Contrastive Code-Comment Pre-training

Xiaohuan Pei*, Daochang Liu†, Luo Qian‡, Chang Xu*
* † *School of Computer Science, Faculty of Engineering, The University of Sydney, Sydney, Australia*
Email: *xpei8318@uni.sydney.edu.au, †finspire13@gmail.com, *c.xu@sydney.edu.au
Shanghai Qi Zhi Institute, Shanghai, China
Email: ‡luoqian19961224@gmail.com

***Abstract*—Pre-trained models for Natural Languages (NL) have been recently shown to transfer well to Programming Languages (PL) and largely benefit different intelligence code-related tasks, such as code search, clone detection, programming translation and code document generation. However, existing pre-trained methods for programming languages are mainly conducted by masked language modeling and next sentence prediction at token or graph levels. This restricted form limits their performance and transferability since PL and NL have different syntax rules and the downstream tasks require a multi-modal representation. Here we introduce C3P, a Contrastive Code-Comment Pre-training approach, to solve various down-stream tasks by pre-training the multi-representation features on both programming and natural syntax. The model encodes the code syntax and natural language description (comment) by two encoders and the encoded embeddings are projected into a multi-modal space for learning the latent representation. In the latent space, C3P jointly trains the code and comment encoders by the symmetric loss function, which aims to maximize the cosine similarity of the correct code-comment pairs while minimizing the similarity of unrelated pairs. We verify the empirical performance of the proposed pre-trained models on multiple downstream code-related tasks. The comprehensive experiments demonstrate that C3P outperforms previous work on the understanding tasks of code search and code clone, as well as the generation tasks of programming translation and document generation. Furthermore, we validate the transferability of C3P to the new programming language which is not seen in the pre-training stage. The results show our model surpasses all supervised methods and in some programming language cases even outperforms prior pre-trained approaches. Code is available at https://github.com/TerryPei/C3P.**

***Index Terms*—contrastive learning, representation learning, pre-training, programming language processing**

I. Introduction

Programming language processing for code-related tasks has attracted rising attention from the deep learning community, given its huge potential to build AI-driven coding applications, e.g., vulnerability detection by deep learning-based systems [1], intelligence code hints for programming learners [2], source code modeling and generation for developers [3] and API modeling for recommendations [4]. Inspired by the success of pre-trained models in natural language processing [5], [6], a few attempts have been made to promote the development of pre-trained models for the programming language, including Code2Seq [7], CuBERT [8], GraphCodeBERT [9] Codex [10], PLBART [11] and CodeT5 [12].

The general idea of these pre-training methods for programming language is either to predict the original code from an artificially masked input code sequence or conduct code syntactic prediction by parsing the textual and structural information from the abstract syntax tree. For example, Code2Seq [7] and CuBERT [8] employed the objectives of mask language modeling (MLM) and next sentence prediction (NSP) to obtain a general representation on the special tokens in the pre-training stage. CodeBERT [13] introduced the objective of replaced token detection (RTD), which utilized both bimodal and unimodal data for training. GraphCodeBERT [9] considered more on the structural information of code by training the objectives of Edge Prediction (EP) and Node Alignment (NA).

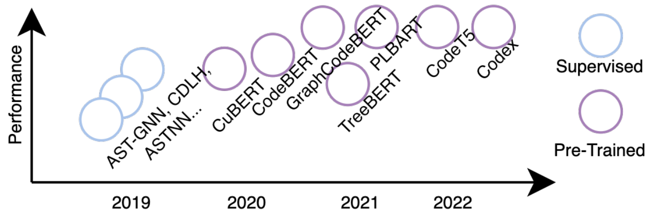


Fig. 1: Progress on Code-Related Tasks.

Despite impressive progress, *code (programming language) and comment (natural language) are often straightforwardly concatenated for the following encoder input without being distinguished in these existing studies*. In fact, programming language and natural language are heterogeneous and they are not sharing common lexical tokens, synonyms, or textual structures. Also, the semantic rules of programming language and natural language are different, and therefore a rashly weight sharing in the encoding stage could degrade the potential model performance on downstream tasks.

In this paper, we propose a contrastive pre-training algorithm to handle the encoding of programming language and natural language. It is natural to consider code and comment as two different views of an instance. Instead of concatenating their intermediate encodings for the optimization of self-supervised objectives, e.g., masked language modeling, we

This work was supported in part by the Australian Research Council under Project DP210101859 and the University of Sydney Research Accelerator (SOAR) Prize. The authors acknowledge the use of the National Computational Infrastructure (NCI) which is supported by the Australian Government, and accessed through the NCI Adapter Scheme and Sydney Informatics Hub HPC Allocation Scheme.

expect these two views to be aligned with each other in the latent space. By maximizing the agreement between two views via a contrastive loss, the embeddings could be well optimized to capture the intrinsic connections between the views. Structural code information can be another important view to describe the semantic information of the code, which can be derived by parsing data flow information from the source code with depth-first search (DFS) and labeling the dependency relation of variables. Given the redundancy between source code and the code structure, we employ a shared code encoder to map the concatenated textual and structural code to code embedding, while the comment information from a completely different natural language domain keeps a single comment encoder to itself. We evaluate our proposed model on four downstream tasks: natural language code search, clone detection, code translation, and document generation, covering the understanding and generation ability of the model on the code syntax. Our model significantly outperforms previous works on these fundamental downstream tasks. To further study the generalization of the learned representations, we set up zero-shot learning tasks by introducing the evaluation over new programming languages that have not been seen during training. The results demonstrate that the proposed contrastive pre-training is competitive with supervised methods that have been pre-trained on the whole programming languages.

II. Related Work

Our contribution is related to prior works about code pre-training, contrastive learning and fundamental code tasks. We start by reviewing programming language pre-training, after which we continue with contrastive methods for pre-training. Lastly, we introduce the fundamental code-related tasks.

A. Programming Language Pre-Training

Early research in code tasks mainly adopted supervised or self-supervised learning on programming language datasets. Inspired by the success of the 'pre-train+fine-tune' paradigm on natural language processing (NLP), many recent studies attempted to transfer these pre-training methods for programming language tasks, such as Code2Seq [7], CodeBERT [13] and GraphCodeBERT [9]. Most of them directly utilized source code for pre-training [7], [13], and some of them considered exploiting the structural logic of the code [9], [14]. Pre-training objectives in these works were mainly designed by masking some textual or structural information such as mask token modeling, and predicting some elements in the source code such as edge prediction on the code graph parsed by an abstract syntax tree (AST) [9].

B. Contrastive Learning for Pre-Training

Recently, contrastive learning methods for pre-training [15]–[17] have shown great potential in improving model robustness when fine-tuned for downstream tasks. The model pre-trained with contrastive learning also has a surprising zero-shot transfer ability [16], [17]. Contrastive text representation pre-training has been widely studied from both supervised and self-supervised perspectives [15]. For supervised contrastive pre-training [18], the input views can be constructed by manual text augmentation [19], or text summarization [20]. For self-supervised contrastive pre-training, the input views can be constructed via automated text augmentation [21], or next/surrounding sentence prediction [22]. There were also research works on cross-modal contrastive representation pre-training [16], [17], where image and text description information were fused into the same representation space.

When we are making this submission, we note a contemporaneous work [23] on arXiv that also studies the code and text contrastive pre-training. However, our approach differs in four ways: 1) This work [23] only feeds in source code tokens, while we incorporate code structure information to enrich the representation of the source code; 2) Mask rules are different. Beyond masking source tokens as in [23], we propose to further mask the variable sequence and nodes of data flow; 3) We not only evaluate the proposed contrastive training for code understanding tasks, but also investigate its performance on the generation tasks via generation decoders; 4) The comparison methods and experiment settings for transferability are different. As the code semantic rule of each programming language could be different in the real world, we test the transferability of the pre-trained encoders over a new programming language that has not been used during training, rather than taking an ordinary zero-shot setting to predict new labels in classification. These four differences make our contribution a unique one to contrastive learning for code pre-training.

C. Downstream Code-Related Tasks

Our work mainly focuses on the following fundamental code-related tasks.

Code Search is the task of answering the natural language query with code snippets. The fundamental problem of this task is modeling the latent correlation between the high-level inheritance in the natural language queries and the low-level semantic code context [24]. One line of related studies about code search mainly focused on supervised objectives [24], while the other line of studies followed the 'pre-trained+fine-tuning' paradigm [9], [13].

Code Clone is responsible for detecting the similarity score given two code fragments [25], and the existing techniques were based on parsing text, token, tree, metric, graph, and hybrid information of code [25]. With the advancement of deep learning, the detection process has been widely modeled by neural networks, including both supervised approaches [26]–[28] and pre-trained+fine-tuning methods [9], [13].

Code Translation aims to build a code-to-code translator that converts source code from an abstract programming language (such as Java or Python) to another [29]. Traditional supervised methods such as statistical phrase-based translation [30] limited the performance on the accuracy. Recent studies on this task mainly targeted introducing unsupervised pre-trained models to improve the performance, readability, and generalization ability [9], [13].

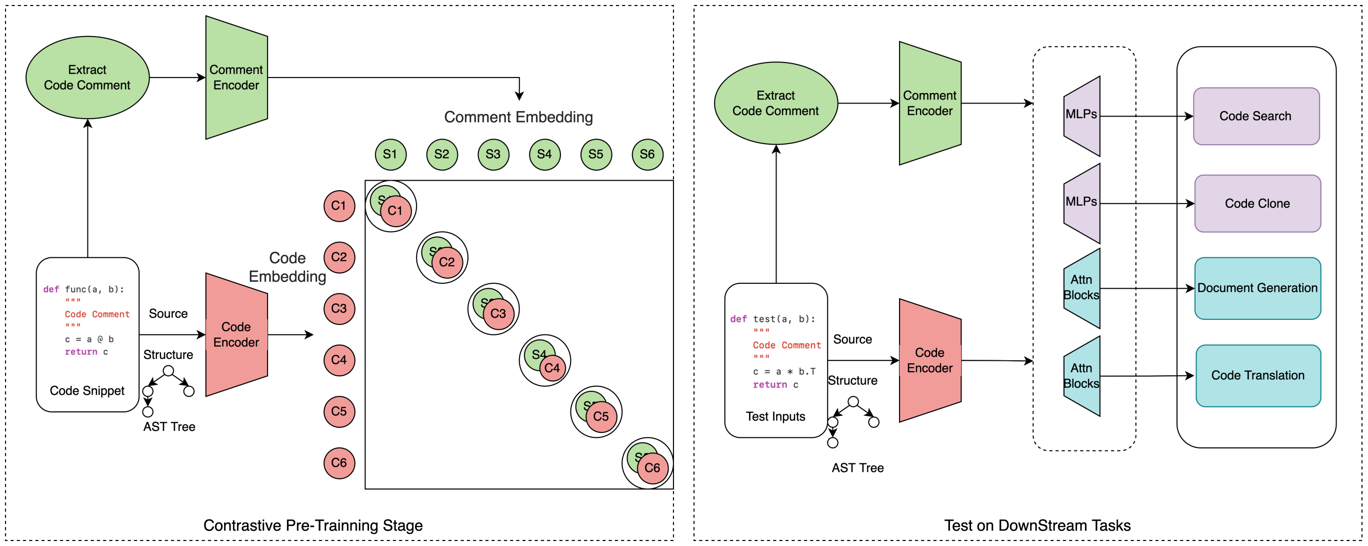


Fig. 2: Overview of Our Proposed C3P Framework. The framework comprises two stages: a code-comment matching pre-training stage and a code-related task fine-tuning stage. In the pre-training stage, we build code embeddings via a Code Encoder learned from textual tokens and structural tokens (nodes and edges) parsed from abstract syntax tree (AST) and we build comment embeddings via a Comment Encoder learned from natural language description. C3P compares the similarity of programming and natural language features for the match. In the fine-tuning stage, we utilize the pre-trained encoders and fine-tune our decoder networks for the downstream tasks. Both in pre-training and fine-tuning stages, the encoders are mapped with linear projection. In the ablation study, we also provide the performance on the Encoder-Only task without fine-tuning.

Document Generation is an intelligence task to auto-generate the natural language summary based on the given code snippet. Recent work has explored the techniques of keywords locator, parsing the code structure and pre-training embedding features for decoding [7], [9], [13].

III. Contrastive Code and Comment Pre-training

The proposed contrastive code and comment pre-training (C3P) learns the latent representations of code and comment by maximizing their agreement. Fig. 2 shows the framework of our proposed pre-training method and relevant downstream tasks. There are two stages in the framework. In the first pre-training stage, we extract the source code (programming language) and comment (natural language) from the given code snippet, which are then fed into the corresponding encoders. The embeddings of code and comment are further aligned in a multi-modal latent space by contrastive learning. In the second fine-tuning stage, we load the pre-trained encoders with additional decoders for different downstream tasks of code search, code clone, code translation and document generation.

A. Code and Comment Encoders

Consider a source code $C = \{c_1, c_2, \cdots, c_n\}$ with n tokens and its comment $W = \{w_1, w_2, \cdots, w_m\}$ with m tokens. To incorporate the structural information of the source code into the embedding, we parse the code syntax with the standard compiler tool treesitter[1]. Our pre-processing mainly refers to the data flow information based on the variable relation graph $G(V, E)$ [9], where $V = \{v_1, v_2, ..., v_l\}$ represents the variable sequence identified from the leaves of abstract syntax tree (AST) by depth-first search (DFS), and E denotes the set of directed edges of V for modeling the dependency relation between variables.

The structural variable sequence (V) can be then concatenated with the source code tokens to enrich the description of the code. To help C3P understand these two types of segments properly, we add $[CLS]$ as a special token for downstream classification tasks and $[SEP]$ as a symbol to separate different segments between textual and structural information. We denote the $[EOT]$ token as an extra end token in each input sequence. In the outputs of code and comment encoder, the activation on the last layer in the encoder for the $[EOT]$ token is treated as the feature representation of the sequence, which is then layer normalized and linearly projected into the multi-modal latent space for the subsequent contrastive learning. The reconstructed code information can be denoted as $X = \{[CLS], C, [SEP], V, [EOT]\}$, and the corresponding comment sequence can be written as $Y = \{[CLS], W, [EOT]\}$.

We plan to keep up to K_X valid tokens in each sequence X in a batch during the training, and thus a mask rule needs to be defined to handle the input sequences of different lengths:

$$[X\_MASK]_j = \begin{cases} 1, & j \leq \min(|X|, K_X), \\ 0, & else, \end{cases} \quad \forall 1 \leq j \leq |X| \tag{1}$$

where $[X\_MASK]_j$ denotes the binary mask value of the j-th token in the sequence X and $|X|$ is the length of the

[1]https://github.com/tree-sitter/tree-sitter

sequence. Similarly, we define the following mask rule for the sequence Y:

$$[Y\_MASK]_j = \begin{cases} 1, & j \leq \min(|Y|, K_Y), \\ 0, & else, \end{cases} \quad \forall 1 \leq j \leq |Y| \tag{2}$$

where $[Y\_MASK]_j$ denotes the binary mask value of the j-th token in the sequence Y, $|Y|$ is the length of the sequence, and K_Y is the comment sequence length to be processed by the encoder.

C3P aims to learn two BERT-based encoders (code encoder $\mathcal{F}_{\theta_X}$ and comment encoder $\mathcal{F}_{\theta_Y}$) parameterized by θ_X and θ_Y, to extract the latent feature representations of the code and comment sequences X and Y respectively. Following the design in [9], the network backbone of the encoders $\mathcal{F}(\cdot)$ is set as a 12-layer 512-max wide model with eight self-attention heads of BERT [31]. The C3P feeds the wrapped batch pairs (X, Y) to the encoders and derive the embeddings of the code and comment in the following process:

$$r_X \leftarrow \mathcal{F}(X \odot [X\_MASK]; \theta_X) \tag{3}$$

$$r_Y \leftarrow \mathcal{F}(Y \odot [Y\_MASK]; \theta_Y), \tag{4}$$

where $\odot$ stands for the mask operation between the binary mask and the corresponding sequence.

B. Contrastive Learning

The contrastive pre-training is designed to discover discriminative code-comment pairwise information in the projection space. We map each encoder with linear projection.

Assume X, Y represent the reconstructed code tokens and source comment tokens. Given a batch of size N of unlabeled samples $\{(X_i, Y_j)\}_{i,j=1}^N$, the C3P is responsible for predicting the correct pairs $\{(X_i, Y_j)\}_{i=j}$ and pulling unmatched pairs $\{X_i, Y_j)\}_{i \neq j}$ away. We adopt cosine similarity to measure similarity between the code latent features r_X and comment latent features r_Y as formulated by:

$$S(r_X, r_Y) = \frac{r_X^T r_Y}{\|r_X\| \|r_Y\|}, \tag{5}$$

where $\|\cdot\|$ is L_2 norm and $S(r_X, r_Y)$ represents the similarity bewteen the code embedding and the comment embedding. For contrastive learning, we follow the symmetric loss settings of [16] and [32]. Let $\mathcal{L}_{X2Y}, \mathcal{L}_{Y2X}$ represent the code-to-comment loss and comment-to-code loss that employ logits softmax function on the latent representation. Given a batch of reconstructed codes and comments, the loss $\mathcal{L}_{X2Y}$ is defined to match each code to its corresponding comment:

$$\mathcal{L}_{X2Y} = -\sum_{i=1}^{N} \log \frac{exp(S(r_X^i, r_Y^i)/\tau)}{\sum_{j=1}^{N} exp(S(r_X^i, r_Y^j)/\tau)}, \tag{6}$$

where τ is the temperature parameter. Similarly, we have $\mathcal{L}_{Y2X}$ to match each comment to its corresponding code:

$$\mathcal{L}_{Y2X} = -\sum_{i=1}^{N} \log \frac{exp(S(r_X^i, r_Y^i)/\tau)}{\sum_{j=1}^{N} exp(S(r_X^j, r_Y^i)/\tau)}. \tag{7}$$

The resulting objective function for batch B is as follow:

$$\mathcal{L}_{(X,Y)\in B} = \frac{1}{2N} \left(\mathcal{L}_{X2Y} + \mathcal{L}_{Y2X} \right). \tag{8}$$

The optimization goal in the pre-training stage is to jointly maximize the similarity between paired code-comment while minimizing the cosine similarity of the projected latent features of the incorrect pairs.

Algorithm 1: Contrastive Pre-Training with Code (C) and Comment (W).

```
Data: Code-comment pairs datasets $\mathcal{D}$ comprised of
      batches
Result: Code and comment encoders $\mathcal{F}_{\theta_X}$, $\mathcal{F}_{\theta_Y}$
        transferable to downstream tasks
1  Initialization of $\mathcal{F}_{\theta_Y}$ with words of BERT-case;
2  Initialization of $\mathcal{F}_{\theta_X}$ with code corpus;
3  for Each batch $B \in \mathcal{D}$ do
4      for Each $C, W \in B$ do
5          Reconstruct code and comment by parsing
            abstract syntax tree:
6          $X \leftarrow \{[CLS], C, [SEP], V, [EOT]\}$;
7          $Y \leftarrow \{[CLS], W, [EOT]\}$;
8          Encode reconstructed pairs with
            $[X\_MASK], [Y\_MASK]$ :
9          $r_X \leftarrow \mathcal{F}(X \odot [X\_MASK]; \theta_X)$;
10         $r_Y \leftarrow \mathcal{F}(Y \odot [Y\_MASK]; \theta_Y)$;
11     Get all reconstructed codes and comments:
        $\{(X_i, Y_j)\}_{i,j=1}^N$;
12     Compute similarities between encodings of codes
        and comments in the batch: $S(r_X^i, r_Y^j)$;
13     Compute symmetric loss with $\mathcal{L}_{X2Y}, \mathcal{L}_{Y2X}$:
        $\mathcal{L}_{(X,Y)\in B} = \frac{1}{2N}(\mathcal{L}_{X2Y} + \mathcal{L}_{Y2X})$;
14     Joint optimization of the symmetric objective:
        $\text{argmin}_{\{\theta_X, \theta_Y\}} \mathcal{L}_{(X,Y)\in B}\left(\{(X_i, Y_j)\}_{i,j=1}^N \middle| \mathcal{F}_{\theta_X}, \mathcal{F}_{\theta_Y}\right)$.
```

The contrastive pre-training process can be summarized as Algorithm 1. With the parsing process in Algorithm 1 from line 5 to line 6, we reconstruct code information with textual tokens and structural graph variables by parsing abstract syntax tree. As the line 11 to line 14 of the pre-training process demonstrated, we employ the symmetric objective in the latent space for the two encoders to jointly learn the multi-modal representation.

C. Fine-Tuning

In the fine-tuning stage, following the pre-trained encoders in C3P, we introduce specialized decoders for different downstream tasks. Each encoder-decoder neural network is optimized by the AdamW optimizer with a scheduler.

Code Search. The backbone decoder for code search is of two fully connected layers. The last layer outputs a normalized probability over all set-id of candidate code snippets. We just fine-tuned one epoch for the task-specific decoder.

Code Clone. The backbone decoder for code clone is a three-layered MLP with the GELU activations that define what is to be fired to the next neural layer. The output of the decoder is a two-element vector that represents the probability distribution of positive and negative. We set only one epoch for fine-tuning following the same settings as [9], [13].

Code Translation. The decoder for translation is of a multi-head attention structure with 6 layers and 12 heads. The decoder is fed with both outputs of encoders on [EOT] tokens and the inputs $\{[CLS], V, [SEP], E, [SEP]\}, \{[X\_MASK]\}$ as shown in the sub-section III-A. We apply the auto-frozen strategy for fine-tuning on this generation task. If the model detects the loss has no sign of dropping on the validation dataset, the model will freeze all layers but the last layer of the encoder and fine-tune the parameters of the decoder.

Document Generation. The architecture of the decoder is of the same number of layers and hidden size as pre-trained models. The output of the decoder is a sequence of tokens that represents the natural language description. We fine-tuned 50 epochs, which follows the experimental pipeline in [13].

Model	Encoder			Decoder of Understanding			Decoder of Generation			
	#H	#D	#Bl	Task	#L	#W	Task	#H	#D	#Bl
C3P	12	768	6	S	2	{512, 128}	T	12	768	6
	12	768	6	C	3	{1024, 512, 64}	G	12	768	6

TABLE I: The Backbone of Decoders for Downstream Understanding and Generation Tasks: Code Search (S), Clone Detection (C), Code Translation (T) and Document Generation (G). #L: number of fully-connected hidden layers. #W: width of each fully-connected hidden layer. #H: number of attention heads. #D: dimensions of each embedding layer. #Bl: number of multi-head attention blocks. We also provide small size on the settings of 3/6 on #H/#Bl and medium size on the settings of 6/6 on #H/#Bl for the ablation study V.

Implementation Details. Table II shows the training recipe in the pre-training and fine-tuning stage. In the pre-training stage, we apply the same training recipe of [9] for C3P. All models are trained for 500 epochs with a batch size of 1024 optimized by AdamW [33]. The learning rate for the optimizer is $1e{-}3$ with cosine decay strategy. We transfer the pre-trained encoders for the understanding task by concatenating with MLP-based decoders and apply the pre-trained encoders fine-tuning on the generation tasks with attention-based decoders, as demonstrated in Table I. Specifically, we freeze the pre-trained embeddings and fine-tune one epoch on each task for the understanding task, while we don't freeze the embeddings in the fine-tuning stage for generation tasks.

	EPs	BS	OP	LR	LD	WU	FE
Pre-Training Task	500	1024	AdamW	2e-4	cosine	10	-
Fine-Tuning on DoU	1	1024	AdamW	1e-6	-	-	✓
Fine-Tuning on DoG	100	1024	AdamW	2e-5	cosine	5	✗

TABLE II: Pre-Training and Fine-Tuning Recipes of the C3P. CS: Whether adding code structure representation; DoU: Decoders of understanding tasks; DoG: Decoders of generation tasks; EPs: Epochs; BS: Batch size; OP: Optimizer; LR: Learning rate; LD: Learning decay; WU: Warm-up; FE: Freeze pre-trained embeddings.

IV. Performance of C3P vs. Other Methods

We evaluate C3P on four fundamental tasks: code search, clone detection, code translation and document generations. All of them are compared with previous well-documented models on the same experimental datasets.

We include the following important baselines for pre-training in the comparison experiments:

- RoBERTa [6]: A Robustly Optimized BERT Pre-training method. We pre-trained RoBERTa with parsed source codes.
- CodeBERT [13]: A BERT-based Model pre-trained with mask language modeling (MLM) and replaced token detection (RTD) on code corpus.
- GraphCodeBERT [9]: A BERT-based Model pre-trained by parsing data flow from abstract syntax tree on programming languages.

Besides these pre-trained methods, we also consider the representative supervised methods that have been particularly developed for each downstream task in the experiments.

A. Code Search

Code search task targets finding correct code snippets from a candidate set by giving a query of natural language description. We evaluate our model on the CodeSearchNet code corpus [34] and the filtering rules follow the settings of GraphCodeBERT [9] on this task. The dataset is composed of both uni-modal and bi-modal data. The statistics of dataset are shown in Table IV.

Since the decoder outputs a list of possible responses to a sample of queries ordered by the probability of matching, we select a widely used evaluation method, i.e., mean reciprocal rank (MRR), to measure the score. For each query q, the score is computed by:

$$MRR = \frac{1}{|C|} \sum_{q \in C} \frac{1}{rank_q} \qquad (9)$$

where C is the candidate set of code snippets and $rank_q$ stands for rank list of query q.

The baselines for code search include supervised and unsupervised pre-trained models that are public and well documented. The first group of Table III shows the previous supervised experimental records on this dataset. NBow is the bag of words model that is particularly good at keyword matching on code. CNN and Bi-RNN were implemented on this task by [34]. The results of Self-Attention [5] obtain the best performance among the supervised models. And then, pre-trained models RoBERTa [6], CodeBERT [13], GraphCodeBERT [9], PLBART [11] and CodeT5 [12] recorded in the second group of Table III boost the performance on this task.

	Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Supervised	NBow [13]	0.162	0.157	0.330	0.161	0.171	0.152	0.189
	CNN [13]	0.267	0.224	0.680	0.242	0.263	0.260	0.324
	BiRNN [13]	0.213	0.193	0.688	0.290	0.304	0.338	0.338
	SelfAtt [13]	0.275	0.287	0.723	0.398	0.404	0.426	0.419
Pre-trained	RoBERTa [13]	0.587	0.517	0.850	0.587	0.599	0.560	0.617
	RoBERTa (code) [13]	0.628	0.562	0.859	0.610	0.620	0.579	0.643
	CodeBERT [13]	0.679	0.620	0.882	0.672	0.676	0.628	0.693
	GraphCodeBERT [9]	0.703	0.644	0.897	0.692	0.691	0.649	0.713
	PLBART [11]	0.675	0.616	0.887	0.663	0.663	0.611	0.685
	CodeT5 [12]	0.719	0.655	0.888	0.698	0.696	0.645	0.716
Ours	**C3P**	**0.756**	**0.677**	**0.906**	**0.704**	**0.759**	**0.684**	**0.748**

TABLE III: Downstream Task 1: Code Search Performance Measured by Mean Reciprocal Rank (MRR).

PL	Train	Valid	Test	Candidates
Python	$251,820$	$13,914$	$14,918$	$43,827$
Php	$241,241$	$12,982$	$14,014$	$52,660$
Go	$167,288$	$7,325$	$8,122$	$28,120$
Java	$164,923$	$5,183$	$10,955$	$40,347$
Javascript	$58,025$	$3,885$	$3,291$	$13,981$
Ruby	$24,927$	$1,400$	$1,261$	$4,360$

TABLE IV: Dataset of Code Search

However, as the last line of Table III shows, our proposed approach achieved better results as compared to other public recorded methods, with the improvement of 3.7%, 2.2%, 0.9%, 0.6%, 6.3%, 3.5% on the programming language Ruby, Javascript, Go, Python, Java, Php.

B. Clone Detection

Code clone detection is to measure whether the two code snippets have the same functionality.

	Train	Valid	Test
Clone Pairs	$901,028$	$415,416$	$415,416$

TABLE V: Dataset of Code Clone

We employ the experiments on the dataset BigCloneBench [35] that consists of known true and false positive clones in a big data inter-project Java repository. We consider a variety of baselines, including supervised learning models and pre-trained models by fine-tuning. The supervised baselines we compared include Deckard [36], RtvNN [37], CDLH [38] and ASTNN [39]. Deckard [36] is an algorithm to obtain the similarity degree from the structure of abstract syntax tree and locality sensitive hashing. RtvNN [37] applies auto-encoder to reconstruct the abstract syntax tree. CDLH [38] transfers the abstract syntax tree to the LSTM network [40]. ASTNN [39] stores a subset of AST information by RNN network and leverages the naturalness of statements to produce the vector representation. FA-AST-GNN [41] applies two different types of graph neural networks (GNN) on the flow-augmented abstract syntax tree (FA-AST).

The pre-trained models contain the RoBERTa [31], CodeBERT [13], GraphCodeBERT [9], PLBART [11] and CodeT5 [12]. Each of them is fine-tuned with a multi-layer perceptron (MLP) decoder. We adopt Precision, Recall, and F1 as evaluation metrics. As Table VI shows, our proposed C3P surpasses both supervised and pre-trained baselines.

Model	Precision	Recall	F1
Deckard [13]	0.93	0.02	0.03
RtvNN [13]	0.95	0.01	0.01
CDLH [13]	0.92	0.74	0.82
ASTNN [13]	0.92	0.94	0.93
FA-AST-GNN [41]	0.96	0.94	0.95
RoBERTa (code) [13]	0.960	0.955	0.957
CodeBERT [13]	0.964	0.966	0.965
GraphCodeBERT [13]	0.973	0.968	0.971
PLBART [11]	-	-	0.972
CodeT5 [12]	-	-	0.972
C3P	**0.979**	**0.978**	**0.979**

TABLE VI: Downstream Task 2: Code Clone Performance Measured by Precision, Recall and F1.

Compared with the best supervised model FA-AST-GNN [41] in the first group, our model achieves 1.9%, 2.8%, 2.9% improvement on Precision, Recall, and F1. Also as listed in the second group of Table VI, C3P is the best performing model among the pre-trained methods on this task, with 0.6%, 1%, 0.8% enhancement on the three evaluation metrics.

C. Code Translation

Code translation aims to migrate the code functionality from one programming language to another, which relieves the workload of transferring existing projects to another programming language.

	Train	Valid	Test
Translation Pairs	$10,300$	500	1000

TABLE VII: Dataset of Code Translation

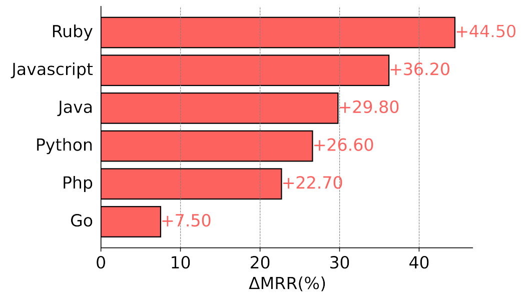
The experimental dataset and settings follow the mppSMT [42], attention-based tree [43], CodeBERT [13] and GraphCodeBERT [9], as shown in Table VII. We report the standard

	Java $\rightarrow$ C#		C# $\rightarrow$ Java	
Model	**BLEU**	**Acc**	**BLEU**	**Acc**
Naive [13]	18.54	0.00	18.6	0.00
PBSMT [13]	43.53	12.50	40.06	16.10
Transformer [13]	55.84	33.00	50.47	37.90
RoBERTa (code) [13]	77.46	56.10	71.99	57.90
CodeBERT [13]	79.92	59.00	72.14	58.00
GraphCodeBERT [9]	**80.58**	59.40	72.64	58.80
C3P	78.91	**59.73**	**73.81**	**59.42**

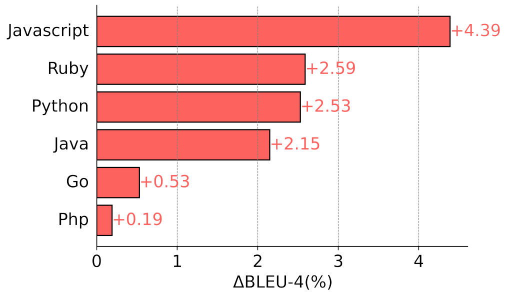
TABLE VIII: Downstream Task 3: Code Translation Performance Measured by BLEU-4 and Accuracy.

generation metrics BLEU-4 and accuracy of the C3P against other previous models in Table VIII. The first group is the results of traditional supervised methods. The naive method is to directly copy original code fragments without modifications. We also cite the Transformer model results [13] on this dataset since the success of the attention mechanism on text processing. The results listed in the first group demonstrate our model significantly outperforms supervised SOTA models with 23.07%, 23.34% improvement on the task of Java transferring to C-sharp, and 26.73%, 26.73% improvement on the task of C-sharp transferring to Java.

For the pre-training approaches, we employ the same decoders of multi-attention backbone as the previous generation decoders [9], [13] to make sure the results can be fairly compared. The results indicate that the C3P is competitive to previous works on the task of Java transferring to C-sharp while outperforming all previous methods evidently on the task of C-sharp transferring to Java.



(a) Code Understanding Task.



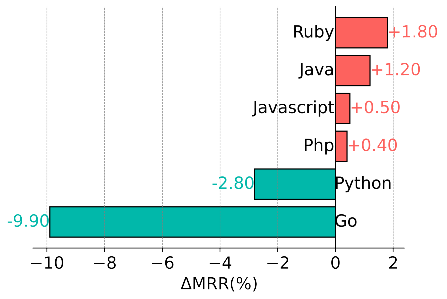
(b) Code Generation Task

Fig. 3: Transferability: Improvement of C3P (w/o PL) over supervised SOTA trained on full datasets including PL.

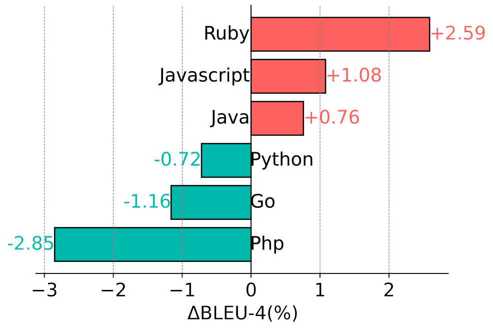
D. Code Document Generation

Document generation aims to auto-summarize the code snippet by producing the natural language description. Since CodeBERT [13] provides a solid and well-documented baseline, our experimental datasets and settings follow their work. Table IX lists the performance of the C3P compared with the previous pre-training works.

Seq2Seq [44] employs a multilayered recurrent neural network to map the sequence to fixed-length vectors and then decodes the latent vectors to a target sequence. The model is transferred well to code2seq [7] that leverages the syntactic structure of programming languages to natural language sequence.



(a) Code Understanding Task.



(b) Code Generation Task.

Fig. 4: Transferability: Improvement of C3P (w/o PL) over pre-trained SOTA using full datasets including PL.

Table IX demonstrates the performance of document generation measured by BLEU-4. The score of our model has remarkably improved compared with the best result of previous works on the six programming languages.

E. Transferability

To study the transferability of our model to new programming languages, we try to train the model C3P (w/o PL) by removing one of the programming languages (PLs) from the pre-training and fine-tuning set. For example, C3P (w/o Python) means the model pre-trained and fine-tuned on the datasets without Python codes, which is then evaluated on Python datasets in downstream tasks. Since C3P (w/o Python) has not seen the code style of Python before the test stage, the transferability of the model (to Python) can be evaluated through the test performance. As shown in Table X, we test the transferability of six programming languages by removing one language at a time in the training stage. We show the performance on two downstream tasks: code search (code understanding) and document generation (code generation), and compare the results with fully supervised / pre-trained SOTA models. For code understanding task, we adopt the same evaluation scripts of MRR as [9]. For code generation task, we adopt the same evaluation function BLEU-4 as [13].

Fig. 3 and Fig. 4 illustrate the improvement brought by our models. We first compare C3P (w/o PL) with fully supervised SOTA models trained on the whole datasets including the

Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Seq2Seq [13]	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 [13]	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT (RTD) [13]	11.42	13.27	17.53	18.29	17.35	24.10	17.00
CodeBERT (MLM) [13]	11.57	14.41	17.78	18.77	17.38	24.85	17.46
CodeBERT (MLM+RTD) [13]	12.16	14.90	18.07	19.06	17.65	25.16	17.83
C3P	**14.67**	**14.98**	**18.43**	**19.12**	**18.09**	**25.88**	**18.52**

TABLE IX: Downstream Task 4: Code Document Generation Performance Measured by Smoothing BLEU-4.

	Model	Ruby	Javascript	Go	Python	Java	Php	Overall
Code Understanding (MRR)	Supervised SOTA [13]	0.275	0.287	0.723	0.398	0.404	0.426	0.419
	Pre-trained SOTA [9]	0.703	0.644	**0.897**	**0.692**	0.691	0.649	0.713
	C3P (w/o Ruby)	0.721	0.662	0.873	0.679	**0.737**	0.659	0.721
	C3P (w/o Javascript)	**0.739**	0.649	0.885	0.681	0.705	**0.672**	**0.722**
	C3P (w/o Go)	0.727	0.637	0.798	0.675	0.694	0.661	0.698
	C3P (w/o Python)	0.716	0.648	0.865	0.664	0.693	0.655	0.706
	C3P (w/o Java)	0.732	0.625	0.874	0.680	0.703	**0.672**	0.714
	C3P (w/o Php)	0.719	**0.652**	0.884	0.690	0.677	0.653	0.712
Code Generation (BLEU-4)	Supervised SOTA [13]	11.18	11.59	16.38	15.81	16.26	22.12	15.56
	Pre-trained SOTA [9]	12.16	14.90	18.07	19.06	17.65	25.16	17.83
	C3P (w/o Ruby)	**13.77**	15.31	17.41	17.81	16.32	**25.73**	17.72
	C3P (w/o Javascript)	12.08	15.98	18.27	**19.35**	**18.51**	25.49	**18.28**
	C3P (w/o Go)	13.36	**16.14**	16.91	16.13	17.92	23.52	17.33
	C3P (w/o Python)	12.79	14.13	18.47	18.34	17.24	24.85	17.63
	C3P (w/o Java)	12.84	14.67	**18.94**	17.44	18.41	23.10	17.56
	C3P (w/o Php)	13.02	13.39	17.91	18.29	16.74	22.31	16.94

TABLE X: Transferability. C3P (w/o PL) is compared with the best available supervised SOTA and pre-trained SOTA.

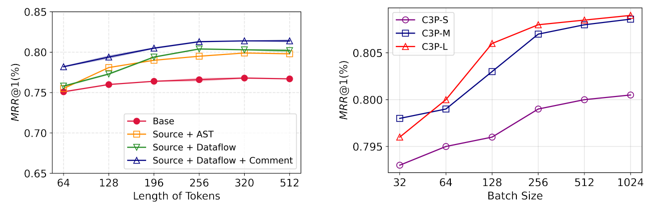
specific PL language. As shown in Fig. 3 (a), in the code understanding task, the scores of our models without prior PL information achieve considerable enhancement, respectively increased by 44.50%, 36.20%, 29.80%, 26.60%, 22.70%, 7.50%. Also, Fig. 3 (b) demonstrates that in the code generation task, C3P (w/o PL) surpasses all supervised specific-task models with different degrees of improvement, with margins from 4.39% to 0.19% on the six programming languages.

We then compare the transfer performance of C3P (w/o PL) with SOTA fine-tuned models pre-trained on the whole datasets including PLs. Fig. 4 (a) shows the comparison of MRR scores on the six programming languages in the code understanding task. We observe that our models achieve enhancement of 1.8%, 1.2%, 0.5%, 0.40% on Ruby, Java, Javascript, Php tasks. However, without prior information on Python and Go, the performance of the model falls behind the model pre-trained and fine-tuned on the whole datasets that include these two languages. Fig. 4 (b) summarises the comparison in the code generation task. Without adding PLs in the pre-training stage, our models are still competitive with fully pre-trained models on Ruby, Javascript and Java tasks, while on Python, Go and Php tasks, our models achieve slightly worse results than fully pre-trained CodeBERT [13].

V. Ablation Study

We present the performance of C3P vs. other methods on a variety of downstream tasks and then analyze the transferability of the models pre-trained without one of programming languages in section IV. In this section, we provide more ablation study results by removing certain components and compare with other training strategies on the same experimental settings as [9].

Impact of Token Length and Batch Size. The performance of contrastive learning is commonly influenced by the length of the inputs and the batch size [16]. We run extensive experiments on studying the hyperparameters for different learning strategies. To remove the impact of specific decoders, we evaluate our approach without fine-tuning on the Encoder-Only Task (Code Search). The Fig. 5(a) shows the impact of token length on the performance using different representations. We observe that the injection of code structure information improves the results on the code search task, and introducing comment encoder boosts the performance further. The Fig. 5(b) demonstrates the effect of batch size on the small, medium and large sizes of C3P. The backbone of large size is the same as experiment settings in section IV. We observe that enlarging batch size in the pre-training stage boosts the performance on the different size of pre-traind models.



(a) Impact of Token Length with Multi-Modalities Representation. (b) Impact of Batch Size on Different Size of Encoders

Fig. 5: Ablation Study on Input Length and Batch Size. Tested on the code search task by pre-trained encoders WITHOUT fine-tuning. The score is measured by MRR(%). C3P-S/C3P-M/C3P-L represent the small/medium/large size of backbone presented in Table I respectively.

Fusion Strategy and Impact of Fine-Tuning. To analyze the fusion of heterogeneous features of code and text, we validate our method compared with the approach of straightforwardly concatenating inputs for the a single encoder with-

Strategy	CS	WU	FT	FE	Valid		Test	
					MRR@1	MRR@5	MRR@1	MRR@5
Single Encoder	✓				72.47	78.19	69.26	75.47
		✓			70.28	81.27	68.05	73.29
	✓		✓		75.83	84.18	71.02	80.36
	✓		✓	✓	75.12	87.42	73.06	85.39
	✓	✓	✓	✓	80.17	88.42	73.53	82.95
Two Encoders	✓				74.15	82.09	69.46	76.54
		✓			73.92	78.44	70.25	78.23
	✓		✓		78.37	88.02	71.98	86.32
	✓		✓	✓	76.26	88.06	74.24	85.73
	✓	✓	✓	✓	82.04	90.07	74.80	87.85

TABLE XI: Ablation Study. Performance comparisons with encoding strategy on the validation set and test set. CS: Whether adding code structure representation; WU: Warp-up; FT: Fine-tuning; FE: Freeze pre-trained embeddings. Each model with ✓ in the FT option is fine-tuned for just one epoch.

out distinguishing codes and comments as in [9]. To ensure the pre-trained models work rather than specific decoders on the downstream task, we also track the impact of proposed models without fine-tuning by specific decoders. The experiment records shown in Table XI demonstrate our pre-training method consistently yields a competitive performance than single encoder on both the valid set and test set. Furthermore, we observe that the C3P boosts the accurate scores more with prior knowledge of natural language description, when enlarging the acceptance rate from top 1 to top 5 (MRR@1 vs. MRR@5).

VI. Case Study

We provide four cases to demonstrate the outputs of C3P on the downstream tasks. For the case of code search in table XII, C3P successfully finds the correct code fragment when given the query of extracting video ID from URL. In terms of code clone demonstrated in Table XIII, the C3P captures the difference on the keywords **'a'** and **'b'** between the two code fragments by giving a relatively low similarity score. A case of comparison between source code and translated code snippet on the code translation task is given in Table XIV. In this example, the C3P transfers a piece of Java code fragment to C-sharp version and successfully captures the difference by adding the definition of type variable. In the case in Table XV, C3P successfully outputs precise natural language description to summarise the code function of determining if the value is within a range.

Query	Extracts video ID from URL.

Code

```
import utils.match as match
def get_vid_from_url (url):
    path1='youtu\.be/ ([^?/]+)'
    path2='youtube\.com/emb/([^/?]+)'
    path3='youtube\.com/v/ ([^/?]+)'
    return match (url, path1) or\
           match (url, path2) or\
           match (url, path3)
```

TABLE XII: A Case of Code Search Output by C3P.

Code1	**Code2**
`import numpy as np`	`import numpy as np`
`def f (array):`	`def f (array):`
`    a=np.sum (array)`	`    a=np.sum (array)`
`    b=np.mean (array)`	`    b=np.mean (array)`
`    return b`	`    return a`
similarity=0.1	

TABLE XIII: A Case of Code Clone Output by C3P.

Java

```
public void removePresentation\
    Format ()
    {
    remove1stProperty (PropertyIDMap\
    .PID_PRESFORMAT);
}
```

C-sharp

```
public void RemovePresentation\
    Format ()
    {
    MutableSection s =  (Mutable\
    Section)FirstSection;
    s.RemoveProperty (PropertyIDMap.\
    PID_PRESFORMAT);
}
```

TABLE XIV: A Case of Code Translation from Java to C-sharp Output by C3P.

Code

```
function inRange  (value, min, max)
{
    const int = parseInt (value, 10)
    return  (
      `${int}` === `${value.\
      replace (/^0/, '')}` &&
      int >= min &&
      int <= max
    )
}
```

NL tokens	['Determine', 'if', 'value', 'is', 'within', 'a', 'numeric', 'range']

TABLE XV: A Case of Document Generation Output by C3P.

VII. Conclusion

In this paper, we introduce a contrastive pre-training method to understand multi-modal representational features of both programming and natural syntax. We conduct extensive experiments on the same pre-training and fine-tuning recipe as benchmarks. It turns out that our model evidently outperforms previous works on multiple code-related downstream tasks including code search, clone detection, code translation and document generation. Our model also has impressive generalization ability and transferability. Its performance on new programming languages that have not been seen during training surpasses all supervised methods, and is competitive with pre-training methods trained with all programming languages. Finally, to investigate the impact factors of our proposed method, we provide detailed ablation study of fusion strategy, code representation, token length and batch size.

References

[1] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[2] J. McBroom, B. Paassen, B. Jeffries, I. Koprinska, and K. Yacef, "Progress networks as a tool for analysing student programming difficulties," in *Australasian Computing Education Conference*, 2021, pp. 158–167.

[3] T. H. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[4] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu, "Pyart: Python api recommendation in real-time," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1634–1645.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[6] 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.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[8] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," 2019.

[9] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[11] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[12] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[14] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[15] N. Rethmeier and I. Augenstein, "A primer on contrastive pretraining in language processing: Methods, lessons learned and perspectives," *arXiv preprint arXiv:2102.12982*, 2021.

[16] M. V. Conde and K. Turgutlu, "Clip-art: contrastive pre-training for fine-grained art classification," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3956–3960.

[17] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," in *International Conference on Machine Learning*. PMLR, 2021, pp. 8748–8763.

[18] N. Rethmeier and I. Augenstein, "Data-efficient pretraining via contrastive self-supervision," 2021.

[19] T. Klein and M. Nabi, "Contrastive self-supervised learning for commonsense reasoning," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7517–7523.

[20] X. Duan, H. Yu, M. Yin, M. Zhang, W. Luo, and Y. Zhang, "Contrastive attention mechanism for abstractive sentence summarization," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3044–3053.

[21] Y. Qu, D. Shen, Y. Shen, S. Sajeev, J. Han, and W. Chen, "Coda: Contrast-enhanced and diversity-promoting data augmentation for natural language understanding," *arXiv preprint arXiv:2010.08670*, 2020.

[22] D. Iter, K. Guu, L. Lansing, and D. Jurafsky, "Pretraining with contrastive sentence objectives improves discourse performance of language models," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4859–4870.

[23] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, J. Heidecke, P. Shyam, B. Power, T. E. Nekoul, G. Sastry, G. Krueger, D. Schnurr, F. P. Such, K. Hsu, M. Thompson, T. Khan, T. Sherbakov, J. Jang, P. Welinder, and L. Weng, "Text and code embeddings by contrastive pre-training," 2022.

[24] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[25] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, "Code clone detection—a systematic review," *Emerging Technologies in Data Mining and Information Security*, pp. 645–655, 2021.

[26] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[27] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[28] 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 *ICSE*, 2019, pp. 783–794.

[29] 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.

[30] P. Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," University of Southern California Marina Del Rey Information Sciences Inst, Tech. Rep., 2003.

[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[32] F. Yu, J. Tang, W. Yin, Y. Sun, H. Tian, H. Wu, and H. Wang, "Ernie-vil: Knowledge enhanced vision-language representations through scene graph," *arXiv preprint arXiv:2006.16934*, 2020.

[33] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[34] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[35] 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 *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[36] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105.

[37] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[38] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[39] 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 *ICSE*, 2019, pp. 783–794.

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

[41] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "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, 2020, pp. 261–271.

[42] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "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, 2015, pp. 585–596.

[43] X.-P. Nguyen, S. Joty, S. C. Hoi, and R. Socher, "Tree-structured attention with hierarchical accumulation," *arXiv preprint arXiv:2002.08046*, 2020.

[44] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.