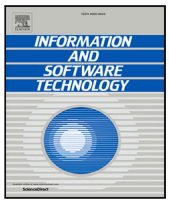
Information and Software Technology 185 (2025) 107775

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Exploring continual learning in code intelligence with domain-wise distilled prompts

Shuo Liu [a], Jacky Keung [a], Zhen Yang [b,*], Fang Liu [c], Fengji Zhang [a], Yicheng Sun [a]

[a] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[b] School of Computer Science and Technology, Shandong University, Qingdao, China
[c] School of Computer Science and Engineering, Beihang University, Beijing, China

ARTICLE INFO

Keywords:
Pre-trained language models
Prompt tuning
Continual learning
Code intelligence

ABSTRACT

Context: Software programs evolve constantly in practice, leading to domain shifts that cannot be fitted in the traditional offline manner. Recently, a few Continual Learning (CL) studies on code intelligence emerged, which learn a sequence of datasets one by one. We criticize existing rehearsal-based CL methods heavily rely on retraining historical samples, bringing about an extra training burden and the risk of data disclosure.
Objective: To overcome the above limitations, in this paper, we leverage the superiority of prompts in eliciting pre-trained knowledge to realize a rehearsal-free method.
Methods: We first explore the performance of vanilla prompt tuning in the CL scenario, finding that inheriting the previous Pre-trained Language Model (PLM) parameters is appropriate and prompt stability should be emphasized. Therefore, we propose an effective method named Prompt Tuning with Domain-wise Distillation (PTDD), which can distill prompts and optimize PLMs with a two-sided learning objective, thus improving PLMs' performance in diverse domains.
Results: We conduct experiments on three widely-studied code intelligence tasks, including Code Summarization, Code Vulnerability Detection, and Code Clone Detection. We evaluate PTDD in comparison with a series of baselines. Experimental results indicate the effectiveness of PTDD. For instance, PTDD surpasses fine-tuning by 2.55%, 11.12%, and 2.25% in the three tasks, respectively. Moreover, we interpret the effectiveness of PTDD by prompt visualization, and discuss its performance in the low-resource scenario, where the improvement of PTDD becomes stark with fewer training samples and can reach up to 69.09%.
Conclusion: To the best of our knowledge, our work conducts the first experimental study to explore the performance of prompt tuning within the CL setting in the code intelligence field. The research findings indicate the effectiveness of PTDD and contribute to a deeper understanding of the capability of prompts.

1. Introduction

Recent years have seen a tendency to leverage Pre-trained Language Models (PLMs), such as CodeBERT [1] and CodeT5 [2], for various downstream code intelligence tasks, including code summarization [3,4], code vulnerability detection [5,6], and code clone detection [7,8]. Typically, these works mostly follow a prevalent paradigm named pre-training and fine-tuning, which first optimizes language models on large-scale code corpora to obtain general knowledge, and then adapts models to acquire task-specific knowledge. Benefiting from the powerful representation capability of PLMs, they can obtain superior performance.

However, previous works primarily focus on an offline setting, tuning a PLM on a pre-collected dataset. The PLM is an expert in the domain of the pre-collected dataset, but performs worse if domain shifting. The domain shift [9], also known as the distributional shift, occurs regularly in the practical application of deep learning models and describes the variation between different datasets. In the field of code intelligence, domain shifts are more common as programs evolve constantly. New repositories and new APIs are created over time [10,11]. If ignoring the dynamic nature of software codebases, the practicability of code intelligence models can be affected. Therefore, it is significant to explore the performance of PLMs in a Continual Learning (CL) setting, where models are trained on a sequence of datasets one by one and are expected to perform well on all introduced domains. As shown in Fig. 1, in the training process, PLMs are adapted to a new domain once inputting a new dataset. The model finally

* Corresponding author.
E-mail addresses: sliu273-c@my.cityu.edu.hk (S. Liu), Jacky.Keung@cityu.edu.hk (J. Keung), zhenyang@sdu.edu.cn (Z. Yang), fangliu@buaa.edu.cn (F. Liu), fengji.zhang@my.cityu.edu.hk (F. Zhang), yicsun2-c@my.cityu.edu.hk (Y. Sun).

https://doi.org/10.1016/j.infsof.2025.107775
Received 1 November 2024; Received in revised form 29 April 2025; Accepted 29 April 2025
Available online 15 May 2025
0950-5849/© 2025 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

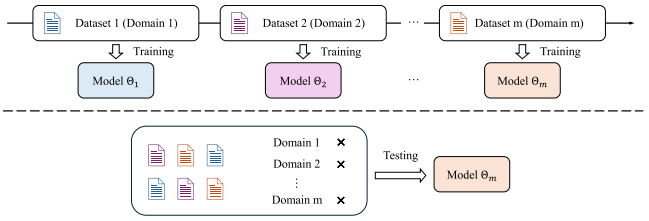


Fig. 1. An illustration of the continual learning scenario. X marks indicate the domain ID that test samples derive from is unknown when testing.

tuned will be evaluated with data samples from all previously learned domains, but which domain a data sample belongs to is not known at test time. This setting is also called domain-incremental continual learning [12]. As far as we know, only a few works are dedicated to the CL scenario in the field of code intelligence.

Continual learning was first proposed in the field of machine learning, and some representative methods have emerged. They can mainly fall into three categories, namely regularization-based [13,14], rehearsal-based [15,16], and representation-based methods [17,18]. The major challenge in the CL scenario is the catastrophic forgetting problem [19], which indicates models' performance on previous domains can be affected when feeding datasets of new domains. To mitigate the problem, in the code intelligence field, Gao et al. [20] resort to storing and retraining some carefully selected previous data, as well as adding an adaptive parameter regularization module. Their method can be seen as a combination of the regularization-based and rehearsal-based approaches. Similarly, Weyssow et al. [21] evaluate the two approaches on API-related tasks. Although it is intuitive to retrain models with previous data and new data to alleviate forgetting, we criticize that previous data samples are not always available because of storage or privacy constraints [22]. Consider a scenario in which several research and development teams share one continual-learned PLM to assist programming, and this PLM needs to evolve based on different teams' development requests. However, in modern software companies, it is prevalent that historical coding samples from different teams are not allowed to be shared. In this case, the above rehearsal-based approach is impracticable. Besides, additional computational and storage resources are needed in the approach. Therefore, to avoid the risk of data disclosure and pursue practicality, we focus on a rehearsal-free method.

In this paper, we resort to the superiority of prompts in eliciting pre-trained knowledge to mitigate the catastrophic forgetting problem. By adding several tunable tokens, prompt tuning can reduce the disparity in the training form between pre-training and downstream tasks, thus providing instructions when tuning PLMs [23]. In the CL scenario, Yadav et al. [24] propose prompt pools, containing diverse prompts for learning domain-specific knowledge. Besides, they also introduce an additional prompt selection and combination module to construct domain-sharing knowledge. We criticize that prompt pools lack the inheritance of previous knowledge and may lead to bias in prompt selection given an example of an unknown domain. Therefore, we expect a method with certain prompts that can instruct both domain-specific and domain-sharing knowledge. We first conduct an experimental study to explore the performance of vanilla Prompt Tuning (PT) and Parameter-Efficient Prompt Tuning (PEPT) in the CL scenario when compared with Fine-Tuning (FT). Experimental results show that PT can surpass FT while PEPT performs worse, enlightening us that it is necessary to tune PLMs when applying PT. Besides, although PT outperforms FT, they still cannot balance plasticity and stability during continuous domain adaptation.

Based on the above findings, we propose an effective method called Prompt Tuning with Domain-wise Distillation (PTDD), which only involves two prompts during training, one for previous domains and one for the current domain, and only the current prompt is utilized when evaluating. Specifically, considering there are correlations between different domains in code intelligence, PTDD first initializes the prompt as its previous one and adds a Multi-Layer Perceptron (MLP) to improve models' representation ability [25]. To distill prompts and optimize the backbone model, PTDD introduces an additional stability learning objective to maintain the learned knowledge in previous domains. PTDD can be tuned with all parameters or just the prompt parameters. We denote the former as PTDD and the latter as parameter-efficient PTDD (PE-PTDD). It is worth noting that the prompt distillation here describes a process that limits the variation of previous prompts in cross-domain adaptations, getting the current prompt that can furthest keep the previous instruction ability and adapt it to the current domain. It is a different term from the knowledge distillation that is often used in the machine learning field.

We conduct experiments on three widely-used code intelligence tasks where domain shifts occur regularly, including code summarization (CS), code vulnerability detection (CVD), and code clone detection (CCD). We evaluate PTDD in comparison with various baselines, and PTDD can obtain consistent improvements among all rehearsal-free methods. For instance, in the CS task, PTDD averagely outperforms FT by 2.55% in terms of the BLEU metric on six datasets. In the CVD and CCD tasks, PTDD outperforms FT by 11.12% and 2.25% in terms of the F1 score. Even when comparing PE-PTDD with other parameter-efficient baselines, our method still shows effectiveness. We also conduct an ablation study and parameter analysis to investigate the performance of PTDD from different perspectives. Furthermore, we visualize the prompts learned by PTDD and PT to give an explanation and discuss the capability of PTDD in the low-resource scenario. In the CS task, when decreasing the number of training samples to 10 in each dataset, the improvement of PTDD can reach up to 69.09% in terms of BLEU. The extensive experiments indicate the superiority of our method in the CL scenario.

The major contributions of this paper can be summarized as follows:

1. To the best of our knowledge, in the code intelligence field, this paper conducts the first experimental study to explore the performance of PT in the CL scenario. The results enlighten us that it is necessary to tune PLMs when applying PT and prompt stability should be emphasized.
2. We propose a rehearsal-free method named PTDD, which can distill and preserve previous knowledge by introducing a stability learning objective, therefore mitigating catastrophic forgetting of previously learned domains.
3. We conduct extensive experiments on three widely-studied code intelligence tasks. Experimental results show the effectiveness of PTDD. We also interpret our method by prompt visualization and discuss its performance in the low-resource scenario.

2. Background

2.1. Continual learning protocols

Generally, in the CL scenario, the language model is presented with a sequence of non-overlapping datasets corresponding to various distinctive domain knowledge, and the goal is to incrementally learn new domain knowledge while not severely forgetting previously learned ones [26,27]. Considering in the code intelligence field, we denote the sequence of datasets as $D = \{D_1, D_2, \ldots, D_m\}$, where m is the total number of datasets. For the tth dataset D_t, it contains data samples $\left(x_i^t, y_i^t\right)_{i=1}^{N_t}$, where x_i^t is an input code snippet or natural language description, and y_i^t is a corresponding label in classification tasks or an output text in generation tasks. N_t represents the dataset size of D_t. We assume that data samples in the same dataset are independent and identically distributed, but they have varied distributions in different datasets. The learning objective is to learn a probabilistic

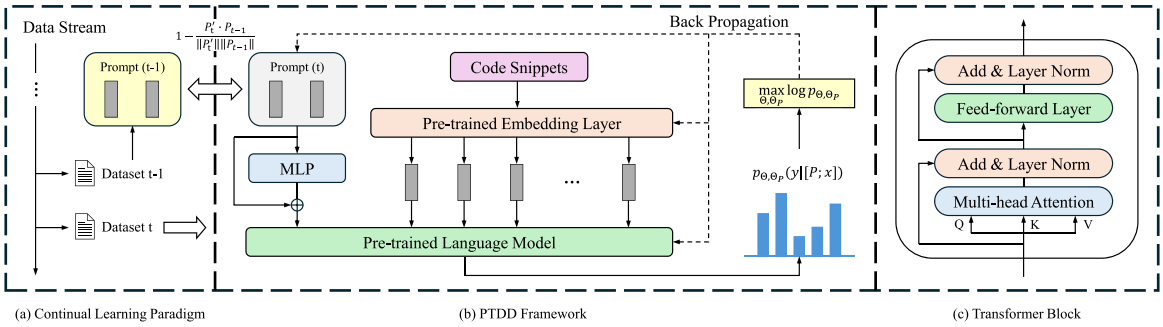


Fig. 2. Overview of the proposed Prompt Tuning with Domain-wise Distillation (PTDD).

model $p(D|\Theta) = \prod_{t=1}^{m} p\left(D_t|\Theta\right)$ and maximize its performance on D. Θ is the model's parameters. Across all datasets, the learning objective can be defined as:

$$\Theta = \arg\max_{\Theta} \sum_{t=1}^{m} \sum_{x,y \in D_t} \log p_{\Theta}\left(y|x\right) \tag{1}$$

In this paper, we deal with the domain-incremental continual learning setting, which learns to solve the same task in different datasets but does not know which dataset a data sample belongs to at test time [12,28]. Specifically, D_t can be divided into a training set, a validation set, and a test set, denoted as $D_t = \{T_t, V_t, S_t\}$. When it is the turn of D_t, the model will be trained and validated on T_t and V_t. Consequently, model Θ_t is obtained. It is worth noting that some existing works [20,27,29] allow the model to access training samples from previous datasets, but in this paper, in order not to violate data privacy, only T_t is accessible. At test time, Θ_t will be evaluated on all learned datasets, namely $\{S_1, S_2, \ldots, S_t\}$.

2.2. Fine-tuning and prompt tuning

Fine-tuning has become the predominant technique for adapting PLMs to specific downstream tasks [30,31]. It initializes models with the parameters from PLMs, and all parameters Θ are updated to learn task-specific knowledge. In many code intelligence tasks, it has shown promising results [32–34]. Formally, given a downstream dataset D' with input text x_i and corresponding output y_i, FT updates all parameters Θ in a supervised manner as:

$$\Theta = \arg\max_{\Theta} \sum_{x_i,y_i \in D'} \log p_{\Theta}\left(y_i|x_i\right) \tag{2}$$

Prompt tuning [35] is an emerging technique and can be regarded as an alternative to fine-tuning. In contrast to FT, it introduces a series of virtual tokens, or soft prompts, to modify the input text. This makes PLMs obtain additional task-specific instructions, so that more pre-trained knowledge can be elicited [36]. In addition, PT can mitigate the gap in learning objectives between the pre-training and the fine-tuning process [23]. It achieves a similar form of the Masked Language Modeling, which is a common pre-training objective. The above-mentioned advantages make PT popular in both NLP [37,38] and code intelligence [39,40]. Consider a dataset $D' = \left\{\left(x_i, y_i\right)\right\}_{i=1}^{N'}$, and denote the soft prompt as P. In prompt tuning, its parameters not only include the PLM parameters Θ, but also contain the extra prompt parameters Θ_P. The learning objective is to maximize the log-likelihood of y_i given the input x_i concatenated with the soft prompt P, which can be formulated as:

$$[\Theta, \Theta_P] = \arg\max_{\Theta,\Theta_P} \sum_{x_i,y_i \in D'} \log p_{\Theta,\Theta_P}\left(y_i|[P; x_i]\right) \tag{3}$$

Moreover, to pursue low computational overhead and high efficiency, there is also an approach called parameter-efficient prompt tuning (PEPT) [25,41]. It tunes only the extra prompt parameters Θ_P on downstream tasks, but keeps the PLM parameters Θ frozen to reduce the computational cost and memory requirements. Although the final performance may be impacted due to the small number of tuned parameters, PEPT is practical in real-world scenarios, especially when training data is scarce [23].

3. Methodology

Fig. 2 shows the overview of the proposed PTDD, which is well-designed to balance the plasticity and the stability learning objective. Receiving a sequence of datasets one by one, PTDD constantly updates the model and the prompt to distill previous knowledge, which can perform well on the current dataset while not forgetting much previously learned knowledge. As there is no overlapping between datasets, here one dataset represents one domain. For each dataset, PTDD first initializes the prompt and embeds it into the prompt space. Then PTDD optimizes the model and the prompt to distill learned knowledge and integrate new knowledge.

3.1. Prompt embedding

In the training process, consider that PTDD is dealing with the tth dataset D_t, whose prompt is denoted as P_t. Different from previous works [17,24] that randomly initialize a prompt pool and calculate the relationship between prompts and datasets, PTDD adopts a straightforward strategy that assigns one prompt for the current dataset and another prompt for all previous datasets. The two prompts alternate when fitting on new domains. Considering that the development experience gained previously significantly contributes to future development in practice, knowledge sharing across domains is vitally important [42]. To accomplish this, the prompt P_t is initialized as its previous prompt P_{t-1} to explicitly inherit previously learned knowledge, but P_0 is randomly initialized from the vocabulary. Formally, $P_t \in \mathbb{R}^{L_P \times D}$, where L_P is the prompt's token length and D is the embedding dimension.

In addition, as some existing works [43,44] find that directly passing the prompt embedding to language models may cause training instability and hinder model performance, PTDD inputs the embedding P_t into a Multi-Layer Perceptron (MLP) and utilizes a residual connection combining P_t and the MLP's output. The residual connection improves language models' representation ability and is beneficial to optimization. As a result, a new prompt embedding P'_t is obtained, which has the same size as P_t and can be formulated as:

$$P'_t = MLP\left(P_t\right) + P_t \tag{4}$$

3.2. Distilled prompt learning

To distill prompts containing previous knowledge, it is important to balance plasticity and stability. Plasticity refers to the adaptability to new domains, while stability prevents knowledge of past domains from forgetting. When training on dataset D_t, plasticity can be simply accommodated by optimizing the model and prompt on the current dataset. Specifically, denote the input code snippet as x_t, its target

Algorithm 1 Prompt Tuning with Domain-wise Distillation

Input: Dataset $D_{1:m}$, pre-trained language model Θ_0, prompt length L_P, factor λ.
Output: Generated text or predicted label.

Initialize $\Theta \leftarrow \Theta_0$.
Randomly select L_P tokens from the vocabulary to initialize P_1.
$P_1' \leftarrow MLP(P_1) + P_1$.
Minimize (5) to update P_1' and Θ.
for $t = 2, \cdots,$ m **do**
  Embed $P_t \leftarrow P_{t-1}'$.
  $P_t' \leftarrow MLP(P_t) + P_t$.
  Calculate cosine similarity between P_t' and P_{t-1}'.
  Minimize (7) to update P_t' and Θ.
end for
Concatenate P_m' with test data to calculate outputs.
return Generated text or predicted label.

output as y_t, and $X = [P_t; x_t]$ is the language model's input embedding. As shown in Fig. 2(c), input embedding X is first transformed to Q, K, V by multiplying learnable projection matrices W_Q, W_K, W_V, and then calculated in different parts including multi-head attention, feed-forward layer, and layer normalization. The encoder output is further sent to a decoder or a classifier according to the task category to generate the final output. The optimizing objective can be defined to minimize the following loss function:

$$\mathcal{L}_1 = \mathcal{L}\left(f_{PLM}\left(\left[P_t'; x_t\right]\right), y_t\right) \tag{5}$$

Where $\mathcal{L}$ is a loss function, which is the language modeling loss for generation tasks and the cross-entropy loss for classification tasks. Besides, $f_{PLM} : \mathcal{X} \rightarrow \mathcal{Y}$ mainly represents the transformation caused by the PLM.

As the findings in Section 5.1, besides plasticity, PTDD also introduces stability into the learning objective. It employs cosine similarity to measure the proximity between the currently updated prompt P_t' and the last prompt P_{t-1}, and expects these two to be similar. In this way, P_t' can retain alike semantics to previous knowledge, thus stabilizing the prompt's representation. To achieve the learning objective, an additional loss function is defined as:

$$\mathcal{L}_2\left(P_t', P_{t-1}\right) = 1 - \frac{P_t' \cdot P_{t-1}}{\|P_t'\| \|P_{t-1}\|} \tag{6}$$

Finally, to combine the two objectives together, PTDD uses a factor λ as the regularization coefficient to balance their relationship. Therefore, the overall loss function is:

$$\mathcal{L} = \mathcal{L}_1 + \lambda \cdot \mathcal{L}_2\left(P_t', P_{t-1}\right) \tag{7}$$

Through backpropagation, PTDD updates the model and the prompt while preserving previous knowledge. Without accessing any past data, PTDD is fully rehearsal-free. It is worth noting that we realize stability from the perspective of prompt, but not like some previous works from the perspective of PLM [13]. As the two perspectives are orthogonal, they can be integrated to further investigate but beyond the scope of this paper. During training, if freezing the PLM parameters Θ and keeping only the prompt parameters Θ_P trainable, the training stage becomes efficient, and we name it as parameter-efficient PTDD (PE-PTDD). Moreover, in the inference stage, as the dataset from which test data comes is unknowable, we directly concatenate the finally learned prompt with input code snippets, and discard the MLP module so as not to interfere with the prompt representation. The overall approach of PTDD is manifested in the Algorithm 1.

4. Experimental setup

4.1. Tasks and datasets

To evaluate the effectiveness of our proposed PTDD, we adopt three downstream code intelligence tasks, namely code summarization, code vulnerability detection, and code clone detection. We utilize the project-level continual learning datasets open-sourced by Gao et al. [20]. In this section, we will describe the three downstream tasks and their continual learning datasets in detail.

4.1.1. Code summarization (CS)

A high-quality summarization is essential to program comprehension and maintenance [45]. When given a code snippet, code summarization aims to automatically generate a natural language description containing the code's functionality and purpose. Recent state-of-the-art approaches commonly resort to sequence-to-sequence deep learning models to achieve promising performance [3,4]. CodeSearchNet [46] is a widely-used dataset for code summarization, which includes open-source code spanning six programming languages. To simulate the continual learning scenario, on each programming language, Gao et al. randomly split the original dataset into five parts according to the project name that code snippets derive from. Their division makes code snippets in different parts belonging to different projects, ensuring project-level continual learning. Datasets for each part are divided into training set, validation set, and test set by the ratio of 8:1:1.

4.1.2. Code vulnerability detection (CVD)

Vulnerability detection aims to identify defective codes timely and is crucial to software security [47]. Recent deep learning-based methods mostly represent it as a binary classification task, to predict whether the input code snippet contains vulnerabilities [6]. Fan et al. [48] curate a large C/C++ code vulnerability dataset, namely Big-Vul. In the same procedure as aforementioned, Gao et al. divide it into five parts and split the training set, validation set, and test set, constructing its project-level continual learning version, called Big-Vul-PC.

4.1.3. Code clone detection (CCD)

Code clone indicates two code snippets have the same functional semantics but with syntactic variations. It makes code error-prone and introduces risks on software maintenance and evolution [49]. The objective of code clone detection is to determine whether a pair of code snippets is functional equivalent, which can also be seen as a classification task. POJ [50] is a large-scale dataset for code clone detection containing 104 different algorithms, and for each algorithm, there are 500 programs written in the C language. Due to the enormous data when pairing code snippets, following the previously used strategy [51], after dividing the 104 algorithms into five parts, Gao et al. randomly sample 40,000/5000/5000 pairs in each part to construct the training set, validation set, and test set. The continual learning dataset is represented as POJ-PC.

4.2. Baselines

To show the effectiveness of our proposed PTDD, we compare our methods with the following representative baselines. Their detailed information is described below. Specifically, in the comparison, we choose CodeT5 [2] as the backbone model for code summarization and CodeBERT [1] for code vulnerability detection and code clone detection. The reason for choosing the two models is based on their popularity and representativeness in the three tasks. As pre-trained on large-scale code corpora, CodeT5 and CodeBERT learn universal representations for programs and have achieved outstanding results. They are widely-used baselines in the research on the three tasks, and become fundamental models to achieve SOTA results [6,8,52]. In the meanwhile, CodeBERT is an encoder-only model and good at

Table 1
Evaluation results of code summarization. M and R are short for Meteor and Rouge-L, respectively.

Methods	Ruby			Javascript			Java			Go			PHP			Python		
	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R
EMR	18.92	14.02	31.88	17.03	12.54	28.46	23.39	17.13	40.27	35.19	22.97	51.76	26.56	18.90	41.09	21.59	16.06	37.37
PEPT	15.89	12.05	28.53	15.14	11.62	26.95	20.85	15.48	38.16	31.67	20.97	49.11	24.62	17.79	39.88	20.49	15.04	36.55
PP-TF	17.06	12.49	29.71	15.65	11.58	27.41	20.14	14.76	37.02	28.65	19.38	46.33	24.28	17.19	39.13	20.22	14.75	35.95
FT	18.67	13.44	31.53	16.87	11.75	27.75	22.72	16.34	39.72	32.84	21.19	49.67	26.25	18.57	40.99	21.24	15.32	37.11
PT	18.75	13.56	31.69	16.61	12.34	28.66	23.28	16.96	40.33	34.48	22.75	51.22	26.49	18.79	41.39	21.63	15.90	37.63
EWC	18.76	13.71	31.77	**16.93**	**12.35**	**28.91**	23.05	16.79	40.08	34.35	22.47	51.20	26.44	18.65	41.16	21.52	15.92	37.54
PTDD	**18.97**	**13.81**	**31.98**	**16.93**	12.31	28.71	**23.35**	**17.03**	**40.37**	**34.61**	**22.84**	**51.34**	**26.55**	**18.80**	**41.41**	**21.72**	**16.04**	**37.71**
PE-PTDD	17.63	12.65	30.60	16.18	11.77	28.02	21.16	15.47	38.40	31.91	21.16	49.31	24.70	17.81	39.93	20.70	15.18	36.71
Ablation	18.93	13.72	31.88	16.89	12.35	28.79	23.26	16.94	40.31	34.48	22.75	51.27	26.53	18.82	41.40	21.67	16.02	37.64

classification tasks like code vulnerability detection and code clone detection, and CodeT5 builds on an encoder–decoder framework so that it excels at generation tasks like code summarization.

- **Fine-Tuning** [30]: FT directly updates all parameters of language models to downstream tasks. In the continual learning scenario, the tuned model in the previous dataset is used as the model's initialization in the next dataset.
- **Prompt Tuning** [35]: For continual learning, PT and PEPT both add extra prompts, but the former tunes all parameters and the latter only updates prompts. Following previous works [17,24], the initialization of prompts is randomly generated.
- **EWC** [13]: As the abbreviation of elastic weight consolidation, EWC slows down learning on certain weights to mitigate catastrophic forgetting. The selected parameters are important for previous tasks and are constrained to stay close to their old values.
- **EMR** [15]: EMR is short for episodic memory replay, a representative rehearsal-based method. It randomly retrains data samples from the previous datasets to ensure the learned knowledge will not be forgotten.
- **PP-TF** [24]: PP-TF, namely prompt pooling with teacher forcing, is a parameter-efficient continual learning approach for generation tasks. It enforces constraints on the prompt selection to make training stable.

It is worth noting that we choose EMR as the representative of rehearsal-based methods but not REPEAT [20]. This is because, considering the advantages that rehearsal-based methods have, namely reutilizing previous data samples, there is no need to compare PTDD with the SOTA rehearsal-based method, i.e., REPEAT. As REPEAT is based on EMR and adds two extra components, including representative exemplars replay and adaptive parameter regularization, we choose the more fundamental baseline EMR in this paper to demonstrate the effect of the rehearsal-based methods.

4.3. Evaluation metrics

For code summarization, we use three metrics for evaluation, i.e. BLEU [53], Meteor [54], and Rouge-L [55]. BLEU compares n-grams of the generated text with n-grams of the ground truth text, counting the number of matches. We set n as 4 in this paper (i.e. BLEU-4). Meteor focuses on the unigrams, and takes into account the precision, recall, and fluency of the generated text. Rouge-L calculates the F-score based on the Longest Common Subsequence (LCS) of the generated text and the ground truth text. For code vulnerability detection and code clone detection, we adopt F1, Precision (P), and Recall (R), which are widely used in previous works [33,34].

Specifically, in the CL scenario, a sequence of models needs to be evaluated on all previously learned datasets. To better assess the method's performance, we calculate the average performance following the same strategy that former works used [20,56]:

$$\Omega = \frac{1}{m}\sum_{i=1}^{m}\Omega_i, \quad \Omega_i = \frac{1}{i}\sum_{j=1}^{i}\Omega_{i,j} \tag{8}$$

where $\Omega_{i,j}$ represents the performance on the jth test set after learning the ith dataset. Ω_i is the average performance of $\Omega_{i,j}$ on all test set from $j = 1$ to $j = i$, and Ω is the overall performance on all m datasets. Here, Ω can be any metric introduced above.

4.4. Implementation details

The experiments are conducted on an Ubuntu GPU server with four RTX 3090 24 GB GPUs. Our code is implemented by the deep learning framework PyTorch.[1] The PLMs we used, namely CodeBERT and CodeT5, are from Huggingface,[2] and we keep their default hyperparameter settings. We reproduce the performance of EWC, EMR, and PP-TF referring to previous works [20,24]. For all three tasks, we set the training epoch to 10 and the batch size to 16. But considering the different complexity of the three tasks and their used PLMs, we set the learning rate to 5e−5 for CS but 2e−5 for CVD and CCD. For the parameter-efficient methods in the CS task, we set the learning rate to 1e−3 as the parameters that can be tuned decrease dramatically. The prompt length is adjusted in the range of {10, 20, 30, 40, 50}, and the factor λ is adjusted in {0.1, 0.5, 1, 5, 10, 50}. An elaborate analysis of the two parameters will be discussed in Section 5.4.

5. Experimental results

In this section, we conduct extensive experiments to answer the following research questions:

RQ1: What is the performance of vanilla prompt tuning in the continual learning scenario?

RQ2: How effective is PTDD when compared to baselines?

RQ3: What are the contributions of different components in PTDD to the performance?

RQ4: How do varied parameter settings influence the performance of PTDD?

RQ5: How does PTDD perform in the low-resource scenario?

5.1. RQ1: Vanilla prompt tuning performance

Prompt tuning has become a popular approach in applying PLMs to specific downstream tasks, and researches on it emerge constantly in the field of software engineering [23,40]. In the CL scenario, prompt-based methods also show advantages [24]. However, we criticize that the existing method lacks inheritance of previous knowledge, and the additional prompt selection module may lead to bias and error. We

[1] https://pytorch.org/
[2] https://huggingface.co/models

Table 2
Evaluation results of code vulnerability detection and code clone detection.

Methods	CVD			CCD		
	F1	P	R	F1	P	R
EMR	32.55	42.90	27.50	85.01	89.11	81.45
FT	27.62	45.67	21.79	83.07	84.50	82.33
PT	28.54	43.64	23.44	84.88	**84.92**	85.45
EWC	28.61	**46.71**	24.07	83.34	83.97	83.43
PTDD	**30.69**	44.89	**25.11**	**84.94**	84.65	**85.94**
Ablation	29.20	42.94	24.08	84.76	83.82	86.28

prefer to explore the effectiveness of using limited prompts. In this section, we first conduct an experimental study to investigate the performance of PT and PEPT, and expect some inspiring findings. We evaluate PT on all three code intelligence tasks, but evaluate PEFT only on the CS task, as they cannot converge on other tasks with tuning a tiny fraction of parameters based on our experiments. Table 1 and Table 2 illustrate their performance.

From the tables, we can observe that PT obtains consistent improvement when compared with the predominant tuning paradigm, i.e., fine-tuning. This finding is in accordance with previous works [23, 40]. For the CS task, Table 1 shows that PT's average performance increases by 1.91%, 3.82%, and 1.83% in terms of BLEU, Meteor, and Rouge-L. The highest improvement is in the Go dataset, where the BLEU metric raises 4.99%. Besides, although the BLEU metric slightly declines in the Javascript dataset, the other two metrics here are obviously improved. Thus, we argue that PT is still beneficial for those codes in Javascript. For the CVD and the CCD task, as shown in Table 2, the enhancement brought by PT can reach 3.33% and 2.18% in terms of the F1 score. The experimental results indicate that PT can achieve better performance in the CL scenario.

Due to the possible capability degradation issue, there are few works exploring the effectiveness of PEPT with PLMs in code intelligence tasks. In the experiment, we evaluate PEPT on the CS task. From Table 1, we can observe that the performance of PEPT is weaker than fine-tuning. The average decline is 1.66, 0.61, and 1.27 in terms of BLEU, Meteor, and Rouge-L, respectively. The observation shows that only depending on adjusting prompt parameters are difficult to adapt across domains, and it is necessary to tune PLMs when applying PT. Besides, although PT outperforms FT, they still lack a learning objective for stability. As shown in Fig. 3, which illustrates the performance variation, their performance decreases when feeding new datasets. It inspires us to balance the plasticity and the stability objective when tuning.

Answer to RQ1: In the CL scenario, PT can consistently outperform FT on the studied three code intelligence tasks. However, the performance of PEPT is slightly weaker. The study inspires us that it is necessary to tune PLMs when applying PT, and we should balance the plasticity and the stability learning objective.

5.2. RQ2: Comparison with baselines

In this section, we conduct comprehensive experiments to evaluate PTDD by comparing it with a series of baselines. We keep the backbone model used in all baselines the same as our method, namely CodeT5 for CS and CodeBERT for CVD and CCD. Table 1 and Table 2 show the detailed evaluation results, and we highlight the highest values among all rehearsal-free methods in bold. The values presented are the average performance calculated by Eq. (8).

For the CS task, it can be observed from Table 1 that the average performance of PTDD on all six datasets can surpass FT by 2.55%, 4.37%, and 2.10% in terms of BLEU, Meteor, and Rouge-L. PTDD also obtains consistent improvements on all three metrics when compared with PT. For the regularization-based baseline EWC, PTDD outperforms it by 0.77% on average regarding the BLEU metric. Considering the additional computation required by EWC to select important parameters, PTDD omits this process, updates models and prompts in a backpropagation step, and can obtain a better performance, indicating its advantage in alleviating the catastrophic forgetting problem. Even in comparison with the rehearsal-based baseline EMR, PTDD can achieve comparable performance. It surpasses EMR in the Ruby and Python datasets regarding the BLEU metric. Due to the need to store previous data, EMR becomes not applicable to data privacy sensitive scenarios. Our proposed PTDD gets rid of the shortcomings and realizes a rehearsal-free approach.

For the CVD task, as shown in Table 2, PTDD dramatically outperforms FT by 11.12% and 15.24% in terms of F1 and Recall, but mildly decreases in terms of Precision. When compared with PT, PTDD can obtain consistent superiority of 7.53%, 2.86%, and 7.13% regarding the F1 score, Precision, and Recall, respectively. Besides, PTDD performs better than EWC but weaker than EMR. This result shows that replaying a fraction of data samples is still effective in such a classification task. Whereas when considering the constraint of data privacy, PTDD performs best among all baselines. For the CCD task, PTDD can surpass FT by 2.25%, 0.18%, and 4.39% in terms of F1, Precision, and Recall. A similar advantage of PTDD can be observed in comparison with PT and EWC. Although it is slightly inferior to EMR, PTDD avoids the risk of data breaches and is practical in real-world scenarios.

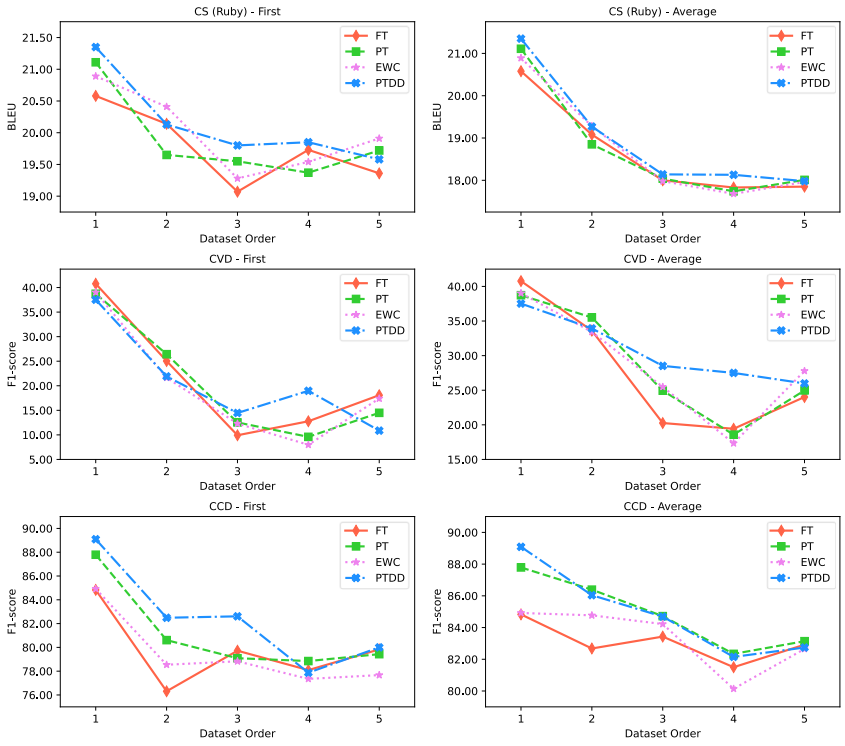
Furthermore, we illustrate the performance variation when inputting a sequence of datasets in Fig. 3. We select the Ruby dataset of the CS task and datasets used in the other two tasks as representatives. The performance trend of the first dataset and the average performance trend on previously learned datasets are presented separately. The horizontal axis is the input order of datasets, and the vertical axis is the value of the BLEU or F1 metric. From Fig. 3, it can be observed that the catastrophic forgetting problem widely exists in the continual learning scenario. Our proposed PTDD can generally improve performance and slow decline trends, indicating it can alleviate forgetting of learned knowledge.

Answer to RQ2: The performance of PTDD can surpass FT, PT, and EWC generally in all three code intelligence tasks, indicating its superiority in the continual learning scenario. However, it is slightly inferior to the rehearsal-based baseline EMR as EMR stores and retrains a fraction of data samples, avoiding our rehearsal-free setting.

5.3. RQ3: Ablation study

In this section, we conduct an ablation study from two aspects. The first is to investigate the contribution of the distilled prompt learning module to the performance of PTDD. In addition, considering the prosperity of exploring parameter-efficient fine-tuning methods in code intelligence, we also detect how effective the PE-PTDD is, expecting advantages of our method in the parameter-efficient scenario. The ablation results are presented at the bottom of Table 1 and Table 2.

To validate the contribution of the distilled prompt learning module, we remove the stability learning objective and train models with the loss function in Eq. (5). In the experiment, following the same experimental setting of RQ2, we use all datasets of the three tasks. For the CS task, as shown in Table 1, there is a consistent decline if discarding the module in terms of BLEU. Except for Javascript and PHP datasets, the same downward tendency occurs in terms of Meteor and Rouge-L. Although the drop is not dramatic, considering the reported values are the average on all five datasets and our prompt embedding strategy is still used, the consistent decline demonstrates the contribution of the module in improving the performance. For the CVD task, from Table 2, there is an obvious degradation of 1.49, 1.95, and 1.03 in terms of F1, Precision, and Recall. For the CCD task, performance drops in terms of F1 and Precision, but an inverse variation takes place in terms of



(a) Performance of the first dataset (b) Average performance of datasets

Fig. 3. Performance recorded by the dataset order.

Recall. As Precision and Recall sometimes conflict, the consistent drop in the F1 score also shows that the module can improve our method.

For the effectiveness of PE-PTDD, we experiment by freezing the backbone PLM and keeping the prompt parameters trainable. Therefore, we only update prompts through input datasets. We evaluate all six datasets of the code summarization task following the same setting of RQ1. As shown in Table 1, when compared with PTDD, PE-PTDD's average performance decreases by 1.64, 1.13, and 1.43 in terms of BLEU, Meteor, and Rouge-L, respectively. Although PE-PTDD leads to degradations, we can observe that it still surpasses parameter-efficient baselines, namely PEPT and PP-TF. PE-PTDD outperforms them by 2.81% and 4.98% on average in terms of BLEU among all datasets (i.e. domains). The results indicate that our method is still effective if only tuning a fraction of the parameters.

Answer to RQ3: The distilled prompt learning module is essential for PTDD. In addition, in the parameter-efficient scenario, our method still shows effectiveness.

5.4. RQ4: Parameter analysis

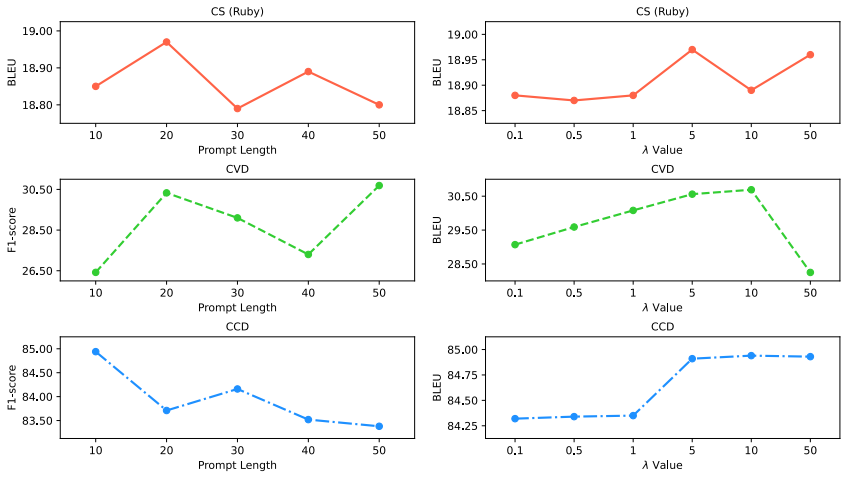
In this section, we investigate the impact of the prompt length L_P and the factor λ on the performance of PTDD. Intuitively, a longer prompt means more trainable parameters and more powerful representation capability. However, excessively lengthy prompts may also tend to overfit the training data and be detrimental to models. Therefore, it is crucial to set an appropriate prompt length to uncover the attainable performance. Besides, the factor λ is used to balance the learning objectives of plasticity and stability. Exploring a proper value is beneficial when applying PTDD in practice. To do so, we consider the Ruby dataset of code summarization and datasets from the other two tasks for investigation, where we only vary the analyzed parameter and keep the others unchanged. The prompt length is set to 10, 20, 30, 40, and 50, and the factor λ is varied in the range of {0.1, 0.5, 1, 5, 10, 50}. The experimental results are illustrated in Fig. 4.

As shown in the figure, we can observe that the optimal prompt length for different datasets is distinguishing. For the Ruby dataset of CS, the best performance is achieved when L_P is set to 20. Larger or lower values do not give better results. For the CVD task, PTDD performs well when L_P equals 20 or 50, but the performance is slightly higher at 50. For the CCD task, setting L_P to 10 can obtain the best result. A performance drop occurs if the prompt length is continuously increased. We suspect that this phenomenon may be related to the complexity of the tasks and the PLMs used, and we recommend adjusting an appropriate prompt length first if generalizing PTDD to other code intelligence tasks. As for the value of the factor λ, we can observe that there is a tendency that larger λ can obtain better results, and PTDD always achieves the best performance on the three datasets when λ is set to 5 or 10. Due to the more complex calculation requirements of the cross-entropy loss, λ needs to increase to emphasize the significance of keeping stability. The two losses become similar when λ is 5 or 10, thus improving PTDD's performance. Through the parameter analysis, we reveal the influence of L_P and λ, as well as select an appropriate setting for our method.

Answer to RQ4: The prompt length L_P relates to the added trainable parameters and the factor λ concerns the significance of keeping stability. Based on the experimental results, we suggest adjusting an appropriate L_P first on new tasks, and adapting λ around 5 and 10.

5.5. RQ5: Performance in the low-resource scenario

Low resource is a common scenario for code intelligence tasks [57]. Especially for the continual learning scenario, consistently constructing large-scale datasets is difficult. In this section, we investigate the



(a) Analysis of the prompt length (b) Analysis of the λ value

Fig. 4. Parameter analysis of the prompt length and λ value.

Table 3
Evaluation results in the low-resource scenario. M and R are short for Meteor and Rouge-L, respectively.

Samples	Methods	Ruby			Javascript			Java			Go			PHP			Python		
		BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R
10 × 5	FT	5.92	5.31	4.61	3.51	2.47	3.99	3.32	1.96	3.25	4.24	2.78	5.53	2.24	1.98	1.63	7.93	6.19	6.93
	PT	9.47	8.32	14.33	4.51	4.60	8.12	4.12	4.28	7.13	6.34	**8.98**	**12.96**	3.03	2.64	3.24	6.94	6.30	11.00
	EWC	9.36	7.16	13.00	4.56	4.54	7.84	1.97	1.90	2.17	5.82	5.36	9.98	2.43	**2.82**	2.21	2.99	4.93	4.97
	EMR	9.19	7.20	13.85	4.41	4.30	7.27	2.03	1.81	2.23	3.73	3.48	6.15	1.97	2.31	1.79	4.34	5.42	5.86
	PTDD	**9.57**	**8.69**	**14.70**	**4.79**	**5.24**	**9.09**	**5.38**	**5.23**	**9.55**	**6.83**	8.43	12.72	**3.19**	2.71	**3.42**	**8.32**	**7.47**	**14.27**
50 × 5	FT	6.29	5.01	7.01	6.16	4.35	8.77	5.24	3.60	7.61	11.76	7.72	19.08	7.42	5.69	10.66	10.46	7.95	13.99
	PT	9.89	8.55	17.20	7.23	6.29	12.65	**9.46**	7.86	17.51	15.28	12.05	27.71	12.69	10.01	19.21	10.34	8.61	17.92
	EWC	9.93	7.84	14.43	8.49	6.58	14.34	3.99	3.65	5.90	10.94	6.84	16.97	6.60	5.29	8.47	10.66	8.12	14.70
	EMR	9.98	7.86	14.37	7.55	5.86	12.55	4.02	3.54	5.97	12.80	8.07	20.07	7.34	5.51	9.07	10.37	7.84	13.97
	PTDD	**10.53**	**8.64**	**17.72**	**9.65**	**7.80**	**17.12**	**9.46**	**7.98**	**17.79**	**16.24**	**12.28**	**28.58**	**12.88**	**10.28**	**19.64**	**10.70**	**8.96**	**18.68**
100 × 5	FT	12.14	9.03	18.96	11.75	8.72	20.17	13.92	10.80	26.19	18.15	12.10	29.84	14.78	11.01	23.52	15.90	11.73	27.39
	PT	14.36	11.23	25.16	12.47	**9.81**	**22.33**	15.22	12.01	29.00	22.85	16.22	38.84	18.52	14.01	30.40	15.73	12.07	28.53
	EWC	12.99	10.05	22.76	11.99	9.51	21.52	13.76	11.26	26.44	23.52	15.33	38.24	17.36	12.64	26.98	14.75	10.74	24.17
	EMR	13.38	10.14	23.33	12.12	9.59	21.75	13.64	11.05	26.11	23.57	15.40	38.18	17.97	13.16	28.55	15.21	11.43	25.73
	PTDD	**14.59**	**11.25**	**25.27**	**12.51**	9.67	21.77	**15.35**	**12.05**	**29.08**	**23.59**	**16.28**	**39.37**	**18.60**	**14.05**	**30.56**	**16.07**	**12.80**	**29.36**

performance of PTDD when given only a small number of training samples, and we use the code summarization task as an example. For each dataset, we randomly select 10, 50, and 100 training samples, but keep the validation set and the test set unchanged. The training set size is about 0.02%–2.3% for their original size. Table 3 presents the evaluation results.

From Table 3, it can be observed that PTDD can generally exceed FT, PT, and EWC, even surpassing EMR in the low-resource scenario. For instance, when compared with FT and the number of training samples is 10, 50, and 100, the BLEU score of PTDD can obtain average promotions of 44.76%, 53.10%, and 15.63%, respectively. Accordingly, the average promotions are 67.88%, 50.13%, and 7.43% when compared with EWC, and reach 69.09%, 45.70%, and 5.67% when compared with EMR. In comparison with PT, the improvements become not drastic and are 11.79%, 8.53%, and 1.43%, respectively. In addition, there is a tendency for the improvement of PTDD to be stark in fewer training samples. The result is explicable because PTDD adds extra prompts that are beneficial to elicit pre-trained knowledge. Thus, even with scarce training data, PTDD can achieve acceptable performance while FT and EWC suffer from not having enough samples to learn task-specific knowledge. PT is affected because it does not mitigate the catastrophic forgetting problem. As for the EMR method, when reducing the training samples, the size of stored samples is also reduced, and consequently, its performance deteriorates. The observations demonstrate that PTDD is more advantageous in the low-resource scenario and enlighten us on the practical value of PTDD.

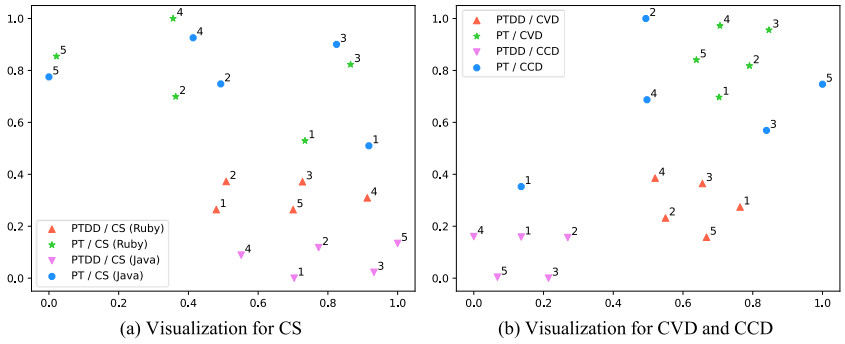
Answer to RQ5: In the low-resource scenario, the performance of PTDD can generally surpass FT, PT, EWC, and even EMR. The average promotion can reach up to 69.09%. In addition, the improvement of PTDD is more stark when training samples are fewer.

6. Discussion

6.1. Visualization for PTDD

To directly show the influence of PTDD in distilling prompts, we visualize the learned prompts in a two-dimensional representation space. Focusing on PT and our proposed PTDD, we carry out experiments on all three code intelligence tasks. In particular, regarding the CS task, Ruby and Java are selected for experiments. Subsequently, we use t-distributed Stochastic Neighbor Embedding (t-SNE) [58] on the learned prompts to obtain their two-dimensional representations. The t-SNE algorithm is a non-linear dimensionality reduction technique for data exploration, giving an intuition on how data is arranged in higher dimensions. Fig. 5 shows the visualization result. As there are five parts in each dataset, We mark out their order in the upper right corner.

As shown in Fig. 5, we can observe that the prompt distribution of PTDD is compact while scattered for PT. The discrete distribution of PT indicates that different datasets belong to different domains, and there are significant diversities that can lead to catastrophic forgetting. However, because PTDD calculates the cosine similarity between current prompts and corresponding previous prompts during training and



(a) Visualization for CS (b) Visualization for CVD and CCD

Fig. 5. Prompt visualization in a representation space. We mark out the order of each prompt in their upper right corner.

aims to minimize their differences, PTDD can restrict newly learned prompts not far from previous embeddings. Therefore, new prompts contain similar semantics to alleviate forgetting of learned knowledge. In addition, it can also be observed that the prompt distribution of PTDD is close to the first prompt embedding of PT. It demonstrates that when training on the first dataset, PTDD and PT can learn prompts with alike semantics. But when a sequence of datasets is inputted continually, PTDD can stabilize prompts' representations while PT cannot. The visualization results directly explain the superiority of PTDD in the CL scenario.

6.2. Case study

The advantage of PTDD comes from its utilization of prompts to instruct the representation of PLMs and thus mitigate the catastrophic forgetting of previously learned knowledge. In this section, as shown in Fig. 6, we present two instances to intuitively show the strength. Evaluating on the Java dataset of the code summarization task, we inspect predictions of the first Java dataset. We compare the initial prediction with the final prediction, which are obtained after training on the first subdataset and all five subdatasets. For the first instance, it can be observed that the initial summarizations of fine-tuning and PTDD are similar and are aligned with the ground truth. But after training on the other four subdatasets, the output of PTDD does not change but fine-tuning varies, outputting "checks" instead of the original "returns true". For the second instance, after training on all five subdatasets, PTDD can still accurately output keywords "TypeConverter", but fine-tuning outputs some extra unnecessary words like "the list of registered converters". We find that many cases that do not work well on fine-tuning but do well on PTDD follow a similar pattern of the second case, i.e., the final fine-tuning summarization outputs some extra words that do not appear initially. These cases indicate that simple fine-tuning tends to forget previously learned knowledge, thus producing low-quality summarization, while PTDD can catch the correct main idea of the source code because the learned knowledge is kept.

```
Code:
public boolean isToolSelected(int tool) {
    boolean selected = false;
    switch(tool) {
        case IBurpExtenderCallbacks.TOOL_PROXY :
            selected = jCheckBoxProxy.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_REPEATER :
            selected = jCheckBoxRepeater.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_SCANNER :
            selected = jCheckBoxScanner.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_INTRUDER :
            selected = jCheckBoxIntruder.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_SEQUENCER:
            selected = jCheckBoxSequencer.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_TARGET :
            break;
        default :
            break;
    }
    return selected;
}
Initial Fine-Tuning Summarization: Returns true if the specified tool is currently selected.
Final Fine-Tuning Summarization: Checks if the specified tool is selected.
Initial PTDD Summarization: Returns true if the specified tool is selected.
Final PTDD Summarization: Returns true if the specified tool is selected.
Ground Truth: Returns true if the requested tool is selected in the GUI.
```

```
Code:
public void removeConverter(final TypeConverter tc) {
    if (tc.getSupportedTypes() == null) {
        untypedTypeEncoders.remove(tc);
        registeredConverterClasses.remove(tc.getClass());
    }
    else {
        for (final Entry<Class, List<TypeConverter>> entry : tcMap.entrySet()) {
            List<TypeConverter> list = entry.getValue();
            if (list.contains(tc)) {
                list.remove(tc);
            }
            if (list.isEmpty()) {
                tcMap.remove(entry.getKey());
            }
        }
        registeredConverterClasses.remove(tc.getClass());
    }
}
Initial Fine-Tuning Summarization: Remove a type converter.
Final Fine-Tuning Summarization: Remove a converter from the list of registered converters.
Initial PTDD Summarization: Remove a type converter.
Final PTDD Summarization: Remove a TypeConverter.
Ground Truth: Removes the type converter.
```

Fig. 6. Examples of summarizations outputted by PTDD and Fine-Tuning.

6.3. Training cost analysis

In this section, we evaluate the training cost of PTDD and compare it with other representative continual learning methods. Specifically, using the code summarization task as an example, we investigate the training time per epoch and the memory consumption of the fifth part of the Java dataset. The evaluation results are illustrated in Table 4. From Table 4 and the conclusion about the performance in Section 5.2, it can be observed that the training cost of PTDD is comparable to that of FT and PT, while it performs better. When compared with EWC, PTDD consumes fewer training resources and obtains better results. Because of the introduction of additional calculations to select important parameters, EWC needs more training time and memory consumption. In comparison with EMR, although the performance of PTDD is slightly inferior, it shows greater efficiency. Without retraining stored data samples, PTDD utilizes prompts to instruct the presentation of PLMs, which is lightweight and shows effectiveness and practicality.

Table 4
Training cost analysis of PTDD and other representative continual learning methods.

Methods	Training time	Memory consumption
FT	11 min22 s	16,994M
PT	11 min33 s	17,784M
EWC	14 min46 s	20,392M
EMR	12 min03 s	18,084M
PTDD	11 min35 s	17,788M

6.4. Implications

This paper presents the first experimental study to investigate the performance of prompt tuning set in the CL scenario in the code intelligence field, and proposes a rehearsal-free CL method named PTDD. In this section, we discuss some implications of this work from the perspectives of developers and researchers.

Implications for developers. This work focuses on the continual learning scenario, where datasets are inputted sequentially to update PLMs so that they can be applied to wider domains. Considering the dynamic nature of software codebases, it is practical for developers in real-world scenarios. In addition, due to the potential restriction of data privacy in practice, this work discards the data selection and data storage module that previous works contain, and leverages the capability of prompt tuning in eliciting pre-trained knowledge to realize a rehearsal-free approach, enlightening developers about how to apply prompt tuning in the CL scenario. Experimental results in the low-resource scenario show the superiority of our approach when meeting the data scarcity problem.

Implications for researchers. This work also obtains several interesting findings that might be instructive for researchers. In the experimental study of prompt tuning set in the CL scenario, we find that freezing the PLM parameters prominently damages models' performance, especially for classification tasks such as code vulnerability detection and code clone detection. The results differ from findings in the NLP field [35], but are in accordance with previous studies in code intelligence [23]. We argue that this phenomenon is related to the size and quality of PLMs, as well as the complexity of downstream tasks. It deserves to be investigated in future works to better utilize prompt tuning. Besides, we also call for more attention to the CL scenario in the code intelligence field when applying PLMs.

7. Threats to validity

In this section, we identify threats to the validity of this work from two aspects.

External Validity mainly concerns the generalizability of our findings to other settings and datasets. One threat to the external validity comes from the limited used PLMs. Considering the computational resource constraints, our experiments are conducted on two widely-used PLMs, namely CodeBERT and CodeT5. However, more PLMs should be considered due to their promising performance, such as DeepSeek-Coder [59] and CodeLlama [60]. As illustrated in the previous work [35], PT becomes more competitive with larger PLMs' scale. It is deserved to explore our proposed PTDD on other PLMs in the future work. Besides, the limited choice of evaluation datasets is another threat. In this work, we adopt the project-level continual learning datasets open-sourced by Gao et al. [20], which are large-scale and their original datasets are well-known. However, as the research about continual learning in code intelligence is just beginning, there is a lack of alternative datasets to enrich our experiments. For example, we follow the same setting to input five datasets sequentially, but it also deserves to be investigated if the total number of datasets becomes more. In the future, we will construct a more comprehensive dataset to expand the experiments.

Internal Validity mainly concerns the aspects that could impact the experimental results. The first threat to the internal validity lies in the hyperparameter settings. Finding the optimal hyperparameters is always a difficult task. In this work, besides some conventional hyperparameters like batch size and learning rate that follow the usual, we conduct specific parameter analysis to adjust the prompt length and the factor λ. Although our analysis does not include all possible parameter values, this aspect is not crucial to the study as we have already shown the performance of PTDD with reasonable hyperparameter values. In addition, the internal validity is also related to the replication of baseline models and evaluation metrics. To mitigate these threats, we strictly follow their designs to reproduce or directly utilize their source code for model construction. Therefore, we conclude that it is a lightweight threat to validity.

8. Related work

8.1. Code intelligence

Code intelligence aims to leverage machine learning or deep learning techniques to facilitate software developers making programming more efficient, thriving in various tasks like code summarization [4,61], vulnerability detection [5,62], code clone detection [7,8], etc. Recently, benefiting from its effectiveness and versatility, the pre-training and fine-tuning paradigm is commonly adopted by state-of-the-art approaches. It first pre-trains language models on large-scale code corpora using elaborate pre-training objectives, like Masked Language Modeling (MLM) and Replaced Token Detection (RTD) [1]. During this process, language models acquire coarse-grained code knowledge and subsequently are fine-tuned on specific downstream tasks to improve their performance. For instance, CodeBERT [1] uses BERT [30], a multi-layer bidirectional Transformer [63], as the language model architecture, pre-trains it with both natural and programming languages, and obtains outstanding performance on two code intelligence tasks. Moreover, CodeT5 [2] builds on the T5 architecture [64], aiming to pre-train a unified Encoder–Decoder model, and shows its superiority in both code understanding and generation tasks.

Based on the pre-training and fine-tuning paradigm, recent studies find that adding prompts in the fine-tuning stage can elicit more pre-trained knowledge and modulate language models' behavior [65,66]. The initial added prompts are hard prompts [65], which use meaningful and task-specific natural language as prompts, but how to design suitable text prompts remains a challenge. To address the issue, Lester et al. [35] proposed prompt tuning, simplifying hard prompts to soft prompts, which are not meaningful words but continuous vectors and can be learned through backpropagation. Their sufficient experiments demonstrate the effectiveness of PT. In code intelligence, Huang et al. [67] first adopted the prompt tuning paradigm in program analysis. They achieved promising performance on the type inference task. Wang et al. [23] broadly evaluated PT on four popular code intelligence tasks, analyzing the different influences of hard prompts and soft prompts. Additionally, they investigated the performance of parameter-efficient prompt tuning. In this paper, based on current research, we further explore the performance of prompt tuning in the CL scenario.

8.2. Continual learning

Continual learning is set to learn a sequence of datasets one by one and behave on them simultaneously [22]. The major challenge is known as catastrophic forgetting [19], where models' performance on previous datasets is prone to decline when adapting the model to new datasets. Existing methods for CL mainly fall into three categories: regularization-based, rehearsal-based, and representation-based methods. Regularization-based methods [13,14] selectively constrain the variation of models' parameters, thus alleviating forgetting of learned knowledge. They usually add penalty terms to the loss function, and through the backpropagation process, those important parameters will be preserved. Rehearsal-based methods [15,16] store a fraction of data samples and retrain them when new datasets come, to ensure models still have the previous knowledge. Representation-based methods [17,18] primarily utilize prompts to instruct the representation of pre-trained models. By accommodating prompts in different datasets, they can elicit task-sharing and task-specific knowledge to relieve forgetting.

Due to the prosperity of prompt-based methods in recent years, representation-based methods have become popular with simplicity, practicality, and good performance. A representative strategy, L2P [17], introduces a prompt pool to store task-independent prompts and dynamically select the most relevant prompts for different inputs in inference. On the basis of L2P, DualPrompt [18] designs a general prompt and an expert prompt to learn the common features among all

tasks and the unique features of each task. Another typical method, Progressive Prompts [44], assigns each incoming task a separate prompt, and sequentially concatenates it with previously learned prompts to instruct the representation.

In code intelligence, as software development constantly evolves, to enhance models with new knowledge, it is essential and practical to investigate models' performance in the CL scenario. However, only a few works are dedicated to it. Gao et al. [20] first studied the problem. They proposed to retrain some carefully selected representative data, also designing adaptive parameter regularization, to prevent catastrophic forgetting. Weyssow et al. [21] evaluated five regularization-based and rehearsal-based approaches on the API call and API usage prediction task. Yadav et al. [24] constructed a CL benchmark, presented a method called prompt pooling with teacher forcing, and assessed its CL abilities on generation tasks. In this paper, focusing on the CL setting in the code intelligence field, we first explore the performance of vanilla prompt tuning, and then on the basis of representation-based methods, we pursue utilizing domain-wise distilled prompts to instruct the representation of PLMs and achieve promising results.

9. Conclusion

Concentrating on the CL scenario, this paper investigates the performance of vanilla prompt tuning and proposes a rehearsal-free CL method named PTDD, which can distill prompts and optimize backbone models sequentially by introducing an extra stability learning objective. From our first experimental study, we find that inheriting previous PLM parameters is beneficial in the CL scenario. For the effectiveness of our proposed method, extensive experiments in three code intelligence tasks indicate that PTDD can surpass a series of rehearsal-free baselines, even in the parameter-efficient setting. We also explain PTDD by prompt visualization and explore its performance when data is scarce. Finally, we summarize our findings and provide implications to enlighten future works. Our source code is publicly available at https://github.com/ishuoliu/IST-25-PTDD.

CRediT authorship contribution statement

Shuo Liu: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Jacky Keung:** Supervision, Project administration, Funding acquisition. **Zhen Yang:** Writing – review & editing, Supervision, Methodology. **Fang Liu:** Writing – review & editing. **Fengji Zhang:** Writing – review & editing. **Yicheng Sun:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong and the research funds of the City University of Hong Kong, Hong Kong (Grant No. 6000796, 9229109, 9229098, 9220103, 9229029), also by the Natural Science Foundation of Shandong Province, China (Grant No. ZR2024QF093) and National Natural Science Foundation of China (Grant No. 62302021).

Data availability

Data will be made available on request.

References

[1] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al., CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[2] Yue Wang, Weishi Wang, Shafiq Joty, Steven C.H. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, Kai-Wei Chang, A transformer-based approach for source code summarization, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 4998–5007.

[4] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, Michael Lyu, Code structure–guided transformer for source code summarization, ACM Trans. Softw. Eng. Methodol. 32 (1) (2023) 1–32.

[5] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, Wei Le, An empirical study of deep learning models for vulnerability detection, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 2237–2248.

[6] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, Shanping Li, Vulnerability detection by learning from syntax-based execution paths of code, IEEE Trans. Softw. Eng. 49 (8) (2023) 4196–4212.

[7] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, Qingkai Shi, Functional code clone detection with syntax and semantics fusion learning, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 516–527.

[8] Chenning Tao, Qi Zhan, Xing Hu, Xin Xia, C4: Contrastive cross-language code clone detection, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 413–424.

[9] Marvin Zhang, Henrik Marklund, Nikita Dhawan, Abhishek Gupta, Sergey Levine, Chelsea Finn, Adaptive risk minimization: Learning to adapt to domain shift, Adv. Neural Inf. Process. Syst. 34 (2021) 23664–23678.

[10] Marius Nita, David Notkin, Using twinning to adapt programs to alternative APIs, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010, pp. 205–214.

[11] Huajie Shao, Dachun Sun, Jiahao Wu, Zecheng Zhang, Aston Zhang, Shuochao Yao, Shengzhong Liu, Tianshi Wang, Chao Zhang, Tarek Abdelzaher, Paper2repo: Github repository recommendation for academic papers, in: Proceedings of the Web Conference 2020, 2020, pp. 629–639.

[12] Gido M. Van de Ven, Tinne Tuytelaars, Andreas S. Tolias, Three types of incremental learning, Nat. Mach. Intell. 4 (12) (2022) 1185–1197.

[13] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al., Overcoming catastrophic forgetting in neural networks, in: PNAS Proceedings of the National Academy of Sciences of the United States of America, National Academy of Sciences, 2017.

[14] Xialei Liu, Marc Masana, Luis Herranz, Joost Van de Weijer, Antonio M. Lopez, Andrew D. Bagdanov, Rotate your networks: Better weight consolidation and less catastrophic forgetting, in: 2018 24th International Conference on Pattern Recognition, ICPR, IEEE, 2018, pp. 2262–2268.

[15] Hong Wang, Wenhan Xiong, Mo Yu, Xiaoxiao Guo, Shiyu Chang, William Yang Wang, Sentence embedding alignment for lifelong relation extraction, in: Annual Conference of the North American Chapter of the Association for Computational Linguistics, Association for Computational Linguistics (ACL), 2019.

[16] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, Christoph H. Lampert, Icarl: Incremental classifier and representation learning, in: Conference on Computer Vision and Pattern Recognition, CVPR, 2017, pp. 5533–5542.

[17] Zifeng Wang, Zizhao Zhang, Chen-Yu Lee, Han Zhang, Ruoxi Sun, Xiaoqi Ren, Guolong Su, Vincent Perot, Jennifer Dy, Tomas Pfister, Learning to prompt for continual learning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 139–149.

[18] Zifeng Wang, Zizhao Zhang, Sayna Ebrahimi, Ruoxi Sun, Han Zhang, Chen-Yu Lee, Xiaoqi Ren, Guolong Su, Vincent Perot, Jennifer Dy, et al., Dualprompt: Complementary prompting for rehearsal-free continual learning, in: European Conference on Computer Vision, Springer, 2022, pp. 631–648.

[19] Michael McCloskey, Neal J. Cohen, Catastrophic interference in connectionist networks: The sequential learning problem, in: Psychology of Learning and Motivation, vol. 24, Elsevier, 1989, pp. 109–165.

[20] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, Chaozheng Wang, Keeping pace with ever-increasing data: Towards continual learning of code intelligence models, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 30–42.

[21] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, Houari Sahraoui, On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 1470–1482.

[22] Liyuan Wang, Xingxing Zhang, Hang Su, Jun Zhu, A comprehensive survey of continual learning: theory, method and application, IEEE Trans. Pattern Anal. Mach. Intell. (2024).

[23] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, Michael R. Lyu, Prompt tuning in code intelligence: An experimental evaluation, IEEE Trans. Softw. Eng. (2023).

[24] Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Parminder Bhatia, Xiaofei Ma, Ramesh Nallapati, Murali Krishna Ramanathan, et al., Exploring continual learning for code generation models, in: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2023, pp. 782–792.

[25] Xiang Lisa Li, Percy Liang, Prefix-tuning: Optimizing continuous prompts for generation, in: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2021, pp. 4582–4597.

[26] James Seale Smith, Leonid Karlinsky, Vyshnavi Gutta, Paola Cascante-Bonilla, Donghyun Kim, Assaf Arbelle, Rameswar Panda, Rogerio Feris, Zsolt Kira, Coda-prompt: Continual decomposed attention-based prompting for rehearsal-free continual learning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 11909–11919.

[27] Wei Yuan, Hongzhi Yin, Tieke He, Tong Chen, Qiufeng Wang, Lizhen Cui, Unified question generation with continual lifelong learning, in: Proceedings of the ACM Web Conference 2022, 2022, pp. 871–881.

[28] Yue Cao, Hao-Ran Wei, Boxing Chen, Xiaojun Wan, Continual learning for neural machine translation, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 3964–3974.

[29] Xu Han, Yi Dai, Tianyu Gao, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, Jie Zhou, Continual relation learning via episodic memory activation and reconsolidation, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 6429–6440.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 2019, pp. 4171–4186.

[31] Tianyi Zhang, Felix Wu, Arzoo Katiyar, Kilian Q Weinberger, Yoav Artzi, Revisiting few-sample BERT fine-tuning, in: International Conference on Learning Representations, 2021.

[32] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, Yihan Liao, Delving into parameter-efficient fine-tuning in code change learning: An empirical study, 2024, arXiv preprint arXiv:2402.06247.

[33] J. Liu, C. Sha, X. Peng, An empirical study of parameter-efficient fine-tuning methods for pre-trained code models, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2023, pp. 397–408.

[34] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, Hongbin Sun, Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 39–51.

[35] Brian Lester, Rami Al-Rfou, Noah Constant, The power of scale for parameter-efficient prompt tuning, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 3045–3059.

[36] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, Graham Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, 2021, arXiv preprint arXiv:2107.13586.

[37] Yihong Ma, Ning Yan, Jiayu Li, Masood Mortazavi, Nitesh V. Chawla, Hetgpt: Harnessing the power of prompt tuning in pre-trained heterogeneous graph neural networks, in: Proceedings of the ACM on Web Conference 2024, 2024, pp. 1015–1023.

[38] Jianing Wang, Chengyu Wang, Fuli Luo, Chuanqi Tan, Minghui Qiu, Fei Yang, Qiuhui Shi, Songfang Huang, Ming Gao, Towards unified prompt tuning for few-shot text classification, in: Findings of the Association for Computational Linguistics: EMNLP 2022, 2022, pp. 524–536.

[39] Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, Beijun Shen, Few-shot code translation via task-adapted prompt learning, J. Syst. Softw. 212 (2024) 112002.

[40] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, Michael R. Lyu, No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 382–394.

[41] Guanzheng Chen, Fangyu Liu, Zaiqiao Meng, Shangsong Liang, Revisiting parameter-efficient tuning: Are we really there yet? in: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 2612–2626.

[42] Aybüke Aurum, Ross Jeffery, Claes Wohlin, Meliha Handzic, Managing Software Engineering Knowledge, Springer Science & Business Media, 2013.

[43] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, Jie Tang, GPT understands, too, AI Open (2023).

[44] Anastasia Razdaibiedina, Yuning Mao, Rui Hou, Madian Khabsa, Mike Lewis, Amjad Almahairi, Progressive prompts: Continual learning for language models, in: International Conference on Learning Representations, 2023.

[45] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, Hongbin Sun, On the evaluation of neural code summarization, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1597–1608.

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

[47] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, Baishakhi Ray, Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (9) (2021) 3280–3296.

[48] Jiahao Fan, Yi Li, Shaohua Wang, Tien N. Nguyen, AC/C++ code vulnerability dataset with code changes and CVE summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.

[49] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, Dae-Kyoo Kim, Deep learning application on code clone detection: A review of current knowledge, J. Syst. Softw. 184 (2022) 111141.

[50] Lili Mou, Ge Li, Lu Zhang, Tao Wang, Zhi Jin, Convolutional neural networks over tree structures for programming language processing, in: Proceedings of the AAAI Conference on Artificial Intelligence, 30, (1) 2016.

[51] Hao Yu, Xing Hu, Ge Li, Ying Li, Qianxiang Wang, Tao Xie, Assessing and improving an evaluation dataset for detecting semantic code clones via deep learning, ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (4) (2022) 1–25.

[52] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, et al., An extractive-and-abstractive framework for source code summarization, ACM Trans. Softw. Eng. Methodol. 33 (3) (2024) 1–39.

[53] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

[54] Satanjeev Banerjee, Alon Lavie, METEOR: An automatic metric for MT evaluation with improved correlation with human judgments, in: Proceedings of the Acl Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/Or Summarization, 2005, pp. 65–72.

[55] Chin-Yew Lin, Rouge: A package for automatic evaluation of summaries, in: Text Summarization Branches Out, 2004, pp. 74–81.

[56] Fei Mi, Liangwei Chen, Mengjie Zhao, Minlie Huang, Boi Faltings, Continual learning for natural language generation in task-oriented dialog systems, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 3461–3474.

[57] Fuxiang Chen, Fatemeh H. Fard, David Lo, Timofey Bryksin, On the transferability of pre-trained language models for low-resource programming languages, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 401–412.

[58] Laurens Van der Maaten, Geoffrey Hinton, Visualizing data using t-SNE, J. Mach. Learn. Res. 9 (11) (2008).

[59] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al., DeepSeek-coder: When the large language model meets programming–the rise of code intelligence, 2024, arXiv preprint arXiv:2401.14196.

[60] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al., Code llama: Open foundation models for code, 2023, arXiv preprint arXiv:2308.12950.

[61] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, Retrieval-based neural source code summarization, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1385–1397.

[62] Kollin Napier, Tanmay Bhowmik, Shaowei Wang, An empirical study of text-based machine learning models for vulnerability detection, Empir. Softw. Eng. 28 (2) (2023) 38.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

[64] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, J. Mach. Learn. Res. 21 (140) (2020) 1–67.

[65] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al., Language models are few-shot learners, Adv. Neural Inf. Process. Syst. 33 (2020) 1877–1901.

[66] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, Sameer Singh, AutoPrompt: Eliciting knowledge from language models with automatically generated prompts, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP, Association for Computational Linguistics, 2020.

[67] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, Qinghua Lu, Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–13.