A Survey on Code Representation

Peter D. Nagy* and Marzieh Ahmadi Najafabadi* and Heidar Davoudi*

Recently, many machine learning models have been proposed to understand and analyze Programming Languages (PLs). While there are some similarities between PLs and Natural Language Processing (NLP), the former one has its own unique challenges. In this survey, we investigate current approaches tackling *representation learning* of codes and associated downstream tasks that can be solved with them. We present and compare the state-of-the-art models specifically designed for embedding PLs in low dimensional space, and demonstrate how these embedding methods are related to representation learning approaches in NLP. We also compare benchmark experiments on multiple code-related tasks and evaluate the models for each specific application.

Keywords: Code Embedding, Code Representation, Transformers, Natural Language Embedding, Programming Languages

1. Introduction

Over the past decade, there has been a significant advancement in Natural Language Processing (NLP) approaches analyzing and comprehending text. Many of these remarkable achievements are due to the power of deep learning techniques applied to various tasks in the NLP domain. In particular, recent pre-trained machine learning models such as BERT[1], BART[2], RoBERTa[3], T5[4], GPT[5], ELMo[6] and XLNet[7] made a huge improvement in the performance of a variety of natural language processing tasks. These pre-trained models learn important contextual representations from different unlabeled texts by optimizing self-supervised objectives (e.g., masked language modeling).

Programming languages (PLs) are different from natural languages (NLs) as they conform to a specific syntax and have rich structural information. Therefore, adapting the existing deep pre-trained NLP models to PLs requires the consideration of the code's structure and relationships between program entities. Moreover, we can build pre-trained models by utilizing both NL and PL pairs (i.e., bimodal data). That is, programming codes and their comments, documentations, and specifications are fed into the model to capture the semantic representations of codes and descriptions together. This paper reviews the current state-of-the-art deep learning models designed to capture the semantic representation of codes in low-dimensional data.

In this survey, we conduct a thorough review of the current state-of-the-art models for code representation and compare them on benchmark downstream tasks. We

*Faculty of Science, Ontario Tech University, Canada {peter.nagy, marzieh.ahmadinajafabadi, heidar.davoudi}@ontariotechu.ca

also present available codes and datasets in the PL field. The paper is organized as follows. In Section 2, we present a short background and preliminaries. In Section 3, we explain the current state-of-the-art pre-trained machine learning models on PL while dividing them into two categories: Transformer type models and other non-transformer architectural models. We also provide available GitHub and HuggingFace links for the presented models. In Section 4, we provide descriptions of the most important and common datasets in the PL domain. In Section 5, we explain different PL downstream tasks along with the performance metrics used to evaluate each task. Furthermore, we compare evaluation metrics reported by previous works on different models and different datasets. Finally, we draw our conclusion in Section 6. Figure 1 shows the structure of this paper.

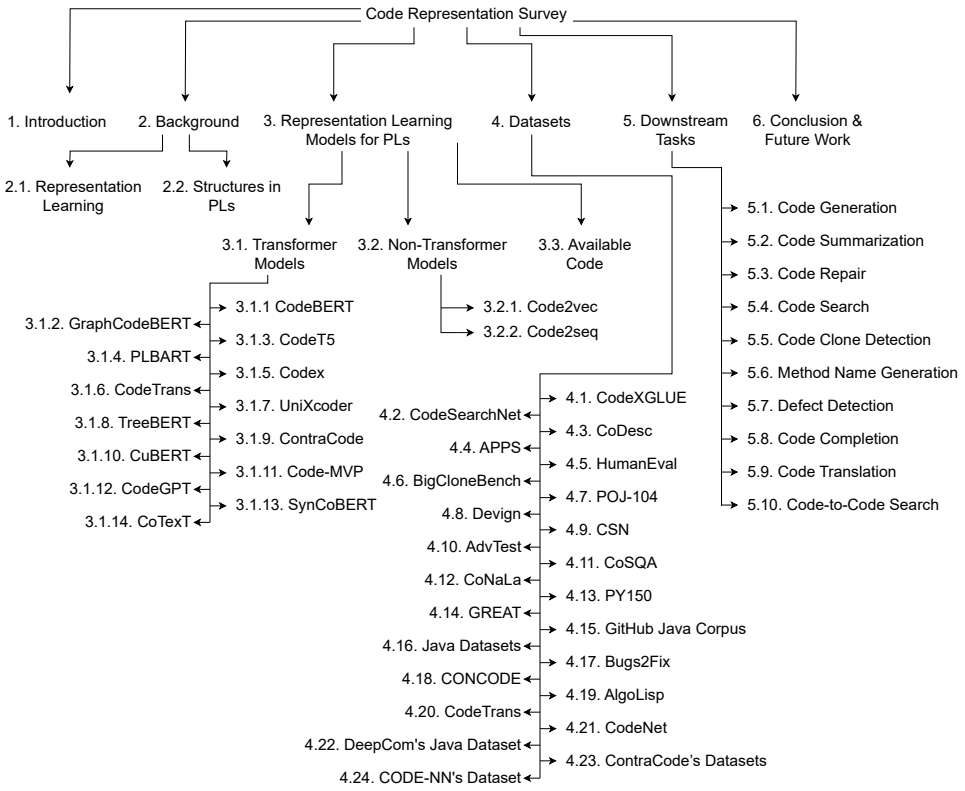

Fig. 1: The structure diagram of the paper.

2. Background

2.1. *Representation Learning*

There has been a growing interest in the field of *representation learning* over the past decade[8–10]. In representation learning, the goal is to capture the semantic representation of instances (e.g., text documents, words, images) in a low-dimensional vector space. That is, each instance is represented by a vector representing its rich set of features. The information captured in the representation vectors (i.e., embedding) can be used in many downstream tasks, such as similarity search[11], classification[12], and clustering[13] and it reduces/eliminates the need for expert knowledge in feature engineering.

For example, in the NLP domain, *context-free* models such as word2vec[9] and GloVe[14] learn word representations without considering the context of words (e.g., the sentence in which they are appearing). However, many words have different meanings depending on the context they are used (e.g., "Apple" for instance, may be the name of a company or a fruit depending on the context). A *context-based* model, such as BERT[1], learns representations based on the context in which words appear. Therefore, BERT is more powerful in learning semantically meaningful representations that capture context-dependent meanings of words.

Most often, models such as BERT are optimized by *supervised-training*. That is, randomly selected tokens are masked from the input sentence before going through the encoder. The encoder learns the embedding vectors for each word/token in the sentence in order to predict the most likely word/token for each masked word/token. This process does not require manually annotated data and can take advantage of large unlabelled training data which is available in many domains.

2.2. *Structures in Programming Languages*

In order to understand the semantics of the code, we need to learn about the basic structures in the code and the way of extracting them. This is important as multiple methods[15–21] have leveraged the code's structural representation in order to incorporate the structure of the code containing crucial code semantics into the embedding space. A lexical analyzer or tokenizer is used to convert the code to a token-based sequence in which the order of tokens follows the order of their appearance in the code. Then, some models opt to utilize a parser, which is also known as a syntax analyzer, that produces an abstract syntax tree (AST) from the token-based sequence based on the grammar rules. Several studies have focused on implementing frameworks leveraging AST representations of code[22–24]. The root node in this tree is the start symbol of the grammar, the interior nodes are the non-terminals in the grammar, and the leaf nodes are the terminals, which are code tokens such as programming language specific keywords, variables, and identifiers defined by the programmer. Finally, a semantic analyzer can utilize the AST representation to generate flow graphs that contain the semantic information of the source code. There are two common flow graphs:

- Control Flow Graph (CFG): this graph illustrates different possible execution paths of a program.
- Data Flow Graph (DFG): this graph presents the relationships between the segments of code where a variable is present. It is used to describe the data dependency relation between variables.

Furthermore, since DFGs are only capable of representing basic blocks without branches (blocks without any conditions in other words), they can be replaced by the basic blocks of a CFG resulting in a control/data flow graph. Utilizing the following piece of code snippet in Python (Figure 2), we constructed the AST (Figure 3) as well as CFG (Figure 4.a) and DFGs (Figure 4.b and (Figure 4.c) to illustrate their definitions.

```
def subtract(a,b):
    x = 0
    if(a > b):
        x = a - b
    else:
        x = a + b
    return x
result = subtract(2,1)
print(result)
```

Fig. 2: A Python code snippet example.

3. Representation Learning Models for Programming Languages

Table 1 presents a list of the newest and most influential pre-trained models in the PL domain reviewed by this paper. The list of models was selected by accounting for the *date of publication* (recent is preferred), *the number of citations* (can be found in Figure 5), and the venue in which they were published. Some of the state-of-the-art models that were published most recently have amassed few citations, which is expected. We present the NL model that served as inspiration, the architecture (i.e., encoder-only, decoder-only, and encoder-decoder), the pre-training tasks or objective functions used to train the model, the parameter size of the model, the corpus used to train, and the programming languages that the model has seen during training.

An **encoder-only** model encodes an input sequence, in our case, a set of code tokens, into internal state vectors, which can be used for program classification or regression tasks. Whereas a **decoder-only** model predicts the next tokens given some previous tokens as context which is mainly used for code completion tasks or

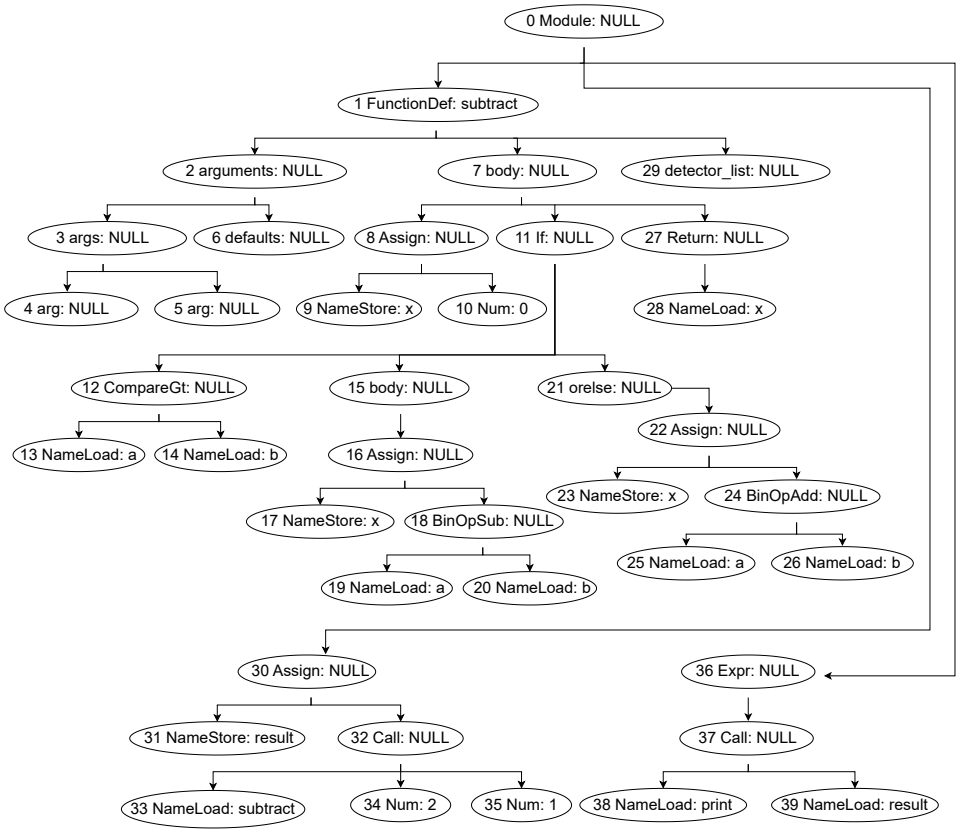

Fig. 3: An example of Abstract Syntax Tree.

code generation tasks. Finally, an **encoder-decoder** model will encode an input sequence and then generate a new output sequence not necessarily as a continuation of the input sequence, but as an original token sequence such as in code summarization or code translation tasks.

We mainly focus on deep-learning models developed for code-understanding tasks. Some models have been developed with networks such as Recurrent Neural Networks (RNNs)[34], more specifically, Long Short-Term Memory (LSTM)[35], bidirectional LSTMs (BiLSTMs), or Graph Neural Networks (GNNs). Bidirectional LSTMs allow the input to flow in both directions through the neural network. GNNs can receive graph representation inputs such as ASTs, CFGs, or DFGs and learn directly from the graphs instead of learning from a sequence representation of the graphs.

3.1. *Transformer Models*

Most recent state-of-the-art models in the field are Transformer[36] based models constructed using the *attention mechanism* connecting the encoder to the decoder and assigning attention weights, which inform the model about the input tokens that are more influential. The encoder associates the input tokens with each other while learning their representation through a self-attention mechanism. This self-attention

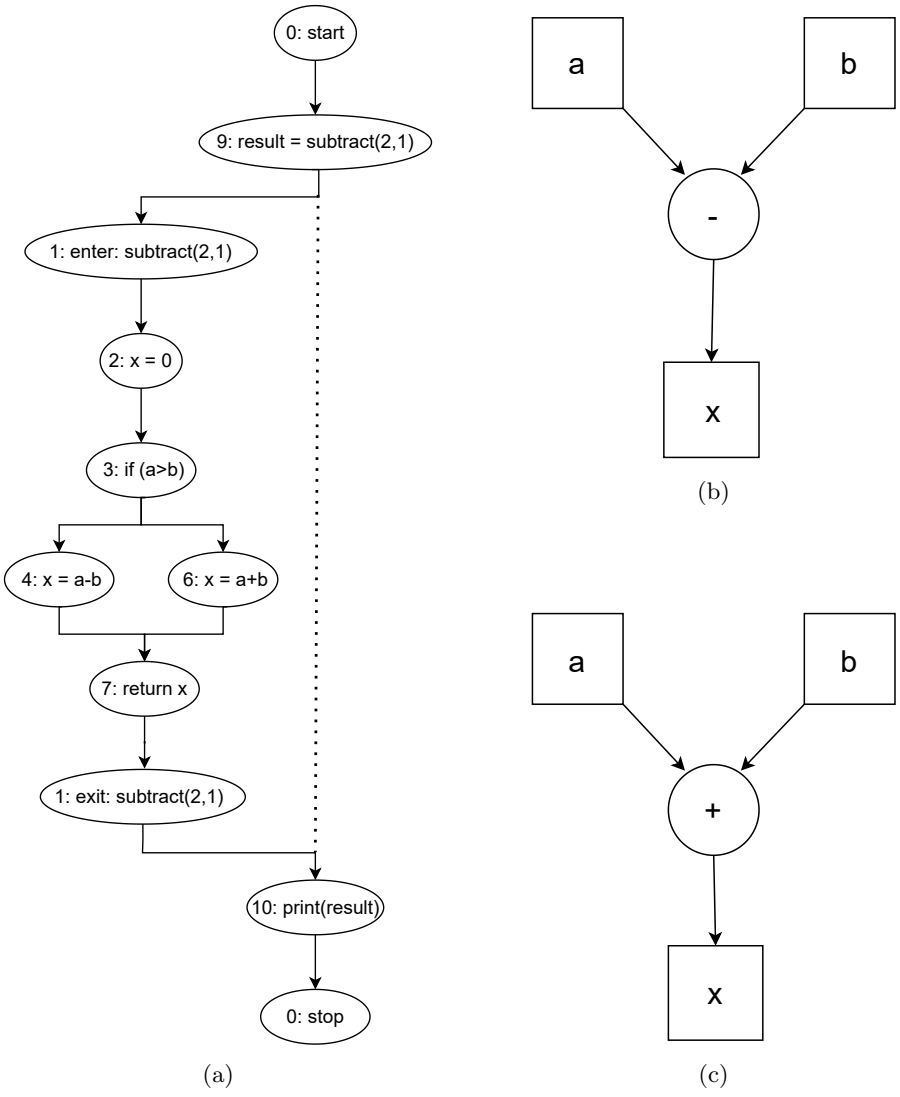

Fig. 4: An example of Flow Graphs. (a) Control Flow Graph, (b) Data Flow Graph of expression x = a - b (c) Data Flow Graph of expression x = a + b

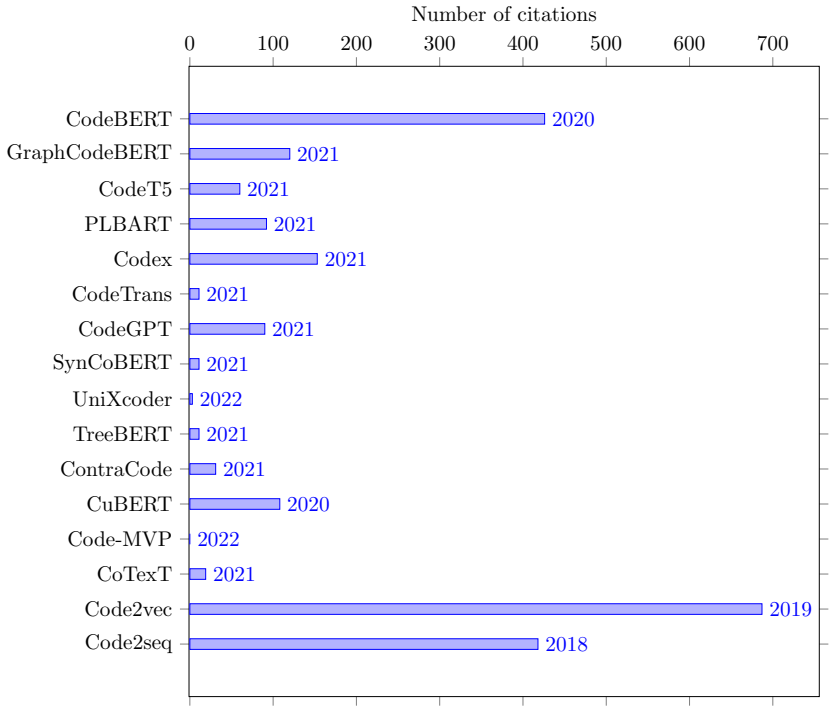

Fig. 5: Number of citations (by July 2022) for models.

mechanism is also used by the decoder when receiving the encoder's output. These models in the PL field were mostly built upon current well-performing transformer-based models in the NL field such as BERT[1], BART[2], GPT[5], and T5[4].

3.1.1. *CodeBERT*

CodeBERT[25] is a pre-trained programming language model utilizing an encoder-only architecture identical to that of the RoBERTa[3] model in the natural language field. CodeBERT is the first NL-PL large language model pre-trained on multiple programming languages. The RoBERTa model builds on top of the BERT[1] model with an optimized training procedure and uses more data to train the model even further to achieve better results on downstream tasks. The CodeBERT model was pre-trained using the Masked-Language Modeling (MLM) objective, which consists of predicting randomly masked code tokens, as well as the Replaced Token Detection (RTD) objective, which consists of predicting whether a token at a specific location appears in the original program or it has been replaced. Both of these objective tasks were originally developed for natural language models and then adapted to programming language models. The adaptations mainly involve addressing both bimodal (the input is NL and PL pairs) and unimodal (the input is PL) types of

Name	NL Derivative	Architecture	Pre-training Tasks	Parameter Size	Corpus	Programming Languages
CodeBERT (Feng *et al.*, 2020)	BERT	Encoder-only	MLM, RTD	125M	CodeSearchNet	Go, Java, JavaScript, PHP, Python, Ruby
GraphCodeBERT (Guo *et al.*, 2021)	BERT	Encoder-only	MLM, EP, NA	125M	CodeSearchNet	Go, Java, JavaScript, PHP, Python, Ruby
CodeT5 (Wang *et al.*, 2021)	T5	Encoder-Decoder	MSP, IT, MIP, BDG	60M, 220M (small, base)	CodeSearchNet, BigQuery's GitHub Dataset	Go, Java, JavaScript, PHP, Python, Ruby, C, C#
PLBART (Ahmad *et al.*, 2021)	BART	Encoder-Decoder	DA	140M	BigQuery's GitHub Dataset, StackOverflow (September 2020)	Java, Python, (Go, JavaScript, PHP, Ruby, C, C++, C#)*
Codex (Chen *et al.*, 2021)	GPT	Decoder-only	Minimizing negative log-likelihood between reference code and generated code	12M, 25M, 42M, 85M, 300M, 679M, 2.5B, 12B	GitHub (May 2020)	Python
CodeTrans (Elnaggar *et al.*, 2021)	T5	Encoder-Decoder	Span Masking	60M, 220M, 770M (small, base, large)	CodeSearchNet, The Public Git Archive, 150k Python Dataset, StaQC, LISP	Go, Java, JavaScript, PHP, Python, Ruby, SQL, LISP, C#
CodeGPT (Lu *et al.*, 2021)	GPT	Decoder-only	Next token prediction	124M	CodeSearchNet	Java, Python
SynCoBERT (Wang *et al.*, 2021)	BERT	Encoder-only	MMLM, IP, TEP, MCL	125M	CodeSearchNet	Go, Java, JavaScript, PHP, Python, Ruby
UniXcoder (Guo *et al.*, 2022)	-	Encoder-only, Decoder-only, Encoder-Decoder	MLM, ULM, DNS, MCL, CMG	125M	CodeSearchNet, C4	Go, Java, JavaScript, PHP, Python, Ruby
TreeBERT (Jiang *et al.*, 2021)	BERT	Encoder-Decoder	TMLM, NOP	-	Python and Java corpus published by CuBERT	Java, Python
ContraCode (Jain *et al.*, 2021)	-	Encoder-Decoder	InfoNCE	18M, 23M (LSTM, Transformer)	CodeSearchNet	JavaScript
CuBERT (Kanade *et al.*, 2020)	BERT	Encoder-only	MLM, NSP	340M	BigQuery's GitHub Dataset	Python
Code-MVP (Wang *et al.*, 2022)	-	Encoder-only	MMLM, MVCL, FGTI	125M	CodeSearchNet	Python
CoTexT (Phan *et al.*, 2021)	T5	Encoder-Decoder	Self-supervised masked span prediction	220M	CodeSearchNet, BigQuery's GitHub Dataset	Go, Java, JavaScript, PHP, Python, Ruby
Code2vec (Alon *et al.*, 2019)	Doc2vec	Encoder-only	Predicting a probability distribution of assigned tags to code snippets	-	GitHub Dataset	Java
Code2seq (Alon *et al.*, 2018)	Seq2seq	Encoder-Decoder	Next token prediction	37M	Java-Small, Java-Med, Java-Large, C# Dataset	Java, C#

Table 1: Representation Learning Models for Programming Languages. * refers to not explicitly trained with the language but used for evaluation.

data. The model has a total of 125M trained parameters and it was trained on the CodeSearchNet[37] dataset with the six available programming languages including Go, Java, JavaScript, PHP, Python, Ruby.

3.1.2. *GraphCodeBERT*

GraphCodeBERT[15] introduces data flow into the architecture with a graph-guided masked attention function. GraphCodeBERT improves on CodeBERT in programming language understanding and code representation by leveraging the code's structure with Data Flow Graphs (DFGs). The code's structure is incorporated

into the model using a graph-guided masked attention function. In addition to the MLM objective, it introduces two new objectives for pre-training in a structure-aware manner. The first one consists of predicting data flow edges between variable nodes which they refer to as Edge Prediction (EP), and the second involves predicting edges between variable nodes and source code tokens, which is referred to as Node Alignment (NA). This encoder-only model follows the BERT architecture using a multi-layer bidirectional Transformer design. They also utilize code comments in their pre-training data, which most pre-trained models omit as it can be misleading and does not affect the execution of the program. This model contains 125M trainable parameters and was trained on the CodeSearchNet dataset using the six given programming languages.

3.1.3. *CodeT5*

CodeT5[26] is an encoder-decoder pre-trained model, which is able to perform both code understanding tasks and code generation tasks. This work improves on the predecessor methods by introducing an encoder-decoder pre-training that is more optimal for generation tasks and takes into account specific characteristics and syntax of programming languages. The architecture of this model is based on the T5[4] NLP model. They introduce new pre-training tasks including Masked Span Prediction (MSP), which masks arbitrary length spans of text and then attempts to predict them. They also make use of the Identifier Tagging (IT) task, where the goal is to predict if a certain code token is an identifier. Furthermore, using the Masked Identifier Prediction (MIP) task, the model learns to predict the missing identifier code tokens from the masked source code. Finally, they employ the Bimodal Dual Generation (BDG) task, which consists of generating NL or PL from an NL-PL pair. This can also be viewed as an MSP task, where the NL or the PL is masked in one single span and the goal is to predict the span. The CodeT5 model was built with two sizes: a small version consisting of 60M parameters and a base version consisting of 220M parameters. It was pre-trained on two datasets: CodeSearchNet and the GitHub fraction of Google BigQuery dataset [a]. In addition to the six programming languages presented in the CodeSearchNet dataset, the model is also trained on C and C# programming languages.

3.1.4. *PLBART*

PLBART[27] is another encoder-decoder pre-trained model originating from the BART[2] model's architecture well established in the NLP field. PLBART develops a general-purpose model focused on tackling program and language understanding and generation tasks. This model is pre-trained with a single objective named Denoising Autoencoding (DA). The objective consists of reconstructing an input text

[a]https://console.cloud.google.com/marketplace/details/github/github-repos

affected by a noise function. The input is modified with noise by masking certain tokens (similar to MLM), by deleting certain tokens, or by masking out spans of tokens (similar to MSP). The PLBART model has 140M parameters and was trained on the Java and Python written GitHub fraction of the Google BigQuery dataset and they used data from StackOverflow to extract NL questions and answers to programming problems. They also evaluated their model on seven additional programming languages (i.e., Go, JavaScript, PHP, Ruby, C, C++, C#), which were not presented to the model during its pre-training phase. The PLBART model performed surprisingly well, even outperforming other models that were explicitly pre-trained on those languages on several tasks.

3.1.5. *Codex*

Codex[28] introduces a decoder-only model that generates code solutions based on a given natural language problem denoted as the context. Codex establishes a GPT style model trained solely on programming language data. This model's architecture was inspired by the GPT-3[38] model family and showed performance improvements on code-related tasks over GPT models, which were partially trained on code. They train this model with the objective of minimizing the negative log-likelihood between the reference code and the generated code. They publish this model in multiple sizes ranging from 12M to 12B parameters. The model is trained on a large dataset (179 GB) of repositories from GitHub using only the Python programming language. This model proves capable of generating solutions to a wide variety of introductory difficulty problems, however, it falls short when tasked with harder problems. It is presumed that the Codex model under-performs a strong student having completed an introductory computer science course.

3.1.6. *CodeTrans*

CodeTrans[29] is an encoder-decoder transformer pre-trained model which follows the T5[4] framework. This work proposes a model focused on tackling six software engineering tasks featuring numerous programming languages. To pre-train their model, they used span masking (similar to MSP) where the spans were composed of three tokens. They release three sizes of their model including a base (220M parameters), a small (60M parameters), and a large (770M parameters) model. The model was trained on a multitude of datasets including CodeSearchNet, the Public Git Archive, the 150k Python dataset, StaQC, and LISP. Hence, it was trained on the six CodeSearchNet programming languages as well as SQL, LISP, and C#. CodeTrans was assessed on Single-Task Learning, Multi-Task Learning, and Transfer Learning where they pre-trained their model in a self-supervised manner before fine-tuning it on downstream tasks. They achieved mixed results between their model learning variations over different tasks.

3.1.7. *UniXcoder*

UniXcoder[17] proposes an encoder-only, a decoder-only, and an encoder-decoder framework for a range of applicable tasks. They take advantage of code comments and abstract syntax trees (ASTs) to enrich their model input. While code comments could be helpful to translate the code's purpose, abstract syntax trees provide the structure of the code, which they then encode into a sequence structure via a one-to-one mapping. They utilize multiple pre-training tasks including Masked Language Modeling (MLM) in the encoder-only mode, where the task is to predict the masked tokens, Unidirectional Language Modeling (ULM) for the decoder-only mode, where the task consists of predicting the next token conditioned on previous tokens. They further employ a DeNoiSing (DNS) objective in an encoder-decoder mode, which consists of predicting random masked spans (similar to MSP). The model also learns semantic embeddings from multi-modal code using Multi-modal Contrastive Learning (MCL), and Cross-Modal Generation (CMG). The former employs a contrastive learning approach based on SimCSE[39] framework and the latter consists of generating code comments. They also mention that while AST representations were crucial in pre-training the model, they are not required when fine-tuning the model to specific downstream tasks. The model includes 125M trainable parameters and follows a transformer-based model architecture. The different behavior modes (encoder-only, decoder-only, encoder-decoder) of the model are enabled using an input prefix ([Enc], [Dec], [E2D]). The UniXcoder model was trained on the CodeSearchNet dataset and on the C4 dataset[40] with the six programming languages available in the datasets (i.e., Go, Java, JavaScript, PHP, Python, Ruby).

3.1.8. *TreeBERT*

TreeBERT[18] provides a tree-based architectural pre-trained encoder-decoder model. They develop a model that incorporates tree structure from abstract syntax trees (ASTs) to improve programming language-oriented generation tasks, outperforming other models in code summarization and code documentation while demonstrating good transferability to unseen programming languages. They pre-train using Tree Masked Language Modeling (TMLM), where they mask the nodes/tokens from the tree/code in a novel way that ensures a diversity in the types of nodes masked as opposed to a standard MLM strategy. The model is also trained using Node Order Prediction (NOP) to learn the syntactical structure of code, where the task consists of exchanging the positions of nodes along paths in the tree and making the model predict whether the nodes are in order or out of order. They employ the Python and Java dataset that was used in training of the CuBERT[32] model to also train the TreeBERT model.

3.1.9. *ContraCode*

ContraCode[31] provides an encoder-decoder pre-trained model along with a contrastive pre-training task, which helps the model learn the code functionality in order to be able to recognize two syntactically different programs with similar objectives/purposes. They enhance the capture of program functionality in source code and improve accuracy in natural code tasks such as code summarization and TypeScript type inference. They use InfoNCE as the pre-training objective, which classifies positive pairs over negative pairs in a contrastive learning setting. They utilize their framework to train an LSTM[35] model (specifically a Bidirectional LSTM), which uses 18M parameters as well as a Transformer model, which uses 23M parameters. This model is trained on the CodeSearchNet dataset with simply the JavaScript language.

3.1.10. *CuBERT*

CuBERT[32] proposes an encoder-only pre-trained model developed from BERT[1]. This work fills the gap in high-quality contextual embeddings for source code and evaluation on program-understanding tasks. The model was trained using similar tasks used to train the BERT model, however, slightly modified to source code. They use the Masked Language Model (MLM) task to predict masked tokens and the Next-Sentence Prediction (NSP) task to predict whether two logical code lines separated by a separator token ([SEP]) follow each other. They trained the model using 340M parameters on the Github portion of Google's BigQuery dataset on solely Python written programs. Unlike most pre-trained models, this model was provided with Python tokenized code with the help of the Python tokenizer library (`tokenize`).

3.1.11. *Code-MVP*

Code-MVP[19] presents an encoder-only pre-trained model, which integrates multiple representations of a program in the model input at the same time with a contrastive learning framework. The motivation of this work is to overcome the limitations of existing code representation learning approaches, achieving superior performance in various code-related tasks compared to state-of-the-art baselines. They extract the natural language (NL) description/comment, the programming language (PL) source code, the abstract syntax tree (AST) representation, the control flow graph (CFG) representation, and the program transformation (PT) variant from a program and use those views as the input to the model. They utilize various functionally-invariant PT techniques to help the model understand functional semantics and also serve as a data augmentation. The model is trained on Multi-View Masked Language Modeling (MMLM), which predicts masked-out tokens from data points. The model is also trained on Multi-View Contrastive Learning (MVCL), which functions differently depending on a single-view or a dual-view approach.

In both cases, they perform contrastive learning which requires positive and negative samples. In a single-view approach, positive samples consist of paired identical programs with different views and negative samples consist of different programs with different views. In the dual-view scenario, the setup is similar, however, they prepend the program pairs with their corresponding NL representation. Finally, the model is trained using the Fine-Grained Type Inference (FGTI) task, which consists of predicting the type information of code tokens. They use a model containing 125M trainable parameters and they train on the CodeSearchNet dataset solely on Python code.

3.1.12. *CodeGPT*

CodeGPT[30] follows identical architecture and objective function as GPT-2[41] and presents a decoder-only pre-trained model with 124M trained parameters. This work provides a platform for evaluating and comparing methods in various programming language tasks. CodeGPT was trained on the CodeSearchNet dataset with the languages Java and Python separately on two different models. The model was trained with the objective of predicting the next token based on a given context. Two extra model variants exist for each programming language. One is trained from scratch with random model weights and a new vocabulary. The other is trained from the GPT-2 weights and vocabulary as starting points.

3.1.13. *SynCoBERT*

SynCoBERT[16] provides a pre-trained encoder-only model derived from BERT. They enhance code representation learning by exploring the properties of programming languages, introducing novel pre-training objectives, and leveraging multi-modal contrastive learning for improved code intelligence tasks. The parameters and initializations are adopted from CodeBERT and GraphCodeBERT. The pre-training dataset and learned programming languages are also identical to that of CodeBERT. SynCoBERT, however, introduces a contrastive pre-training approach, coupled with AST representations, and two novel pre-training tasks. First, the model is trained on Multi-Modal Masked Language Modeling (MMLM), similar to Code-MVP, although, in this case, the modalities used are NL, PL, and AST. They form triplets from these modalities for each data point and predict masked tokens from the data sequences. They further pre-train using the novel Identifier Prediction (IP) task, which consists of predicting whether tokens are identifiers or not as a binary classification task. Furthermore, they use AST Edge Prediction (TEP) to encode structural information directly from AST into the model by predicting masked edges between tree nodes. The main goal is to predict whether two nodes have an edge between them or not. Finally, they employ Multi-Modal Contrastive Learning (MCL), similar to Code-MVP, they create positive and negative samples by pairing different combinations of NL, PL, and AST and train to maximize the similarity for positive pairs while minimizing it for negative pairs.

3.1.14. *CoTexT*

CoTexT[33] offers an encoder-decoder pre-trained model that is trained to learn the representation between NL and PL. They develop a model that understands the relationship between natural language and programming language, enabling it to excel in various NL-PL tasks and achieve state-of-the-art results. This model is trained using a self-supervised method on both unimodal and bimodal data. They pre-trained the model by masking spans of tokens in the input sequence and using the concatenation of sentinel tokens and ground-truth masked tokens as the target sequence. The CodeSearchNet dataset and the GitHub portion of Google's BigQuery dataset were used to train the model on the programming languages: Go, Java, JavaScript, PHP, Python, and Ruby. The model is initialized with the T5[4] model and has a total of 220M trained parameters.

3.2. *Other Non-Transformer Models*

Some influential models have been proposed which do not rely on a Transformer[36] architecture, however, they do utilize the attention mechanism. Although they are not large models, they introduce important and successful code representation models.

3.2.1. *Code2vec*

Code2vec[21] provides an encoder-only trained model, which is designed to aggregate syntactic paths from the AST representation of source code into a single vector. The model architecture is a path-based attention model, which is trained with the objective of predicting a probability distribution of assigned tags to code snippets. The model is inspired by the NL model Doc2vec[8]. Code2vec was trained on a GitHub dataset containing 14M Java methods.

3.2.2. *Code2seq*

Code2seq[20] presents an encoder-decoder model, which is designed to convert source code snippets to natural language sequences. It was inspired by the Seq2seq[42] NL model. Similar to Code2vec, source code snippets are represented as compositional paths in their AST representation. The encoder takes in a vector representation of each path in the AST rather than code tokens. Code2seq was trained to predict the next token based on previously generated tokens as context. The model architecture encapsulates a combination of bidirectional LSTMs coupled with an attention mechanism in the decoder. The model was trained on Java (Java-Small, Java-Med, Java-Large) and C# datasets from GitHub and has a total of 37M trainable parameters.

3.3. *Available Code*

We present the GitHub links to the publicly available code and their availability on HuggingFace for the described models in Table 2. GitHub repositories host code for pre-training and/or fine-tuning the models on specific tasks and datasets. HuggingFace provides the pre-trained models' architecture and trained parameters, which can be simply downloaded and freely used through the HuggingFace API for fine-tuning for any task or dataset.

4. Datasets

We selected the most influential datasets in the PL domain to train and evaluate programming language models. Table 3 consists of programming languages, links, and downstream applications for each dataset.

4.1. *CodeXGLUE*

CodeXGLUE[30] is a collection of 14 datasets curated and filtered following proposed methodologies such as in Wang *et al.*[60]. The dataset collection contains a total of 10 different downstream tasks and it can be accessed at `https://github.com/microsoft/CodeXGLUE`. We omit CodeXGLUE from Table 3 and instead, we include the relevant datasets from the collection independently.

4.2. *CodeSearchNet*

CodeSearchNet[37] is a large collection of datasets that was generated to explore the problem of semantic code search. Given a natural language query, semantic code search refers to the task of retrieving the relevant code. This task requires close connection and cooperation between the language used in code (PL) and natural language, which is used for describing concepts. This dataset contains 2 million (comment, code) pairs extracted from open-source libraries. A comment can be either a NL comment or a top-level function and code is an entire function or method.

4.3. *CoDesc*

CoDesc[43] is a large parallel dataset containing 4.2 million pairs of Java source code and corresponding natural language descriptions with removed noise. This dataset is generated from other similar but noisy datasets such as CodeSearchNet[37], FunCom, DeepCom[58], and CONCODE[55]. The authors provided the noise removal source code, which they used as a pre-processing step, along with the dataset.

4.4. *APPS*

Automated Programming Progress Standard (APPS)[44] is a dataset curated for code generation. Despite previously proposed benchmarks for code generation with

Model	GitHub Link	HuggingFace Link
CodeBERT	https://github.com/microsoft/CodeBERT	https://huggingface.co/microsoft/codebert-base https://huggingface.co/microsoft/codebert-base-mlm
GraphCodeBERT	https://github.com/microsoft/CodeBERT	https://huggingface.co/microsoft/graphcodebert-base
CodeT5	https://github.com/salesforce/CodeT5	https://huggingface.co/Salesforce/codet5-base https://huggingface.co/Salesforce/codet5-small https://huggingface.co/Salesforce/codet5-large https://huggingface.co/Salesforce/codet5-base-multi-sum https://huggingface.co/Salesforce/codet5-large-ntp-py
PLBART	https://github.com/wasiahmad/PLBART	https://huggingface.co/uclanlp/plbart-base https://huggingface.co/uclanlp/plbart-large https://huggingface.co/uclanlp/plbart-java-cs https://huggingface.co/uclanlp/plbart-python-en_XX https://huggingface.co/uclanlp/plbart-csnet https://huggingface.co/uclanlp/plbart-multi_task-python https://huggingface.co/uclanlp/plbart-java-clone-detection
Codex	N/A	N/A
CodeTrans	https://github.com/agemagician/CodeTrans	https://huggingface.co/SEBIS/code_trans_t5_small_program_synthese_transfer_learning_finetune https://huggingface.co/SEBIS/code_trans_t5_base_code_documentation_generation_python https://huggingface.co/SEBIS/code_trans_t5_small_code_documentation_generation_python https://huggingface.co/SEBIS/code_trans_t5_large_code_comment_generation_java_transfer_learning_finetune https://huggingface.co/SEBIS/code_trans_t5_large_source_code_summarization_python_multitask_finetune
UniXcoder	https://github.com/microsoft/CodeBERT	https://huggingface.co/microsoft/unixcoder-base-nine https://huggingface.co/microsoft/unixcoder-base https://huggingface.co/microsoft/unixcoder-base-unimodal
TreeBERT	https://github.com/17385/TreeBERT	N/A
ContraCode	https://github.com/parasj/contracode	N/A
CuBERT	https://github.com/zhihu/cuBERT	https://huggingface.co/zluvolyote/CUBERT
Code-MVP	N/A	N/A
CodeGPT	N/A	https://huggingface.co/microsoft/CodeGPT-small-py-adaptedGPT2 https://huggingface.co/microsoft/CodeGPT-small-java-adaptedGPT2 https://huggingface.co/microsoft/CodeGPT-small-py https://huggingface.co/microsoft/CodeGPT-small-java
SynCoBERT	N/A	N/A
CoTexT	https://github.com/justinphan3110/CoTexT	https://huggingface.co/razent/cotext-1-ccg https://huggingface.co/razent/cotext-1-cc https://huggingface.co/razent/cotext-2-cc
Code2vec	https://github.com/tech-srl/code2vec	N/A
Code2seq	https://github.com/tech-srl/code2seq	N/A

Table 2: Available code and pre-trained model parameters.

Dataset	Programming Languages	Link	Downstream Applications
CodeSearchNet* (Husain *et al.*, 2019)	Go, Java, JavaScript, PHP, Python, Ruby	https://github.com/github/CodeSearchNet	Code Search[25], Code Summarization[25–27,29,30,33]
CoDesc (Hasan *et al.*, 2021)	Java	https://github.com/csebuetnlp/CoDesc	Code Search, Code Summarization
APPS (Hendrycks *et al.*, 2021)	Python	https://github.com/hendrycks/apps	Code Generation[28]
HumanEval (Chen *et al.*, 2021)	Python	https://github.com/openai/human-eval	Code Generation[28]
BigCloneBench* (Svajlenko *et al.*, 2014)	Java	https://github.com/clonebench/BigCloneBench	Clone Detection[15–17,30]
POJ-104* (Mou *et al.*, 2016)	C, C++	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Clone-detection-POJ-104	Clone Detection[16,17,30]
Devign* (Zhou *et al.*, 2019)	C	https://sites.google.com/view/devign	Defect Detection[16,26,27,30,33]
GraphCodeBERT's Code Search Dataset (CSN[17], CodeSearch[16]) (Guo *et al.*, 2021)	Go, Java, JavaScript, PHP, Python, Ruby	https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/codesearch	Code Search[15–17]
AdvTest* (Lu *et al.*, 2021)	Python	https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/NL-code-search-Adv	Code Search[16,17,30]
CosQA* (Huang *et al.*, 2021)	Python	https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/NL-code-search-WebQuery	Code Search[17,30]
CoNaLa (Yin *et al.*, 2018)	Python	https://conala-corpus.github.io	Code Search[19]
PY150* (Raychev *et al.*, 2016)	Python	https://www.sri.inf.ethz.ch/py150	Code Completion[17,30], Method Name Generation[18]
GREAT (Hellendoorn *et al.*, 2020)	Python	https://github.com/google-research-datasets/great	Defect Detection[19]
Github Java Corpus* (Allamanis and Sutton, 2013)	Java	https://groups.inf.ed.ac.uk/cup/javaGithub/	Code Completion[17,30]
Java Datasets (Java-small, Java-med and Java-large) (Allamanis *et al.*, 2016)	Java	https://groups.inf.ed.ac.uk/cup/codeattention/	Method Name Generation[18,20]
Bugs2Fix* (Tufano *et al.*, 2019)	Java	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-refinement	Code Repair[15,26,27,30,33]
CONCODE* (Iyer *et al.*, 2018)	Java	https://github.com/sriniiyer/concode	Code Generation[17,26,27,30,33]
AlgoLisp (Polosukhin and Skidanov, 2018)	LISP-inspired DSL	https://github.com/nearai/program_synthesis/tree/master/program_synthesis/algolisp	Code Generation[29]
CodeTrans* (Lu *et al.*, 2021) and (Guo *et al.*, 2021)	Java-C#	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans	Code Translation[15,16,26,27,30]
CodeNet (Puri *et al.*, 2021)	55 languages; 95% are in C++, C, C#, Python, Java, Ruby	https://github.com/IBM/Project_CodeNet	Code-to-Code Search[17,19], Clone Detection[19]
DeepCom's Java Dataset (Hu *et al.*, 2018)	Java	https://github.com/xing-hu/DeepCom	Code Summarization[18,29]
ContraCode's JavaScript Clone Detection Dataset (Jain *et al.*, 2021)	JavaScript	https://github.com/parasj/contracode	Clone Detection[31]
ContraCode's JavaScript Summarization Dataset (Jain *et al.*, 2021)	JavaScript	https://github.com/parasj/contracode	Method Name Generation[31]
CODE-NN Dataset (Iyer *et al.*, 2016)	Python, SQL, C#	https://github.com/sriniiyer/codenn/tree/master/data/stackoverflow	Code Summarization[29]

Table 3: Datasets used in PL. * refers to datasets included in the CodeXGLUE collection.

restricted settings, this benchmark evaluates the ability of the model to generate an appropriate code given a natural language query. APPS consists of 10,000 coding problems with different levels of difficulty.

4.5. *HumanEval*

HumanEval is an evaluation set released by Chen *et al.* [28] to calculate the functional correctness of generated codes and evaluate the quality of their model. It consists of 164 hand-written programming problems.

4.6. *BigCloneBench*

BigCloneBench [45] is a clone detection benchmark. It consists of known true and false positive clones extracted from the IJaDataset source repository, which is a Big Data inter-project Java repository.

4.7. *POJ-104*

POJ-104, which comes from a programming open judge (OJ) system, was proposed by Mou *et al.* [46]. The open judge system automatically judges the validity of codes for specified problems by executing them. POJ-104 is a clone detection dataset featured in the CodeXGLUE collection.

4.8. *Devign*

Devign [47], or Defects4J [16], is a defect detection dataset, which contains 27,318 functions gathered from two common and diverse C open-source codes among the developers. These functions are labeled as either vulnerable or non-vulnerable.

4.9. *CSN*

CSN, proposed by Guo *et al.* [15], is different from CodeSearchNet [37] dataset setting since the answer to a query is extracted from the entire development and testing code corpus (instead of only retrieving the answer among 1000 candidate codes). Furthermore, in order to improve the quality of the dataset they filtered out the queries with unrelated content to the code.

4.10. *AdvTest*

AdvTest [30] is a code search dataset in Python extracted from CodeSearchNet [37]. This dataset is generated by filtering out all the samples in which the code can not be parsed into an AST as well as all the samples in which the document has less than 3 tokens or more than 256 tokens, contains special tokens, or is not in English.

4.11. *CoSQA*

CoSQA[48] contains 20,604 pairs of code (in Python) and natural language web queries for code search and code question answering. The instances in this dataset are labeled as 1 or 0 based on whether the code can answer the given query or not.

4.12. *CoNaLa*

CoNaLa[49] is the Code/Natural Language Challenge dataset, which contains pairs of natural language intent and corresponding Python snippets. This dataset was extracted from Stack Overflow and was used in the code search task.

4.13. *PY150*

PY150[50] contains 150,000 Python source codes collected from GitHub and can be useful for token-level code completion. Furthermore Jiang *et al.*[18] used this dataset for method name generation.

4.14. *GREAT*

Graph Relational Embedding Attention Transformer[51] proposed the GREAT model as well as a dataset, which is a defect detection dataset. They used the PY150 corpus to extract top-level function definitions and randomly generate up to three bugs per function. Each instance in the GREAT dataset consists of a generated buggy function definition in Python paired with the original non-buggy code.

4.15. *GitHub Java Corpus*

GitHub Java Corpus[52] is a Java code completion dataset, which contains thousands of Java projects extracted from GitHub repositories and filtered based on GitHub's social fork system to reach a higher-than-average quality. However, considering only open-source projects can limit the type and application of included domains.

4.16. *Java Datasets*

Java Datasets[53] were gathered by cloning 11 widely-known open-source Java projects from GitHub. Chosen projects are the most popular Java projects on GitHub since they were selected based on the total z-scores of the number of watchers and forks.

4.17. *Bugs2Fix*

Bugs2Fix[54] is a Java code repair dataset, which contains two subsets (small and medium) based on the function's length. On the source side (pre-commit) it contains Java buggy functions while on the target side (post-commit) it contains the corresponding repaired ones.

4.18. *CONCODE*

CONCODE[55] is a frequently-used Java code generation dataset, in which each instance is a tuple consisting of NL description, code environment, and source code. It consists of over 100,000 examples of Java classes collected from public GitHub repositories.

4.19. *AlgoLisp*

AlgoLisp[56] is a LISP-inspired dataset for code generation, which contains problem descriptions in natural language as well as their corresponding implementations. This dataset is designed to learn basic concepts and rules in the domain-specific language (DSL). However, since the number of homework assignments used to generate this dataset is limited, models trained on AlgoLisp wouldn't be capable of generalizing to new types of algorithms.

4.20. *CodeTrans*

This Java to C# dataset was initially crawled from several open-source projects by Nguyen *et al.*. Although they provided their methodology for the retrieval of the data, they did not publish nor name the dataset. Afterward, Chen *et al.* used the previous approach to crawl the data again, however, once again, the authors did not publish nor name the dataset. More recently, Guo *et al.* followed the same methods to retrieve the dataset from the same open-source projects and published the dataset, however, it is only later named by Lu *et al.* as CodeTrans. The dataset provides functionally equivalent code written both in Java and C#.

4.21. *CodeNet*

Project CodeNet[57] is a large-scale, diverse, and high-quality curated dataset proposed for AI-for-Code research, with approximately 14 million code samples written in over 50 programming languages. This rich dataset can be potentially used for code search and clone detection as well as code-to-code search. From this dataset, they also create 4 benchmark datasets: C++1000, C++1400, Python800, and Java250.

4.22. *DeepCom's Java Dataset*

Hu *et al.*[58] proposed a Java dataset used for generating code comments (code summarization task). This dataset consists of 9,714 Java projects extracted from GitHub.

4.23. *ContraCode's Datasets*

Jain *et al.*[31] introduced a JavaScript clone detection dataset, as well as a type inference TypeScript dataset and a JavaScript code summarization dataset. They

generated their clone detection dataset by extracting JavaScript programs from the HackerRank interview preparation website. Their JavaScript code summarization dataset is collected from the labeled methods in the CodeSearchNet[37] dataset. We decided to omit the type inference task benchmark due to a lack of results.

4.24. *CODE-NN's Dataset*

Iyer *et al.*[63] proposed a code summarization dataset, which was collected from Stack Overflow. This dataset consists of pairs of Python, SQL, and C# code snippets along with their short NL descriptions.

5. Downstream Tasks

In this section, we will cover different PL downstream tasks and benchmarks. Different tasks use different metrics, thus, we will define the metrics accordingly.

5.1. *Code Generation*

Code Generation[55] is the task of predicting a code snippet or program structure through learning from multimodal data sources including codes in a different programming language, incomplete codes, natural language descriptions, or expected program outputs (see Figure 6). A code generation system can produce the programming language code with or without a given natural language query, describing the requirements, as an input[64]. Automatic code generation can promise to cut on the program development costs of programming tools as well as improve the quality of programs.

Metrics such as Exact Match, BLEU Score, and CodeBLEU are used to report the performance of code generation models. In the following, we provide a short description of each of them.

- Exact Match (EM): It measures how exactly the generated code matches the ground truth.
- BiLingual Evaluation Understudy (BLEU)[65]: It is calculated by counting the matching n-grams in the generated text and the reference regardless of the word order. The BLEU score is originally designed to evaluate natural language.
- CodeBLEU[66]: It utilizes the matching n-grams strategy that the BLEU score is using while integrating code syntax through abstract syntax tree and code semantics via data flow.

Table 4 compares the performance metrics of a few state-of-the-art models on the code generation task. These evaluation metrics are reported from previous benchmark experiments by different models on the Concode[55] dataset.

Input:

```
def generate_integers(a, b):
    """
    Given two positive integers a and b, return the
    even digits between a and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """
```

Generated code:

```
    a, b = divmod(a, b)
    if not a and not b:
        return []

    d = [int(i) for i in range(a, b+1)]
    for i in d:
        return [i] + [int(j) for j in i if j % 2 == 0]
```

Fig. 6: An example of code generation.

Models	Concode		
	EM	BLEU	CodeBLEU
CodeGPT (-adapted)	20.10	32.79	35.98
PLBART	18.75	36.69	38.52
CodeT5 (-base)	22.30	40.73	43.20
UniXcoder	22.60	38.23	-
CoTexT (1-CC)	20.10	37.40	40.14

Table 4: Code Generation [17,26,27,30,33]

5.2. *Code Summarization*

Code Summarization [59], also referred to as Code Documentation Generation [25], is a supervised task of generating a description of the functionality of a given input code (see Figure 7). Summaries of the computation a source code is performing can provide a clear understanding to a user, which is exceedingly helpful in applications such as code search. The comprehension of code, which is necessary to generate these sort of comments manually, can be extremely expensive and time-consuming. Hence, automating the process of generating high-level code documentation in natural language can be beneficial.

Smoothed BLEU-4 Score can be used to evaluate the performance of models on the code summarization task. Since BLEU Score calculates a geometric mean of n-

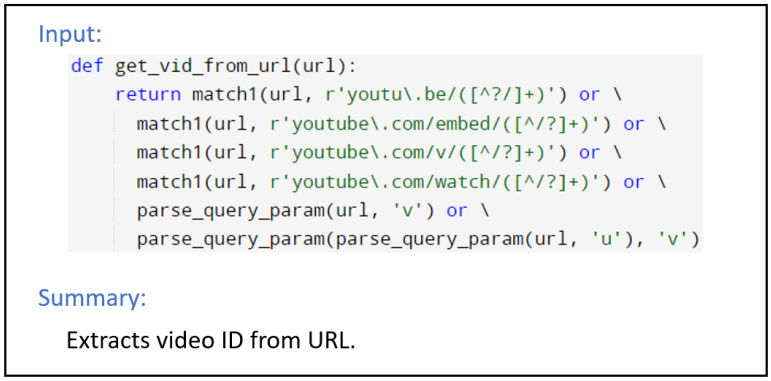

Fig. 7: An example of code summarization.

gram matches, it usually noticeably differs from a human judgment at the sentence level. Thus, different smoothing techniques have been proposed for sentence-level BLEU.

Table 5 compares the performance metrics of a few state-of-the-art models on the code summarization task. These evaluation metrics are reported from previous benchmark experiments by different models on the CodeSearchNet[37] and DeepCom's Java dataset[58].

Models	CodeSearchNet						DeepCom's Java Dataset
	smoothed BLEU-4						BLEU
	Ruby	JavaScript	Go	Python	Java	PHP	Java
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	-
CodeT5 (-base)	15.24	16.16	19.56	20.01	20.31	26.03	-
PLBART	14.11	15.56	18.91	19.30	18.45	23.58	-
CoTexT (1-CC)	14.02	14.96	18.86	19.73	19.06	24.58	-
CodeTrans (-TF-Base)	14.07	18.25	19.50	20.26	20.19	25.84	-
TreeBERT	-	-	-	-	-	-	20.49
CuBERT	-	-	-	-	-	-	17.41
Code2Seq	-	-	-	-	-	-	18.48

Table 5: Code Summarization[18,25–27,29,30,33]

5.3. *Code Repair*

Code Repair[54], also called Code Refinement[15], is the task of learning how to fix common programming bugs (see Figure 8). Finding and fixing the bugs in a code is one of the most difficult and costly tasks for software developers. Thus, code refinement aims at automatically fixing the errors in a program with less effort and cost.

BLEU, Accuracy, and CodeBLEU are the commonly-used metrics for evaluating models on the code repair task.

Buggy code:

```
public Integer getMaxElement(List myList) {
    if (myList.size() >= 0) {
        return ListManager.getFirst(myList);
    }
    return 0;
}
```

Fixed code:

```
public Integer getMaxElement(List myList) {
    if (myList.size() >= 1) {
        return ListManager.max(myList);
    }
    return null;
}
```

Fig. 8: An example of code repair.

- Accuracy: It measures the number of correctly predicted examples out of the total number of examples.

Table 6 compares the performance metrics of a few state-of-the-art models on the code repair task. These evaluation metrics are reported from previous benchmark experiments by different models on the Bugs2Fix[54] Java dataset.

Models	Bugs2Fix (Java)					
	small			medium		
	BLEU	Accuracy	CodeBLEU	BLEU	Accuracy	CodeBLEU
CodeBERT	77.42	16.4	75.58	91.07	5.2	87.52
GraphCodeBERT	80.02	17.3	-	91.31	9.1	-
PLBART	77.02	19.21	-	88.50	8.98	-
CodeT5 (-base)	77.43	21.61	-	87.64	13.96	-
CoTexT (1-CC)	77.79	21.03	76.15	88.4	13.11	85.83

Table 6: Code Repair[15,26,27,30,33]

5.4. *Code Search*

The Code Search[37] task is composed of two subtasks. One of them is finding the most appropriate code snippet for a given natural language description from a list of PL candidates (see Figure 9). The second subtask is analyzing the query-code pairs to indicate if the code answered the given query appropriately or not. This task requires bridging the gaps between the programming language and natural language.

The metric used for evaluating this task is the Mean Reciprocal Rank.

Query:

Adds new tweets to the cache.

Code:

```
def add_tweets(self, url, last_modified, tweets):
    try:
        self.cache[url] = {"last_modified": last_modified, "tweets": tweets}
        self.mark_updated()
        return True
    except TypeError:
        return False
```

Fig. 9: An example of code search.

- Mean Reciprocal Rank (MRR): It is a metric for evaluating any model that generates an ordered list of possible answers (ordered by the probability of correctness).

Table 7 compares the performance metrics of a few state-of-the-art models on the code search task. These evaluation metrics are reported from previous benchmark experiments by different models on the GraphCodeBERT's Code Search Dataset [15], CosQA [48], AdvTest [37], and CoNala [49] dataset.

Models	GraphCodeBERT's Code Search Dataset	CosQA	AdvTest	CoNaLa
	Overall	Python	Python	Python
CodeBERT	69.3	65.7	27.2	38.9
GraphCodeBERT	71.3	68.4	35.2	47.3
SynCoBERT	74.0	-	38.3	48.4
PLBART	68.5	65.0	34.7	45.5
CodeT5 (-base)	71.5	67.8	39.3	47.7
UniXcoder	74.4	70.1	41.3	-
Code-MVP	-	72.1	40.4	50.6

Table 7: Code Search [15–17,19,30]

5.5. *Code Clone Detection*

The Code Clone Detection [67] task consists of detecting whether two programs are functionally or semantically equivalent, in some cases, whether they solve the same problem. This task is evaluated as a binary classification problem with the F1, Mean Average Precision, and Accuracy performance metrics. In the following, we provide a short description of them.

- F1: The F1 score is the harmonic mean between the precision and the recall. Similar to accuracy, it is used for binary classification.

- Mean Average Precision (MAP): Same as AP, this measure is also designed to evaluate the object detection task in general. It compares the ground-truth bounding box with the detected box. In code clone detection, MAP can measure the similarity between two programs.

Table 8 compares the performance metrics of a few state-of-the-art models on the code clone detection task. These evaluation metrics are reported from previous benchmark experiments by different models on the BigCloneBench[45], POJ-104[46], and CodeNet (Python800)[57] dataset.

Models	BigCloneBench	POJ-104	CodeNet (Python800)
	F1	MAP	Accuracy
CodeBERT	94.1	82.67	95.2
GraphCodeBERT	95.0	85.16	95.9
SynCoBERT	-	88.24	96.1
PLBART	93.6	86.27	95.5
CodeT5 (-base)	95.0	88.65	95.7
UniXcoder	95.2	90.52	-
Code-MVP	-	-	97.4

Table 8: Code Clone Detection[15–17,19,30]

5.6. *Method Name Generation*

Method Name Generation, also named as Code Summarization[18,31], refers to the prediction of a method name given the method's body (see Figure 10). This helps developers name functions in a meaningful way to describe their code's functionality. The F1 score is used for evaluating this task.

Input:

```
def _ (self, url):
    """Checks if specified URL is cached."""
    try:
        return True if url in self.cache else False
    except TypeError:
        return False
```

Method name:

is_cached

Fig. 10: An example of method name generation.

Table 9 compares the performance metrics of a few state-of-the-art models on the method name generation task. These evaluation metrics are reported from pre-

vious benchmark experiments by different models on the PY150[50] dataset, Java Datasets (Java-small, Java-med, and Java-large)[53], and ContraCode's JavaScript Summarization Dataset[31].

Models	PY150	Java-small	Java-med	Java-large	ContraCode's JavaScript Summarization Dataset
	F1	F1	F1	F1	F1
TreeBERT	39.04	51.99	61.22	67.25	-
CodeBERT	29.58	41.10	49.64	54.76	-
CuBERT	26.99	38.22	45.99	50.55	-
Code2seq	30.07	43.02	53.23	59.18	9.39
Code2vec	-	-	-	-	9.34
ContraCode	-	-	-	-	17.24

Table 9: Method Name Generation[18,20,31]

5.7. *Defect Detection*

Defect Detection[47] is the task of learning and identifying whether a source code contains any defects such as DOS (denial-of-service) attacks, resources leaks, UFA vulnerabilities (use-after-free vulnerability, which is caused by the incorrect use of dynamic memory during the program's operation), etc. that can be used for software system attacks. Accuracy is used to evaluate this binary classification task.

Table 10 compares the performance metrics of a few state-of-the-art models on the defect detection task. These evaluation metrics are reported from previous benchmark experiments by different models on the Devign[47] and GREAT[51] dataset.

Models	Devign	GREAT
	Accuracy	Accuracy
CodeBERT	62.08	85.5
GraphCodeBERT	63.21	87.5
Code2vec	62.48	-
PLBART	63.18	86.8
CodeT5 (-base)	65.78	87.4
SynCoBERT	64.50	88.2
Code-MVP	-	89.3
CoTexT (1-CC)	65.99	-

Table 10: Defect Detection[16,19,26,27,30,33]

5.8. *Code Completion*

Code Completion[68] is the task of predicting the following tokens based on the given code context. It contains two subtasks: token-level completion and line-level completion task (see Figure 11). In the following table, we will focus on the line-level completion task, which checks the quality of generated line of code. While the token-level completion task tests whether the predicted token was generated

correctly or not.

Input:

```
def get_action_image(op, name):
    """Return the image for the operation."""
    action = _get_action_by_name(op, name)
    if action:
```

Ground truth:

```
        return action.get('imageUri')
```

Fig. 11: An example of line-level code completion.

Exact Match and Edit Sim are two metrics that are used for evaluating the code completion task.

- Edit Sim: The Levenshtein edit similarity measures a distance corresponding to the number of single-character edits necessary to transform one token into another.

Table 11 compares the performance metrics of a few state-of-the-art models on the code completion task. These evaluation metrics are reported from previous benchmark experiments by different models on Py150[50] and Github Java Corpus[52] dataset.

Models	PY150		Github Java Corpus	
	EM	Edit Sim	EM	Edit Sim
CodeGPT (-adapted)	39.65	69.84	26.43	63.03
PLBART	38.01	68.46	26.97	61.59
CodeT5 (-base)	36.97	67.12	24.80	58.31
UniXcoder	43.12	72.00	32.90	65.78

Table 11: Code Completion[17,30]

5.9. *Code Translation*

Code Translation is the task of translating a code from one programming language into another one (see Figure 12). The current benchmark datasets support translation between Java and C#. BLEU, Accuracy, and CodeBLEU are the metrics used frequently for evaluating models on the code translation task.

Table 12 compares the performance metrics of a few state-of-the-art models on the code translation task. These evaluation metrics are reported from previous benchmark experiments by different models on CodeTrans[15,30] dataset.

Input (Java method):

```
public void removePresentationFormat() {
    remove1stProperty(PropertyIDMap.PID_PRESFORMAT);
}
```

Output (C# method):

```
public void RemovePresentationFormat() {
    MutableSection s = (MutableSection)FirstSection;
    s.RemoveProperty(PropertyIDMap.PID_PRESFORMAT);
}
```

Fig. 12: An example of code translation.

Models	CodeTrans					
	Java to C#			C# to Java		
	BLEU	Accuracy	CodeBLEU	BLEU	Accuracy	CodeBLEU
CodeBERT	79.92	59.00	85.10	72.14	58.80	79.41
GraphCodeBERT	80.58	59.40	-	72.64	58.80	-
PLBART	83.02	64.60	-	78.35	65.00	-
SynCoBERT	80.75	60.40	84.85	76.52	61.30	82.22
CodeT5 (-base)	84.03	65.90	-	79.87	66.90	-

Table 12: Code Translation [15,16,26,27,30]

5.10. *Code-to-Code Search*

With Code-to-Code Search [17], we focus mainly on same language code search for consistency and fairness across models. This task involves retrieving the most semantically similar code to a given code query. MAP is the metric which can be used for evaluating code-to-code search.

Table 13 compares the performance metrics of a few state-of-the-art models on the code-to-code search task. These evaluation metrics are reported from previous benchmark experiments by different models on CodeNet [57] dataset.

Models	CodeNet (Python800)	CodeNet (Ruby)	CodeNet (Python)	CodeNet (Java)
	MAP	MAP	MAP	MAP
CodeBERT	86.10	13.55	14.39	7.62
GraphCodeBERT	88.80	17.01	19.34	13.31
PLBART	86.70	18.60	19.55	10.41
SynCoBERT	89.2	-	-	-
Code-MVP	91.50	-	-	-
UniXcoder	-	29.05	30.15	16.12
CodeT5 (-base)	88.10	18.22	17.83	10.18

Table 13: Code-to-Code Search [17,19]

6. Conclusion and Future Work

Automatic understanding and analysis of the structure and semantics of codes written in a Programming Language (PL) is of paramount importance. It helps build smart tools facilitating a more efficient software development process. Moreover, effective code representation learning can serve as a fundamental building block for many downstream tasks in PLs. In this paper, we present the current state-of-the-art models and compare their performance on PL downstream tasks such as code generation, code summarization, etc. We provide the most recent publicly available datasets and available codes. While there are many advancements in representation learning approaches for different PLs, many models, such as Codex [28], ContraCode [31], CuBERT [32], Code-MVP [19] only support one specific programming language. Moreover, some downstream tasks (e.g., code translation, code repair) don't have many programming languages to benchmark in the available datasets (e.g., code translation has only C# to Java, and Code repair has only Java). These could be the subjects of further research in the future.

References

1. J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Minneapolis, Minnesota, 2019, pp. 4171–4186.
2. M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov and L. Zettlemoyer, Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, 2020, pp. 7871–7880.
3. Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov, *CoRR*, 2019, **abs/1907.11692**,.
4. C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. J. Liu, *Journal of Machine Learning Research*, 2020, **21**, 1–67.
5. A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, tech. rep., OpenAI, 2018.
6. M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee and L. Zettlemoyer, *CoRR*, 2018, **abs/1802.05365**,.
7. Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov and Q. V. Le, *CoRR*, 2019, **abs/1906.08237**,.
8. Q. Le and T. Mikolov, Proceedings of the 31st International Conference on Machine Learning, Bejing, China, 2014, pp. 1188–1196.
9. T. Mikolov, I. Sutskever, K. Chen, G. Corrado and J. Dean, Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, Red Hook, NY, USA, 2013, p. 3111–3119.
10. Y. Liao, Y. Wang and Y. Liu, *IEEE Transactions on Image Processing*, 2017, **26**, 2839–2852.
11. N. Reimers and I. Gurevych, Proceedings of the 2019 Conference on Empiri-

cal Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, 2019, pp. 3982–3992.

12. Y. Kim, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 2014, pp. 1746–1751.
13. Y. Zhang, Y. Xia, Y. Liu and W. Wang, Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, 2015, pp. 1262–1267.
14. J. Pennington, R. Socher and C. Manning, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 2014, pp. 1532–1543.
15. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang and M. Zhou, International Conference on Learning Representations, 2021.
16. X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu and X. Jiang, preprint, arXiv:2108.04556, 2021.
17. D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou and J. Yin, Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Dublin, Ireland, 2022, pp. 7212–7225.
18. X. Jiang, Z. Zheng, C. Lyu, L. Li and L. Lyu, Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, 2021, pp. 54–63.
19. X. Wang, Y. Wang, Y. Wan, J. Wang, P. Zhou, L. Li, H. Wu and J. Liu, *ArXiv*, 2022, **abs/2205.02029**,.
20. U. Alon, O. Levy and E. Yahav, preprint, arXiv:1808.01400, 2018.
21. U. Alon, M. Zilberstein, O. Levy and E. Yahav, *Proc. ACM Program. Lang.*, 2019, **3**, 1–29.
22. H.-H. Wei and M. Li, Proceedings of the 26th International Joint Conference on Artificial Intelligence, 2017, p. 3034–3040.
23. J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang and X. Liu, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 783–794.
24. W. Wang, G. Li, B. Ma, X. Xia and Z. Jin, *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, 261–271.
25. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang and M. Zhou, Findings of the Association for Computational Linguistics: EMNLP 2020, Online, 2020, pp. 1536–1547.
26. Y. Wang, W. Wang, S. Joty and S. C. Hoi, Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Online and Punta Cana, Dominican Republic, 2021, pp. 8696–8708.
27. W. Ahmad, S. Chakraborty, B. Ray and K.-W. Chang, Proceedings of the 2021 Conference of the North American Chapter of the Association for Computa-

tional Linguistics: Human Language Technologies, Online, 2021, pp. 2655–2668.
28. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever and W. Zaremba, *CoRR*, 2021, **abs/2107.03374**,.
29. A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes and B. Rost, *CoRR*, 2021, **abs/2104.02443**,.
30. S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu and S. Liu, *CoRR*, 2021, **abs/2102.04664**,.
31. P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez and I. Stoica, Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Online and Punta Cana, Dominican Republic, 2021, pp. 5954–5971.
32. A. Kanade, P. Maniatis, G. Balakrishnan and K. Shi, Proceedings of the 37th International Conference on Machine Learning, 2020, pp. 5110–5121.
33. L. Phan, H. Tran, D. Le, H. Nguyen, J. Annibal, A. Peltekian and Y. Ye, Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), Online, 2021, pp. 40–47.
34. D. E. Rumelhart, G. E. Hinton and R. J. Williams, *Nature*, 1986, **323**, 533–536.
35. S. Hochreiter and J. Schmidhuber, *Neural computation*, 1997, **9**, 1735–80.
36. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser and I. Polosukhin, Advances in Neural Information Processing Systems, 2017.
37. H. Husain, H. Wu, T. Gazit, M. Allamanis and M. Brockschmidt, *CoRR*, 2019, **abs/1909.09436**,.
38. T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, Advances in Neural Information Processing Systems, 2020, pp. 1877–1901.
39. T. Gao, X. Yao and D. Chen, Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Online and Punta Cana, Dominican Republic, 2021, pp. 6894–6910.
40. C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. J. Liu, *Journal of Machine Learning Research*, 2020, **21**, 1–67.

41. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, *OpenAI blog*, 2019, **1**, 9.
42. I. Sutskever, O. Vinyals and Q. V. Le, Advances in Neural Information Processing Systems, 2014.
43. M. Hasan, T. Muttaqueen, A. A. Ishtiaq, K. S. Mehrab, M. M. A. Haque, T. Hasan, W. Ahmad, A. Iqbal and R. Shahriyar, Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, Online, 2021, pp. 210–218.
44. D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song and J. Steinhardt, Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021.
45. J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy and M. M. Mia, 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 476–480.
46. L. Mou, G. Li, L. Zhang, T. Wang and Z. Jin, Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016, p. 1287–1293.
47. Y. Zhou, S. Liu, J. Siow, X. Du and Y. Liu, Proceedings of the 33rd International Conference on Neural Information Processing Systems, Red Hook, NY, USA, 2019, p. 10197–10207.
48. J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou and N. Duan, Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Online, 2021, pp. 5690–5700.
49. P. Yin, B. Deng, E. Chen, B. Vasilescu and G. Neubig, Proceedings of the 15th International Conference on Mining Software Repositories, New York, NY, USA, 2018, p. 476–486.
50. V. Raychev, P. Bielik and M. Vechev, *SIGPLAN Not.*, 2016, **51**, 731–747.
51. V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis and D. Bieber, International Conference on Learning Representations, 2020.
52. M. Allamanis and C. Sutton, Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, p. 207–216.
53. M. Allamanis, H. Peng and C. Sutton, International Conference on Machine Learning (ICML), 2016.
54. M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White and D. Poshyvanyk, *ACM Trans. Softw. Eng. Methodol.*, 2019, **28**, 1–29.
55. S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 2018, pp. 1643–1652.
56. I. Polosukhin and A. Skidanov, *Neural Program Search: Solving Data Processing Tasks from Description and Examples*, 2018, `https://openreview.net/forum?id=B1KJJf-R-`.
57. R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar,

S. Ramji, U. Finkler, S. Malaika and F. Reiss, Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021.

58. X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, ICPC '18: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 200–210.
59. S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany, 2016, pp. 2073–2083.
60. W. Wang, G. Li, B. Ma, X. Xia and Z. Jin, 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 261–271.
61. A. T. Nguyen, T. T. Nguyen and T. N. Nguyen, 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 585–596.
62. X. Chen, C. Liu and D. Song, Proceedings of the 32nd International Conference on Neural Information Processing Systems, Red Hook, NY, USA, 2018, p. 2552–2562.
63. S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany, 2016, pp. 2073–2083.
64. X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang and Q. Liu, Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, 2022, pp. 9–19.
65. K. Papineni, S. Roukos, T. Ward and W.-J. Zhu, Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, USA, 2002, p. 311–318.
66. S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco and S. Ma, *CoRR*, 2020, **abs/2009.10297**,.
67. J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy and M. M. Mia, 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 476–480.
68. M. Allamanis and C. Sutton, 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 207–216.