An Empirical Comparison of Pre-Trained Models of Source Code

Changan Niu*, Chuanyi Li*, Vincent Ng†, Dongxiao Chen*, Jidong Ge*, Bin Luo*
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Email: niu.ca@outlook.com, lcy@nju.edu.cn, MF21320014@smail.nju.edu.cn, gjd,luobin@nju.edu.cn
†Human Language Technology Research Institute, University of Texas at Dallas, Richardson, Texas, USA
Email: vince@hlt.utdallas.edu

***Abstract*—While a large number of pre-trained models of source code have been successfully developed and applied to a variety of software engineering (SE) tasks in recent years, our understanding of these pre-trained models is arguably fairly limited. With the goal of advancing our understanding of these models, we perform the first systematic empirical comparison of 19 recently-developed pre-trained models of source code on 13 SE tasks. To gain additional insights into these models, we adopt a recently-developed 4-dimensional categorization of pre-trained models, and subsequently investigate whether there are correlations between different categories of pre-trained models and their performances on different SE tasks.**

***Index Terms*—Pre-training of Source Code, AI for SE**

I. Introduction

Despite the successful application of deep learning to various Artificial Intelligence (AI) subfields such as natural language processing (NLP) and computer vision in recent years, a large amount of annotated training data is typically needed to train the millions or even billions of network parameters in a deep neural model. For many learning tasks, including those in software engineering (SE), obtaining annotated data is costly. To address this data annotation bottleneck, NLP researchers have come up with an idea that can arguably be considered one of the most exciting developments in recent deep learning research, namely *pre-training* [1]–[4]. Rather than training a model from scratch (i.e., with randomly initialized network weights), which typically requires a lot of task-specific annotated data, one can first pre-train it on one or more so-called self-supervised tasks (i.e., tasks for which annotated data can be automatically generated and therefore large amounts of training data are readily available) so that its weights encode general linguistic and commonsense knowledge about language, and then the resulting pre-trained model can be fine-tuned to learn the target task using (a potentially small amount of) task-specific annotated training data in the usual supervised manner. A large number of pre-trained models of natural language (PTM-NLs) have been developed and widely used in NLP, such as BERT [5], XLNet [6], RoBERTa [7], ELECTRA [8], GPT-2 [9], T5 [10], and BART [11].

Soon thereafter, pre-trained models have made their way into SE research. Initial applications of pre-trained models in SE have primarily involved retraining PTM-NLs on source code [12]–[16]. Nevertheless, employing the resulting retrained models (henceforth PTM-Cs) for SE tasks is not ideal, as there are code-specific characteristics that may not be properly taken into account by these models, such as the syntactic [17], [18] and semantic structures [19] inherent in source code [20]. Consequently, SE researchers have developed a number of pre-trained models of source code (henceforth CodePTMs) that take into account code-specific characteristics in the past few years [21]–[26].

Despite the fact that a large number of CodePTMs have been successfully developed and applied to a variety of SE tasks in recent years, our understanding of CodePTMs is arguably fairly limited. Currently, only one survey of pre-trained models of source code is available from Niu et al. [27], but it just performs a summary and analysis from the results reported by the origin model. While pre-trained models are task-agnostic and therefore can be applied to different SE tasks by design, virtually all CodePTMs have been evaluated on only a handful of SE tasks. For instance, TreeBERT [28], has only been evaluated on code summarization and method name generation. This is by no means ideal: without knowing how TreeBERT performs on the remaining SE tasks, we do not know whether it can achieve state-of-the-art results on any of those tasks. This in turn implies that our understanding of these models could be partial and that the current state-of-the-art could have been very different had we evaluated the existing models on most, if not all, of the available SE tasks. Even when two pre-trained models are being evaluated on the same SE task, a head-to-head comparison of these models could still be made complicated if they are evaluated on different datasets available for this task [29].

With the goal of advancing our understanding of existing pre-trained models of source code, we conduct the first systematic empirical comparison of 19 recently-developed CodePTMs on 13 popular SE tasks. To gain additional insights into these CodePTMs, we employ a recently-developed four-dimensional categorization of CodePTMs [27] to categorize existing the 19 CodePTMs used in our study, and subsequently investigate whether there are correlations between categories of CodePTMs and their performances on SE tasks.

II. Experimental Setup

A. SE Tasks

Table I enumerates the 13 SE tasks we will use in our comparative experiments. These are also the SE tasks that

TABLE I
DETAILS OF EVALUATION TASKS, DATASETS AND METRICS.

Type	I-O	Task	Ab.	ID: Dataset	Metrics
Und.	C-V	Defect Detection	DD	D1: Devign [30]	Acc
				D2: DeepBugs [31]	Acc
				V1: VMC [12]	Acc
				W1: WBO [12]	Acc
				S3: SO [12]	Acc
		Clone Detection	CD	B1: BigCloneBench [32]	F1
				C2: CLCDSA [33]	F1
		Exception Type	ET	K1: Kanade et al. [12]	Acc
	C-C	Code-to-Code Retrieval	CR	P1: POJ-104 [34]	MAP
				C2: CLCDSA [33]	MRR
	NL-C	Code Search	CS	C3: CodeSearchNet (Filtered) [35]	MRR
				A1: AdvTest [35]	MRR
		Code Question Answering	QA	C4: CoSQA [36], WebQueryTest [35]	MRR
				F1: FDM [12]	Acc
Gen.	C-C	Code Translation	CT	C5: CodeTrans [35]	EM/B./C.B.
				T1: TransCoder [37]	CA
				C2: CLCDSA [33]	R.L
		Bug Fixing	BF	B2: BFP [38]	EM/B./C.B.
		Code Completion	CC	P2: PY150 [39]	EM/ES
				C6: CugLM [40]	EM
				S1: SLM [41]	EM
				S2: Svyatkovskiy et al. [14]	PPL
		Mutant Generation	MG	G1: GM [42]	EM/B.
		Assert Generation	AG	A3: ATLAS [43]	EM/B.
	C-NL	Code Summarization	SM	C3: CodeSearchNet (Filtered) [35]	B.
				A2: Attn2FC [44]	B.
				D3: DeepCom [45]	B.
				P2: PY150 [39]	EM
				C7: code2seq [46]	EM
				T2: TL-CodeSum [47]	B.
				M1: Miceli-Barone and Sennrich [48]	B.
	NL-C	Code Generation	CG	C8: CONCODE [49]	EM/B./C.B.

are typically used to evaluate pre-trained models of source code. Following previous work [27], in the first two columns, we classify each task along two dimensions: (1) whether the task concerns *understanding* (**Und.**) or *generation* (**Gen.**); and (2) the type of input assumed by the task and the type of produced output (**I-O**), where **C**, **NL**, and **V** denote code, natural language, and extracted/predicted value, respectively. Table I also shows the abbreviation (Ab.), the dataset, and the main evaluation metrics for each task.

To make the number of experiments manageable in our comparison, whenever there are multiple datasets for a task, we choose the most popular one (shown in Gray in Table I) except for Code Search, where we chose A1 over C3 since A1 is the filtered version of C3 and the results on A1 is more reflective of the generalization ability of a model [35].

B. Evaluation Metrics

For each SE task, we will perform evaluations using the standard metrics listed in the last column of Table I. For classification and retrieval tasks, metrics such as Acc (Accuracy), F1, Precision (P), Recall (R), Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) are used. For generation tasks, metrics developed in the NLP community such as perplexity (PPL), Levenshtein edit similarity (ES) [14], BLEU (B.) [50], as well as variants developed in the SE community, such as CodeBLEU (C.B.) [51], are used. Moreover, some generation tasks have also used variants of Accuracy for evaluation, one of which indicates whether the sequence generated by the model exactly matches (EM) the correct answer, and the other, Computational Accuracy (CA), computes the number of times the hypothesis function generates the same output as the reference when given the same inputs [37].

C. Pre-trained Models

In this subsection, we first present an overview of 26 of the PTMs that have been applied to SE tasks and then enumerate the 19 pre-trained models of source code that we will include in our empirical comparison.

1) Categorization: Table II presents an overview of 26 of the PTMs that are either commonly used and/or developed in SE. As can be seen, these PTMs can be divided into three groups: PTM-NL, PTM-C, and CodePTM. Within each group, we order them chronologically (by the date of the preprint or the official publication). To enable the reader to better understand their similarities and differences, we categorize the PTMs of source code (i.e., PTM-Cs and CodePTMs) along the four dimensions proposed by Niu et al. [27][1]:

(1) Architecture (**Arch.**). Existing network architectures can be divided into *Long Short-Term Memory* (LSTM) [65], *Transformer* (TF) [66], *Transformer-Encoder* (TE, the encoder-only portion of TF), and *Transformer-Decoder* (TD, the decoder-only portion of TF).

(2) **Modality** refers to the type of input a PTM assumes. The possible modalities include *code*, natural language (*NL*) and code *structure*. How these different modalities should be combined is determined by the underlying combination strategy, which can be *together* (+) or *standalone* (&)[2].

(3) Pre-training Tasks (**Tasks**). If more than one task is used, the tasks can be learned *jointly* (+), *sequentially* (&), or *alternately* (/)[3]. The definition of each pre-training task is given in Table III. Following Niu et al, [27], in the first column we classify these tasks into four categories according to their input modalities: (1) Code-Aware or Natural-Language-Aware (**C/NLA**) tasks, which are originated in NLP and can be applied to either NL or Code sequence to mine latent information from NL or Code; (2) Code-Aware Only (**CA**) tasks, which can only be applied to mine latent information from code text; (3) Structure-Aware Only (**SA**) tasks, which aim to learn representations of the code structure; and (4) Cross-Modal-Aware (**CMA**) tasks, which seek to acquire knowledge from multiple input modalities and are further subdivided into three categories based on which input modalities are involved, namely Code-NL (**CN**), Code-Structure (**CS**), and Code-NL-Structure (**CNS**). In the second column, we classify these tasks based on whether they are Generative (**G**, i.e., generate tokens) or Categorical (**C**, i.e., predict labels) in nature.

(4) Programming language (**PL**). We categorize code PTMs depending on whether they are pre-trained on one PL (Monolingual (*Mono*)) or multiple PLs (Multilingual (*Multi*)).

In our empirical comparison, we exclude all LSTM-based models of source code since they do not represent the state of the art, and retain all the Transformer-based models of source code shown in Table II except OSCAR, because OSCAR does not target high-level PLs. This leaves us with 19 PTMs of

[1] Note that three of these four dimensions are also applicable to PTM-NL.

[2] See the supplementary file for details.

[3] See the supplementary file for details on the different ways of pre-training a model when more than one pre-training task is involved.

TABLE II
CATEGORIZATION OF EXISTING PRE-TRAINED MODELS AND THEIR PERFORMANCE ON SE TASKS AS REPORTED IN THEIR ORIGINAL PAPERS. THE STRONGEST RESULT FOR EACH DATASET IS BOLDFACED.

Models	Arch.	Modality	Tasks	PL	Code Understanding Tasks						Code Generation Tasks						
					C-V			C-C	NL-C		C-C					C-NL	NL-C
					DD	CD	ET	CR	CS	QA	CT	BF	CC	MG	AG	SM	CG
RoBERTa [7]	TE	NL Text	MLM	-	D1:61.05	B1:94.9		P1:76.67	C3:61.7 A1:18.33	C4:60.3						C3:16.57	
GPT-2 [9]	TD	NL Text	ULM	-									P2:41.73				17.35
BART [11]	TF	NL Text	DAE	-								11.7					
T5 [10]	TF	NL Text	Seq2Seq MLM	-	D1:61.93							9.7				C3:18.35	18.65
SCELMo [52]	LSTM	Code	BiLM	Mono	D2:93.12												
CuBERT [12]	TE	Code	MLM + NSP	Mono	V1:95.21 W1:92.46 S3:93.36		79.12			F1:98.09						P2:33.48 C7:52.76 D3:17.41	
GPT-C [14]	TD	Code	ULM	Multi									S2:1.65				
C-BERT [13]	TE	Code	MLM	Mono	D1:57.4												
JavaBERT [15]	TE	Code	MLM	Mono													
CodeGPT-adapted [35]	TD	Code	ULM	Multi									P2:42.37				20.1
DeepDebug [16]	TF	Code	Seq2Seq MLM	Mono								15.05					
CodeBERT [53]	TE	Code + Doc	MLM & RTD	Multi				P1:82.67	C3:69.3	C4:65.7	C5:58.5	10.8				C3:17.83 P2:35.97 C7:56.52 D3:17.87	
GraphCodeBERT [54]	TE	Code + Doc + DFG Nodes	MLM + EP + NA	Multi		B1:97.1		P1:85.16	C3:71.3	C4:68.4	C5:59.1	13.2					
CugLM [40]	TE	Code	IMLM + NSP + ULM	Multi									C6:81.91				
DOBF [55]	TF	Code	MLM & Seq2Seq IMLM	Multi		B1:95.9			A1:38.3		T1:46.35					C3:18.65	
T5-learning [56]	TF	Code & Doc	Seq2Seq MLM	Mono								6.5		28	40.5	A2:15	
PLBART [57]	TF	Code & Post	DAE	Multi	D1:63.18	B1:97.2				C4:65.0	C5:64.8	14.10				C3:18.32	18.75
ProphetNet-Code [58]	TF	Code & Doc	FNP	Multi												C3:18.54	
CoTexT [59]	TF	Code + Doc	Seq2Seq MLM	Multi	D1:**65.99**							17.30				C3:18.38	20.1
TreeBERT [28]	TF	Code + AST Paths	TMLM + NOP	Multi												**D3:20.49** **P2:45.81** **C7:67.9**	
OSCAR [60]	TE	IR + AEI	MLM + CCL	Mono				P1:49.17									
CodeDisen [61]	LSTM	Code + AST Seq	VGVAE + CLR + PD + ACP	Multi		C2:90.0		C2:43.6			C2:50.08						
CodeT5 [62]	TF	Code + Doc	Seq2Seq MLM / IT / Seq2Seq IMLM / BDG	Multi	D1:65.78	B1:97.2				C4:67.8	C5:**66.4**	**17.79**				C3:**19.55**	22.30
SynCoBERT [63]	TE	Code + Doc + AST Seq	MLM + IT + TEP + MCL	Multi	D1:64.5	B1:**97.4**		P1:88.24	C3:74.0 A1:38.1		C5:60.85						
SPT-Code [29]	TF	Code + Names + AST Seq	CAP & MASS & MNG	Multi					C3:71.5		C5:62.18	14.2	S1:19.09			C3:15.0 T2:49.1 M1:36.1	
UniXcoder [64]	TE	AST Seq + Doc	MLM / ULM / Seq2Seq MLM / MCL / CMG	Multi		B1:95.2		P1:**90.52**	C3:**74.4** A1:**41.3**	C4:**70.1**						C3:19.30	**22.60**

TABLE III
CATEGORIZATION AND DESCRIPTION OF THE PRE-TRAINING TASKS MENTIONED IN TABLE II.

Type		O.	Task	Full Name and Description
C/NLA		G.	ULM [14]	Unidirectional LM: conditional on words that have already appeared, maximizes the conditional probability of all next words.
			FNP [58]	Future N-gram Prediction: conditional on words that have appeared, maximizes the conditional probability of all next N ($N > 1$) words.
			BiLM [52]	Bidirectional LM: apply ULM to the input and its reversion to maximize the conditional probability of each word in both directions.
			MLM [15]	Masked Language Model: predicts a certain percentage of tokens that have been randomly masked in the input. (Basic version of MLM)
			WWM [13]	Whole Word Masking: a variant of basic MLM, if parts of a word is masked, ensure all subwords/tokens in it be masked.
			MASS [29]	MAsked Seq2Seq: predicts 50% of the content that is randomly masked consecutively in the sentence in the encoder-decoder architecture.
			SMLM [56]	Seq2Seq MLM: sequentially predicts a set of token spans randomly masked in the input in the encoder-decoder framework.
			DAE [57]	Denoising Auto-Encoding: recovers the original input from the one tampered by masking, deleting, and replacing tokens, etc.
		C.	NSP [12]	Next Sentence Prediction: determines whether the two given sentences or logical lines of code appear consecutively in real world.
			RTD [53]	Replaced Token Detection: identifies whether a token in the input is a fake one that is produced by a small generator network.
CA		G.	IMLM [40]	Identifier MLM: predicts a certain percentage of identifiers that randomly masked in the code text (an adaption of basic MLM to code).
			SIMLM [62]	Seq2Seq IMLM: an adaptation of Seq2Seq MLM to source code that masks only a certain percentage of the identifiers in the code text.
		C.	IT/IP [62]	Identifier Tagging/Predicting: determines whether the input token at each position is an identifier or not via binary classification.
			CCL [60]	Code Contrastive Learning: minimizes/maximizes the distances between the representations of similar/dissimilar code snippets.
SA		C.	EP/TEP [54]	Edge Prediction: predicts the edges that are masked by randomly selecting source and target nodes in a DFG or AST.
			NOP [28]	Node Order Prediction: determines if a change occurs in an AST where the order of some randomly selected nodes are changed.
CMA	CN	G.	BDG/CMG [62]	Bimodal Dual Generation/Cross Modal Generation: generates a Natural Language/Code if Code/Natural Language is given.
			MNG [29]	Method Name Generation: produces a name for the given method body by generating sub-tokens with the decoder sequentially.
	CS	G.	CLR [61]	Cross-Language Reconstruction: generates a code snippet in one PL functionally equivalent to the given one in other PLs.
			TMLM [28]	Tree MLM: generates complete code from inputs where some terminal nodes/identifiers in ASTs/code are masked in encoder/decoder.
		C.	VGVAE [61]	vMF-Gaussian Variational Autoencoder: disentangles code semantics from code syntax under the supervision of a masked AST.
			CAP [29]	Code-AST Prediction: determines whether the given code and AST in the input correspond to each other via binary classification.
			NA [54]	Node Alignment: predicts the masked edges connecting randomly sampled nodes in a DFG and its corresponding code token.
			PD [61]	Posterior Distribution: minimizes the difference in semantics distributions of functionally equivalent code snippets in different PLs.
			ACP [61]	Attentive Code Position: predicts the node type in AST of a code token in the input through an attention mechanism.
	CNS	C.	MCL [63]	Multi-modal Contrastive Learning: an adaptation of CCL to (Code,NL)/(Code,Structure) pairs where samples are no longer code pairs.

source code in our comparison. In addition, we will present results of five models that are *not* pre-trained on source code. They include four PTMs-NL (RoBERTa, GPT-2, BART, and T5) and a vanilla Transformer model [66]. Comparing the results of these models and those obtained by the 19 PTMs of source code could shed some light on the gains that can be obtained on each SE task via pre-training on source code.

D. Implementation Details

a) The 19 PTMs of source code: According to the public availability of the artifacts, the 19 models of source code we use in our comparison can be divided into four categories:

(1) For those PTMs that have publicly available pre-trained models and tokenizers, we use them as provided. CuBERT, CodeBERT, GraphCodeBERT, DOBF, JavaBERT, CodeGPT-

adapted, T5-learning, PLBART, ProphetNet-Code, CoTexT, CodeT5, SPT-Code and UniXcoder are in this category. If more than one model is provided, we choose the "base" version consistent with the approach in the original paper.

(2) Of the remaining PTMs, if the source code and datasets are provided, we re-train them according to the setting introduced in the original papers to get the pre-trained models and the tokenizers. TreeBERT is the only model in this category.

(3) For those that have the source code but not the datasets, we collect the required datasets ourselves in the same way as the original authors did, and re-train them according to the settings in the original papers. Only CugLM is in this category.

(4) If no source code is provided, we re-implement and pre-train according to the settings (e.g., tokenizer, hyperparameters, and dataset) described in the original papers. They are GPT-C, C-BERT, DeepDebug and SynCoBERT[4].

When evaluating on a downstream SE task, each of the 19 models is fine-tuned on the training data available for that task.

b) The 5 non-PTMs: As noted above, we also include four PTMs-NL (RoBERTa, GPT-2, BART, and T5) and a vanilla Transformer model in our comparison. For the four PTMs-NL, we use their publicly available implementations. Like the 19 PTMs of source code, these five models are being fine-tuned on task-specific data [66] before applying to each downstream task.

E. Application to SE Tasks

Two aspects need to be considered while applying PTMs to SE tasks, namely, Inputs and Outputs.

1) Inputs: The inputs for different SE tasks are different. When applying a PTM to a SE task, the input of the task should be organized into a form needed by the PTM. The input of the SE tasks in Table II belongs to three types:

(1) **Using only a code snippet as input**: Tasks such as Defect Detection and Code Translation assume input that belongs to this category. Here, we follow the input representation as defined by PTMs. For example, for TreeBERT, we parse the code into an AST and encode each path in the AST before passing it to the Transformer, as described in the original paper; and for PLBART, we add a special symbol indicating the programming language, e.g., "[java]", to the input sequence.

(2) **Using only a natural language description as input**: This is used by tasks such as Code Search and Code Generation. In this case, we input the text sequence directly. But for PLBART, we follow the approach described in its paper and add a special symbol "[en]" to the input.

(3) **Using a code-code pair or a code-NL pair as input**: Tasks like Clone Detection (inputs: code-code) and Code Question Answering (inputs: code-NL) belong to this type. In this case, we prepare the inputs for the two parts separately and then concatenate them to obtain the final input representation.

2) Outputs: The output required by a SE task may not be the same as the output produced by the PTMs. Hence, additional modules or operations may be needed in order to get the output required by SE Tasks. The outputs that need to be provided by PTMs for different SE tasks can be divided into two types:

(1) **Output based on the input representation**: Among the SE tasks, Code Search and Code Question Answering use the input representation directly (to calculate the similarity between two sequences), while the others need a fully connected layer and a softmax layer to be added to obtain a probability distribution. PTMs with different architectures use different ways to get the representation vector for the input. For **TE-based** models, we use the vector that corresponds to the position of the classification symbol in the input (typically "[CLS]") as the representation vector. For **TD-based** models, we use the last time step of the output hidden state (i.e., the position of the special symbol "[endoftext]" in the input sequence). For **TF-based** models, it depends. Since T5-based models (i.e., T5, T5-learning, DeepDebug, CoTexT and CodeT5) formalize all tasks as text-to-text tasks, for *classification tasks* we map all categories to text (e.g. for a binary classification task, 0 is mapped to "false" and 1 to "true"), while for retrieval tasks, we use the output hidden state of the decoder corresponding to the "[EOS]" symbol as the representation vector. In contrast, for BART-based models (i.e., BART, PLBART and SPT-Code), we keep the input of the decoder to be the same as the input of the encoder and use the decoder hidden state of the last timestep as the representation vector. For other **TF-based** models, we only use its encoder and adopt the same method as used in the **TE-based** models.

(2) **Output based on the ultimate output sequence**: For **TE-based** models, we follow Lu et al. [35] to randomly initialize a Transformer Decoder of the same size as the model to form an encoder-decoder architecture. For **TD-based** models, we follow GPT-2 [9]: for training, we concatenate the input and output sequences using a special symbol; and for evaluation, we pass the input sequence concatenated with this special symbol into the model and use the sequence predicted by the model as the output. **TF-based** models can be applied directly to this type of tasks. The **Code Completion** task deserves special mention. Recall that it requires a model to complete the unfinished line given the previous context. However, during training, it follows the GPT-like, casual language modeling manner. This is not applicable to TE- and TF-based PTMs that adopt the encoder-decoder architecture for this task. Therefore, when training TE- and TF-based PTMs, we randomly extract the first 15%-85% of the entire sequence as input (since the input context in the test data is ensured to be at least 15% of the whole length [35]) of the encoder, and the rest is used as the input of the decoder.

F. Other Settings and Data Availability

For other settings, e.g., the hyperparameters and the optimizer, we adopt those used in the provided source code or mentioned in the original paper. If neither of the above is

[4] To verify the validity of the latter two types of models pre-trained by us, we perform fine-tuning on the downstream tasks corresponding to the original paper and use pair-wise t-tests to ensure that the difference between our results and those reported in the original papers are statistically indistinguishable. Details can be found in the supplementary materials.

available, we perform parameter tuning ourselves to maximize model performance on held-out development data[5].

III. Evaluation of PTMs: The Status Quo

The current state of research on applying PTMs, including PTMs-NL, PTMs-C, and CodePTMs, to SE tasks is somewhat unsatisfactory. To understand this status quo, we show in Table II the ID of each dataset that each PTM is evaluated on (see Table I for an explanation of the dataset IDs) and the corresponding results as reported in the original papers. To avoid overloading the reader with information, we (1) omit the dataset ID when the SE task has only one dataset; (2) report results in terms of percentage using the first evaluation metric (see Table I); and (3) average results over all data subsets when a dataset is composed of multiple subsets[6].

Below we discuss the status quo based on the results shown in Table II, focusing our discussion on PTMs of source code given that they are the focus of this paper.

a) Code Understanding Tasks: Among the code understanding tasks, only one PTM of source code is evaluated for Exception Type and Code Question Answering, so we have no idea of the performance of the other models on these tasks. Although three CodePTMs are evaluated on Code-to-Code Retrieval, they all used different datasets, thus making direct comparisons impossible. Consequently, the only tasks for which we can compare PTMs of source code are Defect Detection, Clone Detection and Code Search. For Defect Detection, most of the models are evaluated on Devign with CoTexT achieving the best results. For Clone Detection, most models are evaluated on BigCloneBench with SynCoBERT achieving the best results. For Code-to-Code Retrieval (POJ-104) and Code Search (CodeSearchNet and AdvTest), UniXcoder is the state-of-the-art CodePTM.

b) Code Generation Tasks: Code generation tasks are more popularly used to evaluate PTMs of source code. However, we cannot make valid comparisons on three of the seven generation tasks shown in Table II: four PTMs of source code were evaluated on Code Completion, but they each used different datasets. Only T5-learning was evaluated on Mutant Generation and Assert Generation. Of the four comparable tasks and datasets, CodeT5 achieved the best results in three of them: Code Translation (CodeTrans), Bug Fixing, and Code Summarization (CodeSearchNet). The remaining task, Code Generation, is bested by UniXcoder.

c) Overall: Since different PTMs of source code are evaluated on different downstream tasks and dataset, it is impossible to compare them directly based only on the results reported in existing paper. Consequently, we cannot draw conclusions that are more broadly applicable, the conclusions we draw are not reliable, and we could not know the best performing PTMs of source code on many of these SE tasks. Therefore, achieving a fair and systematic comparison of these PTMs is the main motivation for the work in this paper.

TABLE IV
Current SOTAs and new SOTAs.

	Current SOTA		New SOTA		Δ
	Model	**Value (%)**	**Model**	**Value (%)**	**(pts)**
DD	CoTexT	65.99	**SynCoBERT**	66.25	0.26
CD	SynCoBERT	97.4	SynCoBERT	97.55	0.15
ET	CuBERT	79.12	**CodeT5**	85.00	5.88
CR	UniXcoder	90.52	UniXcoder	90.55	0.03
CS	UniXcoder	41.3	UniXcoder	41.57	0.27
QA	UniXcoder	70.1	UniXcoder	70.3	0.2
CT	CodeT5	66.4	**PLBART**	67.6	1.2
BF	CodeT5	17.79	CodeT5	17.98	0.19
CC	CodeGPT-adapted	42.37	CodeGPT-adapted	43.80	1.43
MG	T5-learning	28	**CodeT5**	34.83	6.83
AG	T5-learning	40.5	**PLBART**	49.94	9.44
SM	CodeT5	19.55	CodeT5	19.71	0.16
CG	UniXcoder	22.60	**CodeT5**	23.43	0.83

IV. Evaluation Results

In this section, we present our evaluation results.

For each SE task, we repeat the fine-tuning and testing experiments on each model three times using three random seeds (i.e., 24, 42 and 81) and report the average results in Tables V, VI, VII, and VIII. Specifically, for each task and each evaluation metric of that task, we show under the "Cur" and "New" columns the current results reported by existing work and the results we obtained through our experiments, respectively. For each "New" column, the best and second best results obtained by each type of pre-trained models (i.e., PTM-NL, PTM-C, and CodePTM) on the corresponding SE task are marked in **bold** and underline respectively[7]. Note that the first row of each table show the results of vanilla Transformer.

V. Discussion

Through our experiments, we obtain the new SOTA[8] on each task. In Table IV we show for each task the current SOTA model (derived from existing work), the new SOTA model (derived from our experiments), as well as their corresponding performances[9]. Moreover, we boldface the new SOTA model if it is different from the current SOTA model. Finally, we show the absolute performance difference between the new SOTA model and the current SOTA model under the "Δ" column.

First, the SOTA models of all 13 SE tasks belong to the type of CodePTM, which covers models specifically designed to capture the unique features of Source Code, except for

[5] Refer to the supplementary materials for details.

[6] The datasets that comprise multiple subsets are: `C3` (which includes subsets corresponding to six languages), `C5` (which contains two subsets corresponding to bidirectional translation between Java and C#), `B2` (which consists of two subsets of different length distributions), `A3` (which includes two subsets corresponding to raw and abstracted source code [43]) and `C7` (which consists of three subsets of different sizes).

[7] For each SE task, we apply the Approximate Randomization Test [67] to every pair of results to determine whether their difference is statistically significant. These results are presented in the supplementary materials.

[8] For generation tasks (e.g., Code Translation, Assert Generation, Bug Fixing, Code Completion, etc.) adopting multiple metrics (e.g., EM, BLEU, CodeBLEU, etc.), we follow the existing papers and use EM (Exact Match) as the metric to determine the SOTA model. The reason is that EM is the most rigorous metric for evaluating the generation performance. Besides, if there are multiple sub-datasets for evaluating one task, the average performance over all sub-datasets is used to determine the SOTA model.

[9] If multiple sub-datasets are used for evaluation, the average performance is reported here.

TABLE V
EXPERIMENTAL RESULTS ON CODE UNDERSTANDING TASKS.

Model	DD		CD		ET		CR		CS		QA	
	Acc		F1		Acc		MAP		MRR		MRR	
	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New
Transformer		64.40		89.27		48.98		64.27		3.12		52.89
RoBERTa	61.05	**64.47**	94.9	95.35	-	**76.94**	76.67	**80.20**	18.33	**18.82**	60.3	**60.28**
GPT-2	-	63.22	-	**96.22**	-	75.54	-	53.30	-	16.38	-	58.06
BART	-	63.81	-	95.11	-	73.68	-	79.63	-	16.65	-	55.57
T5	61.93	61.87	-	94.86	-	74.75	-	69.16	-	16.97	-	45.63
CuBERT	-	64.25	-	94.78	79.12	**79.90**	-	76.87	-	22.26	-	54.33
GPT-C	-	63.77	-	95.46	-	78.26	-	55.23	-	24.39	-	50.32
C-BERT	57.4	64.05	-	95.00	- -	74.57	-	72.91	-	25.34	-	54.81
JavaBERT	-	64.50	-	96.57	-	67.66	-	**77.44**	-	25.02	-	54.04
CodeGPT-adapted	-	**65.64**	-	**96.65**	-	76.71	-	72.63	-	25.97	-	54.24
DeepDebug	-	64.18	-	95.90	-	73.50	-	73.51	-	**30.58**	-	**57.39**
CodeBERT	-	65.02	-	96.77	-	81.25	82.67	85.61	-	38.21	65.7	65.90
GraphCodeBERT	-	65.92	97.1	97.11	-	83.26	85.16	87.73	-	38.76	68.4	68.55
CugLM	-	64.19	-	96.44	-	79.01	-	83.32	-	36.20	-	61.44
DOBF	-	63.86	95.9	96.84	-	79.04	-	87.31	38.3	38.56	-	61.31
T5-learning	-	63.60	-	96.38	-	69.85	-	80.82	-	37.98	-	60.21
PLBART	63.18	64.21	97.2	97.01	-	77.93	-	85.02	-	38.70	65.0	65.01
ProphetNet-Code	-	63.57	-	96.05	-	79.37	-	79.82	-	37.64	-	63.73
CoTexT	65.99	65.68	-	95.96	-	77.21	-	86.65	-	38.13	-	68.70
TreeBERT	-	65.76	-	96.51	-	78.08	-	85.54	-	39.60	-	64.98
CodeT5	65.78	65.82	97.2	97.18	-	**85.00**	-	87.53	-	40.03	67.8	67.91
SynCoBERT	64.5	**66.25**	97.4	**97.55**	-	82.70	88.24	88.52	38.1	39.99	-	69.19
SPT-Code	-	64.88	-	96.40	-	77.11	-	86.54	-	37.05	-	64.55
UniXcoder	-	65.64	95.2	96.32	-	83.47	90.52	**90.55**	41.3	**41.57**	70.1	**70.30**

the Code Completion task whose SOTA model, CodeGPT-adapted, is of type PTM-C, which covers models designed for Natural Language but pre-trained on Source Code.

Second, while many PTMs have been proposed, only five of them have managed to achieve SOTA performance on at least one SE task. They are CodeT5 (SOTA on 5 tasks), UniXcoder (SOTA on 3 tasks), PLBART (SOTA on 2 tasks), SynCoBERT (SOTA on 2 tasks), and CodeGPT-adapted (SOTA on 1 task).

Third, vanilla Transformer's performance relative to the PTMs is different for different SE tasks: (1) on Clone Detection (CD), Error Type prediction (ET), Code Search (CS), Code Translation, Assert Generation, and Code Summarization, vanilla Transformer is surpassed in performance by all types of PTMs (i.e., PTM-NL, PTM-C, and CodePTM); (2) on Code Completion and Mutant Generation, vanilla Transformer is beaten by all PTMs-C and CodePTMs but it outperforms two PTMs-NL, BART and T5; (3) on Code-to-Code Retrieval (CR) and Code Question Answering (QA), vanilla Transformer not only surpasses a PTM-NL (GPT-2 on CR and T5 on QA), but also beats one PTM-C (GPT-C for both tasks); and (4) on Defect Detection (DD) and Bug Fixing, vanilla Transformer even outperforms CodePTMs in addition to PTMs-NL and PTMs-C, beating CugLM, DOBF, T5-learning, PLBART, and ProphetNet-Code on DD, and CugLM on Bug Fixing.

In the following, we discuss in detail the observations obtained from the current and the new results on each task.

A. Defect Detection

SynCoBERT defeats CoTexT and becomes the new SOTA PTM for this task, and Accuracy improves by 0.26.

1) Architecture: While the Top-2 models on this classification task, SynCoBERT and GraphCodeBERT, are both TE-based, **there is not enough empirical evidence for us to conclude that TE is a better architecture for this task than TD or TF**, for several reasons. First, to draw this conclusion, we need to compare the results of two models that differ only w.r.t. architecture, but there do not exist two PTMs on our list that differ only w.r.t. architecture. Second, TF-based CoTexT, which uses MLM as the only pre-traning task, outperforms TE-based UniXcoder, which uses three more complex pre-training tasks, ULM, MCL and CMG. Finally, TF-based DeepDebug achieves better results than TE-based C-BERT when using only code as input and MLM as its only pre-training task.

2) Modality: **Both code structure and NL are shown to have a positive effect on the performance of the models on this task, but the way they are being used also matters.** As an example, TF-based TreeBERT outperforms some of the TF-based models that use code and NL (e.g, DOBF, T5-learning, PLBART) significantly owning to its use of ASTs. As another example, TF-based CoTexT outperforms TF-based T5-learning considerably: CoText concatenates Code and the corresponding Doc as one single input, whereas T5-learning treats the features derived from these two modalities as separate data instances. This suggests that how the information derived from these modalities is used has an impact on performance.

3) Pre-training Tasks: First, the results in the *New* column of this task in Table V reveal that **the most influential pre-training tasks are cross-modal-aware classification tasks as they are being used by the Top-5 models**. These tasks include *TEP/EP* (used by SynCoBERT and GraphCodeBERT), *MCL* (used by SynCoBERT and UniXcoder), and *NA* (adopted by GraphCodeBERT). This observation is different from the conclusion derived from the *Cur* column, where Seq2Seq MLM (the only pre-training task used by the old SOTA model, CoTexT) seems to have the greatest impact on defect detection.

B. Clone Detection

The new results on this task do not change significantly from the current ones, except that PLBART, which is currently tied for second place, has slipped to fourth place. The drop in PLBART's rank seems to suggest that using multiple pre-training tasks is better than using a single pre-training task on this task: while the Top-2 models, SynCoBERT and CodeT5, employ four distinct pre-training tasks, PLBART uses DAE as the only pre-training task. Besides, the new results also enable us to see the performance of TD-based models on this task; in particular, the best TD-based PTM, CodeGPT-adapted, ranks 7th.

C. Exception Type

This task is the only multi-label classification task among our 13 SE evaluation tasks. Currently, only one model (i.e., CuBERT) has been applied to this task, which prevents us from drawing any conclusions about the relative performance of different types of models on a multi-label classification task like this. Fortunately, our results enable us to draw several new conclusions:

1) Architecture: Most notably, according to the new results, **the SOTA performance on this task is not achieved by a TE-based model**. Instead, TF-based CodeT5, which turns the task into a text-to-text form, achieves the best results. The best TE-based model (UniXcoder) and the best TD-based model (GPT-C) rank second and tenth respectively, and their accuracies are 1.53 and 6.74 points lower than that of CodeT5. Recall that in Section II-E, we mentioned that as a T5-based model, CodeT5, when applied to a classification task, maps each label to a unique text string. Specifically for Exception Type, it does not predict the index of each exception, but rather the text string of that exception. In this way, CodeT5 turns this classification task into one of generating NL, which is exactly what CodeT5 is good at. In contrast, for TE-based models (e.g., SynCoBERT, UniXcoder, GraphCodeBERT), most of the tasks they use in pre-training are binary classification tasks (e.g., MCL, TEP/EP, NA), so they may lack the knowledge needed for multi-label classification.

2) Modality: The impact of each modality on this task becomes clear as well. All of the Top-3 models (i.e., CodeT5, UniXcoder, and GraphCodeBERT) use NL as one of the input modalities, while both code and code structure were only used by two of them (CodeT5 and UniXcoder). This seems to suggest that **NL has a better positive impact on this task than the other two modalities**.

3) Pre-training Tasks: **Both the classification pre-training task NSP and the generative pre-training task FNP seem to have positive impacts on this task**. To exemplify, while CuBERT and C-BERT are both TE-based models that use code as the only modality and differ only in their pre-training tasks (CuBERT uses both MLM and NSP whereas C-BERT uses only MLM), CuBERT outperforms C-BERT by as many as 5 percent points in accuracy. As another example, while ProphetNet-Code and PLBART are both TF-based models that use code and NL as input modalities and differ only in terms of their pre-training tasks (ProphetNet-Code uses FNP whereas PLBART uses DAE), ProphetNet-Code surpasses PLBART in performance.

D. Code-to-Code Retrieval

Currently, the relative advantages and disadvantages of different model architectures are not available since only four TE-based models are evaluated on this task. However, with the new results, the conclusion that **TE-based models have more advantages over the other architectures on this task** can be verified, since the Top-3 models of this task are all TE-based (i.e., UniXcoder, SynCoBERT, and GraphCodeBERT). Besides, the performance of the TF- and TD-based models is also measurable. Specifically, the best performing TF-based model (CodeT5) and TD-based one (CodeGPT-adapted) ranks 4th and 20th, respectively.

E. Code Search

1) Architecture: Although the SOTA model on this task is still UniXcoder (TE-based), the rank of CodeT5 (TF-based) improved from third to second in the new results, and the third position is taken by SynCoBERT (TE-based). TreeBERT (TF-based) ranks fourth, GraphCodeBERT (TE-based) ranks fifth, and PLBART (TF-based) ranks sixth. These results seem to suggest that **TE-based and TF-based models perform comparably on this task**, as they alternate in the Top-6. In addition, the performance of TD-based models on this task is now measurable: the best TD-based PTM (CodeGPT-adapted) ranks 15th.

2) Pre-training Tasks: **The MLM pre-training task and its variants, as well as cross-modal-aware tasks demonstrate their necessity in achieving top performance on this task**. Specifically, the pre-training tasks the top-ranked models used all include *MLM* (and its variants such as *Seq2seq MLM*), as well as cross-modal-aware tasks (e.g., *MCL*, *BDG*, *EP*). On one hand, *MLM* and its variants enable a model to generate better input representations. On the other hand, the cross-modal-aware tasks typically allow a model to learn the alignment between different input modalities with the same semantics. These two types of pre-training tasks therefore allow a model to generate a more uniform input representation for multimodal inputs, which is exactly what a model needs to have for Code Search.

3) Modality: **Pre-training on multiple modalities appear to benefit this task** since all of the Top-6 models are pre-trained on two or three modalities. Concretely, UniXcoder is pre-trained on NL and Structure, TreeBERT is pre-trained on Code and Structure, CodeT5 and PLBART are both pre-trained on Code and NL, while SynCoBERT and GraphCodeBERT are pre-trained on all of the three modalities. It is hard to tell which modality has the largest impact on performance, because the absence of any one of them would not prevent a model from becoming the Top-6..

F. Code Question Answering

The new SOTA model remains the same as the current one, i.e., UniXcoder. But our newly reported SOTA performance

TABLE VI
EXPERIMENTAL RESULTS ON CODE TRANSLATION AND ASSERT GENERATION.

Model	Code Translation												Assert Generation							
	Java->C#						C#->Java						abs				raw			
	EM		BLEU		CodeBLEU		EM		BLEU		CodeBLEU		EM		BLEU		EM		BLEU	
	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New
Transformer	33.0	40.8	55.84	60.22	63.74	67.10	37.9	43.9	50.47	54.86	61.59	61.84	-	28.65	-	63.24	-	35.35	-	67.62
RoBERTa	-	**58.9**	-	**79.70**	-	**83.77**	-	**59.5**	-	73.14	-	**78.63**	-	**36.12**	-	**69.11**	-	**50.68**	-	**76.23**
GPT-2	-	54.4	-	65.39	-	76.92	-	56.0	-	69.41	-	71.24	-	35.02	-	66.89	-	48.23	-	74.15
BART	-	49.5	-	67.91	-	74.25	-	55.2	-	72.32	-	70.80	-	33.82	-	65.11	-	48.14	-	74.27
T5	-	45.3	-	69.23	-	76.05	-	53.2	-	**73.84**	-	71.52	-	33.10	-	64.95	-	48.04	-	74.13
CuBERT	-	55.6	-	75.15	-	79.30	-	55.6	-	70.07	-	75.31	-	37.14	-	68.34	-	50.60	-	74.31
GPT-C	-	60.9	-	77.91	-	82.48	-	59.6	-	72.94	-	78.18	-	37.16	-	66.39	-	51.64	-	76.12
C-BERT	-	55.4	-	74.65	-	81.63	-	56.4	-	70.28	-	75.73	-	36.64	-	65.51	-	50.70	-	72.11
JavaBERT	-	61.1	-	80.74	-	80.45	-	58.3	-	70.10	-	77.14	-	38.23	-	**71.22**	-	52.66	-	**78.51**
CodeGPT-adapted	-	**62.0**	-	80.21	-	**85.47**	-	60.3	-	72.93	-	79.02	-	39.42	-	69.61	-	52.85	-	76.12
DeepDebug	-	59.5	-	**81.46**	-	83.95	-	**63.8**	-	**75.10**	-	**82.43**	-	**39.89**	-	71.10	-	**56.65**	-	76.84
CodeBERT	59.0	61.2	79.92	81.16	85.10	85.29	58.0	60.1	72.14	73.73	79.41	80.11	-	38.40	-	70.65	-	53.23	-	77.54
GraphCodeBERT	59.4	62.6	80.58	81.24	-	85.34	58.8	61.5	72.64	73.67	-	80.63	-	38.98	-	70.87	-	53.71	-	77.69
CugLM	-	60.8	-	78.34	-	83.65	-	61.6	-	73.95	-	78.06	-	39.36	-	73.20	-	52.85	-	77.46
DOBF	-	64.8	-	80.27	-	82.77	-	64.6	-	75.44	-	80.53	-	40.01	-	72.39	-	54.71	-	78.40
T5-learning	-	62.9	-	78.19	-	81.13	-	64.8	-	75.64	-	81.09	34	40.95	-	72.70	47	56.85	-	77.09
PLBART	64.6	**67.8**	83.02	84.75	87.92	**88.16**	65.0	**67.4**	78.35	**79.75**	85.27	85.05	-	**42.44**	-	74.21	-	**57.43**	-	79.51
ProphetNet-Code	-	62.5	-	80.38	-	81.64	-	64.5	-	75.68	-	81.04	-	37.38	-	68.45	-	56.26	-	77.64
CoTexT	-	65.7	-	83.35	-	85.63	-	65.4	-	77.98	-	82.31	-	38.19	-	71.51	-	56.04	-	78.55
TreeBERT	-	62.1	-	81.72	-	84.34	-	64.2	-	76.33	-	81.17	-	42.32	-	73.95	-	57.21	-	**79.89**
CodeT5	65.9	67.2	84.03	**84.97**	-	87.50	66.9	66.3	79.87	79.67	-	83.70	-	40.67	-	71.77	-	56.90	-	79.71
SynCoBERT	60.4	64.1	80.75	82.52	84.85	85.60	61.3	63.8	76.52	77.53	82.22	82.36	-	39.10	-	70.42	-	54.66	-	79.27
SPT-Code	64.1	66.6	90.34	83.24	-	85.15	60.3	63.9	86.10	78.82	-	**85.33**	-	42.35	-	**74.53**	-	57.09	-	79.36
UniXcoder	-	64.5	-	81.66	-	85.60	-	64.1	-	77.37	-	82.56	-	39.46	-	71.25	-	54.94	-	78.99

TABLE VII
EXPERIMENTAL RESULTS ON BUG FIXING, CODE COMPLETION AND MUTANT GENERATION.

Model	Bug Fixing												Code Completion				Mutant Generation			
	small						medium													
	EM		BLEU		CodeBLEU		EM		BLEU		CodeBLEU		EM		ES		EM		BLEU	
	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New
Transformer	14.7	14.63	77.21	76.92	-	73.88	3.7	8.46	89.25	89.23	-	86.86	-	37.71	-	67.95	-	24.50	-	78.60
RoBERTa	-	13.88	-	**79.72**	-	**78.31**	-	**9.09**	-	**88.69**	-	**84.05**	-	39.01	-	68.98	-	**28.18**	-	**80.34**
GPT-2	-	14.97	-	64.42	-	68.10	-	5.05	-	74.11	-	72.42	41.73	**41.77**	-	**70.30**	-	26.77	-	79.45
BART	16.7	**15.60**	-	71.34	-	72.43	6.7	6.97	-	82.37	-	81.85	-	30.67	-	56.17	-	23.21	-	77.13
T5	15.3	14.34	-	69.71	-	73.10	4.11	6.49	-	78.22	-	78.20	-	28.58	-	55.24	-	24.17	-	79.16
CuBERT	-	14.87	-	74.93	-	75.28	-	8.92	-	86.12	-	83.09	-	38.32	-	66.97	-	27.08	-	78.66
GPT-C	-	13.08	-	70.06	-	71.83	-	8.26	-	85.41	-	82.47	-	42.82	-	71.35	-	27.24	-	76.55
C-BERT	-	14.04	-	73.19	-	74.54	-	9.37	-	85.57	-	83.87	-	41.07	-	67.82	-	26.63	-	77.43
JavaBERT	-	15.39	-	**78.98**	-	76.02	-	9.41	-	85.33	-	84.42	-	39.16	-	67.67	-	28.14	-	79.33
CodeGPT-adapted	-	13.66	-	76.07	-	**77.13**	-	11.00	-	85.28	-	84.55	42.37	**43.80**	-	**72.54**	-	27.64	-	79.40
DeepDebug	18.7	**18.13**	-	76.64	-	76.91	11.4	**11.09**	-	**87.10**	-	**85.80**	-	39.68	-	65.28	-	**30.11**	-	**79.45**
CodeBERT	16.4	14.66	77.42	78.41	-	78.09	5.2	9.72	90.07	86.94	-	83.88	-	40.77	-	68.58	-	28.61	-	80.59
GraphCodeBERT	17.3	16.85	80.02	**79.61**	-	**79.68**	9.1	10.14	91.31	87.63	-	85.33	-	40.26	-	69.88	-	29.72	-	80.53
CugLM	-	13.78	-	75.36	-	75.20	-	9.08	-	85.20	-	84.82	-	**42.94**	-	**71.89**	-	27.09	-	78.95
DOBF	-	15.45	-	75.52	-	74.62	-	10.28	-	87.93	-	85.01	-	39.35	-	69.30	-	29.16	-	79.10
T5-learning	10	17.62	-	77.05	-	76.30	3	10.94	-	88.46	-	86.42	-	38.06	-	66.58	28	29.90	-	78.44
PLBART	19.21	19.40	77.02	78.03	-	77.58	8.98	11.05	88.5	88.48	-	86.67	-	41.74	-	68.42	-	33.08	-	80.07
ProphetNet-Code	-	17.23	-	75.40	-	75.60	-	10.75	-	86.82	-	84.19	-	39.66	-	67.24	-	29.63	-	77.12
CoTexT	21.58	21.33	77.28	77.20	77.38	77.75	13.03	13.37	88.68	87.13	84.41	85.14	-	40.36	-	70.16	-	31.87	-	80.00
TreeBERT	-	20.73	-	79.38	-	75.17	-	12.89	-	89.05	-	**87.15**	-	41.73	-	70.14	-	33.20	-	80.46
CodeT5	21.61	**21.65**	77.43	77.55	-	77.24	13.96	**14.30**	87.64	**89.23**	-	87.05	-	40.52	-	71.29	-	**34.83**	-	**80.75**
SynCoBERT	-	20.32	-	78.81	-	78.56	-	11.17	-	87.94	-	86.10	-	41.52	-	70.26	-	29.86	-	80.16
SPT-Code	17.54	18.59	75.10	78.51	-	74.97	10.86	12.06	87.88	88.37	-	84.35	-	40.20	-	68.97	-	33.00	-	79.18
UniXcoder	-	19.05	-	79.18	-	79.45	-	13.96	-	87.59	-	86.23	-	41.69	-	69.84	-	29.78	-	80.02

has an improvement of 0.2 percent MRR points. Note that *MCL*, the pre-training task used by the SOTA model UniXcoder, aims to distinguish whether two inputs match each other, which is also the goal of the Code Question Answering task.

G. Code Translation

Although the models are ranked according to their average EM value on the "Java to C#" and the "C# to Java" sub-datasets, we find that the Top-2 models on the two sub-datasets are both PLBART and CodeT5[10]. Besides, the Top 3–6 models on this task are CoTexT, SPT-Code, DOBF, and UniXcoder respectively.

1) Architecture: According to the new results, that **TF-based models take the absolute lead on this task** can be verified, since the Top-5 models are all TF-based, and given that we have more TF-based models in our comparison than before, the rank of the best performing TE-based model (i.e., SynCoBERT in Current and UniXcoder in New) drops from fourth to sixth.

2) Modality: **The importance of NL is well validated**, due to the fact that the Top-4 performers in both the current (i.e., CodeT5, PLBART, SPT-Code and SynCoBERT) and the new results (i.e., PLBART, CodeT5, CoTexT and SPT-Code)

[10] For a discussion of the results w.r.t. other evaluation metrics. See the supplementary file.

TABLE VIII

EXPERIMENTAL RESULTS ON CODE GENERATION TASKS INVOLVING NATURAL LANGUAGE.

Model	Code Summarization												Code Generation					
	Java		Py		JS		PHP		Go		Ruby							
	BLEU		BLEU		BLEU		BLEU		BLEU		BLEU		EM		BLEU		CodeBLEU	
	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New	Cur	New
Transformer	16.26	16.37	15.81	16.47	11.59	10.26	22.12	23.41	16.38	16.46	11.18	11.09	-	6.10	-	21.67	-	26.98
RoBERTa	16.47	17.38	18.14	17.49	11.90	11.75	24.02	24.42	17.72	17.63	11.17	11.26	-	**19.30**	-	31.92	-	**35.40**
GPT-2	-	18.62	-	18.92	-	14.13	-	23.91	-	17.59	-	12.03	17.35	17.55	25.37	23.62	29.69	29.93
BART	-	**18.98**	-	19.37	-	**14.68**	-	**24.46**	-	**18.80**	-	13.65	-	18.90	-	31.38	-	33.72
T5	18.35	18.87	19.26	**19.57**	14.57	14.45	24.59	24.32	19.17	18.70	14.18	**13.72**	18.65	18.70	32.74	**32.02**	35.95	33.26
CuBERT	-	16.75	-	17.77	-	11.34	-	22.76	-	16.09	-	10.46	-	19.05	-	30.14	-	32.82
GPT-C	-	17.18	-	17.78	-	12.01	-	23.42	-	16.96	-	10.54	-	19.85	-	30.45	-	33.10
C-BERT	-	17.44	-	18.39	-	13.14	-	23.90	-	17.47	-	12.14	-	19.80	-	33.62	-	35.99
JavaBERT	-	18.23	-	17.57	-	11.91	-	22.87	-	17.13	-	10.94	-	18.45	-	34.62	-	36.93
CodeGPT-adapted	-	17.68	-	18.46	-	12.91	-	**24.68**	-	17.38	-	12.39	20.10	**20.15**	32.79	35.94	35.98	37.27
DeepDebug	-	**19.00**	-	**18.85**	-	**14.39**	-	23.37	-	**17.68**	-	**13.27**	-	18.40	-	**36.52**	-	**38.90**
CodeBERT	17.65	18.61	19.06	19.23	14.90	14.75	25.16	24.70	18.07	18.26	12.16	12.53	-	21.15	-	31.45	-	35.26
GraphCodeBERT	-	18.93	-	19.39	-	14.90	-	25.64	-	18.50	-	12.63	-	21.00	-	34.33	-	37.55
CugLM	-	18.04	-	18.20	-	14.07	-	24.66	-	18.53	-	11.47	-	21.80	-	33.70	-	35.91
DOBF	19.05	19.18	18.24	18.41	-	13.22	-	23.83	-	18.28	-	13.21	-	20.35	-	35.26	-	37.41
T5-learning	-	18.84	-	18.23	-	13.18	-	23.10	-	17.29	-	12.51	-	18.95	-	35.76	-	38.30
PLBART	18.45	19.31	19.30	19.41	15.56	15.73	23.58	24.47	18.91	19.01	14.11	14.15	18.75	19.85	36.69	36.63	38.52	39.29
ProphetNet-Code	19.39	19.29	17.87	18.20	16.60	15.95	24.57	24.28	18.43	18.31	14.37	14.39	-	21.70	-	37.67	-	39.79
CoTexT	19.10	19.19	19.52	19.72	14.77	15.08	24.47	24.57	19.37	19.13	13.07	14.28	20.10	21.80	36.51	38.74	39.49	40.63
TreeBERT	-	18.90	-	19.44	-	15.05	-	23.82	-	18.74	-	13.57	-	22.05	-	38.27	-	39.73
CodeT5	20.31	**20.35**	20.01	**20.17**	16.16	**16.75**	26.03	**25.97**	19.56	**19.68**	15.24	**15.36**	22.30	**23.40**	40.73	**40.75**	43.20	**43.40**
SynCoBERT	-	18.89	-	18.74	-	14.57	-	25.55	-	18.36	-	13.91	-	21.35	-	38.39	-	41.21
SPT-Code	-	18.90	-	19.71	-	15.28	-	24.78	-	19.17	-	14.38	-	20.00	-	37.91	-	39.90
UniXcoder	-	19.42	-	18.64	-	14.27	-	25.70	-	18.59	-	14.32	22.60	22.65	-	38.73	-	40.86

use NL. The role of code structure, on the other hand, is less clear, since the Top-2 models (i.e., PLBART, CodeT5) are not pre-trained on the Structure modality and the best model using code structure drops from the third place in "Cur" (i.e., SynCoBERT) to the fourth place in "New" (i.e., SPT-Code).

3) Pre-training Tasks: The new results show that **the pre-training objective *DAE* has a more significant impact than *BDG/CMG***. To exemplify, consider PLBART and CodeT5, both of which are TF-based and employ the same modalities (code and NL). They differ only in terms of the pre-training tasks: the former uses *DAE* and the latter uses *BDG/CMG*. The fact that PLBART outperforms CodeT5 can therefore be attributed to the fact that *DAE* is a better pre-training task for Code Translation than *BDG/CMG*. This conclusion is contrary to the conclusion drawn from the CUR results, where *BDG* is believed to have a stronger influence than *DAE* on Code Translation due to the fact that CodeT5 beat PLBART by 1.3 percent EM points.

H. Bug Fixing

Considering the EM value averaged over the "small" and "medium" datasets, the Top-4 models change from CodeT5, CoTexT, DeepDebug, and SPT-Code (listed in decreasing order of performance) to CodeT5, CoTexT, TreeBERT, and UniXcoder. A closer examination of the sub-datasets reveals that UniXcoder outperforms CoText and TreeBERT, achieving the second best performance on the "medium" dataset while ranking 4th on the "small" one.

1) Architecture: The Top-3 performance is achieved by three TF-based models (i.e, CodeT5, CoTexT, and TreeBERT) and the best and second TE-based models (i.e., UniXcoder and SynCoBERT) rank 4th and 5th respectively. Besides, the best TD-based model (i.e., CodeGPT-adapted) only rank 14th. This seems to suggest that **the TF architecture should be considered first when designing high performance pre-trained models for this task**.

2) Pre-training Tasks: **The most useful pre-training tasks for Bug Fixing is the sequence-to-sequence variants of MLM** adapted to the Transformer decoder. They enable a model to acquire the ability to generate target sequences from an incomplete one. As an example, consider the top-3 TF-based models, which all use such pre-training tasks: *Seq2seq MLM* in CodeT5 and CoTexT, *TMLM* in TreeBERT, and *Seq2seq IMLM* in CodeT5. Moreover, by using Seq2seqMLM as the only pre-training task, the second-highest ranked model, CoTexT, achieves better performance than TreeBERT, which uses *NOP* in addition to *TMLM* for pre-training.

I. Code Completion

This is the only SE task among the ones we consider where SOTA performance is achieved by the TD-based model CodeGPT-adapted, and it is the SOTA model in both the current and new results. This seems to suggest the absolute dominance of the TD architecture on this task. Our new results further suggest that **the pre-training objective *ULM* (adopted by the Top-3 models on this task, i.e., CodeGPT-adapted, CugLM, and GPT-C), whose goal is similar to that of code completion, plays an influential role in Code Completion**. As an example, consider the TE-based model CugLM, which outperforms another TE-based model CuBERT (pretrained on *MLM* and *NSP*) and achieves the second best performance by using *ULM* in addition to *MLM* and *NSP*. Moreover, in terms of modality, **Code Completion is the only task where neither NL nor code structure plays a positive role** since all of the Top-3 models use code as the only input modality. We speculate the reason is that there is currently no effective way to combine these two modalities with *ULM*.

J. Assert Generation

Since only T5-learning has been evaluated on this task currently, all conclusions drawn from the new results could be viewed as new findings. First, **the Top-5 performers are all TF-based models** (i.e., PLBART, TreeBERT, SPT-Code, T5-learning, and CodeT5 by order). The best performing TE-based (i.e., UniXcoder) and TD-based (i.e., CodeGPT-adapted) models rank 8th and 16th, respectively. As far as modality is concerned, NL seems to have a greater impact than other modalities, as four of the Top-5 models (i.e., PLBART, SPT-Code, T5-learning, and CodeT5) use NL, whereas only two (i.e., TreeBERT and SPT-Code) use code structures.

K. Mutant Generation

NL and code structure appear to have positive impacts, since the Top-3 models (i.e., CodeT5, TreeBERT, and PLBART by order) use either NL or code structure as one of the inputs in addition to the code. As for pre-training tasks, *DAE* alone is able to help the model (i.e., PLBART) achieve the third best performance. The structure-aware pre-training tasks, such as *TMLM* and *NOP* used by TreeBERT (the second best) and *CAP* used by SPT-Code (the fourth best) clearly have positive impacts on this task.

L. Code Summarization

1) Architecture: The best TE-based model (UniXcoder) ranks 5th, with the Top-4 being TF-based models (i.e., CodeT5, ProphetNet-Code, CoTexT, and SPT-Code), which suggests **the strong positive influence of the TF architecture on this task**. This is not in line with the current results in which the best TE-based model ranked second. With the new results, the best TD-based model (GPT-2) ranks 15th.

2) Modality: **The highest ranks achieved by models pre-trained on NL only (e.g., T5 and BART) are 5th and 9th in the current and new results, respectively**. The reasons why they perform even better than some of the models pre-trained on code or structure in addition to NL (e.g., CodeBERT, GraphCodeBERT, etc.) are two-fold. First, they acquire the ability to generate NL during pre-training, which is required by Code Summarization, and (2) because of the "naturalness" of the source code [13], [68], they are able to understand the code to some extent although they only have the ability to understand NL.

3) Pre-training Tasks: **Cross-modal(Code and Natural Language)-aware generation tasks such as *BDG/CMG* and *MNG* have positive impacts on a model's performance on this task**. As an example, CodeT5, which utilizes *BDG*, and SPT-Code, which utilizes *MNG*, are the top performers among TF-based models, and UniXcoder, which utilizes *CMG*, is the top performer among TE-based models.

M. Code Generation

The SOTA model changes from TE-based (UniXcoder) to TF-based (CodeT5), and the best TE-based (UniXcoder) and TD-based (CodeGPT-adapted) models rank 2nd and 11th, respectively. While the new SOTA model (i.e., CodeT5) for Code Generation is also the SOTA model for Code Summarization, **the ranks of T5 and BART (pre-trained only on NL) on this task are lower than their ranks on Code Summarization**, because understanding code and generating code are fundamentally different in nature. In addition, **the importance of the code and NL modalities for this task is not as clear as that for Code Summarization**, considering that among the Top-3 models (CodeT5, UniXcoder, and TreeBERT), only CodeT5 uses both code and NL: UniXcoder uses code structure and NL and TreeBERT uses code structure and code instead. Moreover, although only NL is the input of this task, **pre-training on code structure has a positive impact on this task**, since both of the second and third best models (UniXcoder and TreeBERT) are pre-trained on tasks such as *MCL*, *CMG*, and *NOP*.

VI. INSIGHTS AND TAKEAWAYS

After analysis and discussion by task, we have some insights and takeaways to provide to subsequent researchers.

- When designing a new model to solve multiple tasks, look up the current SOTA model's architecture, features, and pre-training tasks for each task, and use such information as a starting point.
- Always pre-train on multiple programming languages.
- Always pre-train with NL, since all of the new SOTAs use NL.
- Utilize structure information in PTMs for code understanding tasks.
- Ensure the pre-training tasks are as similar in form as possible to the target downstream task.
- Use different CodePTMs for different target task types since there is no almighty CodePTM, as per our results and Zeng et al. [69].

Particularly, for the following tasks, we have additional takeaways:

- **Clone Detection**: Although the TE-based model achieves the best performance, comparable results are achieved with the TF-based model. Besides, the use of NL and code structure is beneficial. Finally, MLM and its variants have better results on this task.
- **Code-to-Code Retrieval**: Utilize NL and code structure following the "Altogether" strategy. Besides, MLM and its variants, as well as structure-aware pre-training tasks, have positive effects on this task.
- **Code Question Answering**: Prefer TE models and use NL whenever possible.
- **Assert Generation**: NL is not a required modality. The reason is that although the model with the best performance uses NL, NL is not used in the same data sample as other modalities (because of the Standalone strategy).

Seq2seq pre-training tasks, such as DAE, MASS and MNG, should be prioritized.

- **Code Generation**: Besides TF, TE is worth trying. The use of NL-Code generation code pre-training tasks (e.g., BDG and CMG) is mandatory.

Finally, through our experiments, we propose several possible subsequent research directions as follows.

- Design more efficient pre-training tasks to make CodePTMs learn source code features better [20].
- Improve the efficiency of CodePTMs for fine-tuning on downstream tasks [70].
- Make the large CodePTMs lighter [71], [72].
- Improve the robustness of CodePTMs.

VII. Threats to Validity

Construct Validity: As discussed in Section II-D, we have re-implemented some PTMs (category IV) or re-collected some datasets (category III and some in IV). The replication may not be perfect but we have tried our best to do the re-implementation and collect the datasets to minimize the deviations from the original model (See Section II-D). Besides, we adopt the statistical significance testing to measuring the differences between our implementation and the original ones.

Internal Validity: It is widely agreed that, during fine-tuning, hyperparameters have a significant impact on the performance of pre-trained models. For models where hyperparameters for fine-tuning are not available (See Section II-D), the settings we obtain by hyperparameter search may introduce some bias with the performance reported in the original paper. But we have tried best to derive best performance of these models on each SE task.

External Validity: The results and observations we obtained in this work may apply only to the downstream tasks and corresponding datasets we have evaluated. For the other SE tasks and datasets, we cannot guarantee exactly the same results and observations.

VIII. Conclusion

We conducted the first systematic empirical comparison of existing pre-trained models of source code[11]. We believe that the results of our large-scale evaluation and the associated discussion can provide SE researchers with a better understanding of existing PTMs and their relative strengths and weaknesses, as well as a better characterization of the state-of-the-art of each SE task on which PTMs are commonly evaluated.

This paper provides many valuable findings that are either not available based on the existing results alone or completely contrary to current findings. For example, we found that TF-based models have clear advantages for not only code generation tasks but also code understanding tasks. We hope that this paper could provide interested researchers with a comprehensive and comparable insights into the current state of this domain and inspire them to design more powerful pre-trained models of source code.

Acknowledgment

This work was supported by National Natural Science Foundation of China (61802167), Natural Science Foundation of Jiangsu Province, China (BK20201250), Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming, and NSF award 2034508. We also thank the reviewers for their helpful comments. Chuanyi Li and Jidong Ge are the corresponding authors.

References

[1] A. M. Dai and Q. V. Le, “Semi-supervised sequence learning,” *Advances in neural information processing systems*, vol. 28, 2015.

[2] J. Howard and S. Ruder, “Universal language model fine-tuning for text classification,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 328–339.

[3] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237. [Online]. Available: https://aclanthology.org/N18-1202

[4] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” 2018.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *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)*, 2019, pp. 4171–4186.

[6] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” *Advances in neural information processing systems*, vol. 32, 2019.

[7] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

[8] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, “Electra: Pre-training text encoders as discriminators rather than generators,” in *International Conference on Learning Representations*, 2019.

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.

[10] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of Machine Learning Research*, vol. 21, pp. 1–67, 2020.

[11] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7871–7880.

[12] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[13] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, “Exploring software naturalness through neural language models,” *arXiv preprint arXiv:2006.12641*, 2020.

[14] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[15] N. T. De Sousa and W. Hasselbring, “Javabert: Training a transformer-based model for the java programming language,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 90–95.

[16] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, “Generating bug-fixes using pretrained transformers,” in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, 2021, pp. 1–8.

[11] All materials used in our experiments are available at https://github.com/NougatCA/FineTuner and https://doi.org/10.5281/zenodo.7318110.

[17] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[18] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[19] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[20] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1332–1336.

[21] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[22] C. Cummins, H. Leather, Z. Fisches, T. Ben-Nun, T. Hoefler, and M. O'Boyle, "Deep data flow analysis," 2020. [Online]. Available: https://arxiv.org/abs/2012.01470

[23] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[24] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon, "Graphcode2vec: generic code embedding via lexical and program dependence analyses," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 524–536.

[25] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[26] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 378–389.

[27] C. Niu, C. Li, B. Luo, and V. Ng, "Deep learning meets software engineering: A survey on pre-trained models of source code," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022*, 2022, pp. 5546–5555.

[28] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "Treebert: A tree-based pre-trained model for programming language," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, vol. 161. PMLR, 27–30 Jul 2021, pp. 54–63.

[29] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning source code representations," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 01–13.

[30] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[31] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[32] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480.

[33] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1026–1037.

[34] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 1287–1293.

[35] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. 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, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[36] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20,000+ web queries for code search and question answering," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 5690–5700.

[37] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 601–20 611, 2020.

[38] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[39] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 731–747.

[40] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[41] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *International conference on machine learning*. PMLR, 2020, pp. 245–256.

[42] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2019, pp. 301–312.

[43] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[44] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 300–310.

[45] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[46] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019.

[47] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI 2018*, 2018, pp. 2269–2275.

[48] A. V. Miceli-Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," in *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 2017, pp. 314–319.

[49] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 1643–1652.

[50] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[51] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[52] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.

[53] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 1536–1547.

[54] 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, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[55] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, "Dobf: A deobfuscation pre-training objective for programming languages," *arXiv preprint arXiv:2102.07492*, 2021.

[56] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[57] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668.

[58] W. Qi, Y. Gong, Y. Yan, C. Xu, B. Yao, B. Zhou, B. Cheng, D. Jiang, J. Chen, R. Zhang *et al.*, "Prophetnet-x: Large-scale pre-training models for english, chinese, multi-lingual, dialog, and code generation," *arXiv preprint arXiv:2104.08006*, 2021.

[59] L. Phan, H. Tran, D. Le, H. Nguyen, J. Annibal, A. Peltekian, and Y. Ye, "Cotext: Multi-task learning with code-text transformer," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 40–47.

[60] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?" in *International Conference on Machine Learning*. PMLR, 2021, pp. 8476–8486.

[61] J. Zhang, H. Hong, Y. Zhang, Y. Wan, Y. Liu, and Y. Sui, "Disentangled code representation learning for multiple programming languages," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 4454–4466.

[62] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[63] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[64] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[66] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017.

[67] E. S. Edgington, "Approximate randomization tests," *The Journal of Psychology*, vol. 72, no. 2, pp. 143–149, 1969.

[68] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[69] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 39–51.

[70] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "Bridging pre-trained models and downstream tasks for source code understanding," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 287–298.

[71] Z. Zhang, H. Zhang, B. Shen, and X. Gu, "Diet code is healthy: Simplifying programs for pre-trained models of code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1073–1084.

[72] J. Shi, Z. Yang, B. Xu, H. J. Kang, and D. Lo, "Compressing pre-trained models of code into 3 mb," in *The 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022*, 2022.