Zero-Shot Program Representation Learning

Nan Cui, Yuze Jiang, Xiaodong Gu, Beijun Shen*
School of Software, Shanghai Jiao Tong University
Shanghai, China
{cuinan,jyz-1201,xiaodong.gu,bjshen}@sjtu.edu.cn

ABSTRACT

Learning program representations has been the core prerequisite of code intelligence tasks (*e.g.*, code search and code clone detection). The state-of-the-art pre-trained models such as CodeBERT require the availability of large-scale code corpora. However, gathering training samples can be costly and infeasible for domain-specific languages such as Solidity for smart contracts. In this paper, we propose Zecoler, a zero-shot learning approach for code representations. Zecoler is built upon a pre-trained programming language model. In order to elicit knowledge from the pre-trained models efficiently, Zecoler casts the downstream tasks to the same form of pre-training tasks by inserting trainable prompts into the original input. Then, it employs the prompt learning technique to optimize the pre-trained model by merely adjusting the original input. This enables the representation model to efficiently fit the scarce task-specific data while reusing pre-trained knowledge. We evaluate Zecoler in three code intelligence tasks in two programming languages that have no training samples, namely, Solidity and Go, with model trained in corpora of common languages such as Java. Experimental results show that our approach significantly outperforms baseline models in both zero-shot and few-shot settings.

KEYWORDS

Learning Program Representations, Zero-Shot Learning, Prompt Learning, Code Intelligence

ACM Reference Format:
Nan Cui, Yuze Jiang, Xiaodong Gu, Beijun Shen. 2022. Zero-Shot Program Representation Learning . In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524610.3527888

1 INTRODUCTION

Learning program representations has achieved great success in software engineering thanks to recent advances in deep learning [4] and the availability of large-scale code corpora [6]. Program representations, namely, code vectors that reflect their deep semantics, have been widely used in code intelligence tasks [24] such as clone detection [12], code summarization [9], and code search [16]. For example, in the clone detection task, program representations can be used to reflect similar features between two code snippets [43].

Despite showing promising results, the state-of-the-art techniques rely on the availability of large code corpora. However, labeling data samples of some programming languages is often expensive and sometimes infeasible. For example, Solidity is a new language and specifically designed for smart contracts. This language is becoming increasingly popular and hence code representation tools for the Solidity language are highly demanded. However, obtaining labelled Solidity code is challenging as it requires domain knowledge on Blockchain and the collected data is significantly redundant [8]. This restricts the collection of supervised data and causes deep learning models to learn poor representations [26].

One possible solution towards alleviating this issue is to use the pre-trained language models (PLMs) such as CodeBERT [13], PLBART [1], and CodeT5 [38]. PLMs are pre-trained to learn code representations of large common languages such as Java and then are fine-tuned on domain-specific languages such as Solidity. For example, Salza et al. [30] proposed cross-language CodeBERT, which pre-trains CodeBERT on multiple languages and then transfers the program representations to other languages through fine-tuning.

However, fine-tuning a PLM on a specific task is challenging. The learning tasks (e.g., masked language model (MLM)) in the pre-training phase are usually different from the downstream tasks (e.g, code search). As such, the reusability of prior knowledge learned in the pre-training phase may be limited in the fine-tuning phase. This is even more challenging when there is no or insufficient training data for downstream tasks in domain-specific languages. The large pre-trained model can easily overfit scarce data, which leads to poor task fitting.

In this paper, we propose Zecoler (**Ze**ro-shot **co**de representation **le**a**r**ning), a novel approach for learning program representations without labelled data samples of the target language. In order to learn better representations and bridge the gap between pre-training and downstream tasks, we adapt prompt learning [20], a new learning paradigm for PLMs. Prompt learning adapts the downstream task to the same form as that in pre-training through accompanying trainable prompt tokens with the PLM input. The continuous vectors of prompts guide the pre-trained model to elicit knowledge of programming languages efficiently. For example, the input of code clone detection can be converted to an MLM task input based on a template containing prompt tokens. In this way the model can be adapted to the target language while maximizing the use of prior knowledge learned during the pre-training phase. Hence, prompt learning allows few-shot or even zero-shot learning for pre-trained models in new languages with unlabelled data [31].

To evaluate the proposed approach, we experiment on three code intelligence tasks, including code clone detection, code search,

*Beijun Shen is the corresponding author.

and method name prediction. The results show that our approach is substantially effective in zero-shot learning of program representations. The accuracy of the three tasks in Solidity is 79.8%, 67.1%, and 68.1%, respectively, which is around 14.7% greater than the strong baseline CodeBERT. Zecoler also demonstrates great effectiveness when a few data samples are given. Moreover, the learned program representations have good generalizability.

The contributions of this paper can be summarized as:

- To the best of our knowledge, we are the first to propose zero-shot learning of program representations, which does not require manual labeling of data for code intelligence tasks.
- We propose a prompt learning based method for zero-shot program representation.
- We conduct extensive experiments to evaluate the proposed approach. Results show that our approach significantly outperforms the baseline models.

2 BACKGROUND

2.1 Pre-trained Language Models for Code

Pre-trained language models (PLMs) such as BERT [11], GPT [27], and T5 [28] have achieved a great success in the natural language processing field. A PLM is usually pre-trained on a large-scale text corpora through a series of self-supervised learning tasks such as masked language modeling (MLM) and next sentence prediction (NSP). The MLM task masks a random portion of tokens in the input text and tries to predict the masked words, while the NSP task predicts whether or not two given input segments are coherent. The PLM can then be fine-tuned on task-specific datasets. A fine-tuning header on top of the PLM is optimized via supervised learning tasks in a specific domain.

Due to the great success of PLMs, researchers also seek the adaptations of PLMs for programming languages [13, 38]. They customize the pre-training objectives using programming related tasks on a big code corpus. PLMs have been successfully used to learn program representations and further be used in a variety of code intelligence tasks.

For example, CodeBERT is built based on RoBERTa [22] and is pre-trained with both natural and programming languages. Figure 1 illustrates the main pipeline of CodeBERT. The model is first pre-trained on two tasks, namely, MLM and replaced token detection (RTD). In MLM, CodeBERT randomly masks the tokens in natural language and programming language (NL-PL) pairs and learns to predict the original words. For RTD, the model is trained to detect whether tokens are original or not. The pre-trained model is then fine-tuned on data of downstream tasks such as clone detection and code search. A header based on multilayer perceptron (MLP) is added to the PLM and is optimized with downstream tasks.

2.2 Zero-Shot Learning

The standard supervised learning approaches train a model with large-scale labelled samples. However, in many tasks such as recognizing name of a new brand or translating a new language, obtaining sufficient training samples is laborious and often impracticable. Zero-shot learning transfers a learned model to a target domain that has no labelled data, and hence alleviates this "data hungry" problem. It can be realized through a variety of techniques such as data augmentation [5], meta-learning [29, 33], and PLMs [6].

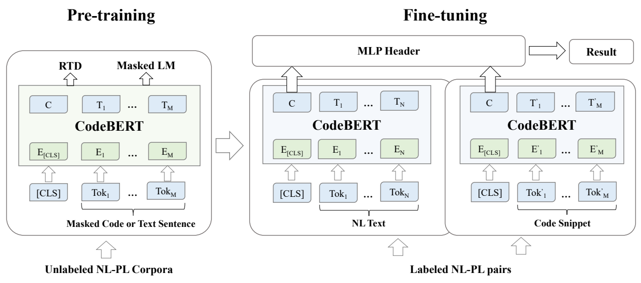


Figure 1: Illustration of the pre-trained code model.

Data augmentation. A direct technique of zero-shot learning is data augmentation, namely, enlarging the data set (e.g., randomly inserting samples and noise) so that the model can have sufficient data samples for training [5].

Meta-learning. Another popular strategy for zero-shot learning is meta-learning. Meta learning is also known as "learning to learn", which aims at training a meta learner which learns the update rules of the target model [14]. This enables a machine learning model to achieve competitive performance even with scarce data. However, meta-learning focuses on learning strategies instead of representations. Hence it will be difficult to be generalized across different code intelligence tasks.

Pre-trained Language Models. PLMs are pre-trained on large-scale text corpora to learn common knowledge of the languages, and can be generalized to specific tasks with only a few training examples. However, PLMs require a fine-tuning phase to adapt the pre-trained model to the downstream tasks, which needs the availability of labelled datasets. Therefore, it's still quite laborious to annotate the data manually.

Prompt-based Learning. To alleviate the data hungry problem of fine-tuning, Brown et al. [6] introduced prompt-based learning, a lightweight alternative of fine-tuning for PLMs. Unlike fine-tuning, which adds fine-tuning header and re-optimizes the PLM using downstream tasks, the prompt-based learning approach converts downstream tasks (e.g., method name detection) to the same form as the pre-training tasks (e.g., MLM) by injecting "prompts" and "[MASK]" to the PLM input. Hence, the PLM generates the target result after minimal adjustment. This encourages downstream tasks to reuse the knowledge from the PLM.

3 APPROACH

3.1 Problem Definition

Program Representation Learning: Let $x = \{x_1, ..., x_N\} \in \mathcal{X}$ denote a program code snippet with N tokens. The goal of program representation learning is to map x into an d-dimensional vector which contains program semantics, namely, $f_\theta : \mathcal{X} \rightarrow R^d$ [4], where f_θ is a function parameterized by θ. f_θ can be implemented using deep neural networks such as the fully-connected networks [3], LSTM [25] and Transformers [36].

The learned program vectors can further be taken as input to machine learning models for code intelligence tasks such as code

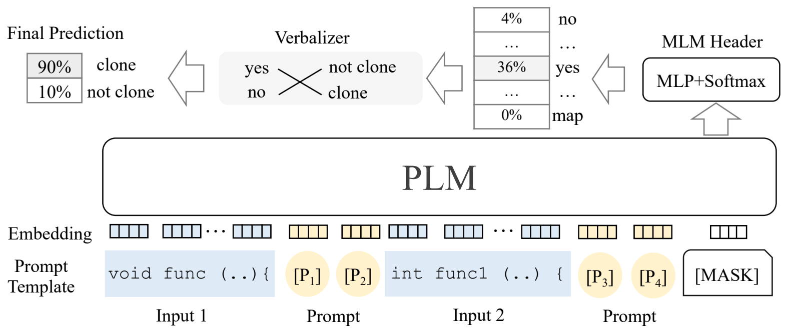


Figure 2: Model Architecture.

clone detection and code search. Given two code snippets x_1 and x_2, we can train a classifier which predicts the class label $y \in \mathcal{Y}$ given their vectors:

$$p(y|x_1, x_2) = g_\phi(f_\theta(x_1), f_\theta(x_2)), \tag{1}$$

where g_ϕ denotes the neural network model for the classification task. For example, in the code clone detection task, x_1 and x_2 stand for two code snippets and y stands for whether they contain clones. In the code search task, x_1 and x_2 stand for a code snippet and a natural language description, respectively, and y stands for whether they are semantically correlated.

Zero-Shot Program Representation Learning: Let $(\mathcal{X}^S, \mathcal{Y}^S)$ denote the labelled dataset in the source language, and $\mathcal{X}^T$ denote the unlabelled dataset in the target language. The goal of zero-shot program representation learning is to train a machine learning model which predicts the labels in the target language given the labelled data in the source language, namely, to estimate p, the probability of the label predicted:

$$p(y^T|x_1^T, x_2^T, \mathcal{X}^S, \mathcal{Y}^S), \tag{2}$$

where $x_1^T, x_2^T \in \mathcal{X}^T$ and $y^T \in \mathcal{Y}^T$. Representations for $\mathcal{X}^S$ might not be useful for $\mathcal{X}^T$ since there could be a lexical gap between $\mathcal{X}^S$ and $\mathcal{X}^T$. However, both of them can be used in pre-training tasks such as the MLM [13] for a PLM. Hence, it is feasible to bridge their representation gap using the common pre-training task. Based on this idea, we cast the problem in Equation 1 as an MLM task for a PLM and train the model on ($\mathcal{X}^S$, $\mathcal{Y}^S$), so that the PLM can also predict the class labels for data in $\mathcal{X}^T$ seamlessly.

3.2 Model Architecture

Domain-specific languages such as Solidity often have insufficient data for code intelligence tasks, while popular languages such as Java and Python have large-scale code corpora. Our goal is to transfer the representations of a popular programming language into a domain-specific language that has no training samples. As such, we keep pre-training the PLM on the task of a popular source language and then transfer the model to tasks in the target language.

Figure 2 illustrates the overall architecture of our Zecoler. The pipeline is comprised of three steps: First, we cast any downstream task to the pre-training task (e.g., MLM) by inserting trainable prompts and a "[MASK]" token into the input of the task (§3.3). Taking the resulting data as input, a PLM is then re-trained using the MLM task, that is, infers the program representation of the input and predicts the word for the "[MASK]" token (§3.4). Finally, a verbalizer is employed to cast the predicted word to class labels (§3.5).

3.3 Casting Downstream Tasks into MLM

Our first step is to cast the downstream task (Equation 1) into the MLM task. Given two input snippets x_1 and x_2 of the downstream task, we concatenate the two input snippets x_1 and x_2, inspired by NSP-BERT [34].

Like that in the MLM task, we also insert a "[MASK]" token into the concatenated input. The "[MASK]" token acts as a placeholder which steers the pre-trained model to generate the classification result y in the code intelligence task. It is notable that the position of the "[MASK]" token is a hyperparameter and we append it at the tail of the input by default. The masked sequence

$$\tilde{x} = [CLS]; x_1; x_2; [MASK] \tag{3}$$

is taken as input to the PLM, which yields the hidden states

$$\mathbf{h}_1, ..., \mathbf{h}_{|\tilde{x}|} = f_\theta(\tilde{x}) \tag{4}$$

for all tokens, where f_θ denotes the pre-trained programming language model parameterized by θ.

Then, the hidden state corresponding to the masked token, namely, $\mathbf{h}_{-1}$, is fed into an MLM header g_ϕ which predicts a token for the masked position:

$$\hat{\mathbf{y}} = \text{softmax}(g_\phi(\mathbf{h}_{-1})). \tag{5}$$

The MLM header g_ϕ is a fully connected neural network parameterized by ϕ that is optimized to minimize the cross-entropy loss:

$$L_{MLM}(\phi|\mathrm{y}, \hat{\mathrm{y}}) = -\sum_{i=1}^{|V|} y_i \log(\hat{y}_i), \tag{6}$$

where $\mathbf{y}$ denotes the ground-truth label of the code intelligence task, and $|V|$ is the vocabulary size.

3.4 Prompt-based Learning

The conventional fine-tuning method for the MLM task optimizes f_θ and g_ϕ in Equation 1 from scratch. This causes the model to overfit scarce task-specific data. Inspired by prompt-based learning [21], we optimize the PLM by merely adjusting its input sequence. More specifically, we insert a number of pseudo tokens called prompt into the input sequence of the PLM, which coaxes the PLM to directly generate the predicted label of the downstream task. By only adjusting the model input, the PLM needs far less optimization cost to fit for the data in the target task, while keeping the most of prior knowledge learned during pre-training.

Based on this idea, we design a number of prompt tokens $P = [P_1, ..., P_m]$ and inject them into the masked sequence $\tilde{x}$ using a pre-defined template $T = [P]; x_1; [P]; x_2; [P]; [MASK]$. Hence, the original inputs x_1 and x_2 are transformed into

$$\tilde{x} = [P_{1:i}]; x_1; [P_{i+1:j}]; x_2; [P_{j+1:m}]; [MASK] \tag{7}$$

through template T. For example, in the task of code clone detection, given two code snippets, "code1" and "code2", the transformed input is

$$[P_1][P_2] \text{ code1 } [P_3]\ [P_4] \text{ code2 } [P_5][P_6][\text{MASK}], \tag{8}$$

which contains six trainable prompts in this case.

Like general words, these prompt tokens are embedded into trainable vectors and are optimized on downstream tasks in the target domain through gradient descend.

In a zero-shot setting, there is no training sample in a low-resource programming language. Instead, we train the PLM using large-scale code corpora in popular languages (e.g., Java) through the converted MLM task, and then directly apply the trained model to tasks in the low-resource language. More specifically, we train the PLM through prompt learning for the converted MLM task in the source domain. Then, we take as input data samples in the target domain into the same model without extra training, and obtain the results of the code intelligence tasks.

3.5 Reverting MLM Outputs to Classification Labels

The MLM task generates a token that is likely to fill into the masked position. In order to obtain the classification result, we need to revert the MLM predictions to classification labels of the downstream task. For this purpose, we employ a verbalizer [31] which realizes such a reversion. Let $\mathcal{V}$ be the vocabulary of the PLM and $\mathcal{Y}$ be the labels of the downstream task such as {true, false}. The verbalizer is defined as a function $v: \mathcal{C} \rightarrow \mathcal{Y}$ that maps each candidate word in the vocabulary to a classification label. The choice of candidate words is arbitrary as long as they are sufficiently different. The model will be trained to map candidate words to true predictions. In our approach, we consider two candidate words $\{yes, no\} \in \mathcal{V}$ as a candidate set $\mathcal{C}$ and only inspect which word in $\mathcal{C}$ is more likely to fill into the "[MASK]" position through PLM predictions. If the word "yes" has a higher probability to fill in the masked position, the verbalizer will map it to the label "true" and hence output a positive prediction for this task.

Take code clone detection as an example. Given two code snippets, the model constructs an input sequence by injecting a number of prompt tokens into the snippets, followed by a "[MASK]" token. The constructed sequence is fed into the PLM to predict the label Y, where $Y \in$ {"cloned", "not cloned"}. The MLM header of the PLM outputs the probability of each candidate word for the masked position. If the candidate word "yes" has a higher probability, the verbalizer will map it to the class label "cloned", yielding the final prediction as "cloned".

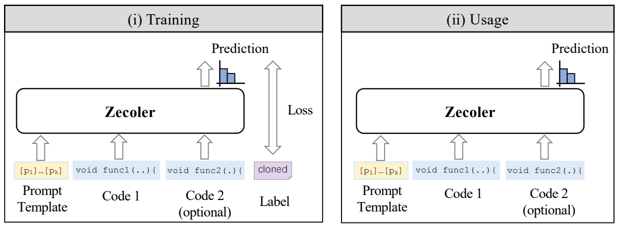


Figure 3: The workflow of Zecoler.

3.6 Training and Usage

Figure 3 shows the workflow of Zecoler. Zecoler follows the general paradigm of learning program representations. In the training phase, Zecoler is given a training set of labeled code snippets. For each snippet (pair), Zecoler augments it using a prompt template. The prompt-augmented code (pair) is taken as input to Zecoler which yields the prediction and calculates the loss function based on the ground-truth label.

In the usage phase, Zecoler is given a code snippet (pair) only. Zecoler augments it using the same prompt template as in the training phase and then gives the prediction for the downstream task.

4 EXPERIMENTAL SETUP

We implement our approach on top of CodeBERT [13], one of the most popular pre-trained PLMs, and evaluate it on three code intelligence tasks.

4.1 Research Questions

The evaluation is designed to answer the following research questions:

- **RQ1: How effective is our approach in zero-shot program representation learning?**
  We evaluate the effectiveness of Zecoler in zero-shot program representation learning. We take Java as the source language and transfer the learned model to Solidity, a domain-specific languages, and Go, an up-and-coming language which are not provided with training samples. The experiments are conducted in three popular code intelligence tasks.
- **RQ2: How effective is our approach in few-shot program representation learning ?**
  In some programming languages and tasks, we can obtain small-scale training datasets. We wonder whether Zecoler is also effective for tasks with scarce (*e.g.*, 100) data. Therefore, we provide the PLM models with a few samples of the target language and conduct the same experiments as in RQ1.
- **RQ3: How effective is our approach in monolingual program representation learning?**
  RQ1 and RQ2 mainly evaluate the effectiveness of Zecoler in a cross-language setting. We further explore how effective is our approach without transfer learning. We train the model in three languages, and test the model in the same language. Besides Solidity, we also want to assess our approach in other languages such as Java and Go when data labels are unavailable.

- **RQ4: How do different hyperparameters impact the performance of our approach?**
  Finally, we evaluate the performance of our approach under different hyperparameters. Specifically, we conduct ablation studies on prompt templates (number and position), source languages, and PLM scales.

4.2 Downstream Tasks

We evaluate our approach on three popular code intelligence tasks:

1) **Code Clone Detection**: a task that determines whether two code snippets are cloned [35] or not. A PLM based clone detection model takes as input two code snippets and outputs their representations. Then, a classification header is built on top of the representations and predicts whether the two code snippets are cloned (=1) or not (=0). There are four types of clones [39]. Our approach can challenge type-3 and type-4 clones, that is, the two snippets are not textually identical, but implement the same functionality.
2) **Code Search**: a task that retrieves a semantically relevant code snippet for a given natural language query [17]. Following CodeBERT [13], we formulate code search as a classification problem. Given a natural language description of the code and a programming language code snippet, this task aims at determining whether this NL-PL pair is related. The binary answer is "related" or "not related". The classification generates a probability score, which can be used for ranking results of code search.
3) **Method Name Prediction**: a task that suggests the function name for a given code snippet [42]. Similar to code search, we transform this task to a binary classification task [10]: given a code snippet, it enumerates all candidate function names (i.e., the vocabulary of code tokens) and constructs a "<snippet, name>" pair. The pair is taken as input to the PLM which outputs a binary prediction whether the name in the pair is "suitable" (=1) or "not suitable"(=0) for the code snippet.

4.3 Datasets

We conduct our experiments on four datasets. Each dataset is used for one or two tasks. Table 1 shows the statistics of each dataset, including sizes, languages, and corresponding tasks.

Smart Contract Clone Detection (SCCD): a manually labelled clone detection dataset for the Solidity language. It contains 10,000 data samples collected from EtherScan, an analytic platform for smart contracts. We build web scrapers to collect Solidity code snippets of smart contracts and label the cloned pairs based on contract information such as contract address and opcode. Each data sample consists of a pair of code snippets that are cloned. One notable feature of this dataset is that most samples are type-3 and type-4 clones.

Smart Contract Summarization (SCS) [41]: a code summary dataset for the Solidity language. The dataset contains 347,410 code-comment pairs. It was originally collected for code summarization. We preprocess SCS to a classification format so as to fit for the code search and method name prediction tasks. We generate the code search dataset by filtering long code and removing code comments. We generate the method name prediction dataset by separating method name from original code snippets in SCS.

CodeNet [26]: a multilingual codebase built from two online judge websites, namely, AIZU[1] and AtCoder[2]. CodeNet contains 8,008,527 code submissions in multiple programming languages such as Java, Go, Ruby, and Python. We use this dataset for the code clone detection and code search tasks. To adapt the dataset to the code clone detection task, we label two code submissions as a cloned pair if they solve the same problem. To adapt the original data to the code search task, we extract NL-PL pairs from problem descriptions and their code submissions, respectively.

CodeSearchNet [17]: a widely used code search dataset, which is used for the method name prediction task in our work. The dataset involves six languages, namely, Java, Go, Python, Javascript, Ruby and PHP, with 2,000,000 code snippets and corresponding method names.

We preprocess these datasets by filtering out comments. Code snippets with more than 250 tokens are filtered out to fit for the PLMs. We also exclude code snippets with less than 125 tokens to accommodate downstream tasks. In order to prevent the model from being biased to one class, we balance the dataset with the same number (1:1) of positive and negative samples. The negative pairs of code snippets ($y = 0$) are created using random combinations of snippets from the positive data samples ($y = 1$).

4.4 Implementation Details

We implement our models on top of the popular CodeBERT which is built based on RoBERTa-base (H=768, A=12, L=12). CodeBERT learns representations of programming languages (Java, Python, JavaScript, PHP, Ruby, and Go) in the pre-training phase. We use the default tokenizer (i.e., Microsoft/codebert-base) of CodeBERT with a vocabulary size of 50,265. We set the maximum sequence length to 512. Our experimental implementation is based on the Huggingface Transformers[3] and P-Tuning [21]. The batch size and the number of epochs are set to 10 and 20 respectively. We insert prompt tokens to the original input of CodeBERT and place them uniformly.

All models are optimized using the AdamW [23] algorithm on a machine with two GeForce RTX 2080 Ti GPUs. The initial learning rate (lr) is set to 3e-5, which linearly increases from 0 during a warm-up period. The iteration number of the warm-up period equals to the number of the training step in first epoch. Then the learning rate decreases to 0 during the rest training process. We measure the performance on the validation set during training. The checkpoint that achieves the best accuracy on the validation set are selected for testing.

4.5 Baseline Models

We compare our approach with five baseline models:

[1]https://onlinejudge.u-aizu.ac.jp/introduction
[2]https://atcoder.jp/
[3]https://huggingface.co/microsoft/codebert-base

Table 1: Overview of datasets.

Datasets	Downstream Tasks *			Size	Programming Languages
	CD	CS	MNP		
SCCD	√			10,000	Solidity
SCS [41]		√	√	347,410	Solidity
CodeNet [26]	√	√		8,008,527	Java, Go, C++, C, Python, Ruby, C#, ...
CodeSearchNet [17]			√	2,000,000	Java, Go, Python, Javascript, Ruby, PHP

* CD = clone detection, CS = code search, MNP = method name prediction.

1) **AVG**: a baseline approach that directly represents programs by averaging their token embeddings. We reuse token embeddings from CodeBERT and represent an input code snippet by the average of all token embeddings. Next, we fine-tune the classifier of downstream tasks using a 3-layer MLP header.
2) **RoBERTa** [22]: a popular pre-trained language model that has also been used for programming languages [13]. The model is constructed with 12 transformer layers and pre-trained on a large English corpus with the MLM objective. We fine-tune it with a 3-layer MLP header over the "[CLS]" position.
3) **RoBERTa-large**[4]: a large version of RoBERTa (H=1024, A=16, L=24) with around 300 million parameters, which is almost twice the size of the normal version. We compare with this model to verify the advantages of Zecoler over large-scale PLMs.
4) **CodeBERTa**[5]: a version of RoBERTa pre-trained with CodeSearchNet, which was proposed by Huggingface. We use its default setting in our experiments.
5) **CodeBERT** [13]: one of the state-of-the-art models for learning program representations. A more detailed description of CodeBERT can be found in Section 2.1. We follow the same experimental setup in its original paper.

We implement these baseline models by referring to the work of CodeXGlue [24]. We construct 3-layer fully connected neural networks as the fine-tuning header which maps the hidden vector of the "[CLS]" token to the class labels of downstream tasks.

5 RESULTS

5.1 RQ1: Effectiveness of Zero-shot Learning

In this experiment, we evaluate the effectiveness of Zecoler in zero-shot program representations learning. We initially train a representation model for each task using data samples of Java. Then, we adapt the trained model to the target languages (i.e., Solidity and Go) directly without extra training. We train the model with both 5,000 and 500 data samples of Java to assess the effects under different data sizes.

Table 2 shows the accuracy of different models in three code intelligence tasks. We can observe that Zecoler significantly outperforms baseline models in all three tasks and all target languages.

[4]https://huggingface.co/roberta-large
[5]https://huggingface.co/huggingface/CodeBERTa-small-v1

Table 2: Accuracy of program representation models on three tasks in the zero-shot setting.

Model	CD		CS		MNP	
	Solidity	Go	Solidity	Go	Solidity	Go
AVG	57.5	49.2	49.0	50.3	50.8	50.0
RoBERTa	60.5	49.4	49.6	49.4	50.0	50.3
RoBERTa-L	47.3	51.0	48.7	48.8	51.7	48.5
CodeBERTa	57.9	67.3	53.2	53.1	49.7	49.0
CodeBERT	65.4	91.7	48.9	46.2	52.1	65.2
Zecoler$_{5000}$	**79.8**	**96.4**	**67.1**	**80.3**	59.2	**98.8**
Zecoler$_{500}$	74.9	82.4	53.3	56.9	**68.1**	90.4

* The target languages (i.e., Solidity and Go) are not provided with training data. All source languages are trained with 5000 samples except the last one which is trained with only 500 samples.

In the code clone detection task, the accuracy of Zecoler is 5%-14% greater than that of CodeBERT, the strongest baseline. The improvement is much more significant in the code search (30% in average) and method name prediction (24% in average) tasks. By contrast, AVG and RoBERTa-large obtain results that are close to random, indicating that they can hardly learn useful knowledge from few data samples.

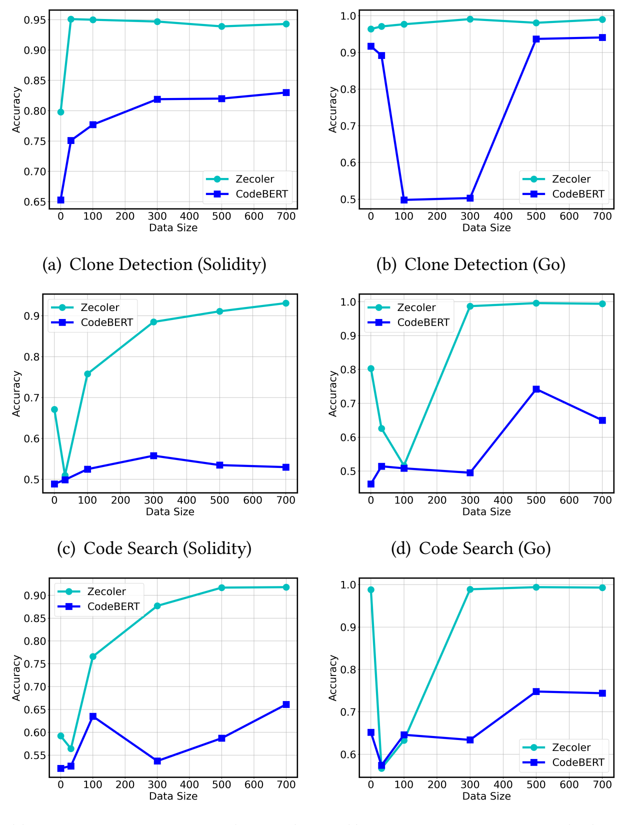
The same trend can be observed when only 500 (1/10) samples of the source language are provided for training. As the data size decreases from 5000 to 500, the accuracy of Zecoler drops in all tasks. Nevertheless, it still significantly outperforms the baseline models. This means that Zecoler can learn representations much more efficiently while requiring smaller data compared with baselines.

It is notable that CodeBERT outperforms RoBERTa and CodeBERTa in both the code clone detection and method name prediction tasks, except for the code search task. We conjecture that CodeBERT is pre-trained on programming languages whereas the RoBERTa is only pre-trained on natural languages. Hence, CodeBERT can be better adapted to PL related tasks.

Answer to RQ1: Our approach shows greater performance than baseline models in code intelligence tasks for no-resource programming languages, affirming the strong ability of Zecoler in zero-shot program representation learning.

5.2 RQ2: Effectiveness of Few-Shot Learning

In this experiment, we evaluate the effectiveness of Zecoler in few-shot learning of program representations. We continue training



(a) Clone Detection (Solidity) (b) Clone Detection (Go)

(c) Code Search (Solidity) (d) Code Search (Go)

(e) Method Name Prediction (Solidity) (f) Method Name Prediction (Go)

Figure 4: Accuracy of program representation models continuously trained with different-scale datasets in few-shot setting.

the model in RQ1 using a few data samples of the target languages. We vary the data sizes from 32 to 700 and evaluate the performance in three code intelligence tasks.

Figure 4 shows the results. Compared to CodeBERT, the strongest baseline in RQ1, Zecoler shows greater strength in all tasks when provided with a few data samples of the target language. When the data size is 700, the accuracy of Zecoler is about 30% greater than that of CodeBERT. This means that Zecoler can learn code representations more effectively when a small number of samples are available in the training dataset. We notice that when the data size is too small (e.g., 32), the model tends to overfit data. In this situation, zero-shot learning is preferred.

Answer to RQ2: Zecoler shows effective performance with a few data samples. As in zero-shot setting, Zecoler still keeps the best accuracy among all compared models in few-shot setting.

5.3 RQ3: Effectiveness of Monolingual Few-Shot Learning

Different from RQ2 in a cross-language few-shot setting, in this experiment we evaluate the effectiveness of our approach in a monolingual few-shot setting. We train models with a few samples of Java, Solidity and Go, and evaluate the performance of the tasks in the same language.

Table 3 shows the accuracy of different approaches in three downstream tasks. We can observe that Zecoler outperforms baselines in the monolingual few-shot setting. Most of the baseline models just predict random answers, with an accuracy of around 50%. This indicates that the baseline models cannot learn meaningful program representations with scarce data. Comparatively, Zecoler achieves 75.7% accuracy in average with only 100 data samples. The results suggest that Zecoler learns program representations efficiently in the monolingual few-shot setting.

Figure 5 shows the performance of Zecoler, CodeBERT, and CodeBERTa with different data sizes in the code clone detection task. We can see that Zecoler outperforms the other two baselines under almost all data sizes. Furthermore, as the data size increases, the accuracy of Zecoler grows faster than that of baseline models. This indicates that Zecoler is effective in learning program representations given only 100 or 300 data samples.

We have also observed that monolingual learning outperforms cross-language learning on small data sizes (*e.g.*, 32 and 100), but achieves similar performance when the data size becomes larger. This is because continuously training on scarce data of a different language can lead to overfitting.

Answer to RQ3: Zecoler is effective in monolingual few-shot learning, and shows much stronger performance than that in the cross-language setting.

5.4 RQ4: Ablation Study

In this experiment, we inspect the performance of Zecoler under different hyperparameters. We vary the prompt templates and numbers of prompt tokens to search for the optimal prompt template. We also explore the impact to the performance by different source languages and different scales of backbone PLMs.

Prompt Templates: We first explore the effect of prompt templates on performance. We vary the position of the prompt tokens $P_{1:k}$ in the prompt template, namely, head: [$P_{1:k}$, x_1, x_2, MASK], middle: [x_1, $P_{1:k}$, x_2, MASK], uniformly: [$P_{1:m}$, x_1,$P_{m+1:n}$, x_2,$P_{n+1:k}$, MASK] and tail: [x_1, x_2, MASK, $P_{1:k}$]. The number of prompt tokens (k) is fixed to 10. We train the model with 700 Java code snippets and evaluate the model on the code clone detection task of Solidity.

As shown in Figure 6(a), placing prompt tokens uniformly achieves the best performance compared to other templates. The reason could be that prompts have more influence to nearby tokens. By placing prompts uniformly, every input token can be influenced by sufficient prompt tokens.

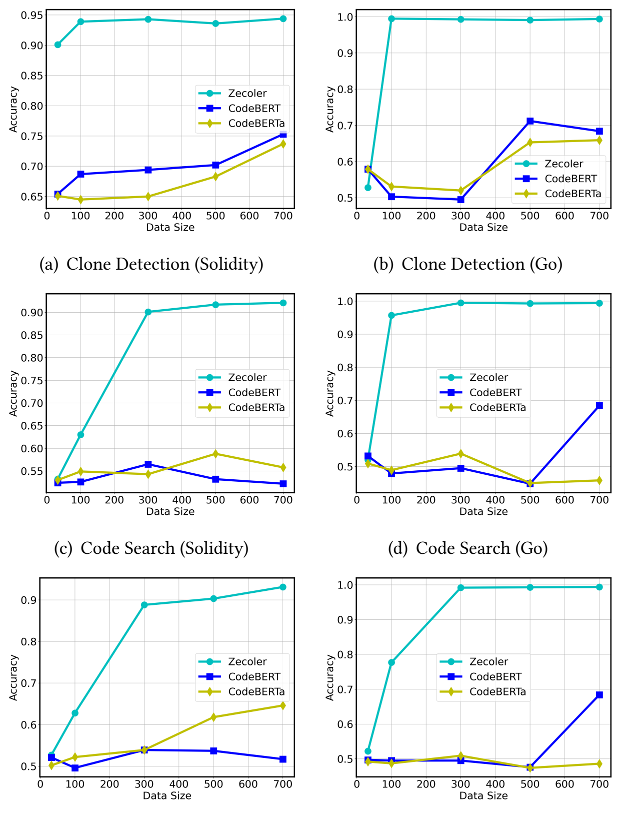
Number of Prompt Tokens: We further assess the impact of prompt numbers. We insert prompt tokens uniformly into the PLM input and vary the number of prompt tokens k from 1 to 20. We train the model with 700 Java code snippets and evaluate it on the code clone detection task of Solidity.

As Figure 6(b) shows, the number of prompt tokens is strongly correlated to the performance of representation learning. Fewer prompt tokens can be insufficient to steer the PLM to yield meaningful prediction, while large numbers of prompts can restrict the

Table 3: Accuracy of various program representation models in a monolingual few-shot setting.

Model	CD			CS			MNP			Average
	Java	Solidity	Go	Java	Solidity	Go	Java	Solidity	Go	
AVG 300	51.6	64.1	50.1	49.1	50.2	50.1	47.9	48.1	50.1	51.3
RoBERTa 300	46.6	73.5	50.1	52.5	55.2	50.1	50.1	53.7	50.1	53.5
RoBERTa-large 300	53.0	75.9	53.9	50.4	57.4	51.4	47.8	55.6	50.1	55.1
CodeBERTa 300	50.7	68.3	65.3	52.0	58.8	45.0	50.8	61.8	47.4	55.6
CodeBERT 300	51.3	69.4	49.5	50.8	56.5	49.5	49.3	53.9	49.5	53.3
Zecoler 300	**85.8**	**94.3**	99.3	51.7	**90.1**	**99.5**	**98.7**	**88.8**	**99.2**	**89.7**
Zecoler 100	63.6	93.9	**99.5**	**52.6**	63.0	95.7	72.8	62.8	77.7	75.7

* In this experiment, we train and test the model in the same programming language respectively. The training data size of source languages is 300 except the last one with only 100 data samples.

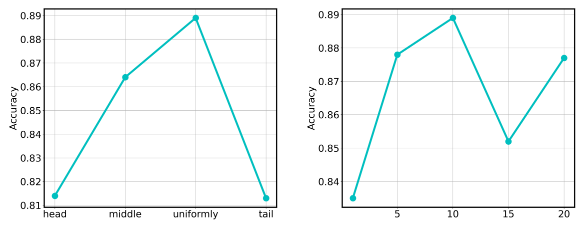


(a) Clone Detection (Solidity) (b) Clone Detection (Go)

(c) Code Search (Solidity) (d) Code Search (Go)

(e) Method Name Prediction (Solidity) (f) Method Name Prediction (Go)

Figure 5: Accuracy of program representation models continuously trained with different-scale datasets in monolingual few-shot setting.



(a) Ablation on the position of prompts (b) Ablation on the number of prompts

Figure 6: Performance of Zecoler under different prompt positions and prompt numbers on the Solidity clone detection dataset (SCCD).

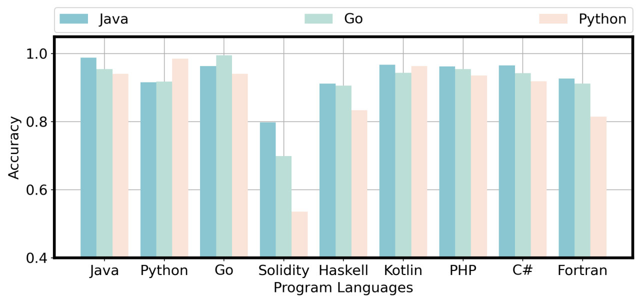


Figure 7: Performance of Zecoler in the code clone detection task with different source languages.

input size. The optimal number of prompt tokens is 10 in our experiments.

Source Languages: To study the impact of different source languages, we train the model using 5,000 data samples of Java, Python, and C++, respectively. We evaluate the performance of zero-shot program representation on the code clone detection task of nine target languages in the CodeNet. Figure 7 shows the results. We can observe that using Java as the source language achieves the best performance. This can be attributed to two reasons. First, CodeBERT is pre-trained using Java. Second, as a common language, Java contains more general features of programming languages compared to other languages. This facilitates the transfer of model to other languages with zero- or few-shot samples.

PLM Scales: Lastly, we study the effect of PLM scales. As Table 2 and 3 indicate, larger PLM scale has a negative effect on the performance. For example, RoBERTa-large is almost twice the size of

```
contract SmartPromise {
    address owner;
    mapping (address => uint256) balances;
    mapping (address => uint256) timestamp;
    constructor() public { owner = msg.sender;}
    function() external payable {
        owner.send(msg.value / 10);
        if (balances[msg.sender] != 0){
            address paymentAddress = msg.sender;
            uint256 paymentAmount = balances[msg.sender]
            *4/100*(block.number-timestamp[msg.sender])/5900;
            paymentAddress.send(paymentAmount);
        }
        timestamp[msg.sender] = block.number;
        balances[msg.sender] += msg.value;
    }
}
```

```
contract Wizard {
    address owner;
    function Wizard() { owner = msg.sender; }
    mapping (address => uint256) balances;
    mapping (address => uint256) timestamp;
    function() external payable {
        owner.send(msg.value / 10);
        if (balances[msg.sender] != 0){
            address kashout = msg.sender;
            uint256 getout = balances[msg.sender]
           *2/100*(block.number-timestamp[msg.sender])/5900;
            kashout.send(getout);
        }
        timestamp[msg.sender] = block.number;
        balances[msg.sender] += msg.value;
    }
}
```

Figure 8: An example of two cloned snippets in Solidity. Zecoler can successfully identify that the given snippets as cloned while CodeBERT cannot.

RoBERTa, but the latter performs even worse than the former. This is because large PLMs can easily overfit few data samples in the zero- or few-shot learning.

Answer to RQ4: The effectiveness of our approach is affected by prompt templates, source languages and PLM scales. Inserting ten prompt tokens uniformly to the original PLM input can steer the PLM to output better representations. Java as the source language can be better generalized to other languages. PLMs with a moderate scale can better fit for the zero- or few-shot learning.

5.5 Example

We now provide a concrete example to demonstrate the effectiveness of Zecoler. Figure 8 shows two cloned code snippets in Solidity. They are similar in functionality but different in key words and structures. For example, "paymentAddress" and "kashout" (highlighted in red) are two equivalent keywords in the two snippets. Because the two words are both domain specific, baseline models such as CodeBERT can hardly detect the clone without prior knowledge. Comparatively, Zecoler successfully detects the clone by reusing prior knowledge from PLMs using prompt learning. The prompts in Zecoler cast the underlying meaning in the PLM to downstream tasks, which help large PLMs capture word semantics even without training data.

6 THREATS TO VALIDITY

Internal Validity. Our approach is built upon CodeBERT. Although CodeBERT is the most popular PLM for learning program representations, other PLMs for code such as GPT-2 (unidirectional Transformers) and CodeT5 (with an encoder-decoder architecture) may have different results. However, we argue that our approach is independent on the PLM architecture itself since we merely modify the format of input and output of the PLM.

External Validity. Zecoler is evaluated on classification tasks such as code clone detection. Other generative tasks such as code summarization might have different performance. However, prompt-based learning has also been shown to be effective in generative tasks [19]. We leave the extensions of our approach to generative tasks for our future work. In our work, the downstream tasks are assumed to be binary classifications. Hence, we represent the binary answers using two candidate words. We can use more candidate words for multi-class classification tasks. The candidate words are manually selected and can be searched to find the most suitable ones.

7 RELATED WORK

7.1 Learning Program Representations

As the core prerequisite for many code intelligence tasks, learning program representations has been extensively explored in software engineering [24]. Table 4 shows typical approaches in learning program representations. Broadly, they can be classified into three categories, including unsupervised for general languages, supervised for specific tasks, and few-shot learning.

The most typical category of work lie in the unsupervised approaches such as code2vec [3], code2seq [2], and InferCode [7]. Code2vec and code2seq aggregate representations of each path in AST (abstract syntax tree) based on attention. InferCode predicts subtrees automatically identified from the contexts of an AST in a self-supervised manner. These methods directly learn program representation from AST paths. They utilize the word embedding techniques in natural language processing and incorporate them with semantic and syntax information in program source code. The limitation of these methods is the lack of adaptions to downstream tasks. The learned code vectors are fixed and cannot be fine-tuned on downstream tasks. Furthermore, these methods are purely trained on code, thus are unsuitable for NL-PL tasks such as code search.

To improve the performance of downstream tasks, researchers have also resorted to task-oriented supervised learning methods [26]. For example, for code clone detection task, Fang et al. [12] caught the similarity of semantics between two code snippets using a supervised deep learning model, which pays attention to caller-callee relationships and learns the hidden syntactic and semantic features of source codes. Zhang et al. [43] disentangled the representation of semantic and syntax with AST and GAN (generative adversarial network), then used only semantic representation to detect code clone. For code search task, Gu et al. [15] proposed a code representation model named CODEnn to learn semantic representations of

Table 4: Models for Learning Program Representations.

Model	Training data				Generalization		Task adaption			
	Unsupervised	Supervised	Few-shot	Zero-shot	General	Task-specific	None	Task-specific	Fine-tuning	Prompt
Code2vec [3]	√				√		√			
Code2seq [2]	√				√		√			
InferCode [7]	√				√		√			
CD [12]		√				√		√		
DCRL [43]		√				√		√		
CM [9]		√				√		√		
CS [15, 16, 18]		√				√		√		
MNP [42]		√				√		√		
DD [44]		√				√		√		
CC [37]		√				√		√		
CodeBERT [13]			√		√				√	
CodeT5 [38]			√		√				√	
Zecoler				√	√					√

* CD = clone detection, CM = code summarization, CS = code search, MNP = method name prediction, DD = defect detection, CC = code completion.

code snippets through jointly embedding with comments, Haldar et al. [16] designed a multi-perspective cross-lingual neural framework, and Li et al. [18] learned code-query interactions. Zhang et al. [42] proposed a hybrid code representation learning approach to resolve program dependence and semantics for predicting method name. Yang et al. [40] learned a unified vector representation of both methods and bug reports for method-level fault localization. Zhou et al. [44] constructed a graph neural network to learn semantic representation of code to identify the vulnerable functions. Wang and Li [37] proposed AST Graph Attention Block to capture different dependencies in the AST graph for representation learning in code completion. These models are trained for the specific downstream tasks, which achieve good performance but lack generality to support multiple tasks with one single model.

The aforementioned methods require a large scale corpus to train the program representation model. To alleviate this problem, pre-trained programming language models are proposed such as CodeBERT [13] and CodeT5 [38]. It is a fine-tuning based few-shot program learning paradigm: PLMs learn a vast amount of knowledge from large scale unlabelled corpora in the pre-training phase, and achieve state-of-the-art accuracy in the fine-tuning phase with a small amount of labelled task-specific data. This gives PLMs the basic generalization ability to handle a wide range of downstream tasks well. Task adaption through fine-tuning adds extra knowledge of specific tasks to PLMs and improves the performance. However, in this paradigm, the gap between the pre-training phase and the downstream task can be significant: the objectives are different, and for the downstream tasks, we usually need to introduce new parameters.

To the best of our knowledge, our Zecoler is the first zero-shot learning method for program representation. Zecoler follows a prompt-based learning paradigm for task adaption. Prompt learning makes it possible for downstream tasks to take the same format as the pre-training objectives and require no new parameters. By narrowing the gap between the two phases, deploying the PLMs on specific tasks becomes much easier with little training data.

7.2 Prompt-based Learning

As a promising method for zero-shot learning, a growing number of prompt-based learning approaches [20] have been proposed in recent years. For example, Schick and Schütze [31] proposed PET which transforms the classification task into an MLM task and uses prompt to elicit knowledge from PLM. But the prompt is manually crafted and hard to select the most suitable words for it. Shin et al. [32] proposed AutoPrompt which automatically searches prompt words discretely using gradient signals in the target task. Although discrete searching retains the semantic of prompt, it also cannot find out the most precise prompts for machine models. For solving this problem, Li and Liang [19] proposed Prefix-Tuning which optimizes a continuous task-specific vector prepended to every layer of the Transformer [36] in PLM and freezes the PLM for saving computation cost. The performance of Prefix-Tuning is excellent but it only focuses on natural language generation tasks.

Comparatively, Zecoler optimizes the prompt vectors in continuous space instead of discrete words or human-writing, making the prompt more suitable for PLMs to understand and more efficient for extracting knowledge. Moreover, Zecoler is the first prompt method to solve programming language understanding tasks.

8 CONCLUSION

In this paper, we propose Zecoler, a novel approach for zero-shot program representation learning via prompt tuning. Zecoler improves traditional pre-trained programming language models by introducing prompt into program representation learning. Experiments show that Zecoler outperforms baseline models in three code intelligence tasks, including code clone detection, code search and method name prediction in both zero-shot and few-shot settings. Program representations learned by Zecoler also demonstrate good generalizability for low/zero-resource programming languages. In the future, we will investigate our approach in more languages and software engineering tasks.

Source code and datasets to reproduce our work are available at: https://github.com/ChrisCN97/zecoler.

ACKNOWLEDGMENTS

This research is supported by National Natural Science Foundation of China (Grant No. 62032004, 62102244).

REFERENCES

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

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of ICLR*.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29.

[4] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 8 (2013), 1798–1828.

[5] Mihaela A. Bornea, Lin Pan, Sara Rosenthal, Radu Florian, and Avirup Sil. 2021. Multilingual Transfer Learning for QA using Translation as Data Augmentation. In *Proceedings of AAAI*. 12583–12591.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Proceedings of NeurIPS*. 1877–1901.

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

[8] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding Code Reuse in Smart Contracts. In *Proceedings of (SANER)*. 470–479.

[9] YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning Sequential and Structural Information for Source Code Summarization. In *Proceedings of ACL/IJCNLP*. 2842–2851.

[10] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. In *Proceedings of MSR*. 243–253.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*. 4171–4186.

[12] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of ISSTA*. 516–527.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of EMNLP*. 1536–1547.

[14] Jiatao Gu, Yong Wang, Yun Chen, Victor O. K. Li, and Kyunghyun Cho. 2018. Meta-Learning for Low-Resource Neural Machine Translation. In *Proceedings of EMNLP*. 3622–3631.

[15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of ICSE*. 933–944.

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

[17] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019).

[18] Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. 2020. Learning Code-Query Interaction for Enhancing Code Searches. In *Proceedings of ICSME*. IEEE, 115–126.

[19] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *Proceedings of ACL/IJCNLP*. 4582–4597.

[20] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *CoRR* abs/2107.13586 (2021).

[21] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. GPT Understands, Too. *CoRR* abs/2103.10385 (2021).

[22] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).

[23] Ilya Loshchilov and Frank Hutter. 2017. Fixing Weight Decay Regularization in Adam. *CoRR* abs/1711.05101 (2017).

[24] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[25] Shivangi Mahto, Vy Ai Vo, Javier S. Turek, and Alexander Huth. 2021. Multi-timescale Representation Learning in LSTM Language Models. In *Proceedings of ICLR*.

[26] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021).

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[28] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[29] Sachin Ravi and Hugo Larochelle. 2017. Optimization as a Model for Few-Shot Learning. In *Proceedings of ICLR*.

[30] Pasquale Salza, Christoph Schwizer, Jian Gu, and Harald C Gall. 2021. On the Effectiveness of Transfer Learning for Code Search. *arXiv preprint arXiv:2108.05890* (2021).

[31] Timo Schick and Hinrich Schütze. 2021. Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference. In *Proceedings of EACL*. 255–269.

[32] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *Proceedings of EMNLP*. 4222–4235.

[33] Jake Snell, Kevin Swersky, and Richard S. Zemel. 2017. Prototypical Networks for Few-shot Learning. In *Proceedings of NeurIPS*. 4077–4087.

[34] Yi Sun, Yu Zheng, Chao Hao, and Hangping Qiu. 2021. NSP-BERT: A Prompt-based Zero-Shot Learner Through an Original Pre-training Task-Next Sentence Prediction. *CoRR* abs/2109.03564 (2021).

[35] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *Proceedings of ICSME*. 476–480.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of NIPS*. 5998–6008.

[37] Yanlin Wang and Hui Li. 2021. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In *Proceedings of AAAI*. 14015–14023.

[38] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of EMNLP*. 8696–8708.

[39] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of IJCAI*. 3034–3040.

[40] Shouliang Yang, Junming Cao, Hushuang Zeng, Beijun Shen, and Hao Zhong. 2021. Locating Faulty Methods with a Mixed RNN and Attention Model. In *Proceedings of ICPC*. 207–218.

[41] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *Proceedings of ICPC*. 1–12.

[42] Fengyi Zhang, Bihuan Chen, Rongfan Li, and Xin Peng. 2021. A hybrid code representation learning approach for predicting method names. *J. Syst. Softw.* 180 (2021), 111011.

[43] Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu, and Yulei Sui. 2021. Disentangled Code Representation Learning for Multiple Programming Languages. In *Proceedings of ACL/IJCNLP*. 4454–4466.

[44] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of NeurIPS*. 10197–10207.