Towards More Trustworthy Deep Code Models by Enabling Out-of-Distribution Detection

Yanfu Yan[§], Viet Duong[§], Huajie Shao, Denys Poshyvanyk
Department of Computer Science, William & Mary
Williamsburg, Virginia, USA
Email: yyan09, vqduong, hshao, dposhyvanyk {@wm.edu}

***Abstract*—Numerous machine learning (ML) models have been developed, including those for software engineering (SE) tasks, under the assumption that training and testing data come from the same distribution. However, training and testing distributions often differ, as training datasets rarely encompass the entire distribution, while testing distribution tends to shift over time. Hence, when confronted with out-of-distribution (OOD) instances that differ from the training data, a reliable and trustworthy SE ML model must be capable of detecting them to either abstain from making predictions, or potentially forward these OODs to appropriate models handling other categories or tasks.**

In this paper, we develop two types of SE-specific OOD detection models, unsupervised and weakly-supervised OOD detection for code. The unsupervised OOD detection approach is trained solely on in-distribution samples while the weakly-supervised approach utilizes a tiny number of OOD samples to further enhance the detection performance in various OOD scenarios. Extensive experimental results demonstrate that our proposed methods significantly outperform the baselines in detecting OOD samples from four different scenarios simultaneously and also positively impact a main code understanding task.

***Index Terms*—OOD detection, Trustworthy ML, Code Models, Contrastive learning**

I. Introduction

Extensive ML models have been developed under the assumption that training and testing data come from the same distribution (*i.e., closed-world assumption*). However, this assumption is often violated in practice, where deployed models may frequently encounter out-of-distribution (OOD) instances that are not seen in training [1]. For instance, a model trained on high-quality code may struggle to comprehend buggy code. Adapting ML models to distribution shifts is possible but challenging and costly due to the constantly evolving data [2]. Moreover, even if the training data is up-to-date, models will still encounter unforeseen scenarios under the open-world setting. Failure to recognize an OOD sample, and consequently to produce incorrect predictions, significantly compromises the reliability of a model. A reliable and trustworthy ML model should not only achieve high performance on samples from known distributions, *i.e.,* in-distribution (ID) data, but also accurately detect OOD samples which can then either abstain from making predictions, or potentially be forwarded to appropriate models handling other distributions or tasks.

OOD detection has been extensively studied in computer vision (CV) [3] and natural language processing (NLP) [4] across a range of tasks (*e.g.,* image/sentiment classification, question answering). Existing OOD detectors typically design a scoring function to derive confidence/ID scores, enabling the detection of OOD samples based on a predefined threshold. These OOD detectors serve as an auxiliary function to the original ML models and ensures a high proportion (*e.g.,* 95%) [5] of ID data finally retained based on the threshold. This is crucial to prevent the OOD auxiliary scoring from adversely affecting ML models' performance on their main image/language-related tasks. Current OOD detection approaches are proposed in supervised, unsupervised, and weakly-supervised regimes depending on the availability of OOD data. Supervised approaches [6] learn a classical binary classifier based on both ID and OOD data, but in practice, it is hard to assume the presence of a large dataset that captures everything different from the ID data. Unsupervised ones [7], [8] only utilize ID data for training, but are likely to suffer from poor performance. Recent studies have demonstrated that weak supervision [9]–[12] can remarkably outperform unsupervised learning methods for anomaly/OOD detection. Some weakly-supervised approaches [10]–[12] generate pseudo-labeled OODs by partially corrupting ID data based on output attention mappings, while others [9] leverage a tiny collection of labeled OODs (*e.g.,* 1% of ID data) to detect specific OOD types in the applications where access to OOD samples is limited and pseudo OOD generation is challenging [13]. However, none of these ML approaches have been applied in the context of SE for code-related tasks.

Existing OOD detection research in SE primarily focuses on anomaly detection or software defect detection. Anomaly detection techniques [14]–[17] are designed to detect anomalous system states (*e.g.,* failed processes, availability issues, security incidents) during system running based on *monitoring data* (*e.g.,* logs, traces), but they still cannot been applied to the code context. There also exists a body of research dedicated to detecting suspicious defects in *source code* (*e.g.,* vulnerability detection [18]–[20], neural bug detection [21], [22]). Although defective source code represents a type of distribution shifts from normal code, current defect detection techniques are not sufficient to cover a broad range of unseen scenarios considered by OOD detection.

Therefore, the goal of this work is to address the OOD detection problem in the context of SE for code-related tasks. While Transformer-based [23] NL-PL (programming

[§]Equal contribution.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00177

language) models have shown remarkable success in code understanding and generation [24]–[26] by utilizing bimodal data (*i.e.,* comment and code), they often assume training and testing examples belong to the same distribution. Thus, these models may not guarantee the robustness against OOD instances in the open world (as evidenced by [27] for NL Transformers). For instance, a code search engine, which is trained on GitHub comment-based queries and code, is likely to fail in user questions and code answers from StackOverflow.

In this paper, we systematically investigate the ability of pre-trained NL-PL models [24], [25], [28] in detecting OOD instances and the impact of OOD detection on a downstream code task (*i.e.,* code search). While NLP OOD detection techniques show promise for adaptation to NL-PL models due to the similarity between NL and PL, they can only detect textual OODs from uni-modal data. However, in the SE context for code-related tasks, distribution shifts can occur in either modality (comment or code) or both of them. An effective OOD code detector should be able to detect OOD from comments, code, or both modalities, by utilizing multi-modal NL-PL pairs. Several multi-modal approaches have been proposed for vision OOD detection [5], [29], utilizing information from both images and their textual descriptions, but they are still designed to detect *only* visual OODs.

To overcome these challenges, we develop two types of multi-modal OOD detection models to equip NL-PL models with OOD code detection capability. The first one is unsupervised (coined as COOD), which fine-tunes the NL-PL models to closely align NL-PL representations solely from ID data [30] based on the multi-modal contrastive learning [31], and then uses their prediction confidences as OOD scores. The contrastive learning objective is expected to effectively capture high-level alignment information within (NL, PL) pairs to detect OODs. To further enhance the OOD detection performance, we propose a weakly-supervised OOD detection model, COOD+, which utilizes a tiny collection of OOD samples (*e.g.,* 1%) during model training. Current techniques in ML typically considered unsupervised contrastive learning [8] or outlier exposure [6], [32], in conjunction with a scoring function, limiting their ability to detect OODs from just one modality. In contrast, our COOD+ integrates an improved contrastive learning module with a binary OOD rejection module in order to effectively detect OODs from NL, PL, or both modalities. OOD samples are then identified by a combination of two different scoring functions: the confidence scores produced by the contrastive learning module and the prediction probabilities of the binary OOD rejection module.

Due to the lack of evaluation benchmarks for OOD code detection, we create a new benchmark tailored for code context following the construction principles in ML [7], [8], but containing more OOD scenarios: (1) aligned (NL, PL) pairs collected from a new domain, *e.g.,* from StackOverflow rather than GitHub, (2) misaligned (NL, PL) pairs, (3) the presence of syntactic errors in NL descriptions, and (4) buggy source code. We first evaluate the proposed models on two real-world datasets, CodeSearchNet-Java and CodeSearchNet-Python, and establish a range of unsupervised and weakly-supervised baselines for comparison. Experimental results show that both COOD and COOD+ models significantly outperform the best unsupervised and weakly-supervised baselines, respectively. Specifically, our unsupervised COOD is moderately capable of detecting OODs from three scenarios but does not perform well across all four scenarios. By integrating two modules, our COOD+ model effectively detects OODs from all scenarios simultaneously.

Furthermore, we apply our approaches to improve the robustness of existing (NL, PL) models for the code search task under the four OOD scenarios described above. By corrupting 15% of the testing dataset with OOD examples, we demonstrate that NL-PL models actually are not robust to OOD samples. Specifically, the performance of a fine-tuned GraphCodeBERT code search model drops by around 5% due to the presence of OODs. Subsequently, we filter the corrupted testing dataset with our COOD/COOD+, and show that our detectors successfully recover this performance loss and also improve the code search performance compared to the original testing set. In summary, the contributions of this paper are:

- A novel OOD benchmark specifically designed for code contexts, encompassing multiple OOD scenarios;
- The first work to address OOD detection for code across four distinct scenarios;
- A multi-modal OOD detection framework for NL-PL pre-trained models, leveraging contrastive learning in both unsupervised and weakly-supervised settings;
- A comprehensive evaluation showcasing the superior performance of our COOD and COOD+ frameworks in detecting OOD samples across four scenarios;
- An online appendix providing the full codebase and experimental infrastructure of our approaches [33].

II. Related Work

We review the related work on OOD detection in various fields such as computer vision (CV), natural language processing (NLP), and software engineering (SE), and then point out unique characteristics of our approach.

A. OOD Detection in SE

To ensure the reliability and safety of large-scale software systems, extensive work [14], [34], [35] has been conducted on anomaly detection to identify anomalous system state (*e.g.,* failed processes, availability issues, security incidents) during system running based on `monitoring data` (not in code format). Specifically, monitoring data includes logs [14], [15], metrics (*e.g.,* response time, CPU usage) [36], traces [16], [17], etc. While some approaches utilize supervised learning techniques [37], [38], others employ unsupervised [39] or semi-supervised learning [40], [41] due to insufficient anomaly labels. However, none of these anomaly detection techniques target code-based OOD detection, the main focus of our work. We mention this research line here since some existing OOD-related work in ML use the terms *anomaly detection* and *(generalized) OOD detection* interchangeably [6], but anomaly detection in SE has distinct characteristics as described above.

Additionally, current defect detection techniques [42] in SE typically identify defects by analyzing `source code` with code semantic features extracted. Research in vulnerability detection focuses on security-related defects, such as buffer overflows and use-after-free. Compared to conventional static tools [43], [44], DL-based techniques [18]–[20], [45] utilize Graph Neural Networks (GNNs) or Transformers to learn implicit vulnerability patterns from source code. Additionally, bug detection techniques [21], [22], [46], [47] also fall under the umbrella of defect detection but typically address semantically-incorrect code (*e.g.,* wrong binary operators, variable misuse) which is not necessarily security-related and probably syntactically feasible. Although our focus is also on source code, defective code is only considered as one scenario within the scope of our OOD detection problem.

More recently, several research has explored the robustness and generalization of source code models to different OOD scenarios [48]–[50]. Hu *et al.* introduced a benchmark dataset to assess the performance of code models under distribution shifts [48], while others investigated fine-tuning strategies like low-rank adaptation [50] and continual learning [49] for enhanced *generalization* on OOD data. However, these studies did not specifically tackle OOD *detection*, and existing unsupervised OOD detectors have shown limited effectiveness for source code data [48]. In short, unlike prior work, our study directly aims to improve the OOD detection performance of existing code-related models, ensuring greater robustness and trustworthiness in the open world where many unseen OOD scenarios may be encountered.

B. OOD Detection in CV and NLP

In the ML community, OOD detection [3], [4], [51], [52] has been extensively studied over the years, leading to a better-defined and formulated task. The primary objective of OOD detection here is to design an auxiliary ID-OOD classifier derived from neural-based visual and/or textual models based on OOD scores. Given that correctly predicted instances tend to have greater maximum softmax probabilities (MSP) than incorrectly predicted and OOD instances, MSP-based OOD scoring function [51], [53] weree initially utilized to identify OOD samples. Subsequently, energy- and distance-based scores [8], [32], [54] have also been utilized to derive OOD scores. For visual OOD data, existing techniques often aim for multi-class classification tasks (*e.g.,* image classification) and learn a $K + 1$ classifier assuming that the unseen space is included in the additional class [55], [56]. The OOD data utilized for evaluation is typically constructed from a completely different dataset (out-domain data) or by holding out a subset of classes in a categorized dataset, where one category is considered normal and the remaining categories are treated as OOD.

In the context of textual data, OOD detection techniques are applied to both classification tasks (*e.g.,* sentiment/topic classification [8], [12]) and selective prediction tasks [57]–[59] (*e.g.,* question answering, semantic equivalence judgments). These techniques rely on various algorithmic solutions including outlier exposure [55], [60], data augmentation [61], [62], contrastive learning [8], [63], *etc.*. Compared to traditional neural-based language models, pre-trained Transformer-based [23] models exhibit greater robustness to distributional shifts and are more effective in identifying OOD instances [27], [64]. Besides the out-domain data, text-based OOD detection also consider syntactic OOD data [7] due to the intrinsic characteristics of sentences. Syntactic OOD and ID data come from the same domain, but the syntactic OOD data has its word order shuffled, which allows for the measurement of OOD detectors' sensitivity to underlying syntactic information while preserving word frequency.

Some studies [65], [66] have explored the incorporation of multi-modal data into neural-based models to improve OOD detection accuracy. Recently, CLIP-based methods [5], [29], [67] have emerged as a promising approach for OOD detection by leveraging vision-language bimodal data, exhibiting superior performance over uni-modal data only. The main intuition behind these approaches is to take advantage of the alignment between visual classes or concepts and their textual descriptions. For instance, Ming et al. [5] detect visual OOD in an unsupervised manner by matching visual features with known ID concepts in the corresponding textual descriptions.

However, these studies typically focus on detecting OOD data from at most two scenarios (*i.e.,* out-domain and shuffled-text OODs) within a *single* modality. Even multi-modal approaches are often limited to detecting only visual OODs by additionally considering accompanying textual descriptions. Our proposed approach aims to effectively identify OOD samples from four distinct scenarios across *two* modalities (*i.e.,* NL and PL). To achieve this, we utilize a combination of different scoring functions from two different modules: cosine similarities of a contrastive learning module and prediction probabilities of a binary OOD classifier.

III. Preliminaries

Below we review some background knowledge and techniques that will be used in the proposed approach.

A. Code Representation Learning

Representation learning refers to the process of learning a parametric mapping from the raw input data domain to a low-dimensional latent space, aiming to extract more abstract and useful concepts that can improve performance on a range of downstream tasks. Recently, self-supervised representation learning has become prominent in the ML community due to the success of large pre-trained models. The similarity between NL and PL has led to the development of numerous NL-PL pre-trained models [24], [25], resulting in significant improvements across various code-related tasks such as code search, generation, and clone detection. CodeBERT [68] stands as the first Transformer-based NL-PL pre-trained model, while GraphCodeBERT [24] further enhances its performance by considering the inherent structure of code (*i.e.,* data flow) during pre-training. Subsequent models like UniXcoder [25] and ContraBERT [28] further enhance the code understanding capabilities. In our research study, we are concerned with extending self-supervised NL-PL representation learning to the OOD detection task, which is outside the scope of existing pre-trained models, but is crucial for ensuring the trustworthiness of these models in real-world applications.

B. Multimodal Contrastive Learning

Contrastive learning [69] is an emerging technique that learns discriminative representations from data that are organized into similar/dissimilar pairs. It has been developed over multiple sub-fields, such as NLP, CV [70], [71], and SE [25], [72]. Recently, researchers have developed multi-modal approaches that combine contrastive learning with multiple data modalities, achieving superior performance over uni-modal modals in various tasks [28], [73]. Due to the prowess of contrastive learning in discriminative tasks (*e.g.,* classification and information retrieval), it naturally fits the OOD detection domain, as shown by studies on both uni-modal and multi-modal data [5], [29], [74]. Inspired by this, we apply contrastive learning to NL-PL data to learn representative and discriminate features for both ID and OOD instances and transfer this knowledge to train our OOD detector.

IV. Approach

In this section, we first formally define the OOD code detection problem for (NL, PL) models (Sec. IV-A), then introduce the overall proposed framework (Sec. IV-B), and finally present details of unsupervised COOD and weakly-supervised COOD+ in Sec. IV-C and Sec. IV-D, respectively.

A. Problem Statement

Since current state-of-the-art code-related models [24], [25] typically extract code semantics by capturing the semantic connection between NL (*i.e.,* comment) and PL (*i.e.,* code) modalities, we formally defined OOD samples involving these two modalities in the SE context by following the convention in ML [3], [8]. Consider a dataset comprising training samples $((t_1, c_1), y_1), ((t_2, c_2), y_2), ...$ from the joint distribution $P((T, C), Y)$ over the space $(\mathcal{T}, \mathcal{C}) \times \mathcal{Y}$, and a neural-based code model is trained to learn this distribution. Here, $((t_1, c_1), y_1)$ represents the first input pair of (comment, code) along with its ground-truth prediction in the training corpus. T, C and Y are random variables on an input (comment, code) space $(\mathcal{T}, \mathcal{C})$ and a output (semantic) space $\mathcal{Y}$, respectively. OOD code samples refer to instances that typically deviate from the overall training distribution due to distribution shifts. The concept of distribution shift is very *broad* [3], [52] and can occur in either the marginal distribution $P(T, C)$, or both $P(Y)$ and $P(T, C)$.

We then formally define the OOD code detection task following [9], [12], [32], [75] as follows. Given a main code-related task (*e.g.,* clone detection, code search, *etc.*), the objective here is to develop an *auxiliary* scoring function $g : (\mathcal{T}, \mathcal{C}) \rightarrow \mathcal{R}$ that assigns higher scores to normal instances where $((t, c), y) \in P((T, C), Y)$, and lower scores to OOD instances where $((t, c), y) \notin P((T, C), Y)$. Based on whether to use OOD instances during the main-task training of pre-trained NL-PL models, we define OOD for code in two settings, namely unsupervised and weakly-supervised learning. For the unsupervised setting, only normal data is used in the main-task training. Conversely, weakly-supervised approaches utilize ID and a tiny collection of OOD data (*e.g.,* 1% of ID data) [9] in training. In this context, the output space $\mathcal{Y}$ is typically a binary set, indicating normal or abnormal, which is probably unknown during inference. Due to the small number of training OOD data, the OOD samples required by our COOD+ and other existing weakly-supervised approaches [9], [76] in ML can be generated at minimal cost and feasibly verified by human experts when necessary.

B. Overview

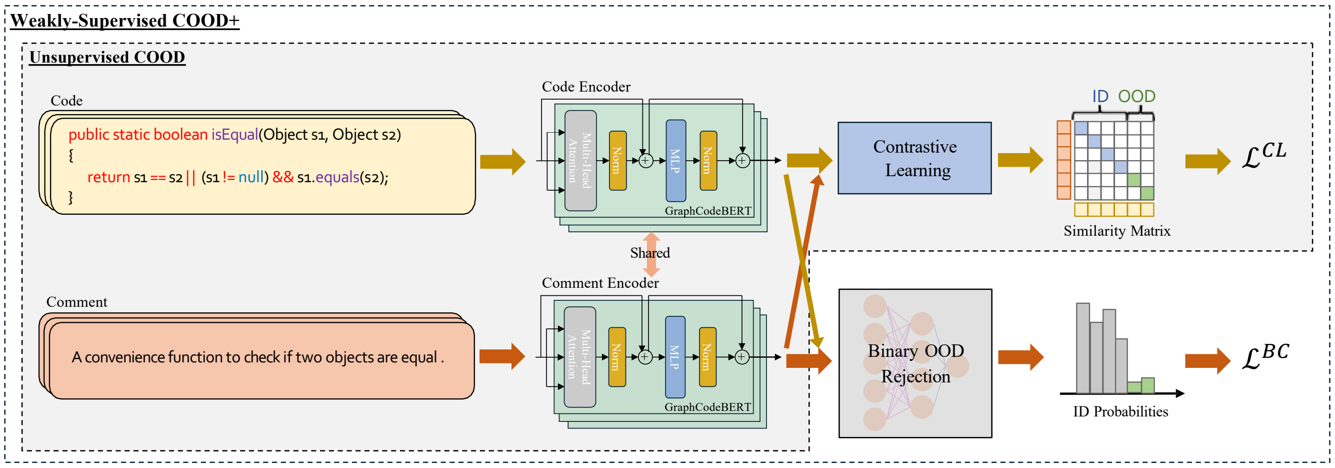
Overall, there are two versions of our COOD approach: unsupervised COOD and weakly-supervised COOD+. Given a multi-modal (NL, PL) input, the unsupervised COOD learns distinct representations based on a contrastive learning module by utilizing a pre-trained Transformer-based code representation model (*i.e.,* GraphCodeBERT [24]). Then, these representations are mapped to distance-based OOD detection scores in order to indicate whether the test samples are OODs during inference. The weakly-supervised COOD+ further integrates a improved contrastive learning module with a binary OOD rejection module to enhance the detection performance by using a very tiny number of OOD data during model training. The OOD samples are then identified by the detection scores produced by the contrastive learning module as well as the prediction probabilities of the binary OOD rejection module.

C. Unsupervised COOD

Our unsupervised COOD approach consists of a contrastive learning (CL) module trained only on ID samples. Specifically, given (comment, code) pairs as input, we fine-tune a comment encoder and a code encoder through a contrastive objective to learn discriminative features, which are expected to help identify OOD samples based on a scoring function.

The (comment, code) pairs are first converted into the comment and code representations, which are processed by the comment and code encoder, respectively. We use the pre-trained GraphCodeBERT model [24] as the encoder architecture (*i.e.,* backbone). GraphCodeBERT is a Transformer-based model pre-trained on six PLs by taking the (comment, code) pairs as well as the data flow graph of the code as input, which has shown superior performance on code understanding and generation tasks. All the representations of the last hidden states of the GraphCodeBERT encoder are averaged to obtain the sequence-level features of comment and code.

Contrastive Learning Module. To achieve the contrastive learning objective, we fine-tune the base (GraphCodeBERT) encoders with the InfoNCE loss [31]. The comment and code encoders follow the Siamese architecture [25] since they are designed to be identical subnetworks with the same



```
public static boolean isEqual(Object s1, Object s2)
{
    return s1 == s2 || (s1 != null) && s1.equals(s2);
}
```

A convenience function to check if two objects are equal .

Fig. 1: The Overview of Our Proposed COOD and COOD+ Approaches for OOD Detection

GraphCodeBERT backbones, in which their parameters (*i.e.,* weights and biases) are shared during fine-tuning. Parameter sharing can reduce the model size and has shown state-of-the-art performance for the code search task [77]. To extract discriminative features for (comment, code) pairs, we organize them into functionally-similar positive pairs and dissimilar negative (unpaired) pairs. Through a contrastive objective, positive pairs are drawn together, while unpaired comment and code are pulled apart. Specifically, for each positive (comment, code) pair (t_i, c_i) in the batch, the code in each of other pairs and t_i are constructed as in-batch negatives, similarly for the comment side. The loss function then formulates the contrastive learning as a classification task, which maximizes the probability of selecting positives along the diagonal of the similarity matrix (as shown in Fig. 1) by taking the *softmax* of projected embedding similarities across the batch. The loss function can be summarized as follows:

$$\begin{aligned}\mathcal{L}^{CL} = -\frac{1}{2N}(&\sum_{n=1}^{N}\log\frac{e^{sim(v_{t_i},v_{c_i})/\tau}}{\sum_{j=1}^{N}e^{sim(v_{t_i},v_{c_j})/\tau}}\\ &+\sum_{n=1}^{N}\log\frac{e^{sim(v_{t_i},v_{c_i})/\tau}}{\sum_{j=1}^{N}e^{sim(v_{t_j},v_{c_i})/\tau}})\end{aligned} \tag{1}$$

where v_{t_i} and v_{c_i} represent the extracted features of the comment t_i and the code c_i. τ is the temperature hyperparameter, which is set to 0.07 following previous work [77]. $sim(v_{c_i}, v_{t_i})$ and $sim(v_{t_i}, v_{c_j})/sim(v_{t_j}, v_{c_i})$ represent the cosine similarities between comment and code features for positive and negative pairs, respectively. N is the number of input pairs in the batch. InfoNCE loss is designed for self-supervised learning and learns to distinguish positive pairs from in-batch negatives. Compared to other contrastive losses [8], [70], it can take advantage of large batch size to automatically construct many diverse in-batch negatives for robustness representation learning, which is more effective to capture the alignment information between comment and code.

Scoring Function. Existing OOD detection techniques in ML derive scoring functions based on model's output, which typically map the learned class-probabilistic distributions to OOD detection scores for testing samples. Maximum Softmax Probability (MSP) [78] is commonly used for OOD scoring. This method uses the maximum classification probability $\max_{l\in L} softmax(f(v_t, v_c))$, where $f(v_t, v_c)$ is the output of the classification model, with low scores indicating low likelihoods of being OOD. However, NL-PL code search models typically utilize the similarity retrieval scores of NL-PL output representations to make predictions. Therefore, to enable simultaneous similarity and OOD inference, we alternatively extract cosine similarity scores of testing NL-PL pairs as OOD detection scores, denoted as $P^{CL} = sim(v_c, v_t)$. The underlying intuition behind this scoring metric is that OOD testing samples should receive low retrieval confidence from the model fine-tuned on ID data, which establishes a closer relationship between ID (comment, code) pairs. Hence, this scoring function also assigns higher scores to ID data and lower scores to OOD data similar to previous scoring methods.

D. *Weakly-Supervised COOD+*

To further enhance the performance of unsupervised COOD, we extend it to a weakly-supervised detection model, called COOD+, which takes advantage of a few OOD examples. Inspired by [79], our COOD+ combines an improved contrastive learning (CL) and a binary OOD rejection classifier (BC). The improved CL module adopts a margin-based loss [80] which enforces a margin of difference between the cosine similarities of aligned and unaligned (comment, code) pairs, and constrains the cosine similarities of OOD pairs below another margin. The BC module integrates features from both comments and code to calculate the probabilities of OOD pairs. The OOD scoring function is then designed by combining the cosine similarity scores from the CL module and the prediction probabilities from the BC module. Below, we detail each component of our weakly-supervised COOD+.

Improved Contrastive Learning (CL) Module. Given a batch of N input pairs (comprising $N-K$ ID pairs and K OOD pairs), the latent representations are first obtained from the comment and code encoders. Then the margin-based loss is leveraged in the CL module to distinguish representations of ID and OOD data by constraining the cosine similarity. Specifically, the margin-based contrastive loss is first applied to $N-K$ ID code to maximize the difference between aligned (comment, code) pairs and incorrect pairs for each batch:

$$\mathcal{L}^{ID} = \sum_{i=1}^{N-K} \left(\frac{1}{N} \sum_{j=1, j \neq i}^{N} \max\left(0, m - s(v_{t_i^+}, v_{c_i^+}) + s(v_{t_j^-}, v_{c_i^+})\right) \right) \quad (2)$$

$s(v_{t_i^+}, v_{c_i^+})$ represents the cosine similarity of representations between each aligned ID pair from all the $N-K$ aligned pairs, and $s(v_{t_j^-}, v_{c_i^+})$ represents the cosine similarity of representations between each ID code and all the other $N-1$ comments (*i.e.,* the comment is either not aligned with the ID code or from OOD comments). Thus, this margin-based loss encourages the difference between the aligned pairs and the incorrect pairs greater than margin m.

Regarding the K OOD code, we enforce a constraint on the cosine similarity between each OOD code and all the comments, ensuring that the similarity remains below a margin m. This constraint is necessary because each OOD code should not align with its corresponding comment, nor with any of the other $K-1$ OOD comments and the $N-K$ ID comments. The loss function is denoted as follows:

$$\mathcal{L}^{OOD} = \sum_{k=1}^{K} \left(\frac{1}{N} \sum_{i=1}^{N} \max\left(0, -m + sim(t_j^-, c_k^-)\right) \right), \quad (3)$$

where $sim(t_j^-, c_k^-)$ represents the cosine similarity between each of the K OOD code and all N comments. Finally, the overall loss for the contrastive module can be expressed as:

$$\mathcal{L}^{CL} = \frac{1}{N} \left(\mathcal{L}^{ID} + \mathcal{L}^{OOD} \right). \quad (4)$$

Binary OOD Rejection (BC) Module. Besides the CL module, we also introduce a classification module under weakly-supervision for identifying OOD samples. Inspired by the Replaced Token Detection (RTD) objective utilized in [68], we bypass the generation phase since our OOD data are generated prior to training. Therefore, we directly train a rejection network responsible for determining whether (comment, code) pairs are OOD or not, which can be framed as a binary classification problem. Our binary OOD rejection network comprises a 3-layer fully-connected neural network with *Tanh* activation, and the input is based on the concatenation of features from the comment and code encoders: $v_i = (v_{t_i}, v_{c_i}, v_{t_i} - v_{c_i}, v_{t_i} + v_{c_i})$. Apart from utilizing the comment and code features, we also incorporate feature subtraction $v_{t_i} - v_{c_i}$ and aggregation $v_{t_i} + v_{c_i}$. Additionally, we apply the sigmoid function to the output layer, producing a prediction probability that indicates whether the sample is OOD. We then use binary cross entropy loss for this module:

$$\mathcal{L}^{BC} = \frac{1}{N} \sum_{i=1}^{N-K} (y_i \log p(v_i) + (1 - y_i) \log(1 - p(v_i))), \quad (5)$$

where $p(v_i)$ is the output probability of the BC module, and $y_i \in [0, 1]$ is the ground-truth label. $y_i = 1$ indicates the input sample is an inlier, while $y_i = 0$ signifies it is an outlier.

Hence, for weakly-supervised COOD+, we combine the objectives of the CL and the BC modules to jointly train our model, where λ is a weight used to balance the loss functions:

$$\mathcal{L} = \mathcal{L}^{CL} + \lambda \mathcal{L}^{BC}. \quad (6)$$

Combined Scoring Function. Similar to the unsupervised COOD approach, we utilize the diagonals of the similarity matrix as the OOD detection scores obtained from the CL module. To further improve the detection performance of the weakly-supervised version, we combine these P^{CL} scores with the output probabilities of the BC module, denoted as P^{BC}. Here, we convert cosine similarity scores into probabilities using the sigmoid function $P^{CL*} = \sigma(sim(v_c, v_t))$, then use multiplication to create the overall scoring function, yielding $P^{ID} = P^{CL*} \times P^{BC}$. We anticipate that higher scores will be assigned to ID pairs, while lower scores will be assigned to OOD pairs. This combined scoring function aims to enhance the discrimination between inliers and outliers, leading to more effective OOD detection.

V. Empirical Evaluation Design

To evaluate the performance of the proposed approaches in four scenarios, we investigate the following research questions:

RQ$_1$: *How effective is our unsupervised COOD when compared to unsupervised baselines?*

RQ$_2$: *How effective is our weakly-supervised COOD+ when compared to weakly-supervised baselines?*

RQ$_3$: *How effective is our weakly-supervised COOD+ when using different modules or encode backbone?*

RQ$_4$: *Is the main task (Code Search) performance affected by our COOD/COOD+ auxiliary, and to what extent?*

A. Datasets

In our experiments, we rely on two benchmark datasets: CodeSearchNet (CSN) [24] and TLCS [81]. CSN contains bimodal data points consisting of code paired with function-level NL descriptions (*i.e.,* first lines of documentation comments) in six PLs (*e.g.,* Python, Java) collected from GitHub repositories. While CSN was originally created for a specific downstream task (*i.e.,* code search), it has since been widely adopted by large (NL, PL) models [24], [25] for pre-training due to the informative nature of bimodal instances. Large NL-PL models are first pre-trained across *all* six languages, and then further fine-tuned for a *specific* PL for some downstream task to enhance performance. For code search, the goal is to retrieve the most relevant code given a NL query, where CSN is widely used to further fine-tune a PL-specific code search model [68].

Salza *et al.* [81] used training samples from CSN for pre-training, and created a new dataset sourced from StackOverflow (SO) for fine-tuning the code search model, involving only *three* PLs: Java, Python, and JavaScript. Specifically, they leverage SO user questions as search queries and accepted answers as retrieved code snippets, which differ from GitHub comments and the corresponding code in CSN. We refer to this new dataset as TLCS. Existing work [82]–[84] investigated code clones between SO and GitHub, demonstrating there exists *only* 1-3% code reuse. Besides code, user questions in SO are typically formulated before code answers, without concrete knowledge of what code answers will be, and are mostly written by end-users. Conversely, in GitHub, method docstrings (*i.e.,* comments) are often written following code snippets, and are mostly written by developers. These distinctions cause performance shortfall when directly applying models trained on CSN to TLCS without further fine-tuning or transfer learning [25], [81], [85].

B. OOD Scenarios

We design four distinct OOD scenarios using the datasets described above, with CSN-Java and CSN-Python as inliers due to their common use for the pre-training of code models.
Scenario 1: Out-domain. Following existing ML work [8], [27], [86], we create an out-of-domain setting by choosing OOD samples from a different dataset than the training data. Thus, samples from TLCS-Java or TLCS-Python are treated as outliers accordingly. Inliers and their corresponding outliers belong to the same PL to ensure approaches don't identify OODs based on syntax differences between PLs but on data domains: GitHub vs. SO. Prior studies [87] show that CSN queries are longer than SO questions on average, so we sampled TLCS questions and answers to match the length distribution of CSN comments and code, to avoid OOD approaches exploiting spurious cues of query length differences. We didn't consider other code search datasets [88]–[90] because they either contain only one of the PLs (Python or Java) or have a smaller dataset size.
Scenario 2: Misaligned. In this scenario, we shuffle normal NL-PL pairs so that each code doesn't match its NL description. Although the NL modality sourced from attached comments in code are typically aligned with the PL modality, documentation errors may still occur and not effectively filtered by handcrafted rules [24].
Scenario 3: Shuffled-comment. For (comment, code) pairs, we modify the syntactic information in each comment by shuffling 20% of selected tokens using a seeded random algorithm [91] with positions of stopwords and punctuations unchanged. No changes are made to the code for this scenario. This scenario is inspired by [7], [92]. [92] discovered that NL pre-trained models are insensitive to permuted sentences, which contrasts with human behavior as humans struggle to understand ungrammatical sentences, or interpret a completely different meaning from a few changes in word order. [7] further introduces syntactic (shuffling) outliers into NL pre-training corpora to enhance OOD robustness and NL understanding performance.
Scenario 4: Buggy-code. We create buggy code using a *semantic* mutation algorithm which injects more natural and realistic bugs into code than other traditional loose/strict mutators [93]. This simulates buggy programs that the model may encounter during testing, typically absent from the training dataset, and should be taken into account by OOD code detectors according to the OOD definition [22]. We avoid using real bug/vulnerability datasets [94]–[96] due to limitations like the absence of paired comments, lack of support for Python or Java, introduction to a new dataset domain *etc.*. We generate buggy code for each code in CSN-Java and CSN-Python using [93] to serve as outliers, ensuring the inliers and outliers are from the same dataset domain with the only difference being normal vs. buggy code. We focus on variable-misuse bugs, as only this mutation algorithm is available for both Python and Java in [93]. Variable-misuses occur when a variable name is used but another was meant in scope, and often remain undetected after compilation and regarded as hard-to-detect by recent bug detection techniques [22], [97]. Comments remain unchanged for this scenario.

C. Model Configurations

For the weakly-supervised COOD+, we experiment with either the contrastive learning module (COOD+_CL) or the binary OOD rejection module (COOD+_BC) to compare against the combined model. All models are trained using the Adam optimizer with a learning rate of $1e-5$, and a linear schedule with 10% warmup steps. The batch size is set to 64, and the number of training epochs is 10. For the COOD+_CL and COOD+, the margins in the margin-based loss are set to 0.2. The balancing value λ is set to 0.2 after a grid search. The hidden layer size in the binary OOD rejection module for COOD+ is 384 ($768/2$). We also explore the robustness and agnosticism of our COOD+ approach to different NL-PL models by replacing the GraphCodeBERT encoder with CodeBERT [68], UniXcoder [25], and ContraBERT [28].

D. OOD detection model training and measurement

For unsupervised COOD, we use only ID data for model training, thus involving all training data from CSN-Python and CSN-Java, with 10% randomly sampled for validation. We avoid using the CSN development dataset for validation due to its smaller size. For weakly-supervised COOD+, we randomly select 1% of the training data and replaced them with OOD samples generated for each scenario (following [9]), resulting in a total of 4% OOD samples and 96% ID samples for training. During inference, both COOD and COOD+ utilize the same ratio (20%) for inliers and outliers from each scenario, which is more convincing than using an imbalanced dataset (*i.e.,* tiny number of OOD data). Detailed dataset statistics are provided in our online appendix [33]. Since all outliers are randomly selected, we report average OOD detection results across *five* random seeds of the test dataset to ensure evaluation reliability and reproducibility.

Following prior work in ML [56], [98], we use two standard metrics to measure the effectiveness of our COOD/COOD+ models: the area under the receiver operating characteristic curve (AUROC) and the false positive rate at 95% (FPR95). AUROC is threshold-independent, calculating the area under the ROC curve over a range of threshold values, representing the trade-off between true positive rate and false positive rate. It quantifies the probability that a positive example (ID sample) receives a higher score than a negative one (OOD sample). Higher AUROC indicates better performance. Additionally, FPR95 corresponds to the false positive rate (FPR) when the true positive rate of ID samples is 95%. FPR95 is threshold-dependent, where OODs are identified by setting a threshold σ with $P^{OOD} < 1 - \sigma$ ($P^{ID} > \sigma$) so that a high fraction (95%) of ID data is above the threshold. It measures the proportion of OOD samples that are mistakenly classified when 95% of ID samples are correctly recalled based on the threshold. Lower FPR95 indicates better performance.

E. Baselines

We compare our COOD/COOD+ against various OOD detection baselines, including adaptations of existing unsupervised NLP OOD approaches on NL-PL encoders (1-2), weakly-supervised approaches based on outlier exposure (3), and neural bug detection techniques (4-5). Since unsupervised approaches (1-2) rely on classification outputs for OOD scoring, we reformulate code search as binary classification to fine-tune the encoders similarly to [68]. (1-2) is supervised for code search, but unsupervised for OOD detection. For weakly-supervised baselines (3-5), we use the same number of OOD samples as COOD+ for a fair comparison. Note that the encoder backbone of (1-3) is also GraphCodeBERT. (4-5) are specifically designed for neural bug detection, thus not requiring other encoder backbone for OOD detection.

1) **Supervised Contrastive Learning For Classification (SCL)** [70]. This method fine-tunes transformer-based classification models by maximizing similarity of input pairs if they are from the same class and minimize it otherwise. Following [8], we adopts MSP, Energy, and Mahalanobis OOD scoring algorithms for OOD detection.
2) **Margin-based Contrastive Learning for Classification (MCL)** [8]. This approach fine-tunes transformer-based classification models by minimizing the L2 distances between instances from the same class, and encouraging the L2 distances between instances of different classes to exceed a margin. We also detect OODs by applying MSP, Energy, and Mahalanobis OOD scoring algorithms.
3) **Energy-based Outlier Exposure (EOE)** [32]. This approach uses a few auxiliary OOD data to fine-tune the classification model with an energy-based margin loss [32], and then utilize Energy scores for OOD detection.
4) **CuBERT** [46]. CuBERT is pre-trained on a large code corpus using masked language modeling, then fine-tuned for bug detection and repair. We adapt CuBERT for OOD classification by alternatively fine-tuning it on our datasets with comments appended to their corresponding code, as CuBERT only accepts single instance inputs.
5) **2P-CuBERT** [22]. This method enhances CuBERT's bug detection accuracy with a two-phase fine-tuning approach. The first phase utilizes contrastive learning on generated synthetic buggy code [21]. For the second phase, we alternatively fine-tune CuBERT to detect OOD using our datasets. Results are reported only for CSN-Python due to the lack of Java bug generation algorithms in [22].

F. Main Task Performance Analysis

An effective OOD detector, serving as an auxiliary component, should identify and reject OOD samples without negatively impacting the original model's performance on the main downstream task with ID data [8]. Consequently, we validate the effectiveness of our COOD/COOD+ auxiliary on the code search task using the official evaluation benchmark [24], [28] by calculating the mean reciprocal rank (mRR) for each pair of comment-code data over distractor codes in the testing code corpus. Specifically, we first measure the performance of original GraphCodeBERT code search model on both ID and OOD data, whose performance is expected to be negatively affected with the presence of OOD samples. Then, we utilize our COOD/COOD+ auxiliary to filter the testing dataset by setting a threshold to retain 95% of ID instances with higher scores (following existing ML work [5] and the FPR95 definition), as real-world deployment typically involves few OODs. Finally, we directly use the fine-tuned encoder in COOD/COOD+ to perform code search but on the retained ID instances, and compare this performance with that on the ground-truth ID instances. If the performance loss is recovered by using COOD/COOD+, we actually enhance the trustworthiness and robustness of the original code search model (as shown in Sec. VI-D). Here trustworthiness and robustness mean that predictions of code models become more reliable when encountering OOD data in real-world deployment. Note that the dataset used for COOD/COOD+ training is the same as that used for PL-specific training of existing SOTA code search models.

VI. Experimental Results

A. RQ1: Unsupervised COOD Performance

In this subsection, we analyze the experimental results to assess the detection performance of our unsupervised COOD model compared with the unsupervised baselines. According to Table I and II, we can observe that COOD outperforms all unsupervised baselines on both CSN-Python and CSN-Java. Notably, COOD effectively detect *out-domain* and *misaligned OOD* testing samples, while other unsupervised approaches only work for the *out-domain* scenario. This is because COOD effectively captures alignment information within (comment, code) pairs through a multi-modal contrastive learning objective with InfoNCE loss and uses similarity scores between comments and code to detect OODs. Specifically, COOD outputs low similarity scores for the out-domain data from

TABLE I: Effectiveness of our COOD and COOD+ models compared with the baselines on the CSN-Python dataset.

Approaches	Out-domain+ID		Misaligned+ID		Shuffled-comment+ ID		Buggy-code+ ID		Overall (All OODs+ID)	
	AUROC↑	FPR95↓	AUROC↑	FPR95↓	AUROC↑	FPR95↓	AUROC↑	FPR95↓	AUROC↑	FPR95↓
Unsupervised										
SCL+MSP	78.21	68.93	49.80	97.23	60.91	90.36	49.23	95.26	60.79	87.05
SCL+Energy	77.76	70.13	65.07	96.11	61.24	90.04	49.58	95.00	65.09	86.95
SCL+Maha	73.34	82.82	73.21	92.13	68.03	87.02	**53.89**	**92.43**	68.72	88.14
MCL+MSP	81.18	65.57	53.51	96.43	62.17	90.44	48.69	94.42	63.11	85.78
MCL+Energy	82.55	64.03	62.57	95.43	63.03	90.52	**48.26**	94.90	66.02	85.16
MCL+Maha	53.05	94.60	62.47	93.12	49.23	95.73	51.13	93.64	54.31	94.35
COOD	**86.60**	**48.25**	**99.85**	**0.16**	**72.82**	**85.18**	49.17	**93.58**	**80.50**	**52.31**
Weakly-supervised										
EOE	**98.96**	**3.26**	91.18	50.03	**98.37**	**4.07**	94.78	24.69	95.95	20.02
CuBERT	92.56	14.46	91.13	17.33	88.92	21.74	60.93	77.73	86.11	27.38
2P-CuBERT	92.26	15.16	84.42	30.83	86.38	26.92	92.87	13.94	88.51	22.65
COOD+	98.80	4.30	**99.53**	**0.40**	98.02	6.52	**97.90**	**5.96**	**98.64**	**4.09**
COOD+_CL	93.91	25.44	99.89	0.17	82.57	72.97	52.56	93.99	85.83	42.57
COOD+_BC	96.49	8.67	74.27	61.38	97.53	5.68	95.71	10.95	90.43	22.98

TABLE II: Effectiveness of our COOD and COOD+ models compared with the baselines on the CSN-Java dataset.

Approaches	Out-domain+ID		Misaligned+ID		Shuffled-comment+ ID		Buggy-code+ ID		Overall (All OODs+ID)	
	AUROC↑	FPR95↓	AUROC↑	FPR95↓	AUROC↑	FPR95↓	AUROC↑	FPR95↓	AUROC↑	FPR95↓
Unsupervised										
SCL+MSP	85.15	63.66	58.93	95.76	58.51	92.59	49.01	95.70	62.91	86.92
SCL+Energy	84.00	66.69	55.27	95.56	60.07	91.82	49.13	95.86	62.12	87.48
SCL+Maha	82.16	73.57	79.20	89.71	64.61	91.68	46.55	97.32	68.13	88.07
MCL+MSP	85.16	65.74	59.12	95.83	59.50	95.73	49.44	95.73	63.31	87.40
MCL+Energy	83.44	68.53	44.85	96.22	59.26	91.79	**49.61**	95.51	59.45	88.01
MCL+Maha	50.36	96.73	67.11	90.61	48.44	96.07	46.68	97.44	53.00	95.21
COOD	**92.27**	**40.14**	**99.41**	**0.39**	**75.78**	**86.88**	48.72	**95.04**	**79.05**	**55.95**
Weakly-supervised										
EOE	**99.49**	**1.59**	88.83	62.12	**98.71**	**3.58**	**92.27**	25.51	94.82	23.22
CuBERT	82.59	62.23	49.38	95.25	50.14	94.93	67.97	94.82	62.53	86.80
COOD+	99.29	2.79	**99.56**	**0.81**	97.05	9.43	91.11	**20.88**	**96.75**	**8.48**
COOD+_CL	93.73	23.76	99.65	0.31	83.24	78.14	50.12	95.53	82.47	47.96
COOD+_BC	98.67	3.92	64.55	78.89	95.35	10.17	94.09	14.40	88.16	26.86

TLCS by additionally considering the knowledge gap difference in (comment, code) pairs between ID and out-domain data. Also, as the misaligned scenario involves misaligned (comment, code) pairs, their similarity scores are naturally low. In contrast, the unsupervised baselines aggregate misaligned information into classification logits and rely on the confidence of the "aligned" class to detect OODs. As previously discussed in Sec. IV-C, the contrastive losses [8], [70] used by them are not as effective for learning alignment information, leading to inferior performance. Additionally, detecting token-level OOD in *shuffled-comment* and *buggy-code* scenarios proves challenging without seeing OOD samples during training, as all unsupervised methods fail to detect these OODs.

B. RQ2: Weakly-supervised COOD+ Performance

We further investigate the performance of our weakly-supervised COOD+ method against several weakly-supervised baselines on CSN-Python and CSN-Java. Table I shows that weak supervision on a tiny amount of OOD data enables COOD+ (and EOE) to not only address unsupervised COOD's shortcomings in detecting finer-grained *shuffled-comment* and *buggy-code* OODs, but also enhance performance for the *out-domain* scenario for CSN-Python. This improvement aligns with previous research [6], [12], [32] which enhances OOD detection by complementing the downstream task objective with an complementary discriminator operating to distinguish IDs from external OODs. While EOE slightly outperforms COOD+ for the *out-domain* and *shuffled-comment* scenarios by utilizing the prediction probabilities from one classification module, our COOD+, which combines the BC and CL modules, delivers consistently high performance across all four scenarios, resulting in superior overall performance. In addition, the BC module can be directly adapted to the overall COOD+ framework without modifying the underlying learning objective, but the outlier exposure-based methods (*e.g.,* EOE) typically require additional engineering (*e.g.,* determining class-probabilistic distributions [6], boundaries for energy scores [32]) to equip ML models with OOD detection abilities. Besides, the bug detection method 2P-CuBERT can reasonably detect OODs, but its performance for the *buggy-code* scenario is negatively impacted by the limited amount of training OOD examples.

On the CSN-Java dataset, our COOD+ also achieves the best overall performance compared to all baselines, despite trailing slightly behind EOE for *out-domain* and *shuffled-comment* OODs. While EOE has higher AUROC score than that of COOD+ for the *buggy-code* scenario, it suffers from a high FPR95, indicating a higher margin of error for OOD inference using a threshold of 95% ID recall. Moreover, similar to CSN-Python, CuBERT fails to detect OODs effectively on CSN-Java either, likely due to the lack of training examples. In summary, the superior performance of our COOD+ model results

TABLE III: Our COOD+ model with different encoders.

Encoders	CSN-Java		CSN-Python	
	AUROC↑	FPR95↓	AUROC↑	FPR95↓
GraphCodeBERT	**96.75**	**8.48**	**98.64**	**4.09**
CodeBERT	95.42	10.27	98.59	4.20
UniXcoder	95.49	10.63	97.83	5.91
ContraBERT	96.19	9.32	98.25	4.76

from the interplay between the CL and BL modules, where contrastive learning captures high-level alignment between NL-PL input pairs that is naturally suitable for *out-domain* and *misaligned* OODs, while the OOD rejection classifier targets lower-level OOD information from *shuffled-comment* and *buggy-code* samples. Furthermore, by utilizing a weakly-supervised contrastive learning objective that jointly optimizes for OOD detection and the code search task, our method enables effective deployment of the code search model in OOD environments, which will be further studied in Sec. VI-D.

C. RQ3: Weakly-Supervised COOD+ Performance with Different Model Components and Encoder Backbone

In this subsection, we evaluate the effect of using only the CL (COOD+_CL) or the BC module (COOD+_BC) against the proposed combined COOD+ model to illustrate how COOD+ generalizes in four OOD scenarios. As shown in Table I and II, COOD+_CL performs well in the *out-domain* and *misaligned* scenarios, which is due to its ability to effectively capture high-level (comment, code) alignment information. COOD+_BC excels in the *out-domain*, *shuffled-comment*, and *buggy-code* scenarios, since it can learn lower-level features from these types of OOD samples. While COOD+_BC maintains acceptable OOD detection performance with high AUROC (>90%) and low FPR95 (<25%), the CL module remains crucial for overall performance, since without it the overall performance of COOD+ will drop below the EOE baseline. Moreover, removing the BC module has a more negative impact on the OOD detection as COOD+ loses the ability to capture the necessary lower-level OOD information for detecting *shuffled-comment* and *buggy-code* OODs. Note that the standalone CL module performs better than the unsupervised COOD overall, demonstrating that our proposed modification to the original CL objective enhance OOD detection by leveraging the margin-based loss. Thus, the combined model's superior performance validates our design choices. That is, the combined scoring function (cosine similarities from CL and the prediction probabilities from BC) is thoughtfully designed to leverage the advantage of each module for high detection accuracy.

Moreover, we compare the detection performance of our COOD+ with various underlying NL-PL pre-trained encoder. Specifically, we compare our choice of GraphCodeBERT [24] against other NL-PL encoders from the literature including its predecessor, CodeBERT [68], and more recent ones such as UniXcoder [25] and ContraBERT [28]. As shown in Table III, all encoders perform within a 1-2% difference, indicating that our COOD+ framework is robust across different encoders.

TABLE IV: Code search performance under the impact of OOD detection. Higher numbers represent better performance

Dataset	Testing Subset	GCB	EOE	COOD	COOD+
CSN-Python	Origin	69.20	50.11	68.47	69.69
	15% outliers	65.85	43.68	64.67	65.67
	Filtered-GT	70.24	44.85	68.95	70.24
	Filtered-OOD-model	–	46.82	70.30	**73.10**
CSN-Java	Origin	69.10	46.29	68.85	69.46
	15% outliers	64.99	37.77	64.86	64.54
	Filtered-GT	69.12	38.94	69.36	69.93
	Filtered-OOD-model	–	39.33	71.02	**73.18**

This demonstrates our framework's flexibility and effectiveness in detecting OODs when deploying various NL-PL encoders for code-related tasks. Furthermore, we investigate key hyperparameters in COOD+, such as m for margin-based contrastive loss and λ in the overall loss function. The detailed results are available in our online appendix [33].

D. RQ4: Main Task Performance

We present the code search performance under the impact of OOD instances by using GraphCodeBERT (GCB), COOD/COOD+, and the closest competitor EOE in Table IV. As described in Sec. V-F, we use the official metric mRR and follow the same testing scheme as the original GraphCodeBERT code search model for evaluation. From Table IV, we first observe that our COOD/COOD+ achieves performance comparable to GraphCodeBERT, while the EOE suffers from a significant reduction in performance, as it reformulates code search as binary classification to gain OOD detection ability. This reveals a critical trade-off between OOD detection and downstream task performance. To further validate the importance of OOD detection for code search, we construct outliers based on the CSN-Java and -Python testing dataset, respectively. Given that code search aims to retrieve the most aligned code from a code corpus given an NL query, the outliers are only sampled from three OOD scenarios: *out-domain*, *shuffled-comment* and *buggy-code*, each replacing 5% ID data of the original testing set. We then show the results when the dataset contains 15% OOD samples (*i.e.,* 15% outliers), discard OOD samples by filtering the testing set by ground-truth labels (*i.e.,* Filtered-GT) or using various OOD detection models (*i.e.,* Filtered-OOD-model). Note that the Filtered-GT dataset is the original CSN's subset with 15% of ID samples removed.

According to Table IV, the performance of the original GraphCodeBERT code search model drops by 4.84% and 5.95% ((69.10-64.99)/69.10) mRR when outliers are present in CSN-Python and -Java, respectively. As a solution to this issue, our COOD/COOD+ detector recover the performance losses by identifying and filtering out the OOD samples without negatively impacting the model's code understanding ability in code search. Specifically, the code search performance of COOD/COOD+ on the Filtered-COOD/COOD+ dataset (70.30%/73.10% and 71.02%/73.18% on CSN-Python and -Java, respectively) is comparable to or even better than GraphCodeBERT on the Filtered-GT dataset (70.24% and 69.12%

on CSN-Python and -Java, respectively). This slight improvement is probably because our detectors filter out additional lower-quality testing samples that resemble outliers. Thus, our COOD/COOD+ enhance the trustworthiness and robustness of the GraphCodeBERT, since the model's predictions become more reliable when encountering OOD data. Note that the original GraphCodeBERT is not equipped with the OOD detection ability, so its corresponding cells for the Filtered-OOD-model in Table IV are left blank.

VII. Discussions

Analysis of the Overconfidence of MSP with Conformal Prediction. DNN models pre-trained on ID data are prone to misclassify OOD samples as ID due to overconfident MSP scores [99], [100]. This issue arises from spurious correlations between OOD and ID features, such as entities or syntactic patterns in NL and PL data [61], [101]. For example, OOD inputs with common ID syntactic patterns may receive high ID scores. To overcome overconfident predictions, previous work in ML explored techniques like temperature scaling [100] and confidence calibration using adversarial samples [99], [102]. In contrast, COOD+ leverages weakly-supervised contrastive learning with a small set of OOD samples to prevent the alignment of OOD pairs, and adopts a binary OOD rejection module to better differentiate OOD and ID representations. We further verify COOD+'s ability to address overconfidence using Conformal Prediction (CP) [103]. Post-hoc CP converts OOD scores into prediction sets with a high user-specified coverage (e.g., 95%), correcting overconfident thresholds during calibration. This ensures prediction sets conform statistically to the desired coverage. We experimentally demonstrated that COOD+ achieves near-optimal prediction set sizes with 95% coverage, effectively identifying true OODs with statistical guarantees. In contrast, MCL+MSP, the best MSP-based method, still struggles with overconfident OODs. Therefore, COOD+ can effectively overcome the overconfidence issue of MSP. Full details of the experiments and result analysis are in the online appendix [33].

OOD detection with large language models (LLMs). It's worth noting that transformer-based code models (*e.g.,* GraphCodeBERT [24]) and LLMs share the same underlying transformer architecture. Scaling up transformer-based code models and training them on vast amounts of code data allows LLMs [104] to perform a wide range of code-related tasks, making coding less labor-intensive and more accessible to end-users. Since LLMs are transformer-based, they are also vulnerable to OOD data, with potentially worse performance degradation due to error accumulation during auto-regressive inference. Thus, identifying OOD samples is crucial to knowing when to trust LLM outputs. Our proposed OOD code framework techniques can be applied to these larger transformer-based code models, similarly as demonstrated in our experiments with different encoders in Table III.

Generalization of COOD/COOD+ to other code-related tasks. During software development, developers often write comments following code snippets (methods/functions). Therefore, from a realistic perspective, our framework is generalizable to many code understanding tasks beyond code search, such as clone detection and defect detection, that use (comment, code) pairs as input. All that is needed is identifying the ID dataset and out-domain data, as the four OOD scenarios outlined in this paper are relevant to most tasks. For instance, in clone detection, (comment, code) pairs could first be processed by our framework to identify OOD samples before checking for clones. Unfortunately, since existing clone and defect detection datasets typically lack comments, we haven't applied our framework to these tasks. However, the framework has strong potential to enhance these tasks as more realistic bi-modal datasets become available in the future.

VIII. Threats to Validity & Limitations

Construct validity: Our COOD/COOD+ framework uses data-driven techniques to synthesize OOD samples, which may not fully reflect real-world SE scenarios. While we include diverse OOD scenarios, a pilot study with developers is necessary. Additionally, the reliability of our OOD benchmark depends heavily on the quality of the OOD datasets used. **Internal validity:** Hyperparameter tuning impacts ML performance. For model fine-tuning, we kept the GraphCodeBERT architecture unchanged due to feasibility reasons, but conducted ablation studies with various model components, encoder backbones, and key hyperparameters. **External validity:** We conduct OOD detection experiments on two large-scale code search datasets. Although our focus on Python and Java limits generalizability, experiments on these two languages partially demonstrate that our approach is PL-agnostic.

IX. Conclusion

We proposed two multi-modal OOD detection aproaches for code-related pre-trained ML models; namely unsupervised COOD and weakly-supervised COOD+. The COOD merely leveraged unsupervised contrastive learning to identify OOD samples. As an extension of COOD, COOD+ combined contrastive learning and a binary classifier for OOD detection using a small number of labelled OOD samples. To reap the benefits of these two modules, we also devised a novel scoring metric to fuse their prediction results. The evaluation results demonstrated that the integration of the rejection network and contrastive learning can achieve superior performance in detecting all four OOD scenarios for multi-modal NL-PL data. Additionally, our models can be applied to the downstream SE task, achieving comparable performance to existing code-related models.

X. Acknowledgement

This research has been supported in part by the NSF CCF-2311469, CNS-2132281, CCF-2007246, CCF-1955853 and CNS-2346357. We also gratefully acknowledge support from Cisco Systems. The opinions, findings, and conclusions expressed in this work are solely those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," in *CVPR*. IEEE, 2011, pp. 1521–1528.

[2] X. Liu, C. Yoo, F. Xing, H. Oh, G. El Fakhri, J.-W. Kang, J. Woo *et al.*, "Deep unsupervised domain adaptation: A review of recent advances and perspectives," *APSIPA Trans. Signal Inf. Proc.*, vol. 11, no. 1, 2022.

[3] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized out-of-distribution detection: A survey," *IJCV*, 2024.

[4] X. Li, M. Liu, S. Gao, and W. Buntine, "A survey on out-of-distribution evaluation of neural nlp models," in *IJCAI*, 2023, pp. 6683–6691.

[5] Y. Ming, Z. Cai, J. Gu, Y. Sun, W. Li, and Y. Li, "Delving into out-of-distribution detection with vision-language representations," *NeurIPS*, vol. 35, pp. 35 087–35 102, 2022.

[6] D. Hendrycks, M. Mazeika, and T. Dietterich, "Deep anomaly detection with outlier exposure," *arXiv:1812.04606*, 2018.

[7] K. T. Mai, T. Davies, and L. D. Griffin, "Self-supervised losses for one-class textual anomaly detection," *arXiv:2204.05695*, 2022.

[8] W. Zhou, F. Liu, and M. Chen, "Contrastive out-of-distribution detection for pretrained transformers," in *EMNLP*, 2021, pp. 1100–1111.

[9] Y. Tian, G. Maicas, L. Z. C. T. Pu, R. Singh, J. W. Verjans, and G. Carneiro, "Few-shot anomaly detection for polyp frames from colonoscopy," in *MICCAI*. Springer, 2020, pp. 274–284.

[10] S. Majhi, S. Das, F. Brémond, R. Dash, and P. K. Sa, "Weakly-supervised joint anomaly detection and classification," in *FG*. IEEE, 2021, pp. 1–7.

[11] J. Kim, S. T. Kong, D. Na, and K.-H. Jung, "Key feature replacement of in-distribution samples for out-of-distribution detection," in *AAAI*, vol. 37, no. 7, 2023, pp. 8246–8254.

[12] J. Kim, K. Jung, D. Na, S. Jang, E. Park, and S. Choi, "Pseudo outlier exposure for out-of-distribution detection using pretrained transformers," in *ACL*, 2023, pp. 1469–1482.

[13] J. Yoo, T. Zhao, and L. Akoglu, "Data augmentation is a hyperparameter: Cherry-picked self-supervision for unsupervised anomaly detection is creating the illusion of success," *TMLR*, 2022.

[14] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" in *ICSE*, 2022, pp. 1356–1367.

[15] X. Wang, J. Song, X. Zhang, J. Tang, W. Gao, and Q. Lin, "Logonline: A semi-supervised log-based anomaly detector aided with online learning mechanism," in *ASE*. IEEE, 2023, pp. 141–152.

[16] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *ESEC/FSE*, 2020, p. 1387–1397.

[17] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu, "Microhecl: High-efficient root cause localization in large-scale microservice systems," in *ICSE-SEIP*, 2021, pp. 338–347.

[18] A. Sejfia, S. Das, S. Shafiq, and N. Medvidović, "Toward improved deep learning-based vulnerability detection," in *ICSE*, 2024, pp. 1–12.

[19] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *ICSE*, 2024, pp. 1–13.

[20] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, and W. Liu, "Coca: Improving and explaining graph neural network-based vulnerability detection systems," in *ICSE*, 2024, pp. 939–939.

[21] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *NeurIPS*, vol. 34, pp. 27 865–27 876, 2021.

[22] J. He, L. Beurer-Kellner, and M. Vechev, "On distribution shift in learning-based bug detectors," in *ICML*. PMLR, 2022, pp. 8559–8580.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *NeurIPS*, vol. 30, 2017.

[24] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*, 2021.

[25] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv:2203.03850*, 2022. [Online]. Available: Unified cross-modal pre-training for code representation

[26] Y. Yan, N. Cooper, K. Moran, G. Bavota, D. Poshyvanyk, and S. Rich, "Enhancing code understanding for impact analysis by combining transformers and program dependence graphs," *Proc. ACM Softw. Eng.*, no. FSE, 2024.

[27] D. Hendrycks, X. Liu, E. Wallace, A. Dziedzic, R. Krishnan, and D. Song, "Pretrained transformers improve out-of-distribution robustness," *arXiv:2004.06100*, 2020.

[28] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, "Contrabert: Enhancing code pre-trained models via contrastive learning," in *ICSE*, 2023, p. 2476–2487.

[29] S. Esmaeilpour, B. Liu, E. Robertson, and L. Shu, "Zero-shot out-of-distribution detection based on the pretrained model clip," in *AAAI*, 2022.

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

[31] A. v. d. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv:1807.03748*, 2018.

[32] W. Liu, X. Wang, J. Owens, and Y. Li, "Energy-based out-of-distribution detection," *NeurIPS*, vol. 33, pp. 21 464–21 475, 2020.

[33] Y. Yan, V. Duong, H. Shao, and D. Poshyvanyk, "Cood online appendix," 2024. [Online]. Available: https://github.com/yanyanfu/COOD

[34] B. Yu, J. Yao, Q. Fu, Z. Zhong, H. Xie, Y. Wu, Y. Ma, and P. He, "Deep learning or classical machine learning? an empirical study on log-based anomaly detection," in *ICSE*, 2024, pp. 1–13.

[35] C. Zhang, T. Jia, G. Shen, P. Zhu, and Y. Li, "Metalog: Generalizable cross-system anomaly detection from logs with meta-learning," in *ICSE*. IEEE Computer Society, 2024, pp. 938–938.

[36] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, "A transformer-based framework for multivariate time series representation learning," in *KDD*, 2021, p. 2114–2124.

[37] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *ESEC/FSE*, 2019, pp. 807–817.

[38] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *DASC/PiCom/DataCom/CyberSciTech 2018*. IEEE, 2018, pp. 151–158.

[39] A. Farzad and T. A. Gulliver, "Unsupervised log message anomaly detection," *ICT Express*, vol. 6, no. 3, pp. 229–237, 2020.

[40] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *ICSE*. IEEE, 2021, pp. 1448–1460.

[41] C. Lee, T. Yang, Z. Chen, Y. Su, Y. Yang, and M. R. Lyu, "Heterogeneous anomaly detection for software systems via semi-supervised cross-modal attention," in *ICSE*, 2023, p. 1724–1736.

[42] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv:2102.04664*, 2021.

[43] "Codeql," 2023. [Online]. Available: https://codeql.github.com

[44] "Checkmarx," 2023. [Online]. Available: https://checkmarx.com

[45] Z. Liu, Z. Tang, J. Zhang, X. Xia, and X. Yang, "Pre-training by predicting program dependencies for vulnerability analysis tasks," in *ICSE*, 2024, pp. 935–935.

[46] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *ICML*. PMLR, 2020, pp. 5110–5121.

[47] Z. Chen, V. J. Hellendoorn, P. Lamblin, P. Maniatis, P.-A. Manzagol, D. Tarlow, and S. Moitra, "Plur: A unifying, graph-based view of program learning, understanding, and repair," *NeurIPS*, vol. 34, pp. 23 089–23 101, 2021.

[48] Q. Hu, Y. Guo, X. Xie, M. Cordy, M. Papadakis, L. Ma, and Y. Le Traon, "Codes: towards code model generalization under distribution shift," in *ICSE-NIER*. IEEE, 2023, pp. 1–6.

[49] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code," *arXiv:2305.04106*, 2023.

[50] H. Hajipour, N. Yu, C.-A. Staicu, and M. Fritz, "Simscood: Systematic analysis of out-of-distribution generalization in fine-tuned source code models," in *NAACL*, 2024, pp. 1400–1416.

[51] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," in *ICLR*, 2017.

[52] M. Salehi, H. Mirzaei, D. Hendrycks, Y. Li, M. H. Rohban, and M. Sabokrou, "A unified survey on anomaly, novelty, open-set, and out of-distribution detection: Solutions and future challenges," *TMLR*, 2022.

[53] M. Hein, M. Andriushchenko, and J. Bitterwolf, "Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem," in *CVPR*, 2019, pp. 41–50.

[54] Y. Sun, Y. Ming, X. Zhu, and Y. Li, "Out-of-distribution detection with deep nearest neighbors," in *ICML*, 2022.

[55] Y. Hu and L. Khan, "Uncertainty-aware reliable text classification," in *KDD*, 2021, p. 628–636.

[56] P. Liznerski, L. Ruff, R. A. Vandermeulen, B. J. Franks, K.-R. Müller, and M. Kloft, "Exposing outlier exposure: What can be learned from few, one, and zero outlier images," *arXiv:2205.11474*, 2022.

[57] A. Kamath, R. Jia, and P. Liang, "Selective question answering under domain shift," in *ACL*, 2020, pp. 5684–5696.

[58] N. Varshney, S. Mishra, and C. Baral, "Investigating selective prediction approaches across several tasks in IID, OOD, and adversarial settings," in *ACL*, 2022, pp. 1995–2002.

[59] J. Xin, R. Tang, Y. Yu, and J. Lin, "The art of abstention: Selective prediction and error regularization for natural language processing," in *ACL-IJCNLP*, 2021, pp. 1040–1051.

[60] Z. Zeng, H. Xu, K. He, Y. Yan, S. Liu, Z. Liu, and W. Xu, "Adversarial generative distance-based classifier for robust out-of-domain detection," in *ICASSP*. IEEE, 2021, pp. 7658–7662.

[61] Y. Zheng, G. Chen, and M. Huang, "Out-of-domain detection for natural language understanding in dialog systems," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 28, pp. 1198–1209, 2020.

[62] L.-M. Zhan, H. Liang, B. Liu, L. Fan, X.-M. Wu, and A. Y. Lam, "Out-of-scope intent detection with self-supervision and discriminative training," in *ACL*, 2021, pp. 3521–3532.

[63] D. Jin, S. Gao, S. Kim, Y. Liu, and D. Hakkani-Tür, "Towards textual out-of-domain detection without in-domain labels," *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, vol. 30, p. 1386–1395, 2022.

[64] K. Xu, T. Ren, S. Zhang, Y. Feng, and C. Xiong, "Unsupervised out-of-domain detection via pre-trained transformers," in *ACL*, 2021, pp. 1052–1061.

[65] L. Sun, K. Yang, X. Hu, W. Hu, and K. Wang, "Real-time fusion network for rgb-d semantic segmentation incorporating unexpected obstacle detection for road-driving images," *IEEE Robot. Autom. Lett.*, vol. 5, no. 4, pp. 5558–5565, 2020.

[66] L. Wang, S. Giebenhain, C. Anklam, and B. Goldluecke, "Radar ghost target detection via multimodal transformers," *IEEE Robot. Autom. Lett.*, vol. 6, no. 4, pp. 7758–7765, 2021.

[67] S. Fort, J. Ren, and B. Lakshminarayanan, "Exploring the limits of out-of-distribution detection," *NeurIPS*, vol. 34, pp. 7068–7081, 2021.

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

[69] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *CVPR*, vol. 2, 2006, pp. 1735–1742.

[70] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," *NeurIPS*, vol. 33, pp. 18 661–18 673, 2020.

[71] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *ICML*. PMLR, 2020, pp. 1597–1607.

[72] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv:2007.04973*, 2020.

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

[74] Y. Qiu, T. Misu, and C. Busso, "Unsupervised scalable multimodal driving anomaly detection," *IEEE TIV*, 2022.

[75] Y.-C. Hsu, Y. Shen, H. Jin, and Z. Kira, "Generalized odin: Detecting out-of-distribution image without learning from out-of-distribution data," in *CVPR*, 2020, pp. 10 951–10 960.

[76] L. Ruff, R. A. Vandermeulen, N. Görnitz, A. Binder, E. Müller, K.-R. Müller, and M. Kloft, "Deep semi-supervised anomaly detection," in *ICLR*, 2020.

[77] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," in *ICSE*, 2023, pp. 2198–2210.

[78] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," *arXiv:1610.02136*, 2016.

[79] V. Duong, Q. Wu, Z. Zhou, E. Zavesky, J. Chen, X. Liu, W.-L. Hsu, and H. Shao, "General-purpose multi-modal ood detection framework," *TMLR*, 2024.

[80] H. Xue, Q. Yang, and S. Chen, "Svm: Support vector machines," in *The top ten algorithms in data mining*. Chapman and Hall/CRC, 2009, pp. 51–74.

[81] P. Salza, C. Schwizer, J. Gu, and H. C. Gall, "On the effectiveness of transfer learning for code search," *TSE*, 2022.

[82] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: Any snippets there?" in *MSR*, 2017, pp. 280–290.

[83] A. Lotter, S. A. Licorish, B. T. R. Savarimuthu, and S. Meldrum, "Code reuse in stack overflow and popular open source java projects," in *ASWEC*, 2018, pp. 141–150.

[84] R. Abdalkareem, E. Shihab, and J. Rilling, "On code reuse from stackoverflow: An exploratory study on android apps," *Inf. Softw. Technol.*, vol. 88, pp. 148–158, 2017.

[85] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20,000+ web queries for code search and question answering," *arXiv:2105.13239*, 2021.

[86] A. Podolskiy, D. Lipin, A. Bout, E. Artemova, and I. Piontkovskaya, "Revisiting mahalanobis distance for transformer-based out-of-domain detection," in *AAAI*, vol. 35, no. 15, 2021, pp. 13 675–13 682.

[87] C. Wang, Z. Nong, C. Gao, Z. Li, J. Zeng, Z. Xing, and Y. Liu, "Enriching query semantics for code search with reinforcement learning," *Neural Netw.*, vol. 145, pp. 22–32, 2022.

[88] Z. Yao, D. S. Weld, W.-P. Chen, and H. Sun, "Staqc: A systematically mined question-code dataset from stack overflow," in *WWW*, 2018, pp. 1693–1703.

[89] R. Li, G. Hu, and M. Peng, "Hierarchical embedding for code search in software q&a sites," in *IJCNN*. IEEE, 2020, pp. 1–10.

[90] N. Rao, C. Bansal, and J. Guan, "Search4code: Code search intent classification using weak supervision," in *MSR*, 2021, pp. 575–579.

[91] K. Sinha, R. Jia, D. Hupkes, J. Pineau, A. Williams, and D. Kiela, "Masked language modeling and the distributional hypothesis: Order word matters pre-training for little," in *EMNLP*, 2021, pp. 2888–2913.

[92] K. Sinha, P. Parthasarathi, J. Pineau, and A. Williams, "Unnatural language inference," *arXiv:2101.00010*, 2020.

[93] C. Richter and H. Wehrheim, "Learning realistic mutations: Bug creation for neural bug detectors," in *ICST*, 2022, pp. 162–173.

[94] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[95] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *NeurIPS*, vol. 32, 2019.

[96] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *ESEC/FSE*, 2020, pp. 1556–1560.

[97] C. Richter and H. Wehrheim, "How to train your neural bug detector: Artificial vs real bugs," in *ASE*. IEEE Computer Society, 2023, pp. 1036–1048.

[98] D. Hendrycks, M. Mazeika, S. Kadavath, and D. Song, "Using self-supervised learning can improve model robustness and uncertainty," *NeurIPS*, vol. 32, 2019.

[99] K. Lee, H. Lee, K. Lee, and J. Shin, "Training confidence-calibrated classifiers for detecting out-of-distribution samples," *arXiv:1711.09325*, 2017.

[100] S. Liang, Y. Li, and R. Srikant, "Enhancing the reliability of out-of-distribution image detection in neural networks," *arXiv:1706.02690*, 2017.

[101] Y. Wu, K. He, Y. Yan, Q. Gao, Z. Zeng, F. Zheng, L. Zhao, H. Jiang, W. Wu, and W. Xu, "Revisit overconfidence for ood detection: Reassigned contrastive learning with adaptive class-dependent threshold," in *NAACL-HLT*, 2022, pp. 4165–4179.

[102] J. Bitterwolf, A. Meinke, and M. Hein, "Certifiably adversarially robust detection of out-of-distribution data," *NeurIPS*, vol. 33, pp. 16 085–16 095, 2020.

[103] A. N. Angelopoulos, S. Bates *et al.*, "Conformal prediction: A gentle introduction," *Found. Trends Mach. Learn.*, vol. 16, no. 4, pp. 494–591, 2023.

[104] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv:2308.12950*, 2023.