CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation

Shuai Lu*
Peking University

Daya Guo*
Sun Yat-sen University

Shuo Ren*
Beihang University

Junjie Huang*
Beihang University

Alexey Svyatkovskiy
Microsoft

Ambrosio Blanco
Microsoft Research Asia

Colin Clement
Microsoft

Dawn Drain
Microsoft

Daxin Jiang
Microsoft

Duyu Tang
Microsoft Research Asia

Ge Li
Peking University

Lidong Zhou
Microsoft Research Asia

Linjun Shou
Microsoft

Long Zhou
Microsoft Research Asia

Michele Tufano
Microsoft

Ming Gong
Microsoft

Ming Zhou
Microsoft Research Asia

Nan Duan
Microsoft Research Asia

Neel Sundaresan
Microsoft

Shao Kun Deng
Microsoft

Shengyu Fu
Microsoft

Shujie Liu
Microsoft Research Asia

ABSTRACT

Benchmark datasets have a significant impact on accelerating research in programming language tasks. In this paper, we introduce CodeXGLUE, a benchmark dataset to foster machine learning research for program understanding and generation. CodeXGLUE includes a collection of 10 tasks across 14 datasets and a platform for model evaluation and comparison. CodeXGLUE also features three baseline systems, including the BERT-style, GPT-style, and Encoder-Decoder models, to make it easy for researchers to use the platform. The availability of such data and baselines can help the development and validation of new methods that can be applied to various program understanding and generation problems [1].

KEYWORDS

program understanding, machine learning, naturalness of software

1 INTRODUCTION

Evans Data Corporation[2] estimated that there were 23.9 million professional developers in 2019 and that the number was expected to reach 28.7 million in 2024. With the population of developers growing at such a rate, code intelligence that leverages artificial intelligence (AI) to help software developers improve the productivity of the development process is becoming increasingly important. It is commonly accepted that benchmarks have a significant impact on the growth of applied AI research. In this paper, we focus on establishing a benchmark dataset for code intelligence.

Automated program understanding and generation could increase the productivity of software developers. In fact, developers who want to find code written by others with the same intent can leverage code search systems [23, 35, 58, 85] to automatically retrieve semantically relevant codes through natural language queries. Similarly, developers who are confused about what to write next can use code completion systems [4, 8, 9, 31, 62, 63, 72, 73] to automatically complete the following tokens based on the edits made to the code. Finally, when developers want to implement the Java code in Python, code-to-code translation systems [11, 41, 46, 54] can help translate their code from one programming language (Python) to another (Java).

In recent years, researchers have increasingly applied statistical models, including neural nets, to code intelligence tasks. Very recently, the application of pretrained models that learn from big programming language data has been inspired by the great success of pretrained models like BERT [16] and GPT [69] in natural language processing (NLP). These models, including CodeBERT [18] and IntelliCode Compose [72], have led to further improvements in code understanding and generation problems, but they lack a benchmark suite that covers a wide range of tasks. The use of ImageNet [15] for computer vision and the use of GLUE [81] for NLP have shown that a diversified benchmark dataset has a significant impact on the growth of applied AI research.

*indicates equal contribution and internship at Microsoft. Authors are listed in alphabeta order. Corresponding authors are Duyu Tang and Shujie Liu.

[1] CodeXGLUE is publicly available at https://github.com/microsoft/CodeXGLUE. Participants can submit their results by emailing to codexglue@microsoft.com.

[2] https://evansdata.com/press/viewRelease.php?pressID=278

Table 1: A brief summary of CodeXGLUE, which includes tasks, datasets, languages, sizes in various states, and baseline systems. Highlighted datasets are newly introduced.

Category	Task	Dataset Name	Language	Train/Dev/Test Size	Baselines
Code-Code	Clone Detection	BigCloneBench [71]	Java	900K/416K/416K	CodeBERT
		POJ-104 [52]	C/C++	32K/8K/12K	
	Defect Detection	Devign [99]	C	21K/2.7K/2.7K	
	Cloze Test	CT-all	Python,Java,PHP, JavaScript,Ruby,Go	-/-/176K	
		CT-max/min [18]	Python,Java,PHP, JavaScript,Ruby,Go	-/-/2.6K	
	Code Completion	PY150 [62]	Python	100K/5K/50K	CodeGPT
		Github Java Corpus[4]	Java	13K/7K/8K	
	Code Repair	Bugs2Fix [75]	Java	98K/12K/12K	Encoder-Decoder
	Code Translation	CodeTrans	Java-C#	10K/0.5K/1K	
Text-Code	NL Code Search	CodeSearchNet [35], AdvTest	Python	251K/9.6K/19K	CodeBERT
		CodeSearchNet [35], WebQueryTest	Python	251K/9.6K/1K	
	Text-to-Code Generation	CONCODE [38]	Java	100K/2K/2K	CodeGPT
Code-Text	Code Summarization	CodeSearchNet [35]	Python,Java,PHP, JavaScript,Ruby,Go	908K/45K/53K	Encoder-Decoder
Text-Text	Documentation Translation	Microsoft Docs	English-Latvian/Danish /Norwegian/Chinese	156K/4K/4K	

To address this problem, we introduce CodeXGLUE, a machine learning benchmark dataset for program understanding and generation research that includes 14 datasets[3], a collection of 10 diversified programming language understanding and generation tasks, and a platform for model evaluation and comparison. CodeXGLUE supports the following tasks:

- **code-code** (clone detection [10, 52, 71, 84, 89, 93, 97], defect detection [47, 57, 61, 82, 83, 99], cloze test [18], code completion [4, 8, 9, 31, 62, 63, 72, 73], code repair [2, 28, 30, 75, 76, 78], and code-to-code translation [11, 41, 46, 54])
- **text-code** (natural language code search [23, 35, 85], text-to-code generation [12, 26, 36, 38, 87, 90, 94, 95])
- **code-text** (code summarization [3, 12, 19, 34, 37, 80, 85–87])
- **text-text** (documentation translation [40])

CodeXGLUE includes eight previously proposed datasets — BigCloneBench [71], POJ-104 [52], Devign [99], PY150 [62], Github Java Corpus [4], Bugs2Fix [75], CONCODE [38], and CodeSearchNet [35]— but also newly introduced datasets that are highlighted in Table 1. The datasets are chosen or created based on the consideration that the task has clear definition, and the volume of the dataset could support the development and evaluation of data-driven machine learning methods. The datasets created by us include (1) two cloze test test sets that cover 6 programming languages, (2) two line-level code completion test sets in Java and Python, respectively, (3) a code-to-code translation dataset between Java and C#, (4) two natural language code search test sets with web queries and normalized function and variable names, respectively, and (5) a documentation translation dataset that covers five natural languages.

[3]We plan to evolve the benchmark over time by extending to more tasks.

To make it easy for participants, we provide three baseline models to help perform the tasks, including a BERT-style pretrained model (in this case, CodeBERT) to supports code understanding problems, a GPT-style pretrained model, which we call CodeGPT, to help solve completion and generation problems, and an Encoder-Decoder framework that tackles sequence-to-sequence generation problems.

2 TASKS OVERVIEW

In this section, we provide a definition for each task.

Clone detection [52, 71]. The task is to measure the semantic similarity between codes. This includes two subtasks: binary classification between a pair of codes and code retrieval, where the goal is to find semantically similar codes.

Defect detection [99]. The objective is to identify whether a body of source code contains defects that may be used to attack software systems, such as resource leaks, use-after-free vulnerabilities, and DoS attack.

Cloze test [18]. This aims to predict the masked token of a code and includes two subtasks. The first one is to measure the accuracy of predicting the masked token from the whole vocabulary. The other is to test the semantic reasoning ability by distinguishing between “max” and “min”.

Code completion [4, 62]. It aims to predict following tokens based on a code context. Its subtasks are token-level completion and line-level completion. The former checks whether the next one token has been predicted correctly, while the latter tests the goodness of the generated line.

Code translation [54]. It involves translating a code from one programming language to a different one.

Code search [35]. It measures the semantic relatedness between texts and codes and is composed of two subtasks. The first one is to find the most relevant code in a collection of codes according to a natural language query. The second subtask entails the analysis of a query-code pair to predict whether the code answers the query or not.

Code repair [75]. Its goal is to refine the code by fixing the bugs automatically.

Text-to-code generation [38]. This aims to generate a code via a natural language description.

Code summarization [37]. The objective is to generate the natural language comment for a code.

Documentation translation [40]. It aims to translate code documentation from one natural language to different one.

3 DATASETS

In this section, we describe the datasets included in CodeXGLUE. Datasets are chosen or created based on the criterion that the volume of the dataset could support the development and evaluation of data-driven machine learning methods.

3.1 Clone detection

Clone detection includes two subtasks. The first subtask is to predict whether two given codes have the same semantics. We use the BigCloneBench [71] dataset for the subtask. The second subtask aims to retrieve semantically similar codes given a code as the query and we use the dataset POJ-104 [52] to perform it.

BigCloneBench is a widely used large code clone benchmark that contains over 6,000,000 true clone pairs and 260,000 false clone pairs from 10 different functionalities. The dataset provided by Wang et al. [84] is filtered by discarding code fragments without any tagged true or false clone pairs, leaving it with 9,134 Java code fragments. Finally, the dataset includes 901,028/415,416/415,416 examples for training, validation and testing, respectively.

POJ-104 dataset [52] comes from a pedagogical programming open judge (OJ) system that automatically judges the validity of submitted source code for specific problems by running the code. We use the POJ-104 dataset, which consists of 104 problems and includes 500 student-written C/C++ programs for each problem. Different from that of the BigCloneBench dataset, the task of POJ-104 aims to retrieve other programs that solve the same problem given a program. We group the datasets in three subsets based on the number of problems they are required to solve (64/16/24) for training, validation, and testing.

3.2 Defect detection

For the task of defect detection, Zhou et al. [99] provide the **Devign** dataset that includes 27,318 manually-labeled functions collected from two large C programming language open-source projects popular among developers and diversified in functionality, i.e., QEMU and FFmpeg. The dataset was created by collecting security-related commits and extracting vulnerable or non-vulnerable functions from the labeled commits. Since Zhou et al. [99] did not provide official training/validation/testing sets for the two projects, we randomly shuffle the dataset and split 80%/10%/10% of the dataset for training/validation/testing. The task is formulated as a binary classification to predict whether a function is vulnerable.

3.3 Cloze test

Figure 1 shows two examples of the cloze test (CT) task in code domain, which aims to assess models' ability to understand a code by asking those models to predict the masked code from several candidates. We focus on two subtasks: CT-all with candidates from a filtered vocabulary and CT-maxmin with the candidates "max" and "min".

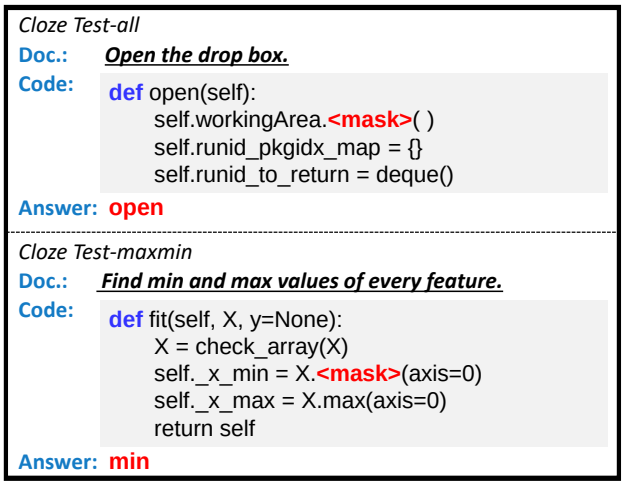

Figure 1: Two examples in the cloze test dataset.

We use the validation and testing sets of CodeSearchNet [35] to create CT-all and CT-maxmin datasets for six programming languages, i.e., Go, Java, JavaScript (JS), PHP, Python and Ruby.

CT-all. To less introduce lengthy variable names and avoid the issue caused by the use of different tokenizers, we select target cloze words by retaining unique words after Byte Pair Encoding [67], and we remove meaningless tokens like punctuations with handcrafted rules. At last, 930 tokens are selected among six languages in total. We select codes containing the 930 tokens and manually set threshold values of token occurrence to balance the frequency of the 930 tokens in CT-all.

CT-maxmin. To further evaluate models' ability to understand code semantics, we introduce CT-maxmin to test how well model can distinguish the difference between *max* and *min*. CT-maxmin comes from the dataset used for the PL-Probing task in CodeBERT[18], which includes codes containing the keywords of *max* or *min*.

The data statistics are listed in Table 2.

3.4 Code completion

We use two influential datasets for code completion, **PY150** in python and **Github Java Corpus** in Java. Both datasets can help achieve token-level code completion. We move further by creating two test sets for the line-level code completion task from the two corpora. The task is to complete an unfinished line. Models should

Table 2: Data statistics about the cloze test datasets.

Task	CT-all	CT-maxmin
Go	25,282	152
Java	40,492	482
JavaScript	13,837	272
PHP	51,930	407
Python	40,137	1,264
Ruby	4,437	38
All	176,115	2,615

be capable of predicting code sequences of arbitrary token types and code structures.

PY150 is a Python dataset [62] containing 150,000 Python source files collected from Github. We follow the data split in Raychev et al. [62], resulting in 100,000 files for training and 50,000 files for testing, consisting 76.3M tokens and 37.2M tokens, respectively. We preprocess the corpora by tokenizing source codes, removing comments, replacing strings with length of more than 15 characters with empty strings, and adding a special token ⟨*EOL*⟩ (end-of-line) to mark the ending of a line explicitly. For line-level code completion, we create 10,000 examples from different files in the test set of PY150 for testing. Since we intend to test model's ability to autocomplete an arbitrary line, we select the line to be predicted at random. We generate a test case by ensuring that there is sufficient context, i.e., at least 15% of the whole file. Models are expected to generate the following line ended by ⟨*EOL*⟩ given the context. The average number of tokens in input and output are 489.11 and 6.56, respectively. Figure 2 shows an example of line-level code completion.

Inputs:

```
def phi_nonTerminal ( self , s ):
    F_s = np . zeros ( self . features_num )
    F_s [ self . activeInitialFeatures ( s ) ] = 1
    bebf_features = self . initial_features
    for ind , ( F_s_ind , feature ) in
```

Ground Truth:

```
enumerate ( zip ( F_s , bebf_features ) ) :
```

Figure 2: An example in the line-level code completion dataset.

Github Java Corpus is a Java dataset mined by Allamanis and Sutton [4], and it collects over 14 thousand Java projects from Github. We follow the settings established by Hellendoorn and Devanbu [29] as well as Karampatsis et al. [42], using 1% of the subset in the corpus. We have 12,934/7,189/8,268 files for training/validation/testing, consisting of 15.8M/3.8M/5.3M tokens, respectively. We do the same preprocessing conducted on PY150, but we don't add the special token ⟨*EOL*⟩ since in Java the symbols ; and } are used to mark the ending of a code statement. For line-level code completion, we create 3,000 examples for testing from different files in the test set of the corpus. Similarly to the process we follow for Python, the line to be predicted is selected at random from the test file. The average numbers of tokens are 350.62 and 10.49 in input and output, respectively.

3.5 Code translation

The training data for code translation is the code pairs with equivalent functionality in two programming languages. In this paper, we provide a dataset consisting of parallel codes between Java and C#. We did not use the dataset of Lachaux et al. [46] because they did not have the data for supervised model training. Following Nguyen et al. [54] and Chen et al. [11], we use the data collected from several open-source projects, i.e., Lucene[4], POI[5], JGit[6] and Antlr[7]. We do not use Itext[8] and JTS[9] due to the license problem. Those projects are originally developed in Java and then ported to C#. They are well-established systems with long developing histories and with both Java and C# versions in use.

The following step is to mine paired functions or methods from those projects. According to our observation, the directory structures and function or method names of the two versions are identical or similar when they are applied to the same project. Therefore, following Nguyen et al. [54], we conservatively search for the functions having the same signatures in the classes with the same/similar names and included in the same/similar directory structures of both versions. We discard duplicate code pairs and the codes having multiple targets searched with the above method. After this step, we remove the pairs whose number of overlapping tokens was less than 1/3 of the sentence length. To make our data more scalable for further syntactic and semantic analysis, we also remove the functions with null function body according to their abstract syntax tree (AST). Then we build the data-flow graph [25] for each function, which represents the dependency between two variables and provides valuable semantic information for code understanding. Finally, a function with no data-flow extracted from the AST of a specific function is also discarded.

At last, the total number of paired functions or methods is 11,800. We randomly select 500 pairs of functions for the development set and another 1,000 pairs for the test set. The average lengths of the Java and C# functions after tokenization are 38.51 and 46.16, respectively [10]. An example of the mined translation pairs from C# to Java is shown in Figure 3.

3.6 Code search

Code search includes two subtasks. The first one is to find the most relevant code from a collection of candidates given a natural language query. We create a challenging testing set, called **CodeSearchNet AdvTest**, from CodeSearchNet corpus [35] for performing this task. An example of this dataset is shown in Figure 4. The second subtask is to predict whether a code answers a given query. We provide a testing set **WebQueryTest** of real user queries. Two examples of the dataset are illustrated in Figure 5.

[4] http://lucene.apache.org/
[5] http://poi.apache.org/
[6] https://github.com/eclipse/jgit/
[7] https://github.com/antlr/
[8] http://sourceforge.net/projects/itext/
[9] http://sourceforge.net/projects/jts-topo-suite/
[10] https://github.com/c2nes/javalang

Input: A Java method

```
public PersianCharFilterFactory(Map<String,String> args)
{
   super(args);
   if (!args.isEmpty())
   {
      throw new IllegalArgumentException("Unknown
                                          parameters: "+ args);
   }
}
```

Output: Its C# version

```
public PersianCharFilterFactory(IDictionary<string, string> args):
base(args)
{
   if (args.Count > 0)
   {
      throw new System.ArgumentException("Unknown
                                         parameters: " + args);
   }
}
```

Figure 3: An example in the code translation dataset.

Query:

Scans through a string for substrings matched some patterns.

Gold Code:

```
def matchall(text, patterns):
   ret = []
   for pattern in patterns:
      match = re.findall(pattern, text)
      ret += match
   return ret
```

Normalized Code:

```
def func(arg0, arg1):
   arg2 = []
   for arg3 in arg1:
      arg4 = re.findall(arg3, arg0)
      arg2 += arg4
   return arg2
```

Figure 4: An example in the CodeSearchNet AdvTest dataset.

CodeSearchNet AdvTest is a Python dataset from the CodeSearchNet [35] corpus. Each example includes a function paired with a document. We follow Husain et al. [35] to take the first paragraph of the documentation as the query for the corresponding function. To improve the quality of the dataset, we filter it by removing the following examples.

(1) Examples whose code could not be parsed into abstract syntax tree.

(2) Examples whose document tokens number is shorter than 3 or larger than 256.

(3) Examples whose document contains special tokens such as "http://".

(4) Examples whose document is empty or not written in English.

At the end of the process, we obtain a dataset with 251,820 / 9,604 / 19,210 examples for training/validation/testing. After normalizing function or variable names with special tokens, we observe that the Mean Reciprocal Rank (MRR) scores of RoBERTa [50] and CodeBERT [18] for the code search task on the CodesearchNet [35] dataset drop from 0.809 to 0.419 and from 0.869 to 0.507, respectively, in Python programming language. To better test the understanding and generalization abilities of the model, we normalize function and variable names in testing and development sets like $func$ for the function name and arg_i for the i-th variable name. Figure 4 shows an example in CodeSearchNet AdvTest dataset. The task aims to search source codes from candidates for a natural language query. In contrast to the testing phase of previous works [18, 35] that only involved 1,000 candidates, we use the entire testing set for each query, which makes **CodeSearchNet AdvTest** dataset more difficult. The training set for this task comes from the filtered CodeSearchNet dataset [35].

WebQueryTest: Most code search datasets use code documentations or questions from online communities for software developers as queries, but these are different from real user search queries. To fix this discrepancy, we provide WebQueryTest, a testing set of real code search for Python. The problem is formulated as a binary classification task and as a complementary setting to the retrieval scenario. Given a pair of query and code function, a model needs to classify whether the code function can answer the query or not.

The data creation process can be divided into two stages: data collection and annotation. We first collect real user queries from the web query logs of a commercial search engine and we keep the queries with "python". Inspired by Yan et al. [91], we design some heuristics based on keyword exact matching to filter out queries without the code search intent. Then we select candidate codes for each query from the Python validation and testing sets in CodeSearchNet. To shrink the candidates to be annotated for each query, we select the top two functions with the highest query-code similarity computed by a CodeBERT-based code retrieval model, which is trained on 148K automated-minded Python Stack Overflow Question-Code (StaQC) [92] with the default parameters provided by Feng et al. [18].

We use a two-stage annotation schema to label each instance. The first step is to judge whether the query has a code-search intent. Instances labeled as "-1" are those without code search intent. The second step is to assess whether the code (with its documentation) can answer the query. Instances labeled as "1" are those where the code can answer the query. Otherwise, they are labeled as "0". Two examples are illustrated in Figure 5. We invite 13 developers proficient in Python to label 1,300 instances, with each annotator dealing with 100 of them. Discussions are allowed during annotation. Finally, the numbers of instances with labels -1, 0 and 1 are 254, 642 and 422, respectively. Since we are more interested in query-code matching, we include only the categories 0 and 1 in our final test set. The training and validation sets we use for this task are from the original CodeSearchNet dataset [35].

3.7 Code repair

Code repair aims to fix bugs in the code automatically. We use the dataset released by Tufano et al. [75]. The source is buggy Java functions, whereas the target is the corresponding fixed functions. To build this dataset, they first download every public GitHub event

Query: ***python measure distance between 2 points***

Code:
```
def vector_distance(a, b):
    """ Euclidean distance between two vectors """
    a = np.array(a)
    b = np.array(b)
    return np.linalg.norm(a - b)
```

Label: 1

Query: ***how to append object in a specific index in list python***

Code:
```
def append(self, item):
    """ append item and print it to stdout """
    print(item)
    super(MyList, self).append(item)
```

Label: 0

Figure 5: Two examples in the WebQueryTest dataset.

between March 2011 and October 2017 from GitHub Archive[11] and use the Google BigQuery APIs to identify all Java-file commits having a message containing the patterns [21]: ("fix" or "solve") and ("bug" or "issue" or "problem" or "error"). For each bug-fixing commit, they extract the source code before and after the fixing process by using the GitHub Compare API[12] to collect the buggy (pre-commit) and the fixed (post-commit) codes. Subsequently, they normalize all the names of the variables and custom methods, which greatly limits the vocabulary size and enables the model to focus on learning bug-fixing patterns. Then, they filter out the pairs that contain lexical or syntactic errors in either the buggy or fixed code, as well as the pairs with more than 100 atomic AST modification actions between the buggy and the fixed versions. To achieve this, they employ the GumTree Spoon AST Diff tool [17]. Finally, they divide the whole dataset into two subsets (*small* with tokens ≤ 50 and *medium* with tokens > 50 and ≤ 100) based on the code length. For the *small* subset, the numbers of training, development, and test samples are 46,680, 5,835, and 5,835, respectively. For the *medium* subset, the numbers are 52,364, 6,545, and 6,545, respectively.

3.8 Text-to-code generation

To carry out this task, we use CONCODE [38], a widely used code generation dataset, which is collected from about 33,000 Java projects on GitHub. It contains 100,000 examples for training and 4,000 examples for validation and testing. Each example is a tuple consisting of NL descriptions, code environments and code snippets. The dataset is tasked with generating class member functions from natural language descriptions (Javadoc-style method comments) and class environments. Class environment is the programmatic context provided by the rest of the class, including other member variables and member functions in the class.

3.9 Code summarization

For code summarization, we use the CodeSearchNet dataset [35], which includes six programming languages; i.e., Python, Java, JavaScript, PHP, Ruby, and Go. The data comes from publicly available open-source non-fork GitHub repositories and each documentation is the first paragraph. We observe that some documents contain content unrelated to the function, such as a link "http://..." that refers to external resources and an HTML image tag "<img ...>" that inserts an image. Therefore, we filter the dataset to improve its quality with the same four rules mentioned in Section 3.6.

The statistics about the filtered CodeSearchNet dataset used in CodeXGLUE are listed in Table 3.

Table 3: Data statistics about the filtered CodeSearchNet dataset for the code summarization task.

Language	Training	Dev	Testing
Go	167,288	7,325	8,122
Java	164,923	5,183	10,955
JavaScript	58,025	3,885	3,291
PHP	241,241	12,982	14,014
Python	251,820	13,914	14,918
Ruby	24,927	1,400	1,261

3.10 Documentation translation

Documentation translation aims to translate code documentation automatically from one natural language (e.g., English) to another natural language (e.g., Chinese), as shown in Figure 7. The dataset we use is crawled from Microsoft Documentation[13], including software and code description documents in different languages. We focus on low-resource language pairs, where parallel data is scarce, and introduce multilingual machine translation tasks, e.g., English $\Leftrightarrow$ Latvian, Danish, Norwegian, and Chinese. To improve the data quality, we filter the corpus by removing the following examples.

(1) Pairs whose source sentence is the same as the target sentence;

(2) Pairs whose length of source language or target language is less than three words;

(3) Pairs whose length ratio between source and target languages is larger than three;

(4) Pairs whose word alignment ratio computed by fast_align[14] is less than 0.6.

The final training data includes 43K, 19K, 44K, and 50K sentence pairs for English $\Leftrightarrow$ Latvian, English $\Leftrightarrow$ Danish, English $\Leftrightarrow$ Norwegian, and English $\Leftrightarrow$ Chinese, respectively. In addition, each language pair has 1K development and test sentence pairs, respectively.

4 BASELINE SYSTEMS

We provide three types of baseline models to perform the previously mentioned tasks, including a BERT-style pretrained model (in this case, CodeBERT), which supports program understanding problems, a GPT-style pretrained model called CodeGPT that helps us solve completion and generation problems, and an Encoder-Decoder

[11] https://www.gharchive.org/
[12] https://developer.github.com/v3/repos/commits/#compare-two-commits
[13] https://docs.microsoft.com/, whose document is located at https://github.com/MicrosoftDocs/.
[14] https://github.com/clab/fast_align.

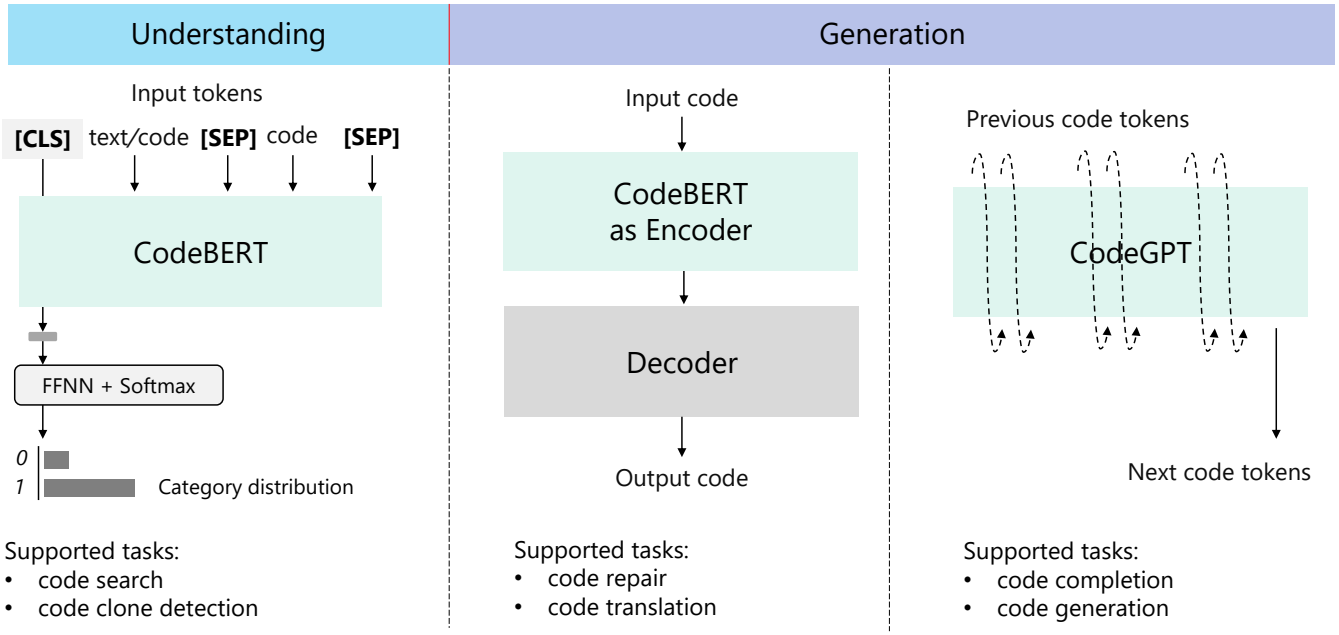

Figure 6: Three pipelines, including CodeBERT, CodeGPT, and Encoder-Decoder, are provided.

Input (English):

Multinomial Logistic Regression (Softmax regression) is used to compute the probabilities of several possible outcomes in classification problems.

Output (Chinese):

多项式逻辑回归（Softmax回归）用于计算分类问题中几种可能结果的概率。

Figure 7: An English-to-Chinese example in the documentation translation dataset.

framework that tackles sequence-to-sequence generation problems. An illustration of these three pipelines is shown in Figure 6.

4.1 CodeBERT

To carry out code understanding tasks like clone detection, defect detection, cloze test, and code search, we use CodeBERT [18] as our encoder. This is a bimodal pretrained model based on Transformer with 12 layers, 768 dimensional hidden states, and 12 attention heads for programming language (PL) and natural language (NL). Feng et al. [18] pretrain CodeBERT by masked language modeling and replaced token detection objectives on the CodeSearchNet dataset [35], which includes 2.4M functions with document pairs for six programming languages. The model supports different types of the sequence input like text/code and code/code with a special token [*CLS*] in front of the sequence and a special symbol [*SEP*] to split two kinds of data types.

The model is publicly available at https://huggingface.co/microsoft/codebert-base.

4.2 CodeGPT

We provide CodeGPT, which is a Transformer-based language model pretrained on programming language (PL), to support the code completion and the text-to-code generation tasks. CodeGPT has the same model architecture and training objective of GPT-2 [59], which consists of 12 layers of Transformer decoders. More model settings are listed in Table 4. We pretrain monolingual models on Python and Java corpora from the CodeSearchNet dataset [35], which includes 1.1M Python functions and 1.6M Java methods. Each function in training dataset has a function signature and a function body. Some functions also contain a natural language documentation.

We train two CodeGPT models for each programming language. One model is pretrained from scratch, so that the BPE (byte pair encoder) [67] vocabulary is newly obtained on the code corpus and the model parameters are randomly initialized. The other model is a domain-adaptive one, which uses the GPT-2 model as the starting point and is continually trained on the code corpus. As a result, the second model has the same GPT-2 vocabulary and natural language understanding ability. We refer to this model as **CodeGPT-adapted**, and regard it as the default one for the code completion and text-to-code generation tasks. Both models are publicly available at https://huggingface.co/microsoft/CodeGPT-small-java and https://huggingface.co/microsoft/CodeGPT-small-java-adaptedGPT2. [15]

[15] Replace "java" with "py" for models pre-trained on python dataset.

Table 4: Parameters of CodeBERT and CodeGPT models.

	CodeBERT	CodeGPT
Number of layers	12	12
Max length of position	512	1,024
Embedding size	768	768
Attention heads	12	12
Attention head size	64	64
Vocabulary size	50,265	50,000
Total number of parameters	125M	124M

4.3 Encoder-Decoder

For sequence-to-sequence generation problems like code repair, code translation, code summarization, and documentation translation, we provide an Encoder-Decoder framework. We initialize the encoder using CodeBERT [18] and use a randomly initialized Transformer with 6 layers, 768 dimensional hidden states and 12 attention heads as the decoder in all settings.

5 EXPERIMENT

In this section, we report accuracy numbers of the baseline systems on 10 tasks. We will also show how long it takes to train the model and to do inference on the model.

5.1 Clone Detection

Setting. We use the **BigCloneBench** and **POJ-104** datasets for clone detection. The task of the **BigCloneBench** dataset is formulated as a binary classification to predict whether a given pair of codes has the same semantics, with the F1 score used as the evaluation metric. The task of the **POJ-104** dataset aims to retrieve 499 codes for a given code from the development/test set for validation/testing, with the Mean Average Precision (MAP) as the evaluation metric. The overall score of the clone detection task is the average value of F1 and MAP scores.

Table 5: Results on the clone detection task.

	BigCloneBench	POJ-104	
Model	F1	MAP	Overall
RtvNN	1.0	-	-
Deckard	3.0	-	-
CDLH	82.0	-	-
ASTNN	93.0	-	-
FA-AST-GMN	95.0	-	-
TBCCD	95.0	-	-
code2vec*	-	1.98	-
NCC*	-	54.19	-
Aroma*	-	55.12	-
MISIM-GNN*	-	82.45	-
RoBERTa	94.9	79.96	87.4
CodeBERT	**96.5**	**84.29**	**90.4**

Results. Results achieved by different models are shown in Table 5. **RtvNN** [89] trains a recursive autoencoder to learn representations for AST. **Deckard** [39] computes vectors for structural information within ASTs and uses a Locality Sensitive Hashing (LSH) [14] to cluster similar vectors. **CDLH** [88] learns representations of code fragments via AST-based LSTM. **ASTNN** [97] uses RNNs to encode AST subtrees for statements. It feeds the encodings of all statement trees into an RNN to learn representation for a program. **FA-AST-GMN** [84] uses GNNs over a flow-augmented AST to leverage explicit control and data flow information. **TBCCD** [96] proposes a position-aware character embedding and uses tree-based convolution to capture both the structural information of a code fragment from its AST and lexical information from code tokens. **Code2vec** [6] learns representations of code snippets by aggregating multiple syntactic paths into a single vector. **NCC** [7] encodes programs by leveraging both the underlying data flow and control flow of the programs. **Aroma** [51] is a code recommendation engine that takes a partial code snippet and recommends a small set of succinct code snippets that contain the query snippet. **MISIM-GNN** [93] learns a structural representation of code from context-aware semantic structure designed specifically to lift semantic meaning from the code syntax.

In this experiment, we use pretrained models like **RoBERTa** [50] and **CodeBERT** [18] to encode source code and take the representation to calculate semantic relevance of two codes through a feed forward network or inner product. Although CodeBERT does not leverage code structure that has proven to be effective in terms of code similarity measure [7, 84, 88, 93, 97], the model still performs better than RoBERTa on the task of clone detection, achieving the overall score of 90.4. These experimental results demonstrate that pretraining is useful for clone detection. There is room for further improvement if code structure is further leveraged.

5.2 Defect Detection

Setting. We use the dataset mentioned in Section 3.2 for defect detection, which aims to predict whether a source code contains defects that may be used to attack software systems. The evaluation metric is accuracy score. We use the CodeBERT baseline to encode source code and take the representation of source code to calculate the probability of being exposed to vulnerabilities.

Results. Table 7 shows the results of the models we implemented. We use Bidirectional LTSM (**BiLTSM**) [32], **TextCNN** [43], **RoBERTa** [50], and **CodeBERT** [18] to encode the representation of a source code, respectively. Then, a two-layer feed forward network followed by a softmax layer is used to calculate the probability of encountering vulnerabilities. As shown in the results, CodeBERT achieve a 62.1 accuracy score, resulting in state-of-the-art performance. However, the improvement achieved by the pretrained models is limited compared with **TextCNN**. A potential direction to improve these pretrained models is to incorporate information from code structures such as Abstract Syntax Tree, data flow, control flow, etc.

5.3 Cloze test

Setting. We use CT-all and CT-maxmin datasets for the cloze test task. Models are expected to predict the masked code token by leveraging documentation and the context of code. Accuracy

Table 6: Results on the cloze test task.

Model	CT-all						CT-maxmin						Overall
	Ruby	JS	Go	Python	Java	PHP	Ruby	JS	Go	Python	Java	PHP	
RoBERTa	47.44	59.96	40.77	54.35	50.73	60.16	73.68	64.71	71.71	59.18	59.75	69.78	62.45
CodeBERT(MLM)	**80.17**	**81.77**	**83.31**	**87.21**	**80.63**	**85.05**	**86.84**	**86.40**	**90.79**	**82.20**	**90.46**	**88.21**	**85.66**

Table 7: Results on the defect detection task.

Model	Accuracy
BiLSTM	59.37
TextCNN	60.69
RoBERTa	61.05
CodeBERT	**62.08**

are reported for each language, with the macro-average accuracy scores for all languages as the overall evaluation metric.

Results. Table 6 shows the results on the CT-all and CT-maxmin datasets. We report the performance of RoBERTa [50] and CodeBERT (Masked Language Modeling, MLM) [18], which is initialized with RoBERTa and further trained with the masked language modeling objective. The results demonstrate that CodeBERT performs better than RoBERTa that only learns from natural language.

5.4 Code completion

Setting. We use the **PY150** and **Github Java Corpus** datasets for token-level and line-level code completion tasks. The token-level task is to predict the next token given context of previous tokens, and predictions are evaluated according to token-level accuracy; whereas the line-level task entails the completion of a whole-line of code, and the quality of the code is evaluated through the metrics known as exact match accuracy and Levenshtein edit similarity [72]. Levenshtein edit similarity measures how many single character edits are required to transform one string into another. This is a critical evaluation metric for the code completion scenario as it measures how much effort it takes for developers to correct an error in the code. The score on each dataset is the average value of the accuracy on token-level completion and the edit similarity on line-level completion. The overall score of code completion task is calculated by averaging the scores on both datasets.

Results. Table 8 shows the results of all models on both datasets. We fine-tune **LSTM** [32], **Transformer** [77], **GPT-2** [59], **CodeGPT** and **CodeGPT-adapted** to generate following tokens. CodeGPT and CodeGPT-adapted models are described in Section 4.2. CodeGPT-adapted achieves a state-of-the-art performance with the overall score of 71.28.

5.5 Code search

Setting. We use the **CodeSearchNet AdvTest** and **WebQueryTest** datasets mentioned in Section 3.6 for code search. To improve efficiency, we separately encode text and code to perform code search. For the **CodeSearchNet AdvTest** dataset, the task is to find the most relevant code from a collection of candidates given a query and it is evaluated through the Mean Reciprocal Rank (MRR) metric. For the **WebQueryTest** dataset, the task is formulated as a binary classification to predict whether a code can answer a given query and we use the F1 and accuracy scores as evaluation metrics. The overall score for code search is the average of the values recorded for the two subtasks.

Results. Table 9 presents the results on the CodeSearchNet AdvTest and WebQueryTest datasets. We report the performance of RoBERTa [50] and CodeBERT [18]. The table shows that CodeBERT performs better than RoBERTa.

5.6 Text-to-code generation

Setting. We use the CONCODE dataset for text-to-code generation. Models are expected to generate source codes of Java class member functions, given natural language descriptions and class environments. We report the exact match accuracy, the BLEU score [56], and the CodeBLEU score [65]. We use the CodeBLEU score as the overall evaluation metric.

Results. Table 10 presents the results on the CONCODE test set. **Seq2Seq** [70] is an RNN-based sequence to sequence model. **Seq2Action + MAML** [26] combines a context-aware retrieval model with model-agnostic meta-learning (MAML). **Iyer-Simp + 200 idoms** [36] extracts code idioms and applies idioms-based decoding. We also report the performance of pretrained models, including **GPT-2** [59], **CodeGPT**, and **CodeGPT-adapted**. CodeGPT-adapted achieves the CodeBLEU score of 27.74, resulting in a state-of-the-art performance.

5.7 Code translation

Setting. We use the dataset we build as described in Section 3.5. The dataset contains matching samples of Java and C# functions. We report the exact match accuracy, the BLEU score [56] and the CodeBLEU score [65] on this task. CodeBLEU is used as the overall evaluation metric.

Results. Table 12 shows the results of models on both translation directions. The **Naive** method directly copies the source code as the translation result. **PBSMT** is short for phrase-based statistical machine translation [44]. **Transformer** uses the same number of layers and hidden size as the pretrained models. The table shows that Transformer initialized with CodeBERT and fine-tuned with the matching sample pairs produces the best result.

5.8 Code repair

Setting. We use the dataset originally released by Tufano et al. [75], which is described in Section 3.7. The dataset contains two

Table 8: Results on the code completion task.

Model	PY150			Github Java Corpus			Overall
	token-level	line-level		token-level	line-level		
	Accuracy	EM	Edit Sim	Accuracy	EM	Edit Sim	
LSTM	58.00	17.93	50.05	56.02	10.30	41.55	51.41
Transformer	73.26	36.65	67.51	64.16	15.33	50.39	63.83
GPT-2	74.22	38.55	68.94	74.89	24.30	60.70	69.69
CodeGPT	74.93	39.11	69.69	76.45	25.30	61.54	70.65
CodeGPT-adapted	**75.11**	**39.65**	**69.84**	**77.13**	**26.43**	**63.03**	**71.28**

Table 9: Results on the code search task.

	AdvTest	WebQueryTest		
Model	MRR	F1	Accuracy	Overall
RoBERTa	18.33	57.49	40.92	33.63
CodeBERT	**27.19**	**58.95**	**47.80**	**40.28**

Table 10: Results on the text-to-code generation task.

Model	EM	BLEU	CodeBLEU
Seq2Seq	3.05	21.31	17.61
Seq2Action+MAML	10.05	24.40	20.99
Iyer-Simp+200 idoms	12.20	26.60	-
GPT-2	17.35	25.37	22.79
CodeGPT	18.25	28.69	25.69
CodeGPT-adapted	**20.10**	**32.79**	**27.74**

subsets established according to the length of the Java functions: $small \leq 50$ and $50 < medium \leq 100$. We report the exact match accuracy, the BLEU score [56] and the CodeBLEU score [65] on this task. The exact match accuracy is used as the overall evaluation metric.

Results. The **Naive** method directly copies the buggy code as the repair result. As for **Transformer**, we use the same number of layers and hidden size as the pretrained models. With regard to the **CodeBERT** method, we initialize the Transformer encoder with pretrained CodeBERT model and randomly initialize the parameters of the decoder and the source-to-target attention. Then we use the training data to fine-tune the whole model. As shown in the table, Transformer with CodeBERT initialization achieves the best performance among all models.

5.9 Code Summarization

Setting. We use the dataset mentioned in Section 3.9 for code summarization. To evaluate the models, we follow Feng et al. [18], who use smoothed BLEU score [49] as evaluation metric, because this is suitable for evaluating short documents. We use the encoder-decoder pipeline to tackle this problem. The max length of input and inference are set as 256 and 128, respectively. We use the Adam optimizer to update the models' parameters. The learning rate and the batch size are 5e-5 and 32, respectively. We tune the hyper-parameters and perform early stopping on the development set.

Results. Table 13 shows the results achieved by different models in code summarization. **Seq2Seq** is an RNN-based sequence to sequence model. **Transformer** and **RoBERTa** use the same setting as **CodeBERT**, but the encoder is initialized randomly and by RoBERTa [50], respectively. All models use Byte Pair Encoding (BPE) [66] vocabulary. In this experiment, CodeBERT obtains a 1.3% gain in the BLEU score over RoBERTa and achieves the state-of-the-art performance on six programming languages.

5.10 Documentation translation

Setting. We use the Microsoft Docs dataset for text-to-text translation tasks, which focus on low-resource multilingual translation between English (EN) and other languages, including Latvian (LA), Danish (DA), Norwegian (NO), and Chinese (ZH). Following Johnson et al. [40], we train a single multilingual model as our baseline. To distinguish between different translation pairs, we add an language token (e.g., $\langle 2en \rangle$, $\langle 2zh \rangle$) at the beginning of the source sentence to indicate the target language the model should translate. We initialize the encoder of the multilingual translation model with XLM-R [13]. Models are evaluated through BLEU [56] score, and the overall score for documentation translation is the average BLEU score on the eight translation directions.

Results. Table 14 shows the results achieved by the models on eight translation directions. **Transformer Baseline** is the multilingual translation model [40]. **pretrained Transformer** initializes the encoder of **Transformer Baseline** with XLM-R[13]. In terms of overall performance on the eight translation directions, **Transformer Baseline** and **pretrained Transformer** obtain the BLEU score of 52.67 and 66.16, respectively. Experimental results demonstrate that pretraining achieves a 13.49 improvement in BLEU score over strong baseline model. Figure 8 shows how long it takes to train the model and to do inference on the model, as well as in other tasks.

6 RELATED WORK

Benchmark datasets have been playing a central role in the growth of applied AI research. For example, the LibriSpeech [55] and the SQuAD [60] datasets drive the development of data-driven models for automatic speech recognition and reading comprehension of

Table 11: Results on the code repair task.

Method	small			medium			Overall
	BLEU	Acc	CodeBLEU	BLEU	Acc	CodeBLEU	
Naive	**78.06**	0.000	-	90.91	0.000	-	0.000
LSTM	76.76	0.100	-	72.08	0.025	-	0.063
Transformer	77.21	0.147	73.31	89.25	0.037	81.72	0.092
CodeBERT	77.42	**0.164**	**75.58**	**91.07**	**0.052**	**87.52**	**0.108**

Table 12: Results on the code translation task.

Method	Java→C#			C#→Java			Overall
	BLEU	Acc	CodeBLEU	BLEU	Acc	CodeBLEU	
Naive	18.54	0.000	-	18.69	0.000	-	-
PBSMT	43.53	0.125	42.71	40.06	0.161	43.48	43.10
Transformer	55.84	0.330	63.74	50.47	0.379	61.59	62.67
RoBERTa (code)	77.46	0.561	83.07	71.99	0.579	**80.18**	81.63
CodeBERT	**79.92**	**0.590**	**85.10**	**72.14**	**0.580**	79.41	**82.26**

Table 13: Results on the code summarization task.

Model	Ruby	Javascript	Go	Python	Java	PHP	Overall
Seq2Seq	9.64	10.21	13.98	15.93	15.09	21.08	14.32
Transformer	11.18	11.59	16.38	15.81	16.26	22.12	15.56
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	**12.16**	**14.90**	**18.07**	**19.06**	**17.65**	**25.16**	**17.83**

Task	Dataset Name	Language	Training Cost	Inference Cost
Clone Detection	BigCloneBench	Java	3 hours training on P100 x2	2 hours on p100 x2
	POJ-104	C/C++	2 hours training on P100 x2	10 minutes on p100 x2
Defect Detection	Devign	C	1 hour on P100 x2	2 minutes on p100 x2
Cloze Test	CT-all	Python, Java, PHP, JavaScript, Ruby, Go	N/A	30 minutes on P100-16G x2
	CT-max/min	Python, Java, PHP, JavaScript, Ruby, Go	N/A	1 minute on P100-16G x2
Code Completion	PY150	Python	25 hours on P100 x2	30 minutes on P100 x2
	GitHub Java Corpus	Java	2 hours on P100 x2	10 minutes on P100 x2
Code Repair	Bugs2Fix	Java	24 hours on P100 x2	20 minutes on P100 x2
Code Translation	CodeTrans	Java-C#	20 hours on P100 x2	5 minutes on P100 x2
NL Code Search	CodeSearchnet, AdvTest	Python	5 hours on P100 x2	7 minutes on p100 x2
	CodeSearchNet, WebQueryTest	Python	5 hours on P100 x2	1 minute on P100 x2
Text-to-Code Generation	CONCODE	Java	30 hours on P100 x2	20 minutes on P100 x2
Code Summarization	CodeSearchNet	Python, Java, PHP, JavaScript, Ruby, Go	On average, 12 hours for each PL on P100 x2	On average, 1 hour for each PL on p100 x2
Documentation Translation	Microsoft Docs	English-Latvian/Danish/Norwegian/Chinese	30 hours on P100x2	55 minutes on P100x2

Figure 8: Training and inference time costs for each task, evaluated on two P100 GPUs.

Table 14: Results on the documentation translation task.

Task	Transformer Baseline	pretrained Transformer
EN → DA	53.31	67.09
EN → LA	37.85	51.92
EN → NO	53.84	68.00
EN → ZH	59.90	70.60
DA → EN	58.73	67.02
LA → EN	50.37	68.30
NO → EN	57.73	71.84
ZH → EN	50.00	64.47
Overall	52.67	66.16

text, respectively. With the growing demand for testing models' generalization ability on a wide range of applications, researchers have created or assembled datasets that cover many tasks. Representative samples of these datasets include ImageNet [15] for computer vision, GLUE [81] for natural language understanding, XTREME [33] and XGLUE [48] for cross-lingual natural language processing. To the best of our knowledge, CodeXGLUE is the first diversified benchmark dataset that can be applied to various code intelligence problems.

Many tasks related to machine learning for software engineering [1] have sufficient amount of data to support the development of data-driven methods, but are not covered by CodeXGLUE. We plan to extend to these tasks in the future. For example, the idiom mining task [5, 36] is to extract code idioms, which are syntactic fragments that recur across software projects and serve a single semantic purpose [5]. Bug localization [27, 61, 76] is to point the error location when a program fails tests. The test case generation task [22, 74] is to generate unit test cases automatically. The program synthesis [20, 45, 53, 64, 68, 79, 98] extends the text-to-code generation task aims to generate programs from a specification [24], such as pseudocode, natural language description, and input/output examples.

7 CONCLUSION

With CodeXGLUE, we seek to support the development of models that can be applied to various program understanding and generation problems, with the goal of increasing the productivity of software developers. We encourage researchers to participate in the open challenge to make progress in code intelligence. Moving forward, we are planning to extend CodeXGLUE to more programming languages and downstream tasks while continuing to develop advanced pretrained models by exploring new model structures, introducing new pretraining tasks, using different types of data, and more.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.

[4] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 207–216.

[5] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 472–483.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems*. 3585–3597.

[8] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 2933–2942.

[9] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 213–222. https://doi.org/10.1145/1595696.1595728

[10] L. Büch and A. Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 95–104. https://doi.org/10.1109/SANER.2019.8668039

[11] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*. 2547–2557.

[12] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. *arXiv preprint arXiv:2010.03150* (2020).

[13] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Unsupervised cross-lingual representation learning at scale. *arXiv preprint arXiv:1911.02116* (2019).

[14] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[19] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824* (2018).

[20] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[21] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 23–32.

[22] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[23] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 933–944. https://doi.org/10.1145/3180155.3180167

[24] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv preprint arXiv:2009.08366* (2020).

[26] Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2019. Coupling Retrieval and Meta-Learning for Context-Dependent Semantic Parsing. *arXiv preprint arXiv:1906.07108* (2019).

[27] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural Attribution for Semantic Bug-Localization in Student Programs. In *Advances in Neural Information Processing Systems*. 11884–11894.

[28] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[29] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[30] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International Conference on Learning Representations*.

[31] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[32] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[33] Junjie Hu, Sebastian Ruder, Aditya Siddhant, Graham Neubig, Orhan Firat, and Melvin Johnson. 2020. Xtreme: A massively multilingual multi-task benchmark for evaluating cross-lingual generalization. *arXiv preprint arXiv:2003.11080* (2020).

[34] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 2269–2275.

[35] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[36] Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086* (2019).

[37] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[38] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

[39] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[40] Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, et al. 2017. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics* 5 (2017), 339–351.

[41] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/2661136.2661148

[42] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. *arXiv preprint arXiv:2003.07914* (2020).

[43] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[44] Philipp Koehn, Franz J Och, and Daniel Marcu. 2003. *Statistical phrase-based translation*. Technical Report. UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST.

[45] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems*. 11906–11917.

[46] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. *arXiv preprint arXiv:2006.03511* (2020).

[47] 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. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[48] Yaobo Liang, Nan Duan, Yeyun Gong, Ning Wu, Fenfei Guo, Weizhen Qi, Ming Gong, Linjun Shou, Daxin Jiang, Guihong Cao, et al. 2020. Xglue: A new benchmark dataset for cross-lingual pre-training, understanding and generation. *arXiv preprint arXiv:2004.01401* (2020).

[49] Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. 501–507.

[50] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[51] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[52] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 1287–1293.

[53] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834* (2015).

[54] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.

[55] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 5206–5210.

[56] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[57] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[58] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

[59] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[60] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[61] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

[62] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *ACM SIGPLAN Notices* (2016), 731–747.

[63] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

[64] Scott Reed and Nando De Freitas. 2015. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279* (2015).

[65] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[66] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[67] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1715–1725.

[68] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*. Springer, 398–414.

[69] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203* (2019).

[70] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[71] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[72] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. *arXiv preprint arXiv:2005.08025* (2020).

[73] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735.

[74] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers. *arXiv preprint arXiv:2009.05617* (2020).

[75] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[76] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720* (2019).

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[78] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 310–325. https://doi.org/10.1145/2908080.2908110

[79] Murali Vijayaraghavan, Chaudhuri Swarat, and Jermaine Chris. 2017. Bayesian Sketch Learning for Program Synthesis. *CoRR.—-2017.—-Vol. abs/1703.05698.—-1703.05698* (2017).

[80] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[81] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[82] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 708–719.

[83] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

[84] 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 *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[85] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranS^ 3: A Transformer-based Framework for Unifying Code Summarization and Code Search. *arXiv preprint arXiv:2003.03238* (2020).

[86] Yanlin Wang, Lun Du, Ensheng Shi, Yuxuan Hu, Shi Han, and Dongmei Zhang. 2020. CoCoGUM: Contextual Code Summarization with Multi-Relational GNN on UMLs.

[87] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems*. 6563–6573.

[88] 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.

[89] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[90] 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. *arXiv preprint arXiv:2004.09015* (2020).

[91] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang. 2020. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 344–354. https://doi.org/10.1109/SANER48275.2020.9054840

[92] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *Proceedings of the 2018 World Wide Web Conference*. 1693–1703.

[93] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marucs, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy Mattson, Tim Kraska, Pradeep Dubey, et al. 2020. MISIM: An End-to-End Neural Code Similarity System. *arXiv preprint arXiv:2006.05265* (2020).

[94] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *International Conference on Mining Software Repositories (MSR)*. ACM, 476–486. https://doi.org/10.1145/3196398.3196408

[95] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[96] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 70–80.

[97] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[98] Ruiqi Zhong, Mitchell Stern, and Dan Klein. 2020. Semantic Scaffolds for Pseudocode-to-Code Generation. *arXiv preprint arXiv:2005.05927* (2020).

[99] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*. 10197–10207.