Learning in the Wild: Towards Leveraging Unlabeled Data for Effectively Tuning Pre-trained Code Models

Shuzheng Gao
The Chinese University of Hong Kong
Hong Kong, China
szgao23@cse.cuhk.edu.hk

Wenxin Mao
Harbin Institute of Technology
Shenzhen, China
maowx5519@mails.jlu.edu.cn

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

Li Li
Beihang university
Beijing, China
lilicoding@ieee.org

Xing Hu, Xin Xia
Zhejiang university
Zhejiang, China
xinghu@zju.edu.cn,xin.xia@acm.org

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

ABSTRACT

Pre-trained code models have recently achieved substantial improvements in many code intelligence tasks. These models are first pre-trained on large-scale unlabeled datasets in a *task-agnostic* manner using self-supervised learning, and then fine-tuned on labeled datasets in downstream tasks. However, the labeled datasets are usually limited in size (i.e., human intensive efforts), which may hinder the performance of pre-trained code models in specific tasks. To mitigate this, one possible solution is to leverage the large-scale unlabeled data in the tuning stage by pseudo-labeling, i.e., generating pseudo labels for unlabeled data and further training the pre-trained code models with the pseudo-labeled data. However, directly employing the pseudo-labeled data can bring a large amount of noise, i.e., incorrect labels, leading to suboptimal performance. How to effectively leverage the noisy pseudo-labeled data is a challenging yet under-explored problem.

In this paper, we propose a novel approach named HINT to improve pre-trained code models with large-scale unlabeled datasets by better utilizing the pseudo-labeled data. HINT includes two main modules: **H**ybr**I**d pseudo-labeled data selection and **N**oise-tolerant **T**raining. In the hybrid pseudo-data selection module, considering the robustness issue, apart from directly measuring the quality of pseudo labels through training loss, we propose to further employ a retrieval-based method to filter low-quality pseudo-labeled data. The noise-tolerant training module aims to further mitigate the influence of errors in pseudo labels by training the model with a noise-tolerant loss function and by regularizing the consistency of model predictions. We evaluate the effectiveness of HINT on three popular code intelligence tasks, including code summarization, defect detection, and assertion generation. We build our method on top of three popular open-source pre-trained code models. The experimental results show that HINT can better leverage those unlabeled data in a *task-specific* way and provide complementary benefits for pre-trained models, e.g., improving the best baseline model by 15.33%, 16.50%, and 8.98% on code summarization, defect detection, and assertion generation, respectively.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**;

ACM Reference Format:
Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R. Lyu. 2024. Learning in the Wild: Towards Leveraging Unlabeled Data for Effectively Tuning Pre-trained Code Models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639216

1 INTRODUCTION

Recently, code intelligence has become a popular research field in software engineering. It aims at improving developers' productivity by providing real-time coding assistance and suggestions for them [9, 28]. The advent of deep learning techniques, especially pre-training techniques [12, 53], has significantly advanced progress in this area. Different from previous supervised learning methods that train the model from scratch [1, 71], these pre-trained code models are first pre-trained on large-scale unlabeled datasets using self-supervised learning tasks and then fine-tuned on labeled datasets in downstream tasks. For example, Masked Language Modeling (MLM) is one of the most popular self-supervised pre-training tasks and is used in many pre-trained code models such as CodeBERT [14] and GraphCodeBERT [21]. It works by training the models to predict the masked tokens based on the context of surrounding words. Since this process does not require human annotation, it can be applied on large-scale unlabeled datasets, enabling the models to acquire a vast amount of general programming knowledge. Equipped with this ability, these pre-trained code models achieve state-of-the-art performance on a variety of code intelligence tasks, such as code summarization and defect detection [14, 17, 20, 21].

Despite the promising results, deep learning models are known to be data-hungry and the size of labeled datasets in downstream tasks is important for the performance of pre-trained models [23, 65]. However, the sizes of labeled datasets in downstream tasks are

*Corresponding author. The author is also affiliated with Peng Cheng Laboratory and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies.

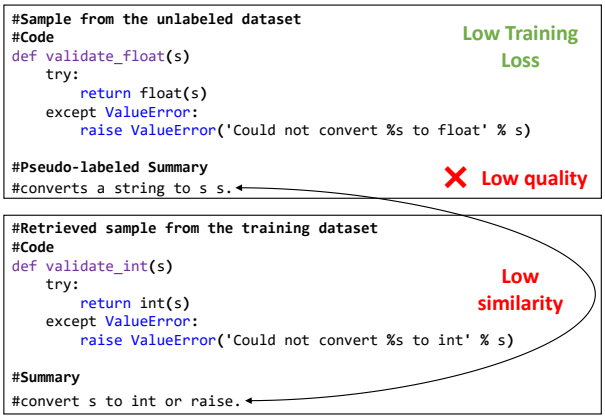
ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639216

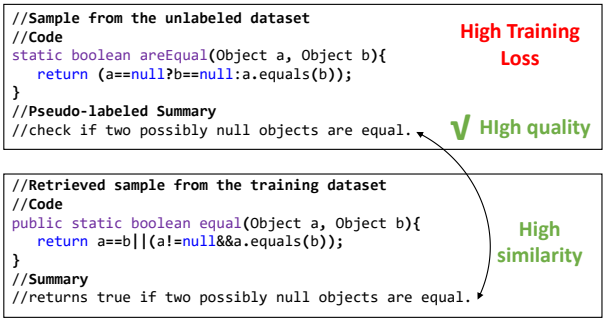
usually limited due to two main reasons. On one hand, the datasets crawled from open-source websites like Github or Stackoverflow are small in size and of low quality. For example, as mentioned in the literature [31], only 6.8% JavaScript code snippets from popular GitHub repositories contain corresponding comments, making only a few of them usable for tasks like code summarization. Furthermore, recent studies have revealed that the quality of existing crawled datasets is also quite poor [11, 56, 57]. For example, as indicated in a recent work [56], over 40% of data in the widely-used code summarization datasets contain various types of noise. On the other hand, due to the requirement of domain expert knowledge, the annotation cost of code intelligence tasks is higher than other tasks in natural language processing or computer vision, such as sentiment analysis and image classification [59]. With insufficient annotated data in downstream tasks, the performance of pre-trained code models is limited.

One possible solution to this problem is to leverage the large-scale unlabeled data in the tuning stage by pseudo-labeling. Pseudo-labeling first trains a base model on the limited labeled dataset, which subsequently serves as a teacher model to annotate the unlabeled dataset [34, 40, 47]. The pseudo-labeled dataset is then merged with the original labeled dataset to help improve the training of a new student model. By replacing the teacher model with the stronger student model, the above process can be iterated multiple times, aiming at improving the models themselves. This technique leverages the unlabeled data in a *task-specific* way and has shown promising results in tasks such as image classification [40] and dialog systems [47]. Although pseudo-labeling can enrich the labeled dataset, directly employing the pseudo-labeled data can bring a large amount of noise [47]. For example, as shown in Figure 1 (a), the pseudo-labeled summary of the top code snippet is not a meaningful sentence and contains redundant tokens. Training with such noisy pseudo labels may amplify the incorrect knowledge in the teacher model and ultimately degrades the model's performance. However, identifying and removing noisy pseudo labels is non-trivial due to the complex semantic of source code. Besides, it is difficult and impractical to ensure that the filtered dataset is noise-free [63, 73]. Therefore, how to effectively leverage the noisy pseudo-labeled data and enable the model to be noise-tolerant for code intelligence tasks is of vital importance, yet under-explored.

In this paper, we propose HINT with two main components, i.e., the **H**ybr**I**d pseudo-labeled data selection module and the **N**oise-tolerant **T**raining module. First, in the hybrid pseudo-labeled data selection module, we propose to combine the training loss of the teacher model and a retrieval-based method for removing the low-quality data. Specifically, we filter out pseudo-labeled samples that present high training loss or low label similarity with the retrieved similar training sample. To further mitigate the influence of data noise on model performance, we propose a noise-tolerant training objective that includes a noise-tolerant symmetric loss function and a consistency regularization of model predictions. To evaluate the performance of HINT, we conduct experiments on three popular code intelligence tasks including code summarization, defect detection, and assertion generation. Following previous work [18, 50, 62], we build our method on top of three popular open-source pre-trained models: CodeBERT [14], CodeT5 [64], and UniXcoder [20].



(a) A Python example of low-quality pseudo-labeled data (top).



(b) A Java example of high-quality pseudo-labeled data (top).

Figure 1: Examples in the code summarization task for illustrating the motivation of the hybrid pseudo-labeled data selection method, which indicates the loss-based data selection strategy alone may incorrectly measure the quality of pseudo labels.

Extensive experiments demonstrate that HINT can consistently improve the performance of pre-trained code models on these code intelligence tasks. For example, HINT improves UniXcoder by 15.33%, 16.50%, and 8.98% in terms of BLEU-4, F1, and EM on code summarization, defect detection, and assertion generation, respectively, indicating that our proposed HINT method can provide complementary benefits for the pre-trained code models.

In summary, the main contributions of this work are as follows:

(1) To the best of our knowledge, we are the first to leverage the large-scale unlabeled data in a task-specific way in the turning phase for code intelligence tasks.
(2) We propose HINT, a novel framework to leverage large-scale unlabeled data for effectively tuning pre-trained code models. It first selects high-quality pseudo-labeled data in a hybrid way and then improves the model's tolerance to noisy data in the training process.
(3) Extensive experiments on three tasks demonstrate that our method can be built on top of a range of existing strong pre-trained models and consistently improve their performance on many downstream tasks.

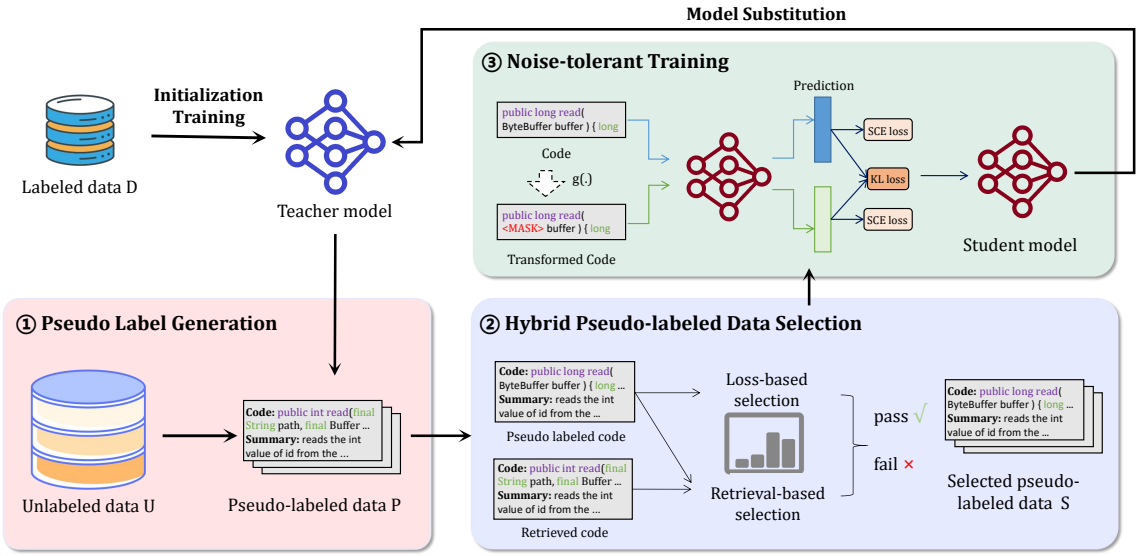


Figure 2: The overview of HINT.

2 PROPOSED APPROACH

2.1 Problem Setup and Overview

In this section, we explicate the detailed design of HINT. Formally, in code intelligence tasks such as code summarization, we have a set of source codes X and summaries Y. Let $D = \{(x^i, y^i)\}_{i=1}^{N}$ denotes the labeled training dataset, where $x^i \in X, y^i \in Y$ and N denotes the size of D. Let $U = \{x^i\}_{i=1}^{M}$ denote the large unlabeled dataset, where M denotes the size of U and $M > N$ in general. Our goal is to learn a model $f : X \mapsto Y$ from both D and U that can well predict the label of input x^i in the test set.

The overall framework of HINT is shown in Figure 2. We first train a teacher model on the original labeled dataset D and ① use the teacher model to generate pseudo labels for the unlabeled dataset. Then, ② a hybrid pseudo-labeled data selection method that contains *loss-based selection* and *retrieval-based selection* is proposed to filter the code with low-quality pseudo labels (introduced in Section 2.2). For further mitigating the influence of noise in pseudo labels during model training, we propose ③ a noise-tolerant training strategy that trains the student model with noise-tolerant symmetric cross entropy loss and consistency regularization (introduced in Section 2.3). The above procedure can be iterated multiple times, enabling the models to be self-improved (introduced in Section 2.4). The algorithm is shown in Algorithm 1.

2.2 Hybrid Pseudo-labeled Data Selection

Once we get a trained teacher model $\mathcal{F}_t$, we use it to generate pseudo labels for unlabeled dataset U, producing a pseudo labeled dataset $P = \{(x^i, \hat{y}^i)\}_{i=1}^{M}$. The pseudo-labeled data cannot be employed directly, since they may contain substantial noise and impact the model performance. Previous studies in machine learning [22, 30] mainly employ *loss-based selection* by filtering the data with high training loss based on the insight that neural models can well distinguish the quality of each sample (i.e., noisy data are generally associated with higher training loss). However, code intelligence models are known to suffer from the robustness issue [25], so solely relying on the model training loss for noise filtering is ineffective. For the example in Figure 1 (a), we can observe that although the quality of this generated summary is pretty poor, its loss is low in value. Specifically, when comparing the loss of all the pseudo-labeled data, it exhibits a lower loss than 83% of the pseudo-labeled data. Besides, in Figure 1 (b), the generated pseudo summary can well describe the meaning of checking the equivalence of two objects in the Java code snippet but its training loss value is relatively high, i.e., surpassing 52% of the pseudo-labeled data.

Considering that code reuse is widespread in software development [35, 37], apart from the *loss-based selection*, we propose to further select high-quality data through a retrieval-based method. As shown in Figure 1, by comparing the pseudo-labeled summaries and retrieved summaries, we can systematically identify the pseudo-labeled data in Figure 1 (b) as a high-quality sample and filter the low-quality pseudo-labeled data in Figure 1 (a). Specifically, in the *retrieval-based selection*, for each unlabeled data x^i, we first use the widely-used BM-25 method [44] to retrieve the most similar code x^j in the labeled training set. Then we propose to compare the similarities of x^i and x^j and their corresponding pseudo label $\hat{y}^i$ and groud truth label y^j through normalized edit distance:

$$NED(x, y) = \begin{cases} \frac{edit\_distance(x,y)}{||x||} & \textbf{if } x, y \in sequence \\ \mathrm{I}\{x \neq y\} & \textbf{if } x, y \in label \end{cases} \quad (1)$$

where $||.||$ denotes the length of the sequence and $\mathrm{I}\{.\}$ is an indicator function that returns 1 if the condition is true and 0 otherwise. Specifically, if both $NED(x^i, x^j)$ and $NED(\hat{y}^i, y^j)$ are not higher than the threshold t, we consider this sample $(x^i, \hat{y}^i)$ as a correctly predicted sample and add it to the selected dataset S. On the contrary, if $NED(x^i, x^j)$ is lower than t while $NED(\hat{y}^i, y^j)$ is above $1 - t$, we choose to filter it as it has a higher probability of being a noisy data (Line 9-11 in Algorithm 1). Here t is a hyperparameter

Algorithm 1 Algorithm of HINT

Input: labeled dataset D, unlabeled dataset U, threshold of edit distance t, the threshold in loss-based selection K, code transformation function g, iteration number I
Output: neural model $\mathcal{F}$

1: Train the teacher model $\mathcal{F}_t$ on D
2: **for** each i in I **do**
3: Generate pseudo data P for U using $\mathcal{F}_t$
4: Calculate the loss of samples in P using $\mathcal{F}_t$
5: TK $\leftarrow$ samples with the least top $K\%$ loss value in P
6: $S \leftarrow \emptyset$
7: **for** each sample $\{x_i, y_i'\}$ in P **do**
8: Retrieve the most similar sample (x_j, y_j) from D
9: **if** $NED(x_i, x_j) \leq t \wedge NED(y_i, y_j') \leq t$ **then**
10: $S.insert(\{x_i, y_i'\})$
11: **else if** $NED(x_i, x_j) \leq t \wedge NED(y_i, y_j') \geq 1 - t$ **then**
12: **continue**
13: **else if** $\mathcal{F}_t(x_i, y_i') \in$ TK **then**
14: $S.insert(\{x_i, y_i'\})$
15: **end if**
16: **end for**
17: $L \leftarrow D \cup S$
18: Train the student model $\mathcal{F}_s$ with dataset L and transformation function g through Equation 4
19: $\mathcal{F}_t \leftarrow \mathcal{F}_s$
20: **end for**
21: **return** model $\mathcal{F}_t$

to control the filtering threshold. For samples that cannot be decided by the *retrieval-based selection*, we first calculate the training loss of each pseudo-labeled data by re-feeding each code x^i into the teacher model $\mathcal{F}_t$ and using the generated pseudo label $\hat{y}^i$ as the ground-truth label. We then select the top $K\%$ data with the lowest loss values among all pseudo-labeled data and add them to S (Line 4-5, 13-14 in Algorithm 1). Finally, we obtain a dataset $S = (x^i, \hat{y}^i)_{i=1}^{M'}$ containing high-quality pseudo-labeled data. The dataset S is employed to train the student model, together with the labeled dataset D.

2.3 Noise-tolerant Training

Despite the dedicated data selection effort, it is still difficult and impractical to ensure that the selected samples S are noise-free. Besides, the pseudo-labeled samples with minor noise are not completely harmful and can also provide rich information for model training. For example, as shown in Figure 3, the assertion statement generated by the teacher model mistakenly predicts the assertion type as "*assertionEquals*". If we directly use it as ground truth to train the model, the model may be misled. Nevertheless, this sample still contains much valuable information since the predictions on other positions such as the parameters are correct. Therefore, instead of directly discarding these samples with a more strict filtering process, we propose to leverage the pseudo-labeled data with noise-tolerant loss function and consistency regularization during model training.

```
//Code from the Unlabeled dataset:
testPreserveProperty(){
    java.lang.Object value = new java.lang.Object();
    adapter.preserve("prop",value);
    "<AssertPlaceHolder>";
}
getPreserved(java.lang.String)
{
    throw new java.lang.UnsupportedOperationException();
}
```

```
Ground truth assertion:
org.junit.Assert.assertSame(value,adapter.getPreserved("prop"))
```

```
Pseudo-labeled assertion:
org.junit.Assert.assertEquals(value,adapter.getPreserved("prop"))
```

Figure 3: An example of a pseudo label with a minor error.

Noise-tolerant Loss Function: Previous studies [63, 73] have found that the widely-used cross entropy (CE) loss function is sensitive to noisy training data. Specifically, the coefficient in the gradient of the CE loss function $-\frac{1}{f_{y^i}(x^i;\theta)}\nabla f_{y^i}(x^i;\theta)$ assigns larger weights to samples with higher loss and smaller weights to samples with lower loss. Since noisy data often obtain a high training loss in the training process [22, 30], models trained with CE loss easily focus on those noisy data and tend to be misled. However, the re-weighting coefficient in the gradient of CE is also beneficial for model training. Directly removing it as $-\nabla f_{y^i}(x^i;\theta)$ might bring the slow convergence problem [73]. To deal with this problem, we propose to employ the Symmetric Cross Entropy loss (SCE) [63]:

$$l_{sce}(x, y) = -\sum_{c=1}^{C}(p(c|x_i)\log(q(c|x)) + q(c|x_i)\log(p(c|x))), \quad (2)$$

where $q(c|x)$ is the prediction of the model and $p(c|x)$ is the corresponding ground truth label in the dataset. C denotes the number of classes in the classification task or the size of vocabulary in the generation task. The former item represents the CE loss, while the latter item corresponds to the Reverse Cross Entropy (RCE) [63] which assigns the same weights for all samples, i.e., $-\nabla f_{y^i}(x^i;\theta)$. Note that $\log p(c|x)$ is 0 for the ground truth class and the same negative value for other classes. Based on the relation $p(c^*|x) = 1 - \sum_{c=1,c\neq c^*}^{C} p(c|x)$ where c^* denotes the ground class, we can achieve the above result. In this way, the student model is less likely to be influenced by minor errors in pseudo labels.

Consistency Regularization: According to Equation 2, the noise-tolerant loss relies on the quality of pseudo labels. Considering that such supervision signals from pseudo labels might be noisy and unreliable, we further propose to add consistency regularization between predictions of the original code and the transformed code. It could enrich the supervision signals and provide the student model with a more reliable objective function without manual labels. Specifically, given a code snippet x, we first apply a code transformation function $ct(\cdot)$ and obtain the transformed input $ct(x)$. Then, the consistency regularization is applied to align the distributions of the predictions $\mathcal{F}_s(x)$ and $\mathcal{F}_s(ct(x))$:

$$l_{cr} = KL(\mathcal{F}_s(x)||\mathcal{F}_s(ct(x))) + KL(\mathcal{F}_s(ct(x))||\mathcal{F}_s(x)), \quad (3)$$

where $KL(\cdot||\cdot)$ denotes the Kullback-Leibler Divergence [38]. For generation tasks, we approximate it by the average value per token. For code transformation methods, we follow previous work [55]

and employ four effective transformation methods, including Dynamic Masking, Dynamic Replacement, Dynamic Replacement of Specified Type, and Dynamic Masking of Specified Type. For each sample, we randomly select one transformation function in each epoch. Different from previous task-agnostic contrastive learning pre-training methods that only focus on aligning the representation of code and transformed code [20, 33], our method directly constrains the prediction $\mathcal{F}_s(x)$ and $\mathcal{F}_s(ct(x))$ of code and transformed code on downstream tasks and regularizes them in a task-specific way.

Finally, HINT trains the student model by combining the noise-tolerant loss function and consistency regularization as follows:

$$l = \sum_{(x,y)\in D\cup S} [\ l_{sce}(x,y) + l_{sce}(ct(x),y) + \mu \cdot l_{cr}\ ], \tag{4}$$

where μ is a hyperparameter to balance the training signals from pseudo labels and the consistency regularization.

2.4 Iterative Training

Based on the aforementioned process, we can obtain a student model that has better performance than the teacher model. Then, we can build upon this student model and repeat the process described above to further boost the models themselves (① → ② → ③ → ①). Specifically, at the end of each iteration, the student model substitutes the teacher model, which is then employed to generate pseudo labels for the unlabeled dataset D in the subsequent iteration. In general, the better the base model, the higher the quality of the pseudo labels. In this way, the student model in the next iteration is more likely to be trained on pseudo-labeled data with higher quality and thus achieves better performance. We follow previous work [47] and reinitialize the new student model from the pre-trained code models in every iteration. After all iterations, the student model in the last iteration will be used as the final model for predictions on the test set.

3 EXPERIMENTAL SETUP

3.1 Research Questions

As we claimed above, HINT is a generic framework that works without imposing specific assumptions regarding data distribution, underlying models, or task characteristics, except for the requirement that the input needs to be in the form of code. To validate the generalizability of HINT, we propose to evaluate the performance of HINT on a variety of pre-trained code models and three semi-supervised code intelligence tasks with code as input. As for the data distribution, we also evaluate the performance of HINT on cross-domain scenarios that do not have sufficient training data and may present data distribution gap between training and unlabeled data. Furthermore, we also explore the effectiveness of each component in HINT and the influence of hyperparameters on its performance. In summary, we evaluate HINT by addressing the following four research questions:

- **RQ1:** How much improvement can HINT provide to existing pre-trained code models?
- **RQ2:** What is the impact of each component on the performance of HINT?
- **RQ3:** How well does HINT perform in cross-domain scenarios?
- **RQ4:** How does HINT's performance vary under different parameter settings?

3.2 Evaluation Tasks

We conduct experiments on three representative code intelligence tasks: code summarization, defect detection, and assertion generation, for covering different task types, i.e., Code → Text, Code → Label, and Code → Code. Due to the space limitation, we provide a more detailed description of evaluation metrics and statistics of the benchmark datasets in our replication packages [26].

3.2.1 Code Summarization. Code Summarization aims to generate useful comments for a given code snippet. It can help alleviate the developers' cognitive efforts in comprehending programs [7, 19].

Datasets. In this study, we conduct experiments on two popular benchmark datasets JCSD and PCSD, which contain Java and Python source code, respectively. The JCSD dataset we used is publicly released by Hu et al. [27], which contains 87,136 pairs of Java methods and comments collected from 9,714 GitHub repositories. The PCSD dataset comprises 92,545 functions with their respective documentation, which is originally collected by Barone et al. [3] and later processed by Wei et al. [68]. For our experiments, we directly used the benchmark datasets released by previous studies [1, 27], in which the datasets are divided into training, validation, and test sets in a ratio of 8 : 1 : 1 and 6 : 2 : 2 for Java and Python, respectively. As reported in previous work [48, 54], there are duplicated data in the training and test set of the JCSD dataset. Therefore, following them, we remove the test samples that also appear in the training or validation set and finally get a deduplicated test set with 6,489 samples. Since there has been no dataset for the evaluation of code intelligence tasks in a semi-supervised setting, we propose to simulate it by extending existing datasets. Specifically, following previous studies [36, 47], we randomly dividing the initial training data into two subsets: labeled training data and an unlabeled dataset, with the ratio of 9:1.

Metrics. For code summarization, we follow previous work [1, 16, 48] and use four popular metrics BLEU-4 [52], ROUGE-L [43], METEOR [2], and CIDEr [60] for evaluation.

3.2.2 Defect Detection. Defect detection aims at identifying the vulnerabilities in the given program, which is crucial to defend a software system from cyberattack [13, 74].

Datasets. In our experiments, we utilize the widely-used Big-Vul dataset created by Fan et al. [13]. This dataset contains 188,636 C/C++ code snippets sourced from more than 300 GitHub projects dating from 2002 to 2019 in Common Vulnerabilities and Exposures (CVE) database. Following previous studies [13], we partition the dataset into training, validation, and test sets with a ratio of 8:1:1. Same with code summarization, we also further construct the labeled training data and unlabeled data by dividing the original training set of Big-Vul with a ratio of 1:9.

Metrics. We follow previous work [41, 74] and evaluate the results by Precision (P), Recall (R), and F1.

3.2.3 Assertion Generation. Assertion Generation is the task of automatically generating meaningful assert statements for unit

Table 1: Experimental results on code summarization. "*" denotes statistical significance in comparison to the base models (i.e., two-sided t-test with p-value< 0.01).

Approach		JCSD				PCSD			
		BLEU-4	ROUGE-L	METEOR	CIDEr	BLEU-4	ROUGE-L	METEOR	CIDEr
CodeBERT	Base model	13.30	26.75	8.10	0.58	17.94	32.35	9.79	0.59
	+HINT$_{(1)}$	14.58*	**29.06***	8.74*	0.69*	18.81*	34.18*	10.52*	0.69*
	+HINT$_{(5)}$	**14.64***	29.00*	**8.87***	**0.71***	**18.86***	**34.25***	**10.87***	**0.72***
CodeT5	Base model	16.67	34.28	11.39	1.05	21.13	40.27	15.69	1.22
	+HINT$_{(1)}$	18.32*	35.49*	**12.36***	1.22*	22.33*	41.42*	**16.31***	1.35*
	+HINT$_{(5)}$	**18.48***	**35.63***	12.29*	**1.24***	**22.55***	**41.67***	16.21*	**1.36***
UniXcoder	Base model	17.16	32.56	11.05	1.11	22.42	35.84	15.38	1.31
	+HINT$_{(1)}$	18.90*	35.16*	12.38*	1.28*	23.77*	41.67*	16.64*	1.48*
	+HINT$_{(5)}$	**19.79***	**35.83***	**13.12***	**1.36***	**23.98***	**41.93***	**16.83***	**1.50***

tests. It can reduce the manual efforts in writing test cases and facilitate faster detection and diagnosis of software failures [46, 72].

Datasets. For assertion generation, we follow previous work [46, 72] and use the ATLAS dataset [67]. It contains 188,154 real-world test assertions obtained from open-source projects in GitHub. The dataset is composed of eight categories of assertions, and each sample in ATLAS is comprised of a focal method and a test method which serve as the context for generating a single assertion for the given test method. We use the original partition of ATLAS and split it into three subsets: training, validation, and test, in an 8:1:1 ratio. The construction of an unlabeled dataset for assertion generation is also the same as the above two tasks. We randomly extract 90% of the training data for the construction of the unlabeled dataset and use the remaining data as the labeled dataset.

Metrics. We follow previous work [49, 72] in this field and use Exact Match (EM), Longest Common Subsequence (LCS), and Edit Distance (ED) as evaluation metrics.

3.3 Baselines

We evaluate the performance of HINT by building it on the top of three popular open-source pre-trained code models, namely CodeBERT [14], CodeT5 [64], and UniXcoder [20]. **CodeBERT** is a representative pre-trained code model that is pre-trained with six programming languages and uses Masked Language Modeling and Replace Token Detection as pre-trained tasks. **CodeT5** is a sequence-to-sequence pre-trained model which involves two code-related pre-training objectives: identifier tagging and masked identifier prediction. It achieves state-of-the-art performance in many sequence generation tasks. **UniXcoder** is a unified cross-modal pre-trained model which incorporates code semantic and syntax information from AST. It is pre-trained with two new pre-training tasks multi-modal contrastive learning and cross-modal generation to learn code fragment representation. These models are all pre-trained on CodeSearchNet [31] and CodeT5 is also pre-trained with C/CSharp code snippets from BigQuery [4].

3.4 Implementation Details

We reproduce the results of all pre-trained models based on the official repositories released by the model authors. In order to facilitate a fair comparison, we ensure that the hyperparameters such as training epochs and learning rates for the models with and without HINT are exactly the same. In our experiments, we set μ and t to 0.5 and 0.4, respectively. The maximum iteration is set to five. To determine the percentage of selected samples, we tune the threshold K in 10, 15, 20, 25, 30, or 35 and select the best results for different datasets. Our rationale for hyperparameter selection is discussed in Section 4.3. When applying our pseudo-labeled data selection methods to the classification task, we conduct Algorithm 1 for each class respectively and balance the class distribution by random down-sampling [5]. All the experiments are conducted on an Ubuntu 20.04 server with an Intel Xeon Platinum 8276 CPU, and 4 Nvidia Tesla A100 GPUs which have 40 GB graphic memory.

4 EXPERIMENTAL RESULTS

4.1 RQ1: Performance Evaluation

In this section, we evaluate the effectiveness of HINT on three code intelligence tasks including code summarization, defect detection, and assertion generation. We present the results of HINT on the first iteration and the best results of HINT on all the five iterations, namely HINT$_{(1)}$ and HINT$_{(5)}$. The results are displayed in Table 1-3. HINT consistently improves three pre-train code models on all the tasks and metrics. In particular, HINT achieves 15.33%, 16.50%, and 8.98% improvements in BLEU-4, F1, and EM over the best pre-trained model UniXcoder on the three datasets, respectively. We detail the results on each task respectively as below.

Code Summarization. As shown in Table 1, HINT can significantly improve the performance of different existing pre-trained code models on all datasets and metrics even with only one iteration. For example, HINT$_{(1)}$ improves the BLEU-4 score of CodeT5 by 9.90% and 5.68% on two datasets, respectively. Meanwhile, compared with the most powerful pre-trained model UniXcoder, HINT$_{(1)}$ can still achieve consistent improvement, e.g., improving UniXcoder on JCSD dataset by 10.14%, 12.04%, 7.99%, and 15.32% with respect to BLEU-4, METEOR, ROUGE-L, and CIDEr, respectively. This indicates that HINT is effective in leveraging the unlabeled data and benefits the strong task-agnostic pre-trained code models in the downstream tasks.

Defect Detection. Table 2 presents the results of defect detection. We can observe consistent improvement on overall performance as in the defect detection task: HINT$_{(5)}$ improves the F1 of

Table 2: Experimental results on defect detection. Statistical significance is not applicable to these metrics [10].

Approach		Precision	Recall	F1
CodeBERT	Base model	29.64	17.63	22.11
	+HINT$_{(1)}$	30.81	21.52	25.34
	+HINT$_{(5)}$	**32.09**	**22.36**	**26.35**
CodeT5	Base model	31.38	20.32	24.66
	+HINT$_{(1)}$	36.79	22.36	27.81
	+HINT$_{(5)}$	**37.66**	**22.36**	**28.06**
UniXcoder	Base model	31.30	17.63	22.55
	+HINT$_{(1)}$	**33.28**	20.96	25.73
	+HINT$_{(5)}$	32.04	**22.26**	**26.27**

Table 3: Experimental results on assertion generation. "*" denotes statistical significance in comparison to the base models (i.e., two-sided t-test with p-value< 0.01).

Approach		EM	LCS	ED
CodeBERT	Base model	31.82	65.99	21.68
	+HINT$_{(1)}$	37.75*	69.46*	**19.05***
	+HINT$_{(5)}$	**38.58***	**69.48***	19.20*
CodeT5	Base model	43.64	72.56	20.30
	+HINT$_{(1)}$	46.53*	74.32*	18.47*
	+HINT$_{(5)}$	**47.66***	**75.22***	**18.17***
UniXcoder	Base model	43.64	72.67	17.82
	+HINT$_{(1)}$	47.13*	74.72*	16.61*
	+HINT$_{(5)}$	**47.56***	**74.76***	**16.21***

three pre-trained models by 19.18%, 13.79%, and 16.50%, respectively. This indicates that HINT can help pre-trained models to capture the patterns of vulnerable code snippets. Besides, by comparing the results of HINT$_{(1)}$ and HINT$_{(5)}$, we can also observe that after multiple iterations, HINT can achieve better performance, e.g., improving HINT$_{(1)}$ by 2.10% F1 in average on UniXcoder.

Assertion Generation. For assertion generation, as shown in Table 3, we can observe that HINT can improve all baseline pre-trained models by a large margin. On average, HINT$_{(1)}$ and HINT$_{(5)}$ improve the EM of these models by 11.09% and 13.14%, respectively. Specifically, on CodeBERT, HINT$_{(5)}$ improves its baseline by 21.24%, 5.29%, and 11.44% in terms of EM, LCS, and ED, respectively. This indicates that the ability to better utilize the unlabeled data of HINT is also beneficial to generate accurate assertion statements.

Answer to RQ1: HINT consistently improves three pre-trained code models on all tasks and metrics, indicating its effectiveness in leveraging unlabeled data for the pre-trained code models.

4.2 RQ2: Ablation Study

In this section, we explore the contribution of the hybrid pseudo-labeled data selection and the noise-tolerant training modules proposed in HINT. We use UniXcoder as the base model since it shows the best performance in the first research question. Besides, considering the time and resource limitation of multiple iterations, we use HINT with one iteration for the following experiments. Due to the page limit, we only present the results on Java in this paper for code summarization, with results for other languages and pre-trained models presented on our GitHub repository [26].

4.2.1 Impact of hybrid pseudo-labeled data selection. We compare HINT with four other data selection methods including *Random selection*, *HINT w/o retrieval-based selection*, *HINT w/o loss-based selection*, and *HINT w/o data selection*. In HINT w/o loss-based selection and HINT w/o retrieval-based selection, we validate the effectiveness of two methods in hybrid pseudo-labeled data selection respectively. In HINT w/o data selection, we remove the whole data selection process and directly use all the generated pseudo-labeled data, which aims at verifying the benefit of data selection. In Random selection, we randomly select a subset from pseudo-labeled data that has the same size as the subset selected by HINT. This is usually used in controlled experiments to eliminate the potential confounding effect of dataset size [56]. The experimental results are presented in Table 4.

Loss-based selection. We conduct this experiment by removing the loss-based selection (Line 4-5, 13-14 in Algorithm 1). From Table 4, we can observe that, without loss-based selection, the performance of HINT decreases consistently on all the tasks. Specifically, removing this component leads to an obvious decrease in defect detection, with the decrease at 19.26%, 11.02%, and 14.42% regarding Precision, Recall, and F1, respectively. This demonstrates the benefits of removing the noisy data by the training loss.

Retrieval-based selection. We conduct this experiment by removing the retrieval-based selection (Line 9-11 in Algorithm 1). As can be seen in Table 4, excluding the retrieval-based selection process leads to a consistent drop in all tasks and metrics. The results demonstrate the effectiveness of involving the retrieval-based strategy for data selection.

Data selection and Random selection. As shown in Table 4, the model suffers from a large degradation after removing the data selection procedure. Specifically, on defect detection, the F1 of using all pseudo-labeled data is only 22.29, much lower than the results of our method, i.e., 25.73. The performance of random selection is even worse. For example, on assertion generation, random selection has a decrease of 3.95%, 1.43%, and 2.95% with respect to EM, LCS, and ED, respectively, indicating the importance of the data selection process in HINT. This also demonstrates that directly using pseudo-labeling cannot achieve promising results on code intelligence tasks.

4.2.2 Impact of noise-tolerant training. In this section, we validate the effectiveness of two components of the noise-tolerant training module, i.e., noise-tolerant loss function and consistency regularization.

Noise-tolerant loss function. We conduct this experiment by removing the noise tolerant loss in Equation 4, i.e., directly using the cross entropy loss. From Table 4, we can observe that removing the noise-tolerant loss results in a performance decrease in the vast majority of cases. For example, on defect detection, HINT without the noise-tolerant loss suffers from a decrease of 4.47% in terms of F1. This shows the importance of using noise-tolerant loss to mitigate the negative impact of errors in pseudo labels on the model performance.

Consistency regularization. We conduct this experiment by removing the noise tolerant loss in Equation 4, i.e., only use the first

Table 4: Ablation study of HINT. Best and second best results are marked in bold and underline respectively.

Approach	Code Summarization				Defect Detection			Assertion Generation		
	BLEU-4	ROUGE-L	METEOR	CIDEr	Precision	Recall	F1	EM	LCS	ED
UniXcoder+HINT	18.90	**35.16**	12.38	1.28	33.28	**20.96**	**25.73**	**47.13**	**74.72**	16.61
Random selection	18.34	34.11	11.70	1.23	34.39	16.14	21.97	45.27	73.65	17.10
-w/o loss-based selection	18.54	34.09	12.34	1.24	26.87	18.65	22.02	45.55	73.96	17.10
-w/o retrieval-based selection	18.71	34.91	12.21	1.26	32.33	19.94	24.67	45.03	73.50	17.16
-w/o data selection	18.27	34.44	12.03	1.21	**34.71**	16.42	22.29	47.03	74.64	16.65
-w/o noise tolerant loss	**18.93**	34.96	12.26	**1.29**	33.02	19.57	24.58	46.64	74.46	**16.56**
-w/o consistency regularization	18.78	35.03	**12.40**	1.27	33.82	19.29	24.57	46.81	74.55	16.70

Table 5: Experimental results on cross-domain scenario. "*" denotes statistical significance in comparison to the base models (i.e., two-sided t-test with p-value< 0.01).

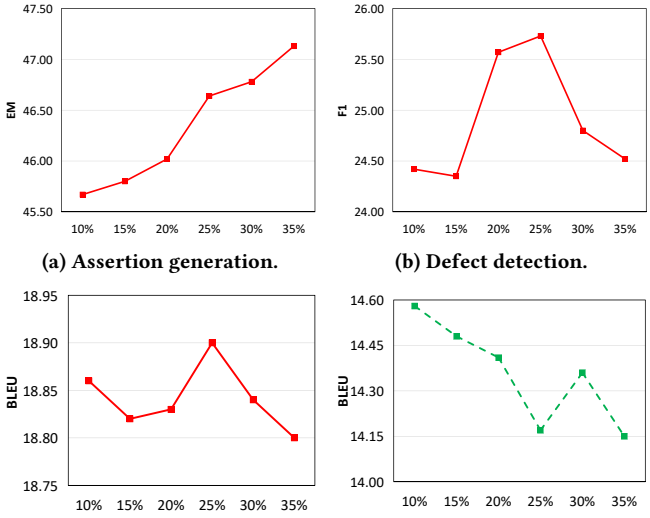
Approach	Python → Java				Java → Python			
	BLEU-4	ROUGE-L	METEOR	CIDEr	BLEU-4	ROUGE-L	METEOR	CIDEr
CodeBERT	9.98	20.01	4.99	0.21	12.65	22.35	7.02	0.25
CodeBERT+HINT	**13.82***	**27.71***	**8.10***	**0.60***	**16.14***	**30.33***	**9.89***	**0.58***
CodeT5	7.75	12.55	7.12	0.29	14.81	30.59	9.66	0.84
CodeT5+HINT	**14.09***	**22.51***	**9.28***	**0.88***	**16.85***	**33.70***	**10.77***	**1.07***
UniXcoder	12.68	26.03	8.33	0.66	13.18	20.94	10.70	0.64
UniXcoder+HINT	**16.33***	**32.22***	**10.54***	**1.03***	**17.26***	**28.28***	**13.14***	**0.97***

term in Equation 4. As can be seen in Table 4, removing the adaptive regularization also leads to a drop in most tasks and metrics. Specifically, removing consistency regularization leads to a decrease of 0.63%, 4.51%, and 0.68% on three tasks regarding BLEU-4, F1, and EM, respectively, indicating the effectiveness of providing reliable training objectives for leveraging the pseudo-labeled data.

Answer to RQ2: All components in hybrid pseudo-labeled data selection module and noise-tolerant training module demonstrate a positive effect on the performance of HINT.

4.3 RQ3: Evaluation on Cross-domain Scenario

In some programming languages, there is often a shortage of training data. For the data-limited scenarios, transfer learning is a popular solution which transfers the knowledge of similar domains with sufficient data to the target domains [42, 61]. In this section, we conduct experiments to study the effectiveness of HINT in cross-domain scenarios, in which the model is trained on the source domain and tested on the target domain with a different programming language. We use the code summarization task for evaluation as it contains two kinds of programming language. Specifically, we first train a model on the Java/Python dataset as the source domain and then evaluate its performance on the test set of the other (Python/Java) dataset as the target domain. As shown in Table 5, HINT can improve the performance of pre-trained code models in the cross-domain scenario by a large margin. Specifically, HINT improves the BLEU-4 score of UniXcoder by 28.79% and 30.96% on Java to Python and Python to Java, respectively, indicating that HINT can effectively utilize the knowledge in those unlabeled data



(a) Assertion generation. (b) Defect detection.

(c) Code summarization (UniXcoder). (d) Code summarization (CodeBERT).

Figure 4: Parameter analysis on threshold K.

by pseudo-labeling. This also shows HINT's ability to enhance pre-trained code models in new programming languages, regardless of any disparities in their data distributions.

Answer to RQ3: HINT can substantially boost the performance of pre-trained code models in cross-domain scenarios where no annotated data exist in the target domain.

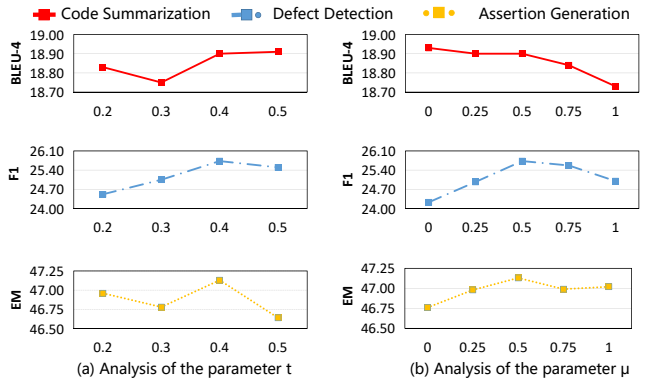


Figure 5: Parameter analysis on t and μ.

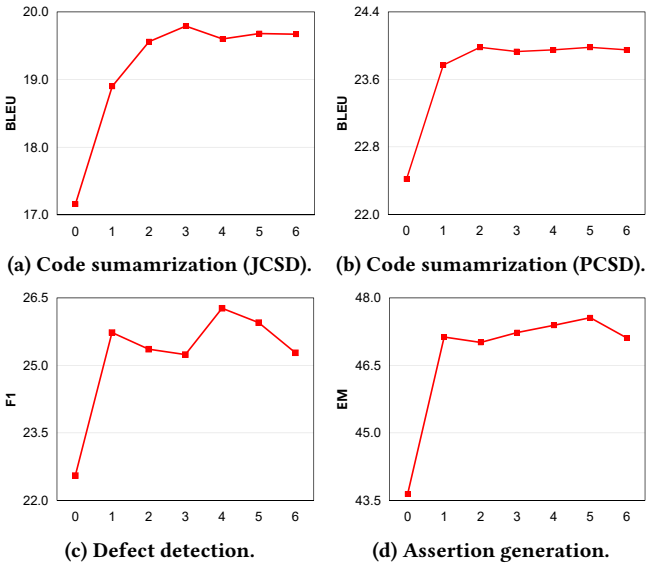


Figure 6: Performance on each Iteration.

4.4 RQ4: Parameter Analysis

In this section, we study the impact of four parameters on the performance of HINT, including the threshold K in loss-based data selection, the edit distance threshold t in retrieval-based data selection, the weight of consistency regularization μ, and the iteration number I. Due to the page limitation, we only present the results of UniXcoder and JCSD dataset for t and μ on code summarization, with results for other languages and pre-trained models presented on our GitHub repository [26].

The threshold K in loss-based data selection. We conduct experiments to evaluate how HINT performs under different thresholds, i.e., 10%, 15%, 20%, 25%, 30%, and 35%. The larger the K is set to, the more the pseudo labeled samples will be selected. As shown in Figure 4, the model performance shows a similar trend along with the increase of K on all pre-trained code models and tasks. HINT first increases and achieves its peak, and then sharply descends with a larger K. Larger K has the risk of involving more noisy data while smaller K might be too strict and filter many high-quality samples. Besides, we can find that the optimal value of K for different models and tasks varies a lot. For example, on code summarization, UniXcoder achieves the best performance when K is set to 25%, while the optimal value for CodeBERT is 10%. We suggest that it is because the capability of the base model on each task is different. Specifically, the performance of UniXcoder on code summarization is very strong, i.e. achieving 17.16 BLEU-4 on the Java dataset, while for CodeBERT, its performance on the Java dataset is only 13.30. The poorer the performance of the based model is, the lower the quality of pseudo-labeled data is. Therefore, when applying HINT on different pre-trained code models, a relatively larger K can be used on a stronger base model and vice versa.

The edit distance threshold t. We study the effect of t, as introduced in Section 2.3, by varying it from 0.2 to 0.5. As shown in Figure 5 (a), for both defect detection and assertion generation, HINT achieves the best performance when t is set to 0.4. Larger or lower values do not give better results. On code summarization, setting t to 0.5 only performs slightly better than 0.4. This indicates that setting t to 0.4 is more appropriate for HINT. Thus, we set t to 0.4 in this work.

The consistency regularization weight μ. To study the impact of μ in HINT, we vary it from 0 to 1 and show the results in Figure 5 (b). Larger μ tends to give a stronger regularization to the model. For both defect detection and assertion generation, HINT achieves the best performance when μ is set to 0.5. However, on code summarization, increasing μ leads to a decrease in performance. Therefore, we set μ to 0.5 to enable HINT to produce relatively better results on different tasks.

The iteration number I. We evaluate the performance of HINT on different iterations by setting the maximum iteration to six, and present the results in Figure 6. Iteration 0 represents the baseline results that do not use HINT. From the results, we can observe that HINT can get better results with the growth of iterations and achieves the peak at around the fifth iteration, indicating that HINT can achieve self-improvement by leveraging the unlabeled data.

Answer to RQ4: Different settings of hyperparameters can influence the performance of HINT on different tasks. Our hyperparameter settings achieve relatively better results.

5 DISCUSSION

5.1 What Makes HINT Work?

5.1.1 HINT can better utilize the unlabeled data for downstream tasks. To better understand how pseudo-labeling benefits pre-trained code models, we give two examples in Figure 7 and Figure 8. The case in Figure 7 shows a Java code snippet with summaries generated by UniXcoder and UniXcoder+HINT. From the example, we can see that the summary generated by UniXcoder only contains a simple description without a detailed introduction to the parameters. HINT can avoid this problem and give a more precise prediction since it can learn from more ⟨unlabeled data, pseudo label⟩ pairs that have a similar summary pattern. We also present another case in the assertion generation task in Figure 8. The assertion statement generated by UniXcoder mistakenly predicts the assertion type as "*assertionTrue*" since it does not learn the meaning of "*empty*" well. However, since UniXcoder+HINT uses the code snippet in Figure 8 as training data which has the same assertion

types as this test sample, it can correctly predict the assertion type in Figure 8.

5.1.2 HINT can select pseudo-labeled data with higher quality. Another advantage of HINT comes from our data selection process. HINT can select high-quality pseudo labels for model training. As shown in Figure 1 and 2, HINT identifies low-quality pseudo-labeled data by employing both the implicit loss-based selection and explicit retrieval-based selection. To further validate this, we calculate the edit distance of the pseudo labels generated by UniXcoder to the ground truth labels of the unlabeled dataset, and use the average distance on the whole selected dataset to measure the quality of our selected dataset. Specifically, on JCSD the average edit distance of all pseudo labels without filtering is 53.70, which is much higher than the dataset selected by HINT, i.e., 37.16. The results on assertion generation are the same. HINT achieves an average edit distance of 5.34 while the average edit distance of all pseudo labels is 17.86. This further shows that HINT can filter noisy data and select pseudo-labeled data with higher quality for model training.

5.2 Limitation of HINT

To gain a deeper understanding of HINT's behavior and limitations, we further investigate cases where HINT fails to make accurate predictions and conclude two possible limitations of HINT.

The first limitation pertains to HINT's inability to introduce additional knowledge and rectify factual knowledge errors. From the example in the above Figure 9, UniXcoder misinterprets the term "*bucket_acl*" as the name of a bucket and fails to rectify this misunderstanding even after additional training on pseudo-labeled data. This shows that without external feedback HINT is hard to identify and rectify the problem on factual knowledge, which also aligns with recent findings on the limited self-correction ability of large language models [29]. To potentially alleviate this limitation, integrating factual knowledge into pre-trained code models via the interaction with a knowledge base or search engine could be further studied.

The second limitation of HINT is the reliance on the capacity of the base model. HINT aims at autonomously synthesizing more labeled data for model training. However, when the base model lacks sufficient capacity, the benefits of additional training data are diminished. As depicted in the Figure 10, despite the presence of training sample in the pseudo-labeled data illustrating the usage of "*assertEquals*", UniXcoder still fails to learn this and erroneously generates "*assertThat*" for the given function. We attribute this limitation to the inherent constraints of the model's capacity and believe that it could be mitigated by using more advanced pre-trained code models.

5.3 Threats to Validity

We identify four main threats to validity of our study:

(1) **The selection of code intelligence tasks.** We evaluate HINT on three commonly-used code intelligence tasks: code summarization, defect detection, and assertion generation. We aim to expand the validation of HINT in the future by testing it on more code intelligence tasks.

(2) **The selection of pre-trained code models.** In this paper, we select three popular open-source pre-trained code models CodeBERT, CodeT5, and UniXcoder for evaluation. These models are all representative and have shown state-of-the-art performance on benchmarks [20, 64]. Recent studies propose pre-trained models with much larger sizes such as ChatGPT [6] and GPT-4 [51] which also show impressive programming ability. However, since the weight of these models is not publicly available, we cannot evaluate our framework on those large language models. Besides, our framework is flexible and easy to be applied to different pre-trained code models.

(3) **The selection of languages.** The datasets that we choose in experiments only contain two kinds of languages, i.e., Java and Python. They are both popular languages. Additionally, our method is language-agnostic and can be easily adapted to other programming languages.

```
//Code from the test set:
public Exchange(final Request request, final Origin origin){
    this.currentRequest = request;
    this.origin = origin;
    this.timestamp = System.currentTimeMillis();
}

Summary generated by Unixcoder:
constructs a new exchange object.
Summary generated by Unixcoder+HINT:
constructs an exchange with the given request and origin.
Ground truth summary:
creates a new exchange with the specified request and origin.

Code from the Unlabeled dataset:
public ScannerException(ErrorMessages message, int line){
    this(null, ErrorMessages.get(message), message, line, -_NUM);
}

Pseudo-labeled summary:
creates new scannerexception with message and line number.
```

Figure 7: Case study on the code summarization task. The green texts highlight the similar part between the prediction of UniXcoder+HINT and pseudo-labeled summary.

```
//Code from the test set:
hasSubscriptionId_emptyID(){
    when(fixture.getSubscriptionId()).thenReturn("");
    "<AssertPlaceHolder>";
}
hasSubscriptionId(){
    return
((getSubscriptionId())!=null)&&(!(getSubscriptionId().isEmpty()));
}

Assertion generated by Unixcoder:
org.junit.Assert.assertTrue(fixture.hasSubscriptionId());
Assertion generated by Unixcoder+HINT:
org.junit.Assert.assertFalse(fixture.hasSubscriptionId());
Ground truth assertion:
org.junit.Assert.assertFalse(fixture.hasSubscriptionId());

Code from the Unlabeled dataset:
hasNext_on_an_empty_collection__returns_false(){
    com.artemis.utils.IntBagIterator intBagIterator = new
com.artemis.utils.IntBagIterator(new com.artemis.utils.IntBag(99));
    "<AssertPlaceHolder>";
}
hasNext(){
    return  (this.cursor)<(size);
}

Pseudo-labeled assertion:
org.junit.Assert.assertFalse(intBagIterator.hasNext());
```

Figure 8: Case study on the assertion generation task. The red and green texts highlight the difference in predictions made by UniXcoder and UniXcoder+HINT.

```
# Code from the test set:
def print_bucket_acl_for_user(bucket_name, user_email):
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    bucket.acl.reload()
    roles = bucket.acl.user(user_email).get_roles()
    print roles
```

Summary generated by Unixcoder:
print the current users name for the specified user.
Summary generated by Unixcoder+HINT:
prints the name for the specified bucket and user.
Ground truth summary:
prints out a buckets access control list for a given user.

Figure 9: Error case on the code summarization task.

```
Focal-test from test set:
testIdAccessor(){
    java.lang.Long id = 3L; instance.setId(id);
    "<AssertPlaceHolder>";}
getId(){
    return id;
}
```

Assertion generated by Unixcoder+HINT:
org.junit.Assert.assertThat(id, instance.getId());
Ground truth assertion:
org.junit.Assert.assertEquals(id, instance.getId());

```
Focal-test from Unlable dataset:
testGetName(){
    java.lang.String id = "id"; togglePanelItem.setId(id);
    "<AssertPlaceHolder>";}
getId(){
    return id;
}
```

Generated pseudo label:
org.junit.Assert.assertEquals(id, togglePanelItem.getId());

Figure 10: Error case on the assertion generation task.

(4) **The limitation of selected metrics.** We evaluate HINT using a variety of commonly used metrics for different tasks. However, these metrics are mainly used for evaluating accuracy and may not reflect other evaluation aspects such as the diversity of generated code summaries. In the future, we plan to conduct human studies to provide a more comprehensive evaluation.

6 RELATED WORK

6.1 Code Intelligence

In this section, we introduce related neural code models in three tasks that are covered in our work, including both non-pre-trained code models and pre-trained code models,

6.1.1 Non-pre-trained code models. Iyer et al. [32] formulate code summarization as a neural machine translation (NMT) problem and propose CODE-NN to translate code snippets to code summaries. For better utilizing code structure information, many works [15, 39] in code summarization also incorporate code-related graphs and GNN to boost performance. Recent studies [58, 69] further incorporate various code structure information into the Transformer model and achieve promising performance. As for vulnerability detection, many deep learning-based methods [41, 74] are proposed. For example, Devign [74] is proposed to learn the various vulnerability characteristics with a composite code property graph and graph neural network. IVDetect [41] uses the program dependency graph and feature attention GCN to detect vulnerabilities in the code. In assertion generation, recent studies adopt the T5 transformer model and achieve promising results [45, 46]. Yu et al. [72] further involve information retrieval to generate more accurate assertion statements.

6.1.2 Pre-trained code models. Recently, a series of pre-trained code models [14, 21, 64] are proposed and achieve state-of-the-art performance on various code intelligence tasks such as code summarization and defect detection. CodeBERT [14] is a pioneer work that is pre-trained with six programming languages and uses Masked Language Modeling and Replace Token Detection as pre-trained tasks. CodeT5 [64] is a sequence-to-sequence pre-trained model which involves two code-related pre-training objectives: identifier tagging and masked identifier prediction. UniXcoder [20] is a unified cross-modal pre-trained model which incorporates code semantic and syntax information from AST.

6.2 Pseudo-labeling

Pseudo-labeling is one of the most widely-used semi-supervised learning methods. It has been applied to different kinds of tasks such as image classification [40, 70], machine translation [24, 34], and dialog systems [47]. To further boost the performance of self-training in sequence generation tasks, He et al. [24] and Mi et al. [47] explore the data augmentation technique and use random noise or gradient-based data augmentation to improve the generalization of the student model. Another line of work [8, 34, 66] focus on the data selection procedure and propose to select high-quality pseudo labeled data based on the uncertainty or the model confidence, respectively. However, these methods mainly filter the pseudo-labeled data only with training loss and do not take the noisy data problem into consideration. Different from them, we propose a hybrid data selection method with the training loss and a retrieval-based method based on the code reuse. Additionally, we also propose a noise-tolerant training module to further mitigate the influence of noise on model performance.

7 CONCLUSION

In this paper, we investigate leveraging large-scale unlabeled datasets for effectively tuning pre-trained code models by pseudo-labeling. We propose a method called HINT which consists of two main components, the hybrid pseudo-labeled data selection module and the noise-tolerant training module. Extensive experiments on three code intelligence tasks show that HINT can be built on a variety of pre-trained models and provide complementary benefits for them. Our replication package including our source code, experimental data, and detailed experiment results is at [26].

ACKNOWLEDGMENT

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). The work was also supported by Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), Shenzhen International Cooperation Project (No. GJHZ20220913143 008015), and the Major Key Project of PCL (Grant No.PCL2022A03).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*. Association for Computational Linguistics, 4998–5007.

[2] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*. Association for Computational Linguistics, 65–72.

[3] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[4] BigQuery. 2022. BigQuery. https://console.cloud.google.com/marketplace/details/github/github-repos.

[5] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[6] ChatGPT. 2022. ChatGPT. https://openai.com/blog/chatgpt.

[7] Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *J. Syst. Softw.* 82, 6 (2009), 981–992.

[8] Yiming Chen, Yan Zhang, Chen Zhang, Grandee Lee, Ran Cheng, and Haizhou Li. 2021. Revisiting Self-training for Few-shot Learning of Language Model. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 9125–9135.

[9] Matteo Ciniselli, Luca Pascarella, Emad Aghajani, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. Source Code Recommender Systems: The Practitioners' Perspective. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2161–2172.

[10] William Jay Conover. 1999. *Practical nonparametric statistics*. Vol. 350. john wiley & sons.

[11] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 121–133.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186.

[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. ACM, 508–512.

[14] 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 *Findings of the Association for Computational Linguistics: EMNLP 2020 (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.

[15] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[16] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael R. Lyu. 2023. Code Structure-Guided Transformer for Source Code Summarization. *ACM Trans. Softw. Eng. Methodol.* 32, 1 (2023), 23:1–23:32.

[17] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 761–773.

[18] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 30–42.

[19] Golara Garousi, Vahid Garousi-Yusifoglu, Günther Ruhe, Junji Zhi, Mahmood Moussavi, and Brian Smith. 2015. Usage and usefulness of technical software documentation: An industrial case study. *Inf. Softw. Technol.* 57 (2015), 664–682.

[20] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 7212–7225.

[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021*. OpenReview.net.

[22] Bo Han, Quanming Yao, Xingrui Yu, Gang Niu, Miao Xu, Weihua Hu, Ivor W. Tsang, and Masashi Sugiyama. 2018. Co-teaching: Robust training of deep neural networks with extremely noisy labels. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 8536–8546.

[23] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[24] Junxian He, Jiatao Gu, Jiajun Shen, and Marc'Aurelio Ranzato. 2020. Revisiting Self-Training for Neural Sequence Generation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[25] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2022. Semantic Robustness of Models of Source Code. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 526–537.

[26] HINT. 2023. Replication package of HINT. https://github.com/shuzhenggao/HINT.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 2269–2275.

[28] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' Expectations on Automated Code Comment Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1693–1705.

[29] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large Language Models Cannot Self-Correct Reasoning Yet. *CoRR* abs/2310.01798 (2023).

[30] Jinchi Huang, Lie Qu, Rongfei Jia, and Binqiang Zhao. 2019. O2U-Net: A Simple Noisy Label Detection Approach for Deep Neural Networks. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 3325–3333.

[31] 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).

[32] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016*. The Association for Computer Linguistics.

[33] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 5954–5971.

[34] Wenxiang Jiao, Xing Wang, Zhaopeng Tu, Shuming Shi, Michael R. Lyu, and Irwin King. 2021. Self-Training Sampling with Monolingual Data Uncertainty for Neural Machine Translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*. Association for Computational Linguistics, 2840–2850.

[35] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28, 7 (2002), 654–670.

[36] Pei Ke, Haozhe Ji, Zhenyu Yang, Yi Huang, Junlan Feng, Xiaoyan Zhu, and Minlie Huang. 2022. Curriculum-Based Self-Training Makes Better Few-Shot Learners for Data-to-Text Generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. ijcai.org, 4178–4184.

[37] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. ACM, 187–196.

[38] Solomon Kullback. 1997. *Information theory and statistics*. Courier Corporation.

[39] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 184–195.

[40] Dong-Hyun Lee et al. 2013. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on challenges in representation learning, ICML*, Vol. 3. 896.

[41] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 292–303.

[42] Zhiming Li, Xiaofei Xie, Haoliang Li, Zhengzi Xu, Yi Li, and Yang Liu. 2022. Cross-lingual transfer learning for statistical type inference. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 239–250.

[43] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81.

[44] Christopher D Manning. 2009. *An introduction to information retrieval*. Cambridge university press.

[45] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering* (2022).

[46] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[47] Fei Mi, Wanhao Zhou, Lingjing Kong, Fengyu Cai, Minlie Huang, and Boi Faltings. 2021. Self-training Improves Pre-training for Few-shot Learning in Task-oriented Dialog Systems. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 1887–1898.

[48] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2022. Automatic Comment Generation via Multi-Pass Deliberation. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 14:1–14:12.

[49] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. (2023).

[50] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2136–2148.

[51] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).

[52] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318.

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

[54] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the Evaluation of Neural Code Summarization. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1597–1608.

[55] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. CoCoSoDa: Effective Contrastive Learning for Code Search. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2198–2210.

[56] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 107–119.

[57] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1609–1620.

[58] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zheling Zhu, and Bin Luo. 2022. AST-Trans: Code Summarization with Efficient Tree-Structured Attention. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 150–162.

[59] Amazon Mechanical Turk. 2023. Amazon Mechanical Turk. https://www.mturk.com/.

[60] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. CIDEr: Consensus-based image description evaluation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 4566–4575.

[61] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. 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, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 382–394.

[62] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 5–16.

[63] Yisen Wang, Xingjun Ma, Zaiyi Chen, Yuan Luo, Jinfeng Yi, and James Bailey. 2019. Symmetric Cross Entropy for Robust Learning With Noisy Labels. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 322–330.

[64] 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 the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. Association for Computational Linguistics, 8696–8708.

[65] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2021. Generalizing from a Few Examples: A Survey on Few-shot Learning. *ACM Comput. Surv.* 53, 3 (2021), 63:1–63:34.

[66] Zhongyuan Wang, Yixuan Wang, Shaolei Wang, and Wanxiang Che. 2022. Adaptive Unsupervised Self-training for Disfluency Detection. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*. International Committee on Computational Linguistics, 7209–7218.

[67] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[68] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-based Neural Comment Generation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 349–360.

[69] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*. Association for Computational Linguistics, 1078–1090.

[70] Qizhe Xie, Minh-Thang Luong, Eduard H. Hovy, and Quoc V. Le. 2020. Self-Training With Noisy Student Improves ImageNet Classification. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 10684–10695.

[71] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Association for Computational Linguistics, 440–450.

[72] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated Assertion Generation via Information Retrieval and Its Integration with Deep learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 163–174.

[73] Zhilu Zhang and Mert R. Sabuncu. 2018. Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 8792–8802.

[74] 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 *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*. 10197–10207.