: an overlapping-aware code retriever. In *ASE*. 883–894.
[275] 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.
[276] Xiaoning Zhu, Chaofeng Sha, and Junyu Niu. 2022. A Simple Retrieval-based Method for Code Comment Generation. In *SANER*. 1089–1100.
[277] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *TDSC* (2019).

[278] 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.

[279] 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*.