An Empirical Study on Noisy Label Learning for Program Understanding

Wenhan Wang
Nanyang Technological University
Singapore
wwhjacob@hotmail.com

Yanzhou Li
Nanyang Technological University
Singapore
yanzhou001@e.ntu.edu.sg

Anran Li*
Nanyang Technological University
Singapore
anran.li@ntu.edu.sg

Jian Zhang
Nanyang Technological University
Singapore
jian_zhang@ntu.edu.sg

Wei Ma
Nanyang Technological University
Singapore
ma_wei@ntu.edu.sg

Yang Liu*
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

ABSTRACT

Recently, deep learning models have been widely applied in program understanding tasks, and these models achieve state-of-the-art results on many benchmark datasets. A major challenge of deep learning for program understanding is that the effectiveness of these approaches depends on the quality of their datasets, and these datasets often contain noisy data samples. A typical kind of noise in program understanding datasets is label noise, which means that the target outputs for some inputs are incorrect.

Researchers have proposed various approaches to alleviate the negative impact of noisy labels, and formed a new research topic: noisy label learning (NLL). In this paper, we conduct an empirical study on the effectiveness of noisy label learning on deep learning for program understanding datasets. We evaluate various NLL approaches and deep learning models on three tasks: program classification, vulnerability detection, and code summarization. From the evaluation results, we come to the following findings: 1) small trained-from-scratch models are prone to label noises in program understanding, while large pre-trained models are highly robust against them. 2) NLL approaches significantly improve the program classification accuracies for small models on noisy training sets, but they only slightly benefit large pre-trained models in classification accuracies. 3) NLL can effectively detect synthetic noises in program understanding, but struggle in detecting real-world noises. We believe our findings can provide insights on the abilities of NLL in program understanding, and shed light on future works in tackling noises in software engineering datasets. We have released our code at https://github.com/jacobwwh/noise_SE.

KEYWORDS

program understanding, deep learning, noisy label learning

---

*Corresponding authors

1 INTRODUCTION

With the rapid development of artificial intelligence, more and more deep learning models are applied in program understanding. Deep learning-based program understanding is capable of solving a wide range of tasks on program source code, from classification tasks such as program classification [28] and vulnerability prediction [49], to generation tasks such as code summarization [17].

Many deep learning models for program understanding tasks are usually trained in a supervised learning paradigm, *e.g.*, either from scratch or fine-tuned from a pre-trained model trained on a large training dataset, and then evaluated on a test dataset. It is necessary to acquire high-quality datasets for both the training phase and the testing phase [19, 22]. On the one hand, noisy training data can lead to severe model accuracy deterioration. On the other hand, noisy testing data cannot accurately evaluate the capability of the trained model. However, acquiring high-quality and noise-free datasets for program understanding tasks is extremely challenging [25]. For example, in some software engineering applications, such as code summarization and code search, researchers collect large-scale datasets from open-source platforms, such as GitHub and StackOverflow. Those collected datasets are often too large to be thoroughly inspected by human experts, making it difficult to filter out noises. Most noises in supervised learning tasks can be treated as label noises where input samples do not match their target labels. Label noises widely exist in program understanding tasks with different input-target data modalities. For example, in classification tasks, it refers to the samples with incorrect class labels, while in generation tasks, *e.g.*, code summarization, it refers to the inputs, *i.e.*, code snippets, with inconsistent semantic to target outputs, *i.e.*, natural language summary. Furthermore, the proportion of noisy labeled data in program understanding datasets can be very high: *e.g.*, in some vulnerability detection datasets, up to 70% of the vulnerability samples could be mislabeled [8].

To overcome the negative impact of noisy labels on deep learning, researchers proposed a group of noisy label learning (NLL)

approaches. Noisy label learning, or learning from label noises, has become an active field in machine learning, which aims to improve deep learning models' robustness against label noise and detect mislabeled samples from the datasets [36]. Recently, NLL has attracted the attention of software engineering researchers [7, 10, 25]. Although these works claim to successfully apply NLL to software engineering, their evaluations still have the following limitations:

- All of these works only consider classification tasks (e.g. program classification, defect prediction), while generation tasks, e.g., code summarization, have not been investigated. Actually, generation tasks have long been neglected by the NLL research community.
- Deep learning models for program understanding can be categorized into two types: trained-from-scratch models with small parameter size (we can call them "small models" in brief), and large models pre-trained on massive program language/natural language corpus (we call them "large models"). Many of the NLL works mainly adopt small models, while large models have already dominated many program understanding tasks [27].
- Most of these works focus on tackling training data noise [10, 25], because they often assume that it is not difficult to build a small-scale test set with no noisy labels. However, in many program understanding tasks, e.g., vulnerability detection, obtaining accurate labels is difficult and time-consuming even for domain experts. Even though the test sets are usually much smaller than training sets, the time and manpower costs for manually annotating these datasets are still too high for researchers.

As previous works of noisy label learning for software engineering have the above limitations, a systematic study of NLL in software engineering is urgently needed. In this paper, we aim to focus on program understanding tasks, investigate how label noises affect these tasks, and how NLL can improve them. We first study the impact of different types of noises on different models for program understanding. To achieve this, we inject different synthetic noises into a noise-free program classification dataset, because in this way can we control the rate and pattern of the label noises. We further study the performance of NLL in detecting noisy samples and improving performances on clean test set for program classification. The performances on synthetic noises may not accurately reflect the situation in real-world scenarios, so we take a further step to study NLL on real-world noises in two program understanding tasks: vulnerability detection and code summarization.

We aim to answer the following research questions:

- **RQ1:** How do different types of synthetic label noises in program classification affect the performance of deep learning models when NLL is not introduced?
- **RQ2:** How do existing NLL approaches perform on different synthetic noises in program classification?
- **RQ3:** How do NLL approaches perform on program understanding tasks with real-world noises?

To answer these questions, we conduct experiments on 5 NLL approaches with different deep learning models frequently used for software engineering. These models include both small trained-from-scratch models and state-of-the-art large pre-trained models. Our study covers datasets in three different tasks: program classification, vulnerability detection, and code summarization. In summary, our contributions are as follows:

(1) We are the first to conduct a comprehensive empirical study on noisy label learning in program understanding for both classification and generation tasks.
(2) We evaluate NLL approaches for program understanding in both detecting noisy-labeled data and improving the performance of downstream tasks. Our study unveils the strengths and weaknesses of existing NLL approaches in program understanding, and provides suggestions for future exploration in this field.

2 RELATED WORK

2.1 Learning from Noisy Labeled Data

The importance of data quality has long been noticed by artificial intelligence researchers. In supervised learning, label noise in the training data can impair the performance of deep learning models. Therefore, researchers have proposed various approaches to tackle this issue. Table 1 shows a taxonomy of existing NLL approaches. These approaches can be categorized into the following groups:

1) **Gradient-based approaches:** Some methods tried to improve the robustness of neural networks against label noises by manipulating the gradient. For example, Koh et al. [20] proposed Influence Function (IF), which measures the contribution of each training sample by its influence on the gradients of test data. Although their original motivation is to explain the prediction results of deep learning models, researchers also use IF for detecting mislabeled data [23, 24]. For a training set with mislabeled samples, the samples with high influence scores are more likely to be noisy. Pruthi et al. [32] proposed a similar approach TracIn, which outperforms IF in detecting mislabeled data for image classification datasets.

2) **Meta-learning:** Some other approaches use meta-learning to improve robustness against noise. These approaches aim to guide the update of model parameters with a small, noise-free dataset [21]. Shu et al. [35] proposed Meta-Weight-Net, which uses a small neural network (meta-net) to learn the weight of each training sample. Both the main model and the meta net are trained on a metaset that is much smaller than the original training set. If the metaset is clean and class-balanced, then the model learned with Meta-Weight-Net is robust to both label noise and class imbalance. Zheng et al. [48] further proposed Meta Label Correction (MLC), which aims to identify mislabeled data and predict their correct class labels. The main obstacle preventing meta-learning-based approaches from wide application is that obtaining a noise-free metaset is difficult or intractable for many tasks.

3) **Sample selection during training:** Some approaches aim to directly select possible clean data samples from datasets and remove the noisy samples during training [22]. Han et al. [14] proposed Co-teaching, which uses two identical neural networks to dynamically select possibly clean samples during training. Yu et al. [44] proposed Co-teaching+, which further improves Co-teaching by focusing on disagreement samples between two networks. Li

Table 1: A taxonomy for existing NLL approaches. These approaches differ in the following ways: whether they require noise-free examples (column Clean) and whether they require training on noisy datasets (column Train).

Category	Approach	Clean	Train
Gradient-based	IF [20]	✗	✓
	TracIn [32]	✗	✓
Meta-learning	Meta-Weight-Net [35]	✓	✓
	MLC [48]	✓	✓
Sample selection during training	Co-teaching [14]	✗	✓
	Co-teaching+ [44]	✗	✓
	RobustTrainer [25]	✗	✓
Sample selection (separate)	Confident Learning [30]	✗	✓
	Simifeat [50]	✗	✗

et al. [25] proposed RobustTrainer, which is the first approach for learning on software engineering data with label noises.

4) **Sample selection (separate training and detection):** Another group of works focuses on separating noisy sample detection with training on noisy data. For instance, Northcutt et al. [30] proposed Confident Learning, which utilizes the class probabilities predicted by a trained model to identify possible mislabeled samples. Zhu et al. [50] proposed Simifeat, which can detect noisy samples with a pre-trained model. Instead of using class probabilities, Simifeat employs the features generated by the feature extractor network to identify noisy samples.

2.2 Label Noise in Software Engineering

Researchers in software engineering have recognized the negative impact of label noises, and have proposed different approaches to eliminate the noisy-labeled samples in software engineering datasets. Typically, these approaches are specific to particular tasks or datasets and cannot be generalized to other domains.

The investigation of label noises in software engineering datasets emerges from simple binary classification tasks, such as bug report classification and defect prediction. Herzig et al. [15] manually checked over 7,000 issue reports from existing bug databases and found that more than 40% of them are mislabeled (e.g., an issue report of "Refactoring" is misclassified to "Bug"). Yatish et al. [43] compared the labels in defect datasets labeled by heuristic rules with software developing teams, and find out that more than 50% of defective code snippets are mislabeled by heuristic rules. These works tried to prune noisy labels in software engineering datasets by human experts, so they highly depend on domain experts and cannot be applied to large-scale datasets.

For large-scale "big code" [3] datasets (e.g., large datasets for code summarization, code search), researchers have proposed several automatic approaches to improve their dataset quality by removing noises. For example, Shi et al. [34] proposed an approach for improving the quality of code summarization datasets. The authors identified several typical types of low-quality code summarization samples and developed rule-based approaches to detect them. The detected low-quality samples are either removed or refined for the summarization task. Sun et al. [38] proposed a learning-based approach for filtering high-quality code-natural language pairs for
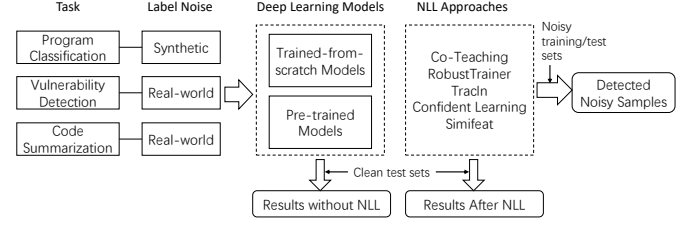

Figure 1: An overview of our study.

code search. The filtering of high-quality data was carried out in two steps: a rule-based syntactic filter and a semantic filter based on a variational autoencoder (VAE). However, this approach only focuses on the quality of natural language queries (whether the query is similar to human-written queries in StackOverflow), but does not consider the semantical relatedness between source code and natural language texts.

Recently, researchers have attempted to investigate the effectiveness of task-agnostic NLL methods on software engineering tasks. Dau et al. [10] investigated the performance of influence function-based approaches on detecting label noise in program classification datasets. The authors introduced random label noise into the datasets and evaluated on two code representation models: ASTNN[46] and CodeBERT[11]. The results showed that for both models, influence-based methods achieved high accuracy in detecting noisy-labeled training samples. However, the authors did not evaluate the performance gain after applying influence-based approaches for the classification task. Li et al. [25] proposed RobustTrainer and evaluated its performance on two tasks: bug report classification and defect prediction. Although RobustTrainer successfully improved the classification metrics on these two tasks, the datasets used by the authors were small, containing only hundreds or thousands of samples, and the deep learning models used were basic (simple MLP and LSTM). Thus, it is still questionable whether RobustTrainer can be applied to larger datasets and more complex models. Croft et al. [7] applied noisy label learning approaches for detecting security vulnerabilities. Their approach is based on existing NLL methods (e.g., confident learning) and an assumption for the vulnerability datasets: vulnerable code snippets may be labeled as clean, but clean samples are unlikely to be labeled as vulnerable. Due to the size of their datasets, they only use NLL with deep learning in detecting noisy samples and did not use deep learning in the vulnerability detection task.

3 RESEARCH METHOD

The overview of our study is shown in Fig. 1. Our study aims to investigate: 1) the impact of label noises in different deep learning-based program understanding models, and 2) the abilities of existing NLL approaches in improving model robustness (improving performances on clean test sets) and detecting noisy-labeled samples in different program understanding tasks.

3.1 Preliminaries

When using deep learning to solve program understanding tasks, we first abstract the task into a mapping task from input space $\mathcal{X}$

Table 2: A summary of the program understanding tasks and datasets in this paper. Dataset sizes are shown in the number of training/validation/test samples.

Task	Style	Size	Noise
Program classification	classification	45,000/15,000/15,000	synthetic
Vulnerability detection	classification	21,854/2,732/2,732	real-world
Code summarization	generation	69,708/8,714/8,714	real-world

to output space $\mathcal{Y}$. In most program understanding tasks, $\mathcal{X}$ is the space of program source code. For classification tasks, $\mathcal{Y}$ is the set of class labels; for summarization tasks, $\mathcal{Y}$ is the space of natural language summaries. We train a deep learning model $f(x \in \mathcal{X}) = y \in \mathcal{Y}$ on the training set $\mathcal{D}_{train}$ to solve the software engineering task. We perform early stop during training by observing the results on the valid set $\mathcal{D}_{valid}$. The trained model is further evaluated on the test set $\mathcal{D}_{test}$.

In a mislabeled data sample $d = \{x, y, \hat{y}\}$, y is the ground truth label, and $\hat{y}$ is the noisy label from the dataset. Notice that in some tasks, the ground truth label y is absent. For example, in the code summarization task, it is nearly impossible to assign a "perfectly correct" summary for a code snippet. In the traditional settings of noisy label learning, only $\mathcal{D}_{train}$ is corrupted with label noises, while $\mathcal{D}_{valid}$ and $\mathcal{D}_{test}$ are clean. However, in some software engineering datasets, $\mathcal{D}_{valid}$ and $\mathcal{D}_{test}$ can also be noisy.

3.2 Tasks and Datasets

We choose three different program understanding tasks: program classification, vulnerability detection, and code summarization to evaluate the performance of NLL approaches. The choice of these three tasks allows us to evaluate NLL under both classification-style and generation-style tasks, and under both synthetic and real-world noises. The summary of the datasets in this study is shown in Table 2

3.2.1 Program Classification. In the program classification task, given a set of programs, we need to classify them into different categories based on their functionalities. For this task, we adopt the CodeNet [33] dataset. CodeNet is collected from two online judging platforms, where programmers are asked to solve various programming questions. For the classification task, each programming question can be seen as a class type. The CodeNet dataset contains four program classification benchmarks. In these four benchmarks, we choose the Java250 dataset for our experiments. Java250 dataset contains 75,000 Java code snippets in 250 classes and is split in 3:1:1 for train/validation/test sets. The creators of CodeNet have cleansed their datasets to ensure that each code snippet exactly matches its class label, so this dataset can be seen as noise-free.

To investigate the impact of label noises and the performance of NLL approaches, we inject the following types of synthetic label noises into the Java250 dataset:

- **Random label noises**: we randomly select $k\%$ samples and randomly change their label y to a different class $\hat{y} \neq y$.
- **Flip label noises**: in some realistic case scenarios, samples in class y are more likely to be mislabeled into a similar class

$\hat{y} = g(y)$ (g is a mapping function). We use a simple mapping function to imitate these cases: we randomly choose $k\%$ samples and change their labels from y to $(y + 1) \mod N$ (N is the total number of classes in the program classification dataset). When adopting flip label noises, the noise rate must be lower than 50% because otherwise, the model will recognize the noisy class as the "true" class label.

3.2.2 Vulnerability Detection. In vulnerability detection, we use deep learning models to classify code snippets as vulnerable or non-vulnerable, so this task can be seen as a binary classification task. Previous studies in vulnerability detection datasets recognized that label noises are prevalent in vulnerability datasets, especially for non-vulnerable labels [8, 9, 29]. For this task, we adopt the Devign [49] dataset. The code snippets in Devign are collected from two open-source projects in C, and 46% of them are labeled as vulnerable. Croft et al. [8] manually studied 70 **vulnerable-labeled** samples in the Devign dataset and found that 20% of them are mislabeled.

3.2.3 Code Summarization. In the code summarization task, a deep learning model generate the natural language summary of a given code snippet. Different from classification tasks, it is difficult to determine whether a code snippet is "mislabelled". In this paper, if a natural language summary cannot describe the functionality of the corresponding program, we treat this data sample as "noisy". For this task, we adopt the TLC [18] dataset, which is a Java code summarization dataset frequently used in previous works. A previous study [34] claims that the TLC dataset contains around 41% of noise, but their observation is based on rule-based patterns and may not reflect the standard of human developers, so we conduct a human evaluation of this dataset. As the original TLC dataset is too large for a complete human evaluation, we randomly sample 100 code-summary pairs from the training set of TLC and ask six Ph.D. students and postdocs in software engineering to score the quality of these data pairs from 0 (lowest) to 2 (highest). All six labelers are asked to score all 100 test samples, and the final quality score is acquired by averaging the scores given by all labelers. Different scores are defined by:

- 2: The summary describes the functionality of the code snippet. Human developers are possible to generate this summary without extra context information.
- 1: The summary partially describes the functionality of the code snippet / The summary may be correct, but it relies on extra context / The summary may be correct, but it contains non-functional descriptions (e.g., security attributes, I/O regulations).
- 0: The summary does not describe the code, or The summary heavily relies on extra context, making it impossible to determine whether the summary is correct.

Fig. 2 shows examples for each score in the TLC dataset. Generally, if the average human evaluation score of a sample is not larger than 1, we can regard this sample as "noisy" and vise versa. Fig. 3 demonstrates the histogram of human evaluation scores on the aforementioned subset with 100 samples. The average quality score for these samples is 1.42. Different from the rule-based analysis [34], we find that human evaluators are generally satisfied

```
Code:
public static boolean startsWith(String s1,
String s2){
  if (s1 == null || s2 == null) {
    return false;
  }
  return s1.startsWith(s2);
}

Summary:
Check if one string starts with another string.
```

```
Code:
public S2Cap complement(){
  double cHeight=isFull() ? -1 : 2 -
Math.max(height,0.0);
  return S2Cap.fromAxisHeight(S2Point.neg(axis),cHeight);
}

Summary:
 Return the complement of the interior of the cap. A cap and its
complement have the same boundary but do not share any interior
points. The complement operator is not a bijection, since the
complement of a singleton cap (containing a single point) is the same as
the complement of an empty cap.
```

```
Code:
protected void engineSetPadding(String padding)
throws NoSuchPaddingException {
  if (!padding.equalsIgnoreCase("NoPadding")) {
    throw new NoSuchPaddingException("Padding "
+ padding + " unknown.");
  }
}

Summary:
 should never be called.
```

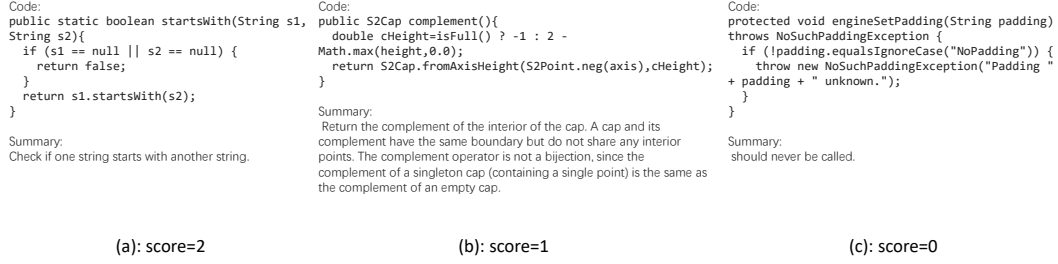(a): score=2   (b): score=1   (c): score=0

Figure 2: Examples for code summarization data samples with each score. (a): a typical example with score=2. The summary clearly describes the functionality of the code snippet. (b): an example with score=1. The summary describes the code, but it requires additional context (e.g., API doc) to map the code to the summary. Moreover, the summary contains a large amount of non-functional descriptions. (c): an example with score=0. The summary is irrelevant to the code snippet's functionality. Note that this sample cannot be detected by the rule-based approach [34].
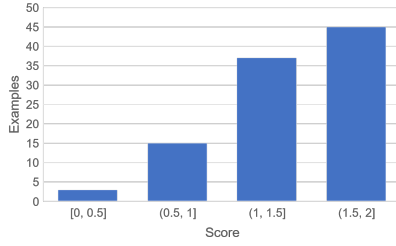

Figure 3: Histogram of the average human evaluation scores on a small subset of TLC.

with the summarization quality of the TLC dataset. Over 80% of code-summary pairs are scored larger than 1, while heavily noisy samples (with a score $\leqslant 0.5$) only consist of less than 5%.

3.3 Noisy Label Learning Approaches

We choose at least one NLL approach from all categories in Table 1 except Meta-learning because for many program understanding tasks, such as code summarization and vulnerability detection, it is often unfeasible to build noise-free metasets with enough sizes. Namely, we adopt the following NLL approaches:

(1) **TracIn [32].** Similar to Influence Function [20], TracIn aims to predict the influence score of a training sample on a test sample. To detect noisy-labeled samples, we calculate the self-influence, which refers to the influence score of a sample on itself. Given the expected noise rate $k\%$, we choose the $k\%$ samples with the highest influence scores as the noisy samples and remove them from the dataset.

(2) **Co-teaching [14]** maintains two separate neural networks with identical structures. In each training batch of Co-teaching, both networks rank all the samples in the batch by their training losses. Suppose the expected noise rate is $k\%$ and the batch size is B, each network selects $(1 - k\%)B$ samples with the smallest loss and feeds them to its peer network for further training.

(3) **RobustTrainer [25]** uses the learned features to identify noise samples during training. The RobustTrainer algorithm consists of two steps: the robust learning algorithm for handling label noises, and class-balanced sampling for addressing data imbalance. As our study focuses on label noises, we only adopt the robust learning step of RobustTrainer. In each training epoch, after the model acquires the features for all training samples, RobustTrainer calculates the similarities between sample features and their class prototypes. The authors further use a Gaussian Mixture Model (GMM) to split the samples into two distributions by the similarity score: samples with high similarity scores are likely to be correctly labeled, while low similarity scores indicate that the sample may be noisy. The selected clean samples are used for the training epoch.

(4) **Confident Learning [30]** is an approach for identifying label noises in classification datasets. It is based on the assumption of Classification Noise Process [5]: label noise is class-conditional, it only depends on the true class label, not the data itself. Confident Learning first builds the confidence joint matrix between noisy labels and latent "true" labels, and then prunes mislabeled data based on the off-diagonals of the confidence joint matrix.

(5) **Simifeat [50]** is a training-free noise detection approach which is based on features from pre-trained models. As this approach requires a pre-trained deep learning model, it can not be applied to trained-from-scratch models. Simifeat uses two different algorithms: voting and ranking to select the most possible "true" label for each sample based on its k-NN neighbours in the feature vector space. If the selected label is different from the label $\hat{y}$ from the dataset, this sample is considered noisy.

3.4 Base Models

When evaluating the performance of NLL approaches, we must consider the base deep learning models used for prediction tasks. Label noises may have different impacts on various neural models, and hence, we included several different deep learning models in our study. These models include sequence-based and tree/graph-based models commonly used for program language data.

For the classification-style tasks, specifically, we have selected the following neural network models:

(1) **Trained-from-scratch small models:** In the early days of deep learning-based program understanding, researchers often treat programs as sequences of tokens and use simple neural networks to model program token sequences. One of the most frequently used sequential model is **LSTM** [16], which we will evaluate in our study.

In the past few years, researchers have noticed that programs contain rich structural information, and integrating deep neural networks with program structures can boost the performance of various program understanding tasks. Many works parse programs into ASTs and add control or data flow edges to build program graphs [4, 33, 41]. Many types of graph neural networks (GNN) have been adopted to learn on programs graphs, such as graph convolutional network (GCN), gated graph neural network (GGNN), and heterogeneous graph neural networks. We choose **Graph Isomorphism Network (GIN)** [42] for our experiments. GIN has been selected as a strong baseline for many program comprehension works, and has outperformed other GNN baselines in the program classification task [33, 47]. As the CodeNet dataset has provided parsed program graphs, we use GIN on the program classification task.

(2) **Large pre-trained models:** In recent years, large pre-trained models have achieved success in natural language and program language-related tasks. Most of these models are based on the Transformer [40] architecture. We choose three pre-trained Transformer encoder models: **CodeBERT** [11], **Graph-CodeBERT** [13], and **UnixCoder** [12]. These three models use the same architecture as Roberta [26]. CodeBERT is the first bi-modal pre-trained Transformer model for natural language and program language. It adopts two pre-training objectives: masked language model (MLM) and replaced token detection. GraphCodeBERT further augments Code-BERT with data flow information: it creates a simple data flow graph for input programs during pre-training. Unix-Coder uses abstract syntax tree (AST) information in the pre-training stage. Furthermore, UnixCoder tried to use the same model for both its encoder and decoder, which means that its pre-trained model can be applied to both program understanding and generation tasks.

For the code summarization task, we adopt the following encoder-decoder models:

(1) **Transformer:** we use a vanilla trained-from-scratch Transformer encoder-decoder for this task. The Transformer encoder takes the code token sequence as input, and the decoder generates the natural language summary tokens. Similar architectures have been applied to various code summarization datasets and achieved desirable results [1].

(2) **PLBART [2]** is a large Transformer encoder-decoder model pre-trained on bimodal data in program languages and natural language. It is able to complete various sequence generation tasks in software engineering, code summarization being one of them. PLBART has achieved state-of-the-art

Table 3: A summary of the program understanding base models and their parameter sizes used in our study.

Type	Model	Parameters
	LSTM	7M
Small trained-from-scratch	GIN	7M
	Transformer (summarization)	30M
	CodeBERT	125M
Large pre-trained	GraphCodeBERT	125M
	UnixCoder	125M
	PLBART	140M

results on code summarization datasets in multiple program languages.

Table 3 shows the summary of our adopted program understanding models in their parameter sizes. All the pre-trained models included in our study belong to the pre-train and fine-tune paradigm: a model is first trained on a large corpus with self-supervised learning, then fine-tuned for the downstream task using the training set of downstream tasks. The sizes of these models range from 125M to 140M parameters. Recently, decoder-only large language models (LLMs) with larger than 5B parameters have seen successes in some software engineering tasks. We do not investigate these models in our study for the following reasons: first, some LLMs [6, 31] are closed-source and cannot be trained on large labeled datasets. Second, though LLMs have shown satisfying results on program generation tasks, they still do not show superiority over smaller fine-tuned models on some program understanding tasks [37, 45].

4 RESEARCH QUESTIONS AND FINDINGS

RQ1: How do different types of label noises in program classification affect the performance of deep learning models when NLL is not introduced?

Experiment Setup: To answer this RQ, we first run our selected models on the original noise-free Java250 dataset. Then we inject noises with different patterns into the Java250 training dataset and run these models again. We adopt 8 different noise rates: random noises range from 10% to 50%, and flip noises from 10% to 40%.

Results: Table 5 demonstrate the classification results on Java250 with different models, noises, and NLL approaches. We can observe that label noises have different impacts on different neural models. Small trained-from-scratch models, i.e., LSTM and GIN, are prone to label noises in training data. However, large pre-trained models are highly robust against label noises, especially random noises. Even when the noise ratio reaches 50%, the accuracy drop for these models is only 1%-2%. Between the two types of noises, all models in our experiment are more prone to label-dependent noise (flip noise) than label-independent noise (random noise). Although pre-trained models are hardly affected by random label noises, their accuracy can still drop over 10% when trained with 40% flip noises.

To further analyze the differences between small models and large pre-trained models on training with noisy labels, we take a deeper look into the training process of different models. Fig. 4 shows the average training loss and validation set accuracy of LSTM and CodeBERT during training on the noisy Java250 dataset. We can

Table 4: The classification accuracy (%) and (accuracy difference with noise-free training data) on Java250 dataset with different noise rates and deep learning models.

Model	No noise	10%random	20% random	30%random	50% random	10% flip	20%flip	30% flip	40% flip
LSTM	93.04	86.68(6.36)	82.88 (10.16)	78.69(14.35)	56.57 (36.47)	83.64(9.40)	77.59 (15.45)	68.22(24.82)	56.32 (36.72)
GIN [42]	93.23	89.16(4.07)	86.72 (6.51)	83.64(9.59)	78.73 (14.50)	85.17(8.06)	81.51 (11.72)	73.81(19.42)	61.42 (31.81)
CodeBERT [11]	98.05	97.57(0.48)	97.26 (0.79)	96.87(1.18)	96.05 (2.00)	97.43(0.62)	96.05 (2.00)	91.22(6.83)	85.58 (12.47)
GraphCodeBERT [13]	98.26	97.62(0.64)	97.60 (0.66)	97.15(1.11)	96.48 (1.78)	97.77(0.49)	96.41 (1.85)	92.87(5.39)	87.85 (10.41)
UnixCoder [12]	98.39	97.86(0.53)	97.29 (1.10)	97.18(1.21)	96.67 (1.72)	97.75(0.64)	96.52 (1.87)	93.06(5.33)	90.21 (8.18)
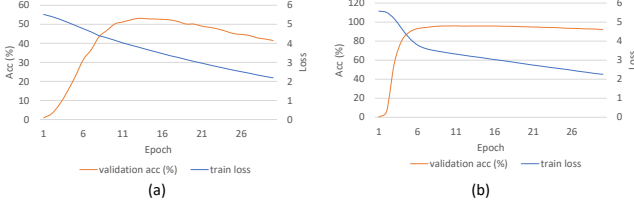
Figure 4: The training loss and validation accuracy during training on 50% random label noise. (a): LSTM. (b): CodeBERT.

see that for the small model LSTM, the training loss gradually drops during the whole training process, while the validation accuracy reaches the peak in around 15 epochs, then starts to drop. On the other hand, for CodeBERT, its loss decreases quickly in the first few epochs and its validation accuracy simultaneously reaches the maximum. Then, the drop of training loss becomes much slower, and the validation accuracy remains high for many epochs. This shows that compared to trained-from-scratch models, large pre-trained models are less likely to overfit the label noises. This is mainly because that pre-trained models have gained stronger generalization abilities from pre-training on unlabeled data. Pre-trained program understanding models demonstrate similar behavior to pre-trained models on natural language processing tasks [39]: the model training has a long "settling " phase in which the model continues to learn on correctly-labeled samples without memorizing mislabeled samples.

> **Answer to RQ1:** In program classification, large pre-trained models achieve significantly higher accuracies than small trained-from-scratch models under different training label noises. Label-dependent flip noises have a larger influence on most program understanding models than label-independent random noises.

RQ2: How do existing NLL approaches perform on different synthetic noises in program classification?

Experiment Setup: Most existing NLL approaches focus on two goals: detect **training** label noises or improve the models' robustness against them. We will first evaluate the capability of NLL on program classification in these two objectives. We follow the experiment settings in previous works: the training set is corrupted with label noises, and the validation/test sets are noise-free. We run the adopted NLL approaches on different models with the same noise settings in RQ1. We evaluate the effectiveness of NLL approaches

with two metrics: the program classification accuracy after applying NLL, and the precision/recall/F1 in detecting mislabeled samples.

For Co-teaching and RobustTrainer, the NLL approaches are integrated into the training process of the model $f(x)$. As for TracIn, confident Learning and Simifeat, the NLL approaches are separated from model training. These approaches follow the "detect-retrain" pipeline. We first train/fine-tune the model $f(x)$ on the noisy training set $\mathcal{D}_{train}$ (for TracIn and Confident Learning), then apply NLL approaches on $f(x)$ to detect noisy samples. We generate a "clean" training set $\mathcal{D}'_{train}$ by removing the detected samples, and re-train/fine-tune a new model $f'(x)$ on $\mathcal{D}'_{train}$ for evaluation.

To gain a clear understanding of the effectiveness of NLL approaches on program classification, we introduce a simple baseline approach: treat all data samples with the predicted label y' that does not match the label in the dataset $\hat{y}$ as noisy samples. After noise detection, we remove the suspected noisy data samples and retrain the model.

Beyond evaluating NLL on noisy training sets, we take a further step of detecting label noises in test sets. In many program understanding datasets, training, validation, and test sets are split randomly or based on simple rules without further data processing, so they may share similar noises. In our experiment, we make the following assumption: the training, validation, and test sets have the same noise distribution. In the program classification dataset with synthetic noise, this means that all three subsets are corrupted with the same noise pattern and rate.

Results: From the results in Table 5, we can find that after applying noisy label learning approaches, the classification accuracies in most models and noise settings are improved. Generally, the improvements on small models are significant, especially for LSTM. For example, with 50% random noises, most NLL approaches can improve accuracy by approximately 20%. In contrast, large pre-trained models only achieve marginal improvements on most NLL approaches because they are already robust against label noises.

When the noise rate increases, most NLL approaches achieve greater improvements over no NLL, but they also exhibit some extent of differences in their performances. Generally, Confident Learning and TracIn are the best-performing approaches in program classification. Especially, TracIn performs much better than other approaches on 40% flip noises. Although RobustTrainer outperforms its baseline method Co-teaching on small binary classification datasets [25], it does not show superiority on the noisy program classification task. Sometimes it even yields negative improvements on large pre-trained models.

As for the simple baseline approach, which directly treats predicted label $y' \neq \hat{y}$ as noisy labels, it actually performs quite well

Table 5: The classification accuracy (%) and (the improvement after NLL) on the noise-free Java250 test set with different NLL approaches.

Model	Approach	10%random	20% random	30%random	50% random	10%flip	20% flip	30%flip	40% flip
LSTM	Co-teaching	90.26(3.58)	88.25 (5.37)	83.37(4.68)	**77.85 (21.28)**	87.35(3.71)	84.27 (6.68)	**76.67(8.45)**	68.25 (11.93)
	TracIn	90.17(3.49)	89.46 (6.58)	84.05(5.36)	76.81 20.24)	88.81(5.17)	**88.62 (11.03)**	73.30(5.08)	**69.26 (12.94)**
	Confident Learning	90.59(3.91)	86.96 (4.08)	81.72(3.03)	65.12 (8.55)	87.49(3.85)	86.41 (8.82)	75.26(7.04)	60.28 (3.96)
	RobustTrainer	91.37(4.69)	**91.29 (8.41)**	83.98(5.29)	75.64 (19.07)	**88.96(5.32)**	88.47 (10.88)	75.41(7.19)	61.79 (5.47)
	Baseline-class	**91.82(5.14)**	90.23 (7.35)	**85.11(6.42)**	73.13 (16.56)	88.58(4.94)	85.43 (7.84)	74.64(6.42)	57.69 (1.37)
GIN [42]	Co-teaching	91.28(2.12)	90.73 (4.01)	87.18(3.54)	**86.42 (7.69)**	89.25(4.08)	88.41 (6.90)	**81.02(7.21)**	74.03 (12.61)
	TracIn	90.74(1.58)	90.35 (3.63)	**87.39(3.75)**	83.26 (4.53)	89.79(4.62)	**88.47 (6.96)**	78.88(5.07)	**75.78 (14.36)**
	Confident Learning	92.08(2.92)	88.95 (2.23)	86.94(3.30)	80.78 (2.05)	90.51(5.34)	87.68 (6.17)	80.01(6.20)	67.13 (5.71)
	RobustTrainer	89.65(0.49)	90.51 (3.79)	84.75(1.11)	82.86 (4.13)	89.48(4.31)	85.44 (3.93)	76.37(2.56)	59.51 (-1.91)
	Baseline-class	**92.21(3.05)**	90.95 (4.23)	87.02(3.38)	85.63 (6.90)	**90.55(5.38)**	87.48 (5.97)	77.00(3.19)	65.23 (3.81)
CodeBERT [11]	Co-teaching	97.58(0.01)	97.40 (0.14)	97.05(0.28)	96.43 (0.38)	97.52(0.09)	96.95 (0.90)	91.29(0.07)	86.28 (0.70)
	TracIn	97.85(0.28)	97.80 (0.54)	97.44(0.47)	96.83 (0.78)	97.90(0.47)	**97.78 (1.73)**	96.70(5.48)	**97.06 (11.48)**
	Confident Learning	**98.05(0.48)**	97.81 (0.55)	97.47(0.60)	96.69 (0.64)	**98.02(0.59)**	97.71 (1.66)	**96.79(5.57)**	94.27 (8.69)
	Simifeat	97.60(0.03)	96.76 (-0.50)	96.66(-0.21)	96.21 (0.16)	97.49(0.06)	96.79 (0.74)	91.12(-0.10)	89.83 (4.25)
	RobustTrainer	97.65(0.08)	97.05 (-0.21)	96.92(0.05)	97.01 (0.96)	97.71(0.28)	96.61 (0.56)	93.19(1.97)	84.87 (-0.71)
	Baseline-class	97.91(0.34)	**97.81 (0.55)**	**97.76(0.89)**	**97.23 (1.18)**	97.94(0.51)	97.25 (1.30)	96.42(3.20)	91.79 (6.21)
GraphCodeBERT [13]	Co-teaching	97.64(0.02)	97.65 (0.05)	97.35(0.20)	96.91 (0.43)	97.77(0.00)	97.66 (1.25)	92.87(0.00)	90.19 (2.34)
	TracIn	98.16(0.54)	**98.14 (0.54)**	97.30(0.15)	97.16 (0.68)	98.00(0.23)	97.82 (1.41)	**97.44(4.57)**	**97.21 (9.36)**
	Confident Learning	**98.24(0.62)**	97.99 (0.39)	97.76(0.61)	97.13 (0.65)	**98.26(0.49)**	98.05 (1.64)	97.10(4.23)	94.63 (6.78)
	Simifeat	97.67(0.05)	97.22 (-0.38)	97.13(-0.02)	96.54 (0.06)	97.85(0.08)	97.35 (0.94)	94.12(1.25)	90.50 (2.65)
	RobustTrainer	97.96(0.34)	97.59 (-0.01)	97.51(0.36)	**97.66 (1.18)**	97.96(0.19)	96.78 (0.37)	93.03(0.16)	88.13 (0.28)
	Baseline-class	98.12(0.50)	98.06 (0.46)	**97.94(0.79)**	97.47 (0.99)	98.02(0.25)	97.72 (1.31)	95.89(3.02)	92.73 (4.88)
UnixCoder [12]	Co-teaching	97.99(0.13)	97.62 (0.33)	97.26(0.08)	97.05 (0.38)	97.89(0.14)	97.39 (0.87)	94.05(0.99)	92.41 (2.20)
	TracIn	98.25(0.39)	**98.25 (0.96)**	97.59(0.33)	97.49 (0.82)	98.06(0.31)	98.18 (1.66)	97.35(4.29)	**97.31 (7.10)**
	Confident Learning	**98.32(0.46)**	98.06 (0.77)	**97.85(0.67)**	97.37 (0.70)	**98.29(0.54)**	98.20 (1.68)	97.37(4.31)	95.19 (4.98)
	Simifeat	97.86(0.00)	97.25 (-0.04)	97.22(0.04)	96.55 (-0.12)	97.81(0.06)	97.31 (0.79)	93.01(-0.05)	92.69 (2.48)
	RobustTrainer	97.95(0.09)	97.48 (0.19)	97.07(-0.11)	97.06 (0.39)	97.85(0.10)	97.42 (0.90)	94.18(1.12)	91.59 (1.38)
	Baseline-class	98.26(0.40)	98.02 (0.73)	97.71(0.53)	**97.81 (1.14)**	98.11(0.36)	98.03 (1.51)	95.22(2.16)	93.67 (3.46)

in our experiments. On the random noise settings, this baseline often outperforms most NLL approaches. However, on flip noises, its performance is lower than TracIn and Confident Learning. The classification accuracies on Simifeat with pre-trained models are lower than other approaches. We believe this is because that Simifeat does not require training/fine-tuning on the classification dataset.

Table 6 shows the precision, recall, and F1 in detecting label noises in the Java250 training dataset. We only include the approaches that separate model training from noise detection (or do not involve model training), because for approaches integrated with training, their "predicted noisy samples" are continuously changing during different training epochs.

From the results, we can see that most NLL approaches perform well in detecting noisy training data and achieve an F1 score over 0.85, except for Simifeat. The results in noisy data detection are consistent with the accuracies on the classification task: NLL approaches with higher classification accuracies tend to have higher F1 values in noisy data detection. Confident Learning performs best in most settings with low noise rates, while TracIn excels in high noise rates. Nevertheless, the baseline achieves comparable results in most noise settings. Compared to NLL approaches, the baseline's recall is higher, and precision is lower, indicating that it is likely to report more false positive noise samples. Notice that TracIn has equal precision/recall/F1 in each experiment setting: this is because the assumed noise rate is equal to the actual noise rate, resulting in the same number of actual noise samples and samples detected as noise, which leads to the same precision and recall. With most

NLL approaches perform similarly. large models perform better than trained-from-scratch models in noise detection, suggesting that program understanding models with stronger capabilities are also better at detecting noisy labels.

Table 7 demonstrates the evaluation metrics in detecting test data noises for the program classification dataset. Generally, the performances of NLL approaches are similar, but a little lower than their results in detecting training data noises. An unexpected difference is that TracIn performs much worse on test sets than on training sets, especially in low noise rates. Confident Learning is the best-performing NLL approach in most noise settings except for 40% flip noises.

> **Answer to RQ2:** Most existing noisy label learning approaches can marginally improve performance on clean test sets for program classification, but they do not show significant superiority over a simple baseline on many noise settings. The improvements of NLL approaches on small trained-from-scratch models are larger than on large pre-trained models. Large pre-trained models integrated with NLL approaches can serve as well-performing label noise detectors for detecting both training and test data noise in the program classification task, although a simple baseline can achieve similar results.

RQ3: How do NLL approaches perform on program understanding tasks with real-world noises?

Table 6: The precision/recall/F1 in training data noise detection with different noise rates, models, and noisy label learning approaches on the Java250 dataset.

Model	Approach	10% random			20% random			30% random			50% random			10% flip			20% flip			30% flip			40% flip		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
LSTM	TracIn	74.17	74.17	74.17	82.51	82.51	82.51	80.05	80.05	80.05	**85.70**	**85.70**	**85.70**	72.96	72.96	72.96	84.27	84.27	**84.27**	71.82	71.82	71.82	**69.11**	69.11	**69.11**
	Confident Learning	**84.78**	79.33	81.96	**90.95**	84.58	**87.65**	**88.57**	76.31	81.99	82.18	82.34	82.26	**85.23**	72.81	78.53	77.50	87.72	82.30	**87.15**	68.0	76.39	59.98	65.94	62.82
	Baseline-class	76.37	**94.11**	**84.32**	72.65	**99.96**	84.15	71.38	**98.53**	**82.79**	74.04	**99.83**	85.02	64.25	**97.72**	77.53	60.74	**91.43**	72.99	65.48	**94.74**	**77.44**	56.10	**70.55**	62.50
GIN [42]	TracIn	74.00	74.00	74.00	84.42	84.42	84.42	80.20	80.20	80.20	84.56	84.56	84.56	75.11	75.11	75.11	73.57	73.57	73.57	**76.21**	76.21	76.21	**69.71**	69.71	**69.71**
	Confident Learning	**92.21**	80.27	85.83	**90.65**	85.72	**88.12**	**92.53**	81.90	86.89	**91.66**	84.24	87.10	**91.18**	78.82	**84.55**	77.03	89.09	82.62	75.11	85.27	79.87	59.86	65.11	62.38
	Baseline-class	82.68	**94.16**	**88.05**	75.50	**99.98**	86.03	76.98	**96.67**	**88.62**	85.71	**99.94**	**91.36**	78.15	**99.47**	73.55	62.54	**91.58**	74.33	66.91	80.91	73.25	56.43	68.41	61.85
CodeBERT [11]	TracIn	80.53	80.53	80.53	94.21	94.21	94.21	91.13	91.13	91.13	98.37	98.37	98.37	84.87	84.87	84.87	93.69	93.69	93.69	**94.01**	94.01	**94.01**	**94.35**	94.35	**94.35**
	Confident Learning	**98.99**	95.60	**97.26**	**99.04**	93.97	96.43	**99.22**	94.46	96.78	**99.18**	94.61	96.84	96.78	96.93	**96.85**	**96.80**	96.56	**96.68**	93.34	93.82	93.58	91.79	89.74	90.75
	Simifeat	67.42	61.09	64.10	61.10	78.56	68.74	60.94	68.25	64.39	82.75	60.20	69.70	63.77	55.51	59.35	56.67	70.10	62.67	64.87	73.64	68.98	65.23	51.94	57.83
	Baseline-class	88.41	**100.0**	93.85	92.73	**100.00**	96.23	95.34	**99.99**	**97.61**	96.88	**99.99**	**98.41**	88.36	**99.73**	93.70	89.91	**98.57**	94.04	83.99	**97.88**	90.41	82.37	**95.30**	88.37
GraphCodeBERT [13]	TracIn	85.47	85.47	85.47	96.41	96.41	96.41	93.02	93.02	93.02	99.17	99.17	99.17	84.31	84.31	84.31	95.82	95.82	95.82	94.97	94.97	94.97	**93.08**	93.08	**93.08**
	Confident Learning	**98.98**	94.71	**96.79**	**99.16**	95.04	97.06	**99.17**	94.94	97.01	**99.35**	93.53	96.36	**97.06**	97.08	**97.07**	**96.92**	97.04	**96.98**	**95.22**	92.87	**94.03**	92.22	89.41	90.79
	Simifeat	66.39	59.51	62.76	57.36	85.26	68.58	67.77	60.84	64.12	79.31	73.06	76.06	63.31	69.33	66.18	62.18	59.67	60.90	73.66	61.79	67.21	69.37	61.26	65.06
	Baseline-class	87.29	**99.98**	93.20	94.47	**99.98**	**97.15**	95.43	**99.99**	**97.66**	97.29	**99.99**	**98.62**	89.82	**99.82**	94.55	90.23	**98.57**	94.21	87.23	**97.98**	92.29	84.11	**94.49**	89.00
UnixCoder [12]	TracIn	90.24	90.24	90.24	97.43	97.43	**97.43**	92.95	92.95	92.95	**99.31**	99.31	**99.31**	83.71	83.71	83.71	96.62	96.62	**96.62**	93.15	93.15	93.15	**95.83**	95.83	**95.83**
	Confident Learning	**96.96**	92.87	94.87	**98.57**	89.23	93.67	**99.80**	92.75	**96.15**	99.30	92.07	95.55	**97.91**	94.02	**95.93**	95.14	93.90	94.51	**97.87**	91.53	**94.59**	93.69	92.66	93.17
	Simifeat	63.78	70.87	67.14	68.56	81.67	74.54	62.30	66.10	64.14	85.72	65.20	74.06	65.54	46.80	54.61	60.19	72.38	65.72	56.42	61.12	58.67	70.50	47.78	56.96
	Baseline-class	93.48	**98.13**	**95.74**	94.03	**99.98**	96.91	92.23	97.87	94.96	96.72	**99.99**	98.32	92.35	**98.93**	95.53	87.86	**100.00**	93.54	90.61	**98.16**	94.23	91.15	**95.98**	93.50

Table 7: The precision/recall/F1 in test data noise detection with different noise rates, models, and noisy label learning approaches on the Java250 dataset.

Model	Approach	10% random			20% random			30% random			50% random			10% flip			20% flip			30% flip			40% flip		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
LSTM	TracIn	67.31	67.31	67.31	76.15	76.15	76.15	**81.02**	81.02	**81.02**	**85.82**	85.82	**85.82**	**80.83**	80.83	**80.83**	**79.91**	79.91	**79.91**	67.95	67.95	67.95	**71.04**	71.04	**71.04**
	Confident Learning	**81.09**	78.33	79.69	**80.37**	80.67	**80.52**	78.24	73.84	75.98	73.42	84.48	78.56	74.75	77.52	77.52	62.62	84.00	71.75	69.4	72.23	**75.31**	55.48	65.17	**71.53**
	Baseline-class	70.42	**96.20**	**81.32**	60.54	**99.93**	75.40	69.99	**97.16**	81.36	67.38	**99.73**	80.43	60.33	**97.16**	74.44	48.77	**89.83**	63.82	52.62	**88.24**	65.93	52.35	**71.53**	60.46
GIN [42]	TracIn	71.06	71.06	71.06	78.63	78.63	78.63	79.90	79.90	79.90	86.85	86.85	86.85	71.95	71.95	71.95	67.30	67.30	67.30	71.46	71.46	71.46	**67.10**	67.10	**67.10**
	Confident Learning	**85.72**	78.56	81.98	**83.50**	83.33	83.42	**89.20**	74.09	80.95	**91.08**	83.49	87.12	**86.16**	81.2	**83.61**	68.76	86.07	**76.45**	**81.64**	73.18	**77.18**	57.09	65.28	60.91
	Baseline-class	80.71	**91.93**	**85.95**	65.57	**99.93**	79.19	79.16	**89.20**	**83.88**	83.23	**99.92**	**90.81**	78.99	**84.98**	81.88	54.43	**91.03**	68.13	71.43	**74.61**	72.98	54.24	**70.27**	61.26
CodeBERT [11]	TracIn	73.23	73.23	73.23	79.83	79.83	79.83	85.76	85.76	85.76	94.56	94.56	94.56	83.60	83.60	83.60	79.90	79.90	79.90	87.52	87.52	87.52	**91.90**	91.90	**91.90**
	Confident Learning	**99.76**	91.47	**95.43**	96.69	92.47	94.53	98.57	91.27	**94.78**	**98.89**	92.76	95.73	**98.06**	85.27	**91.22**	95.40	96.10	**95.75**	**99.09**	81.48	**89.43**	82.29	79.80	81.03
	Simifeat	54.22	58.02	56.05	51.24	76.60	61.40	65.16	61.04	63.04	81.62	55.23	65.88	56.52	73.88	64.04	46.95	69.03	55.89	80.28	57.34	66.9	60.91	53.98	57.24
	Baseline-class	89.51	**95.80**	92.55	89.15	**100.00**	94.27	88.75	**100.0**	94.04	95.88	**100.00**	**97.89**	79.32	**91.89**	85.14	88.43	**99.90**	93.82	82.06	**97.87**	89.27	73.16	**83.18**	77.85
GraphCodeBERT [13]	TracIn	76.31	76.31	76.31	81.17	81.17	81.17	83.88	83.88	83.88	95.24	95.24	95.24	84.17	84.17	84.17	81.20	81.20	81.20	89.46	89.46	89.46	**93.20**	93.20	**93.20**
	Confident Learning	**99.10**	88.36	93.42	**98.53**	91.70	94.99	**99.98**	90.45	94.98	**99.24**	92.55	95.77	**98.24**	80.78	88.66	95.98	95.50	**95.74**	**99.15**	84.46	91.22	89.74	87.75	88.73
	Simifeat	57.69	64.0	60.68	46.87	68.13	55.53	55.04	67.93	60.81	77.84	47.69	59.14	83.6	59.83	69.74	44.32	64.33	52.48	54.12	82.76	65.44	59.50	51.85	55.41
	Baseline-class	89.17	**97.33**	93.07	91.60	**99.97**	**95.60**	93.57	**98.67**	**96.05**	96.47	**99.92**	**98.20**	90.11	**94.11**	**92.07**	89.11	**99.03**	93.81	88.78	**99.31**	**93.75**	82.90	**91.87**	87.15
UnixCoder [12]	TracIn	82.97	82.97	82.97	90.00	90.00	90.00	87.77	87.77	87.77	97.39	97.39	97.39	80.06	80.06	80.06	89.80	89.80	89.80	92.37	92.37	92.37	**96.03**	96.03	**96.03**
	Confident Learning	**96.78**	91.6	**94.12**	**98.13**	90.97	94.41	**98.21**	91.9	94.95	**98.67**	90.81	94.58	**93.98**	91.89	**92.92**	95.79	94.57	**95.17**	**97.56**	88.16	92.62	92.83	92.17	92.50
	Simifeat	49.88	64.78	56.36	58.50	84.30	69.07	52.88	64.75	58.21	88.85	65.81	75.61	55.56	72.16	62.78	52.74	74.27	61.68	67.15	57.36	61.87	52.67	55.85	54.21
	Baseline-class	88.95	**96.04**	92.36	92.45	**99.97**	**96.06**	94.83	**99.3**	**97.01**	95.45	**100.00**	**97.58**	88.41	**96.27**	92.17	90.10	**100.00**	94.79	88.75	**97.96**	**93.13**	91.42	**97.98**	94.59

Table 8: The precision/recall/F1 in detecting noisy-labeled samples in the human-verified subset within the Devign training set.

Model	Approach	P	R	F1
LSTM	TracIn 20%	33.33	36.36	34.78
	TracIn 50%	26.67	72.73	39.02
	Confident Learning	**75.0**	54.55	**63.16**
	Baseline-class	53.85	**63.64**	58.33
CodeBERT [11]	TracIn 20%	50.00	54.55	52.17
	TracIn 50%	30.00	81.82	43.90
	Confident Learning	**88.89**	72.73	**80.00**
	Baseline-class	69.23	**81.82**	75.00
GraphCodeBERT [13]	TracIn 20%	66.67	72.73	69.57
	TracIn 50%	36.67	100.0	53.66
	Confident Learning	**87.5**	63.64	73.68
	Baseline-class	75.0	**81.82**	**78.26**
UnixCoder [12]	TracIn 20%	66.67	72.73	69.57
	TracIn 50%	36.67	100.0	53.66
	Confident Learning	69.23	81.82	75.00
	Baseline-class	66.67	**90.91**	**76.92**

(a) Losses

(b) TracIn scores

Figure 5: A comparison between NLL approaches (TracIn and loss) and human evaluation on the TLC code summarization test subset. (a): Losses. (b): TracIn scores.

To answer this RQ, we adopt two tasks with real-world noises in their datasets: vulnerability detection and code summarization.

Experiment Setup: For vulnerability detection, we investigate the performances of TranIn, Confident Learning, and Simifeat in detecting noisy samples, similar to our experiment in RQ2. Croft

et al. [8] created a subset with 70 vulnerable-labeled samples, in which 20% of them are mislabeled. Among these 70 examples, 60 of them are in the training set of Devign, and we use them to measure the capability of NLL approaches to detect noisy-labeled samples. As our prior knowledge of Devign noise rate may not be accurate, we adopt two different suspected noise rates: 20% and 50% when evaluating TracIn.

Nearly all of the existing NLL approaches are built for classification tasks, so most of them cannot be applied to code summarization. We only inspect TracIn in this experiment because TracIn only relies on data gradients without prior assumptions on the task type. We compare TracIn with a simple baseline: use the loss values to determine the likelihood of noisy samples. Samples with higher losses are more likely to be noisy or low-quality summarization data. We use the subset with 100 samples and their human evaluation scores in Section 3.2.3 for our experiment. The human evaluation scores are computed by averaging the scores of the six annotators ranging from 0 to 2.

For the code summarization task, there is no clear boundary between "noisy" and "clean" samples, so we cannot perform quantitative analysis like detecting noisy samples for code classification. Instead, we analyze the distribution of scores given by NLL approaches and check whether they are consistent with the distribution of human evaluation scores.

Results: Table 8 exhibits the results of noisy label detection in the human-verified Devign subset [8]. We do not evaluate Simifeat because the subset is too small to perform clustering algorithms. We find that the performances of NLL approaches in detecting noises are much lower than in program classification datasets with synthetic noises. Meanwhile, TracIn, which performed very well on synthetic noises, falls behind other approaches and performs much worse than our naive baseline. On the other hand, Confident Learning performs better and achieved F1 scores similar to the baseline.

Fig. 5 shows the distribution of TracIn scores and losses across different human evaluation scores. We further adopt Spearman's rank correlation coefficient to measure the relevance between human evaluation and losses/TracIn scores. If loss/TracIn are effective for detecting noises in code summarization, they should have negative correlations with human evaluation scores. We find that for both models, their losses negatively correlate with human evaluation scores (correlations are negative, and $p < 0.05$). This means that code summarization samples with high quality tend to have lower losses. However, there is no significant correlation between the TracIn influence scores and human evaluation ($p > 0.05$), indicating that TracIn is ineffective for detecting noisy samples for code summarization datasets, and is a worse noise detector than simply using losses.

To better understand NLL's ability to identify noisy code summarization samples, we use several examples to demonstrate the similarities and differences between NLL approaches and human evaluation. Fig. 6 shows three examples in our 100-sample code summarization subset. We can observe that for examples (b) and (c), the human evaluation scores are consistent with TracIn influence and loss. Samples with high human evaluation scores have low TracIn influence/loss and vice versa. However, for example (a), NLL approaches give wrong predictions. Its summary does not describe the code, but the TracIn influence and loss are relatively low. We assume this is because this summary pattern frequently occurs in the training set, which makes the model generate an output similar to the low-quality summary in this sample. This will result in a low loss and gradient, thus leading to a low TracIn score.

Figure 6: Cases from TLC's subset with human evaluation scores and TracIn/loss rankings. The TracIn influences and losses are ranked from low to high within the 100 samples. The TracIn influence, loss, and generated summary are produced from PLBART.

> **Answer to RQ3:** Existing NLL approaches still have much room for improvement when dealing with real-world label noises in program understanding, especially in generation-style tasks.

5 THREATS TO VALIDITY

Usage of synthetic label noises. For the program classification task, we only use synthetic label noise because the original dataset is clean, and we are able to investigate the impact of different noise rates and patterns with synthetic noises. To mitigate this threat, we use multiple noise patterns to mimic different cases of noises in different program understanding datasets. We also conduct experiments on two different tasks with real-world noises: vulnerability detection and code summarization.

Human annotation of datasets. In our code summarization experiments, we use human annotations to judge the quality of code-summary pairs, the scores given by human annotators may not accurately reflect the relatedness between the code and summary. To alleviate this threat, we choose six annotators whose expertise is in software engineering and use their average scores as the final human evaluation score. In the future, we hope to leverage the ability of large data annotation teams to build high-quality, noise-free code summarization datasets on larger scales for accurate evaluation.

Limitation in the choice of NLL approaches. In the AI community, there exists a large number of NLL approaches, some of

which are not discussed and evaluated in our study. To make our study representative of most NLL approaches, we make a taxonomy of existing NLL approaches based on their machine learning techniques, and pick representative approaches within each category suitable for program understanding tasks.

6 CONCLUSION

In this paper, we conduct the first empirical study on noisy label learning for program understanding. In particular, we are the first to investigate NLL for generation tasks in program understanding, which has not been explored before. Our evaluation results show that NLL approaches are more helpful to small trained-from-scratch models than large pre-trained models. Although some approaches achieve satisfactory results on synthetic noises, they still struggle in detecting real-world noises in program understanding datasets.

In the future, the main trend in deep learning for program understanding is the increasing popularity of large pre-trained models, and prompt learning with GPT-based models is likely to surpass traditional supervised learning in many tasks. Therefore, rather than using NLL to enhance model robustness against training label noises, we believe it is more meaningful to utilize NLL to assist human experts in constructing higher-quality program understanding datasets for evaluation and pre-training. Another direction is to explore the possibility of NLL in software engineering beyond the scope of program understanding, such as code generation and program repair.

ACKNOWLEDGEMENT

REFERENCES

[1] Wasi 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*. 4998–5007.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[5] Dana Angluin and Philip Laird. 1988. Learning from noisy examples. *Machine Learning* 2 (1988), 343–370.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Roland Croft, M Ali Babar, and Huaming Chen. 2022. Noisy label learning for security defects. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 435–447.

[8] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[9] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2022. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1044–1063.

[10] Anh TV Dau, Nghi DQ Bui, Thang Nguyen-Duc, and Hoang Thanh-Tung. 2022. Towards Using Data-Influence Methods to Detect Noisy Samples in Source Code Corpora. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–3.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[12] 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)*. 7212–7225.

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[14] Bo Han, Quanming Yao, Xingrui Yu, Gang Niu, Miao Xu, Weihua Hu, Ivor Tsang, and Masashi Sugiyama. 2018. Co-teaching: Robust training of deep neural networks with extremely noisy labels. *Advances in neural information processing systems* 31 (2018).

[15] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 392–401.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

[18] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).

[19] Huda Khayrallah and Philipp Koehn. 2018. On the impact of various types of noise on neural machine translation. *arXiv preprint arXiv:1805.12282* (2018).

[20] Pang Wei Koh and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *International conference on machine learning*. PMLR, 1885–1894.

[21] Anran Li, Yue Cao, Jiabao Guo, Hongyi Peng, Qing Guo, and Han Yu. 2023. FedCSS: Joint Client-and-Sample Selection for Hard Sample-Aware Noise-Robust Federated Learning. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–24.

[22] Anran Li, Lan Zhang, Juntao Tan, Yaxuan Qin, Junhao Wang, and Xiang-Yang Li. 2021. Sample-level data selection for federated learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[23] Anran Li, Lan Zhang, Junhao Wang, Feng Han, and Xiang-Yang Li. 2021. Privacy-preserving efficient federated-learning model debugging. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2021), 2291–2303.

[24] Anran Li, Lan Zhang, Junhao Wang, Juntao Tan, Feng Han, Yaxuan Qin, Nikolaos M Freris, and Xiang-Yang Li. 2021. Efficient federated-learning model debugging. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 372–383.

[25] Zhong Li, Minxue Pan, Yu Pei, Tian Zhang, Linzhang Wang, and Xuandong Li. 2022. Robust Learning of Deep Predictive Models from Noisy and Imbalanced Software Engineering Datasets. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[28] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[29] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, and Haoyu Wang. 2023. Understanding and Tackling Label Errors in Deep Learning-Based Vulnerability Detection (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 52–63.

[30] Curtis Northcutt, Lu Jiang, and Isaac Chuang. 2021. Confident learning: Estimating uncertainty in dataset labels. *Journal of Artificial Intelligence Research* 70 (2021), 1373–1411.

[31] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[32] Garima Pruthi, Frederick Liu, Satyen Kale, and Mukund Sundararajan. 2020. Estimating training data influence by tracing gradient descent. *Advances in Neural Information Processing Systems* 33 (2020), 19920–19930.

[33] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[34] 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*. 107–119.

[35] Jun Shu, Qi Xie, Lixuan Yi, Qian Zhao, Sanping Zhou, Zongben Xu, and Deyu Meng. 2019. Meta-weight-net: Learning an explicit mapping for sample weighting. *Advances in neural information processing systems* 32 (2019).

[36] Hwanjun Song, Minseok Kim, Dongmin Park, Yooju Shin, and Jae-Gil Lee. 2022. Learning from noisy labels with deep neural networks: A survey. *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[37] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *arXiv preprint arXiv:2305.12865* (2023).

[38] 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 *Proceedings of the 44th International Conference on Software Engineering*. 1609–1620.

[39] Michael Tänzer, Sebastian Ruder, and Marek Rei. 2022. Memorisation versus Generalisation in Pre-trained Language Models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7564–7578.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[41] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[42] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[43] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining software defects: Should we consider affected releases?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 654–665.

[44] Xingrui Yu, Bo Han, Jiangchao Yao, Gang Niu, Ivor Tsang, and Masashi Sugiyama. 2019. How does disagreement help generalization against label corruption?. In *International Conference on Machine Learning*. PMLR, 7164–7173.

[45] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).

[46] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[47] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 378–389.

[48] Guoqing Zheng, Ahmed Hassan Awadallah, and Susan Dumais. 2021. Meta label correction for noisy label learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 11053–11061.

[49] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[50] Zhaowei Zhu, Zihao Dong, and Yang Liu. 2022. Detecting corrupted labels without training a model to predict. In *International Conference on Machine Learning*. PMLR, 27412–27427.