HedgeCode: A Multi-Task Hedging Contrastive Learning Framework for Code Search

Gong Chen*, Xiaoyuan Xie*‡, Daniel Tang†, Qi Xin*, Wenjie Liu*
*School of Computer Science, Wuhan University, China
†University of Luxembourg, Luxembourg
chengongcg@whu.edu.cn, xxie@whu.edu.cn, xunzhu.tang@uni.lu, qxin@whu.edu.cn, wenjieliu@whu.edu.cn

***Abstract*—Code search is a vital activity in software engineering, focused on identifying and retrieving the correct code snippets based on a query provided in natural language. Approaches based on deep learning techniques have been increasingly adopted for this task, enhancing the initial representations of both code and its natural language descriptions. Despite this progress, there remains an unexplored gap in ensuring consistency between the representation spaces of code and its descriptions. Furthermore, existing methods have not fully leveraged the potential relevance between code snippets and their descriptions, presenting a challenge in discerning fine-grained semantic distinctions among similar code snippets.**

To address these challenges, we introduce a multi-task hedging contrastive Learning framework for Code Search, referred to as *HedgeCode*. *HedgeCode* is structured around two primary training phases. The first phase, known as the representation alignment stage, proposes a hedging contrastive learning approach. This method aims to detect subtle differences between code and natural language text, thereby aligning their representation spaces by identifying relevance. The subsequent phase involves multi-task joint learning, wherein the previously trained model serves as the encoder. This stage optimizes the model through a combination of supervised and self-supervised contrastive learning tasks. Our framework's effectiveness is demonstrated through its performance on the CodeSearchNet benchmark, showcasing *HedgeCode*'s ability to address the mentioned limitations in code search tasks.

***Index Terms*—code search, relevance detection, contrastive learning, multi-task learning**

I. Introduction

Code search is one of the most important tasks in the field of software engineering [1], [2]. The goal of code search is to retrieve codes from the codebase that match the intent of developers. With the emergence of a large number of available code libraries (such as GitHub and Stack Overflow), developers often need to spend about 19% of their working time to search code from existing code libraries based on their intents for code reuse [3]–[5]. How to search codes that confirm the intent of the query provided in natural language from a rich pool of candidates becomes a challenge [6]–[9].

To contribute to the effectiveness and efficiency of code search, various previous works concerning code search have been proposed. In earlier studies, researchers utilized information retrieval (IR) techniques to search relevant codes [10]–[12]. Specifically, they use TF-IDF methods to represent queries and codes as sparse vectors and calculate the lexical similarity between them. However, these methods are token-sensitive and lack the ability to understand the semantics of codes and the intent of queries. With the development of deep learning (DL), researchers adopt deep neural networks for code search [3], [6], [13]–[19]. The DL-based methods represent codes and queries as dense vectors and then calculate the cosine similarity of vectors to measure semantic relevances [2]. Recently, code large language models (LLMs) pre-trained on massive source code-related data have shown impressive performance in code intelligence and achieved excellent performance on code search task [20]–[22]. Although these code LLMs have achieved advanced performance, the semantic relevance between code and its natural language description is not fully utilized [23], [24]. So it is difficult to distinguish functional equivalent codes with different implementations or code snippets with similar tokens but unequivalent in semantics.

Inspired by the successful application of Contrastive Learning (CL) in the fields of natural language processing (NLP) and computer vision (CV) [25]–[27], some researchers have introduced CL into programming language processing [23], [24], [28]–[32], and achieve advanced performance in code search. However, these methods also have some limitations. Firstly, code is a structured form of language that differs significantly from natural language texts, making them distinct modalities. Despite this, the consistency in representation spaces between code and natural language texts remains an area that has not yet been fully investigated. Fig. 2 illustrates that current approaches are unable to bridge the gap in representation spaces resulting from the inherent differences between code and natural language texts.

Secondly, the relationship between code and its natural language description remains inadequately explored, leading to a situation where current techniques struggle to grasp the finer nuances of their correlation. While these methods do manage to establish a link between code and text using self-supervised CL methods [25], supervised CL [33], by contrast, has the advantage of leveraging prior knowledge contained in supervised signals. This benefit is evident in Fig. 1, where supervised CL employs these signals to capture more semantic detail, thus enabling it to differentiate subtle semantic distinctions more effectively. However, a significant challenge with supervised CL is the difficulty in acquiring the necessary supervised signals.

‡Corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00008

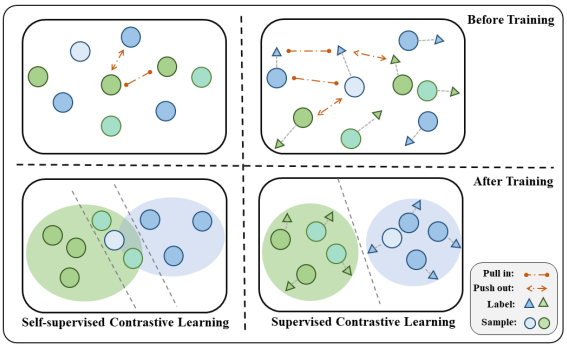

Fig. 1. Comparison of supervised and self-supervised CL.

This paper. To address these limitations, we propose HedgeCode*: A Multi-Task Hedging Contrastive Learning Framework for Code Search*. *HedgeCode* consists of two training stages. In the representation alignment stage, we build a dataset for relevance detection and design a supervised CL method, hedging contrastive learning (HCL), to capture fine-grained differences between the code and text and align the representation spaces through relevance detection. In the multi-task joint learning stage, we employ the trained model as the encoder and adopt the joint learning paradigm to optimize it through three tasks: code-text contrastive learning task (CTC), code-text relevance detection task (CTRD), and code search task (CS) based on supervised and self-supervised contrastive learning. The experimental results on the CodeSearchNet benchmark show the effectiveness of *HedgeCode*. Specifically, for the three integrated pre-trained models (CodeBERT [20], Unixcoder [23], CoCoSoDa [24]), the MRR is improved by 3.7, 2.6, and 1.5 respectively. The visualization results show that *HedgeCode* can effectively align the representation space of code and text.

Our main contributions are as follows:

❶ We propose HedgeCode. Through aligned representation space and multi-task hedging contrastive learning by *HedgeCode*, the integrated code search model is optimized.

❷ We build a dataset and design a supervised contrastive learning method, hedging contrastive learning for relevance detection task. And we innovatively utilize HCL to align representation space and optimize code-text relevance detection task.

❸ Our experimental results on the dataset demonstrate the effectiveness of our framework. The code and datasets are available at: https://github.com/repo-anonymous/hedgecode.

The remainder of this paper is organized as follows. Section II presents the motivations of our work. Section III overviews our proposed approach. The experimental setup and results are then described in Sections IV and V respectively. We provide discussions in Section VI. We presented the relevant work in Section VII and concluded in Section VIII.

II. Motivation

Our motivation stems from two primary concerns:

Addressing the inconsistency in representation spaces due to data diversity. Deep learning-based models typically encode codes and natural language texts into dense vectors to assess their semantic similarity through cosine similarity [2]. Ideally, a natural language text and its corresponding code should exhibit similar vector representations in a multidimensional space. Nonetheless, it has been noted that some widely used models, such as CodeBERT [20], display discrepancies in the representation distributions of code and natural language texts.

Specifically, we adopt the t-SNE algorithm [34] to visualize the representation distributions of codes and their descriptions embedded by CodeBERT, as illustrated in Fig. 2. The figure highlights the disparity in representation distributions; code representations tend to be more clustered, whereas natural language text representations are more scattered. This discrepancy in distribution can lead to two issues: Firstly, the dense clustering of code representations complicates the identification of nuanced semantic differences between codes. Secondly, the mismatch in representation distributions can skew the similarity assessments between codes and texts, potentially leading to incorrect matches.

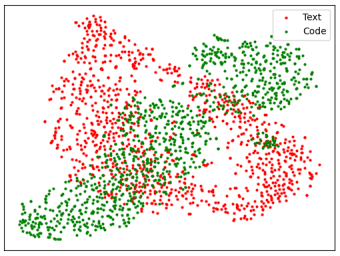

Fig. 2. The representation distributions of CodeBERT.

Identifying distinctions between similar yet not identical codes. Listing 1. presents two instances from the code search dataset. The first instance illustrates that codes with highly similar tokens can have diametrically opposed functional semantics. The second instance exemplifies a typical programming concept, code overload, where codes share similar tokens and basic functions but differ in semantics due to variations in inputs and outputs. These instances demonstrate a prevalent issue: codes with similar lexical components may convey different or subtly distinct semantics. This similarity can deceive neural networks, making the differentiation of nuanced semantic variations a significant challenge.

Driven by these challenges, we introduce *HedgeCode* to delve into the nuanced correlations between code and natural language texts and to align their representation spaces more accurately.

Listing 1. Similar but unequivalent code snippets.

```
Example 1-1:
public boolean add(T o){
   boolean result = super.add(o);
   sizeOf += (result) ? o.sizeof() : 0;
```

```
    return result;}
-----------------------------------------------
Example 1-2:
public void add(int index, T element){
    super.add(index, element);
    sizeOf += element.sizeof();}
-----------------------------------------------
Example 2-1:
Public LongAssert isGreaterThan(long other){
    if(actual > other){return this;}
    failIfCustomMessageIsSet();
    throw failure(
       unexpectedLessThanOrEqualTo(actual,
           other));}
-----------------------------------------------
Example 2-2:
public IntAssert isLessThanOrEqualTo(int
     other){
    if(actual <= other){return this;}
    failIfCustomMessageIsSet();
    throw
        failure(unexpectedGreaterThan(actual,
        other));}
-----------------------------------------------
```

III. Approach

To align the representation spaces and capture the fine-grained semantic relevance between code and natural language text, we propose *HedgeCode*. As illustrated in Fig. 3, *HedgeCode* consists of three stages: representation alignment, multi-task joint, and code search. *HedgeCode* aligns the representation spaces of code and text in the first stage and then further optimizes the representation learning of code and text in the second stage. Finally, it uses the trained model for code search. We will fill in the details of each stage in the subsections.

A. Representation Alignment Stage (RA)

In the representation alignment stage, we align the representation spaces of code and text by detecting relevance between them. We define the code-text relevance detection task (CTRD) and build the dataset. In order to capture the fine-grained differences between the code and text, we design a supervised contrastive learning method named hedging contrastive learning (HCL).

1) Task Formalization: Specifically, we take the relevance detection task as a binary classification problem. And it is defined as Definition III-A1.

> ***Definition III-A1 (Code-Text Relevance Detection Task (CTRD)).*** Code-text relevance detection is a binary classification task. Given a pair of $code$ and $description$, we decide whether $code$ and $description$ are related or not. If they are related, the detection result is expressed as $relevance = 1$, otherwise $relevance = 0$.

2) Detection Dataset Building: We build the detection datasets based on the CodeSearchNet benchmark. The CodeSearchNet contains six programming language (PL) datasets. We build detection datasets for each of them. The raw data contains a code snippet and its description. To construct detection pairs for each PL, we first take a code snippet as an anchor, denoted $code^+$, and identify the code snippets that are similar to the token sequence of the anchor in the dataset. We take the top K similar code snippets as the negative samples set $C^- = \{code_1^-, ..., code_k^-\}$. Then, we take the pair of the anchor as a positive sample pair $(description, code^+)$ and label its relevance as 1. We combine each code snippet in the negative sample set with the anchor's description as a negative sample pair $(description, code^-)$ and label their relevance as 0. Finally, we collect the positive sample pair and the K negative sample pairs of each anchor as detection pairs.

It is worth noting that the objective of the CTRD task is to distinguish codes with similar vocabularies but nuanced semantic differences through more fine-grained relevance detection. Compared to these codes, the anchor is more relevant to its description. So we take the codes that have a similar token sequence to the anchor as its negative samples. Specifically, we adopt BERT [35] to identify codes with similar vocabularies. We first encode all codes with BERT and output their vector representations. Then we use cosine similarity to measure their similarities. We set $K = 2$ and take the first two codes with the highest similarities with the anchor as negative samples.

3) Hedging Contrastive Learning: The detection model is designed to accept a pair of code and its description, embedding their token sequences through an encoder and predicting their relevance using a classification head. We enhance this pipeline with HCL to refine the model's sensitivity to the subtleties distinguishing code from text.

The overview of the HCL is shown in Fig. 4. This approach targets the identification and prioritization of hard negative samples, which closely resemble positive samples but belong to a different class, thus posing a significant challenge for discrimination.

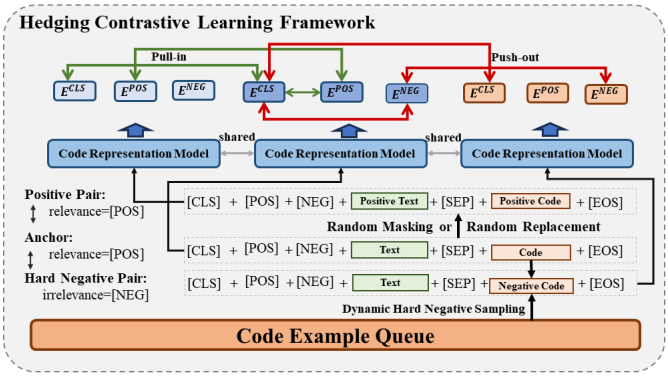

Fig. 4. The overview of the HCL.

In the HCL, we enhance the original dual contrastive loss (DualCL) [36] by incorporating the data augmentation strategy. DualCL is an advanced supervised contrastive learning algorithm designed for text classification tasks (such as emotion classification). However, different from these text classification tasks, there may be no correlation between detection pairs, so the CTRD task cannot simply benefit from the labels of pairs.

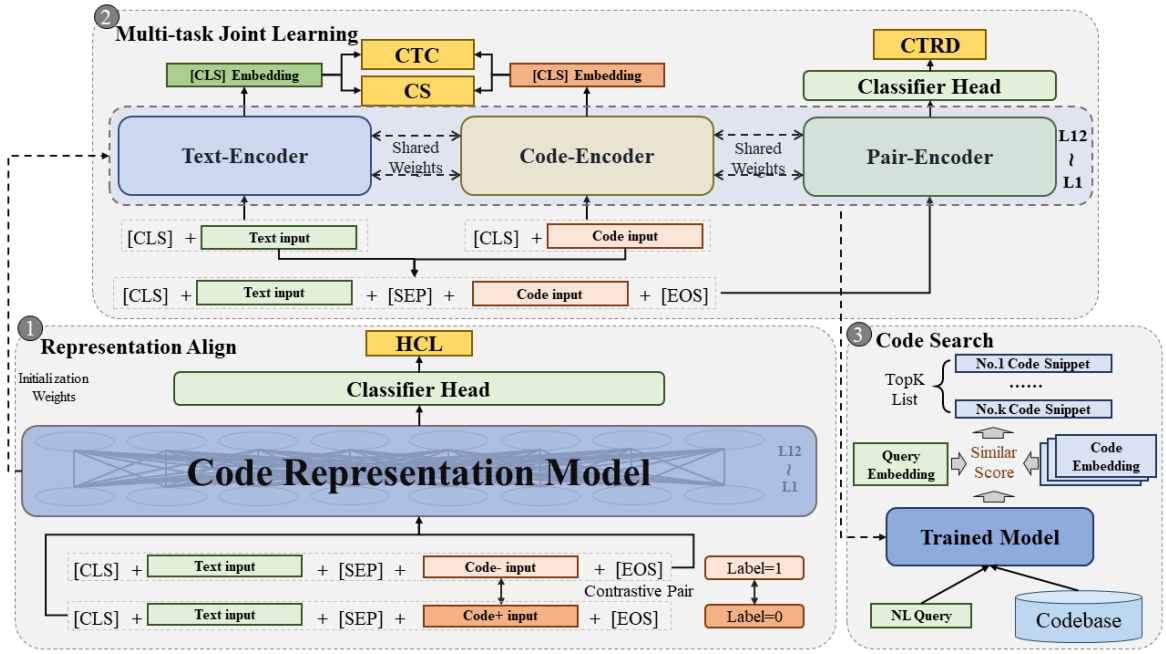

Fig. 3. The overview of the *HedgeCode*. *HedgeCode* consists of three stages. ❶ In the RA stage (Section III-A), we align the representation spaces of code and text through the CTRD task and design a HCL method to capture fine-grained differences between the code and text. ❷ In the MJL stage (Section III-B), we employ joint learning to optimize the CTRD task, CTC task, and CS task. ❸ In the CS stage (Section III-C), we employ the trained encoder to search codes from the codebase.

Due to this data discrepancy, the DualCL cannot be directly applied to the CTRD task. We design a data augmentation strategy to make DualCL suitable for CTRD task.

a) Data Augmentation: Our data augmentation strategy consists of two parts, positive sample augmentation and dynamic hard negative sampling.

Positive Sample Augmentation. We propose two positive sample augmentation methods inspired by Masked Language Models [35]. During training, in order to increase the diversity and randomness of positive samples, we randomly choose a data augmentation method to augment the input.

Random Masking (RM): randomly sample 20% of tokens of the sequence and replace each token with a [MASK] token.

Random Replacement (RR): randomly sample 20% of tokens of the sequence and replace each token with a random token.

Dynamic Hard Negative Sampling. The effectiveness of negative samples is crucial for contrastive learning. We design a dynamic hard negative sampling strategy (DHNS). In particular, we construct and maintain a sample queue for DHNS. We sample the hard negative sample from this queue. The queue is randomly initialized and updated with batch-size data at each training batch. We rank the negative samples based on their similarity to the positive sample and identify the top-ranked sample as the hard negative sample at each training batch. It's a dynamic process.

b) Hedging Contrastive Loss: In the HCL framework, given a detection sample, we insert [CLS] at the beginning and insert the binary labels (relevance and irrelevance) between [CLS] and the token sequence X as the input for the classifier, denoted as $\big[[\mathrm{CLS}], \mathrm{relevance}, \mathrm{irrelevance}, \mathrm{X}\big]$. In the CTRD task, for an input sample x_i, the classifier t_i outputs its feature representation z_i, and predicts its label through the softmax transform of $t_i^T \cdot z_i$. Our objective is to maximize the consistency between the prediction of the classifier and the ground-truth label of x_i. Meanwhile, we also expect to maximize the dot product between the representation of ground-truth label t_i^* of the input token sequence and the feature representation z_i. To achieve this, we introduce an advanced contrastive loss that leverages the relationship between training samples based on label congruence. Specifically, we propose two contrastive views: classifier contrastive view and feature contrastive view.

In the feature contrastive view, given an anchor z_i, we define the set of positive samples as $\{t_j^*\}_{j\in P_i}$ and the set of hard negative samples as $\{t_j^*\}_{j\in H_i}$. The contrastive loss is expressed as:

$$\mathcal{L}_z = -\frac{1}{N}\sum_{i\in I}\frac{1}{|P_i|}\sum_{p\in P_i}\log\frac{\exp\left(t_p^*\cdot z_i/\tau\right)}{\sum_{a\in A_i}\exp(t_h^*\cdot z_i/\tau)} \tag{1}$$

where A is the set of the hard negative samples and P is the set of the positive samples.

Similarly, in the classifier contrastive view, for an anchor t_i^*, the contrastive loss is expressed as:

$$\mathcal{L}_\theta = -\frac{1}{N}\sum_{i\in I}\frac{1}{|P_i|}\sum_{p\in P_i}\log\frac{\exp\left(t_i^*\cdot z_p/\tau\right)}{\sum_{a\in A_i}\exp(t_i^*\cdot z_h/\tau)} \tag{2}$$

The Hedging Contrastive Loss integrates both enhancements to ensure a balanced learning focus between easy and hard

examples and enhance the model's ability to discern subtle differences across classes. Finally, the Hedging Contrastive Loss $\mathcal{L}^{'}_{\mathrm{HCL}}$ is given by:

$$\mathcal{L}^{'}_{\mathrm{HCL}} = \mathcal{L}_z + \mathcal{L}_\theta \tag{3}$$

To combine the HCL objective with the detection objective, we introduce a modified cross-entropy loss $\mathcal{L}_{\mathrm{CE}}$:

$$\mathcal{L}_{\mathrm{CE}} = -\frac{1}{N}\sum_{i\in I}\log\frac{\exp(\theta_i^* \cdot \boldsymbol{z}_i)}{\sum_{k\in K}\exp(\theta_i^k \cdot \boldsymbol{z}_i)} \tag{4}$$

This loss function prioritizes the correct detection of instances by maximizing the dot product between the instance's feature representation z_i and the correct classifier weight θ_i^*. K is the number of types of labels. In the CTRD task, $K = 2$.

Finally, we combine these training losses to jointly optimize the classifier.

$$\mathcal{L}_{\mathrm{HCL}} = \mathcal{L}_{\mathrm{CE}} + \lambda\mathcal{L}^{'}_{\mathrm{HCL}} \tag{5}$$

where λ is a weighting factor to balance the detection objective and the HCL objective.

B. Multi-task Joint Learning Stage (MJL)

In the multi-task joint learning stage, we design a multi-task contrastive learning framework containing three tasks: code-text relevance detection task (CTRD), code-text contrastive learning task (CTC), and code search task (CS). We follow the paradigm of joint learning and employ the model trained by the representation alignment stage as the encoder. These three tasks share a common goal, which is to learn the semantic correlations between code and text. Among them, CS is the main task. CTC learns the semantic associations and distinctions between code and text from two views, cross-modal and unique-modal, through self-supervised contrastive learning. CTRD learns the fine-grained differences between code and text through supervised signals. They are related and complementary.

1) Model Architecture: The model architecture is shown in Fig. 3. The model consists of three encoders with shared weights: text-encoder, code-encoder, and pair-encoder. Among them, text-encoder and code-encoder are used for the CS task and CTC task, and a classification head is attached after pair-encoder for the CTRD task.

Following Section III-A3a, we randomly employ RM or RR to get the positive sample and we also adopt DHNS to sample the negative sample.

2) Mulit-task Joint Learning: We optimize the model by joint learning three tasks.

Code-Text Contrastive Learning Task (CTC) aims to minimize the distance of similar sample representations and maximize the distance of distinct samples. The model employs the text-encoder and code-encoder to embed code descriptions and code snippets and then employs self-supervised contrastive learning to optimize their representations.

We take the given text/code as an anchor and randomly select RM and RR for data augmentation, the augmented texts/codes are used as positive samples of the anchor. And we randomly sample negative samples from the mini-batch. Following previous work [26], we employ InfoNCE loss [37] to optimize this task. Specifically, we propose two contrastive views, unique-modal and cross-modal, and their loss function as follows:

$$\mathcal{L}_{\mathrm{cross}} = -\log\frac{\exp\left(t_i \cdot c_i/\tau\right)}{\sum_{i,j\in I}\exp\left(t_i \cdot c_j/\tau\right)} \tag{6}$$

$$\begin{aligned}\mathcal{L}^{\mathrm{text}}_{\mathrm{unique}} &= -\log\frac{\exp\left(t_i \cdot t^+/\tau\right)}{\sum_{i,j\in I}\exp\left(t_i \cdot t_j/\tau\right)}\\ \mathcal{L}^{\mathrm{code}}_{\mathrm{unique}} &= -\log\frac{\exp\left(c_i \cdot c^+/\tau\right)}{\sum_{i,j\in I}\exp\left(c_i \cdot c_j/\tau\right)}\end{aligned} \tag{7}$$

where c and t are the embeddings of the code and text, respectively. In cross-modal contrastive view, t_i is an anchor, c_i is its positive sample, and c_j is the negative sample. In unique-modal contrastive view, c^+ and t^+ are the positive samples of c_i and t_i, c_j and t_j are negative samples. And τ is the temperature.

Finally, the loss function of CTC is as follows:

$$\mathcal{L}_{\mathrm{CTC}} = \mathcal{L}_{\mathrm{cross}} + \mathcal{L}^{\mathrm{text}}_{\mathrm{unique}} + \mathcal{L}^{\mathrm{code}}_{\mathrm{unique}} \tag{8}$$

Code-Text Relevance Detection Task (CTRD) aims to capture the fine-grained relevance between code and its description. As stated in Definition III-A1, CTRD is a binary classification problem. *HedgeCode* employs the pair-encoder to learn code and text features simultaneously and employs a classification head to predict whether the code-text pairs are relevant or not.

Following the III-A3a section, for a given pair, we label its relevance as 1 and adopt the same data augmentation strategy to sample the positive and negative samples. We adopt HCL (as shown in III-A3 section) to optimize the CTRD.

$$\mathcal{L}_{\mathrm{CTRD}} = \mathcal{L}_{\mathrm{HCL}} \tag{9}$$

The CTRD task is included in the RA and MJL stages. In the RA stage, we build a detection dataset through global token similarity identification and align the representation space of code and text through the CTRD task. In the MJL stage, we adopt the CTRD task to further capture more fine-grained relevance between code and text to optimize the representation learning of code and text.

Code Search Task (CS) is the main task. Following previous works [6], [20], [21], [24], we employ cross-entropy loss to optimize this task.

$$\mathcal{L}_{\mathrm{CS}} = -\sum_{i=1}^{k}\left[\log\frac{\exp\left(cos\left(q_i, c_i\right)\right)}{\Sigma_{j=1}^{k}\exp\left(cos\left(q_i, c_j\right)\right)}\right] \tag{10}$$

where q_i and c_i are the embeddings of the query and code, respectively. $cos(\cdot)$ is the cosine similarity score. k is the batch size.

Joint Learning. Finally, we follow the joint learning paradigm and optimize the three training tasks simultaneously.

TABLE I
STATISTICS OF DATASETS.

Languages	CodeSearchNet				Relevance Pairs		
	Train	Valid	Test	Candidate	Train	Valid	Test
Ruby	24,927	1,400	1,261	4,360	74,781	4,200	3,783
Javascript	58,025	3,885	3,291	13,981	17,4075	11,655	9,873
Java	164,923	5,183	10,955	40,347	494,769	15,549	32,865
Go	167,288	7,325	8,122	28,120	501,864	21,975	24,366
PHP	241,241	12,982	14,014	52,660	723,723	38,946	42,042
Python	251,820	13,914	14,918	43,827	755,460	41,742	44,754

The final training loss is as follows:

$$\mathcal{L}_{\text{joint}} = \lambda_1 \mathcal{L}_{\text{CS}} + \lambda_2 \mathcal{L}_{\text{CTRD}} + \lambda_3 \mathcal{L}_{\text{CTC}} \tag{11}$$

C. Code Search Stage (CS)

After training, we use the trained model to search code. We input the query q, and search codes that match the intent in the given codebase $C = \{c_1, c_2, ..., c_n\}$.

Specifically, the trained model embeds the query q and each code c_i of codebase C as vectors respectively, and computes the cosine similarity between e_q and e_{c_i} using Eq. (12). Finally, the model sorts the code snippets by cosine similarity scores and outputs the top K results which most relevant to the query q.

$$\text{Cosine Similarity}(q, c_i) = \frac{e_q \cdot e_{c_i}}{\|e_q\| \cdot \|e_{c_i}\|} \tag{12}$$

IV. EXPERIMENTAL SETUP

We first enumerate the research questions that we investigate to assess *HedgeCode*. Then, we describe the datasets and baselines used for answering the research questions. Next, we present the evaluation metrics used in our study. Finally, we describe the hyperparameter and experimental environment.

We investigate the advancement and effectiveness of *HedgeCode* through the following four RQs.

- **RQ-1: How does *HedgeCode* perform in code search task compared with state-of-the-art code search models?**
- **RQ-2: How does *HedgeCode* perform when integrating different code representation models?**
- **RQ-3: How effective is each stage of *HedgeCode*?**
- **RQ-4: How effective is *HedgeCode* in special code search scenarios?**

A. Datasets & Baselines

1) Datasets: CodeSearchNet [38] is a widely used benchmark for code search task. It contains six different programming languages. Following previous works [14], [20], [21], [23], [24], [31], we choose CodeSearchNet to train and evaluate *HedgeCode*. The statistical information of CodeSearchNet is shown on the left of Table I.

As described in Section III-A2, we also construct a relevance pair dataset for relevance detection task based on the CodeSearchNet. The statistical information of the relevance pair dataset is shown on the right of Table I.

2) Baselines: In this experiment, we selected nine advanced code search models to compare with *HedgeCode*. The brief introductions of the chosen models are as follows:

- **RoBERTa [39]** is a widely used Transformer-based pre-trained language model.
- **CodeBERT [20]** is a Transformer-based pre-trained programming language model used for multiple code-related tasks.
- **GraphCodeBERT [21]** is a pre-trained model that combines code tokens and data flow information during pre-training.
- **UniXcoder [23]** is a unified contrastive pre-training method that leverages multi-modal content to support code-related tasks.
- **SYNCOBERT [31]** constructs a syntax-guided contrastive pretraining approach.
- **CodeT5 [22]** is a unified pre-trained encoder-decoder transformer model and leverages developer-assigned identifiers to learn the code semantics relevance.
- **CodeRetriever [28]** combines unimodal and bimodal contrastive objectives to learn function-level semantic representations of code snippets.
- **CodeT5+ [32]** is an advanced encoder-decoder LLM for code and it supports a wide range of code understanding and generation tasks.
- **CoCoSoDa [24]** proposes a soft data augmentation method and a multi-modal contrastive objective for the code search task.

B. Metrics

Accuracy. We take a widely used evaluation metric Accuracy to evaluate the relevance detection task of the representation alignment stage. The Accuracy metric is calculated as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{13}$$

where TP, TN, FP and FN denote number of true positive results, true negative results, false positive results, and false negative results, respectively.

MRR. Following previous works [20]–[23], [32], we adopt Mean Reciprocal Rank (MRR) to evaluate the code search task.

$$MRR = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{rank_i} \tag{14}$$

where N is the total number of queries, $rank_i$ is the rank of the ground-truth in all search results.

C. Hyperparameter & Environment

We set the learning rate of the relevance detection task of the representation alignment stage to 10^{-6} and the learning rate of the multi-task joint learning stage to 2×10^{-5}. We set the hyperparameters $\tau = 0.1$, $\lambda = 0.5$, $\lambda_1 = 0.8$, $\lambda_2 = 0.1$, and $\lambda_3 = 0.1$.

We adopt the Adam optimizer to optimize the model. We set the batch size to 64. We set the queue size of DHNS to

5 times the batch size. We adopt the early stop strategy, set the maximum training epochs to 100, and stop the training when the MMR no longer improves in 10 consecutive epochs. All experiments were performed on a machine with two 48G NVIDIA A6000 GPUs.

V. Experiments & Results

We conduct several experiments to answer the four research questions.

A. Effectiveness of HedgeCode

[Experiment Goal]: The goal of RQ-1 is to explore the effectiveness of *HedgeCode*. We answer RQ-1 by comparing the *HedgeCode*'s performance with baselines on CodeSearchNet.
[Experiment Design]: Following previous works, we input the codebase and queries of the test dataset to the trained model. The model outputs the vector representations of queries and codes. Then we compute the cosine similarity between them. For each query, we rank the codes of the codebase according to their cosine similarity score (as shown in Section III-C). Finally, we evaluate the effectiveness of *HedgeCode* by comparing its MRR with baselines.

It is worth noting that *HedgeCode* is model-agnostic and can integrate any code representation model simply. CoCoSoDa [24] is the most advanced code representation model for code search task. So, we implement *HedgeCode* by employing CoCoSoDa as the encoder.
[Experiment Results]: The performances of *HedgeCode* and baselines on CodeSearchNet are shown in Table II. By comparing the MRR scores in the table, it can be found that *HedgeCode* achieves state-of-the-art performance and CoCoSoDa achieves best results in baselines. Specifically, *HedgeCode*'s MRR is 1.5 higher than the most advanced CoCoSoDa on average.

In this RQ, we integrate and optimize CoCoSoDa with *HedgeCode* and *HedgeCode*'s MRR is improved on each programming language of CodeSearchNet compared with the original CoCoSaDa. It shows that *HedgeCode* can improve the representation ability of code LLM (such as CoCoSoDa) through its meticulously designed representation optimization method.

> ✍ **Summary.** *HedgeCode* is effective and advanced. *HedgeCode* can improve the code search performance and achieve the new state-of-the-art performance on code search.

B. Portability of HedgeCode

[Experiment Goal]: The goal of RQ-2 is to explore the portability of *HedgeCode*. We answer RQ-2 by integrating different code representation models and comparing their performances with the original models.
[Experiment Design]: In this experiment, we further choose two wildly used code representation models as encoders and integrate them with *HedgeCode*. Among them, CodeBERT [20] is the first pre-trained code representation model. UniXcoder [23] is an advanced and widely used code representation model.

TABLE II
The MRR scores of HedgeCode and baselines on CodeSearchNet.

Language	Ruby	Javascript	Go	Python	Java	PHP	Avg.
$RoBERTa$	62.8	56.2	85.9	61.0	62.0	57.9	64.3
$CodeBERT$	67.9	62.0	88.2	67.2	67.6	62.8	69.3
$GraphCodeBERT$	70.3	64.4	89.7	69.2	69.1	64.9	71.3
$CodeT5$	71.9	65.5	88.8	69.8	68.6	64.5	71.5
$SYNCOBERT$	72.2	67.7	91.3	72.4	72.3	67.8	74.0
$UniXcoder$	74.0	68.4	91.5	72.0	72.6	67.6	74.4
$CodeT5+$	77.7	70.8	**92.4**	75.6	76.1	69.8	77.1
$CodeRetriever$	77.1	71.9	92.4	**75.8**	**76.5**	**70.8**	77.4
$CoCoSoDa$	**81.8**	**76.4**	92.1	75.7	76.3	70.3	**78.8**
HedgeCode	**82.5**	**77.1**	**92.7**	**77.6**	**78.5**	**73.8**	**80.3(1.5↑)**

TABLE III
The MRR scores of the original models and their HedgeCodes.

Language	Ruby	Javascript	Go	Python	Java	PHP	Avg.
$CodeBERT$	67.9	62.0	88.2	67.2	67.6	62.8	69.3
$UniXcoder$	74.0	68.4	91.5	72.0	72.6	67.6	74.4
$HedgeCode_{CodeBERT}$	72.7	66.3	90.2	71.3	71.2	66.2	73.0(3.7 ↑)
$HedgeCode_{UniXcoder}$	76.2	73.2	91.8	74.6	75.5	70.8	77.0(2.6 ↑)

[Experiment Results]: The performances of *HedgeCode* integrated with different code representation models and their original models are shown in Table III. Specifically, for the two integrated code representation models CodeBERT and UniXcoder, the MRR scores are improved by 3.7 and 2.6 on average, respectively. The results show that *HedgeCode* can effectively improve different code representation models.

Among the models, CodeBERT has significant improvement, because this model does not design pre-training objects to learn the relevance between code and text, so it can further understand the relevance between code and text by integrating it with *HedgeCode*. The improvement of the UniXcoder is limited because UniXcoder can learn the relevance between code and text to a certain extent through unified pre-training and self-supervised CL. However, due to the heterogeneity of code and text and the lack of more fine-grained semantic learning, the understanding of the relevance is still insufficient. So the more fine-grained relevance between code and text can be learned by integrating it with the *HedgeCode*.

In summary, the above results show that understanding the relevance between code and text is crucial to the code search task. *HedgeCode* can better understand the relevance between code and text, so as to improve the performance of code search.

> ✍ **Summary.** *HedgeCode* has good portability. It can easily integrate different code representation models and improve their performance on code search task.

C. Ablation Study of HedgeCode

[Experiment Goal]: The goal of RQ-3 is to study the effectiveness of each training stage of *HedgeCode*. Specifically, we study the following two research questions:

TABLE IV
ACCURACY OF RELEVANCE DETECTION TASK ON REPRESENTATION ALIGNMENT STAGE.

Language	Ruby	Javascript	Go	Python	Java	PHP	Avg.
$CodeBERT_{CE}$	85.67	86.87	86.74	85.27	87.98	84.37	86.15
$UniXcoder_{CE}$	87.92	88.04	88.23	88.26	89.27	86.82	88.09
$CoCoSoDa_{CE}$	90.55	89.42	90.42	89.91	91.34	89.13	90.13
$CodeBERT_{HCL}$	87.51	88.68	88.2	88.31	89.21	88.92	88.47
$UniXcoder_{HCL}$	89.86	89.92	90.42	90.98	90.78	88.85	90.14
$CoCoSoDa_{HCL}$	92.67	91.56	92.36	93.18	93.56	91.59	92.49

- **RQ-3.1: Whether the semantic space of code and text be aligned through the relevance detection task in the representation alignment stage?**
- **RQ-3.2: What impact does each task of the multi-task joint learning stage have?**

[Experiment Design (RQ-3.1)]: We answer RQ-3.1 by evaluating the accuracy of the relevance detection task and visualizing the representation spaces of code and text.

Firstly, we evaluate the relevance detection task. The relevance detection task takes the pair of code and its description as input and then outputs the detection result. We use detection accuracy to evaluate the detection performance. To evaluate the effectiveness of HCL, we train the model with the cross-entropy loss (CE) and HCL, respectively.

Then, we visualize the representation spaces of code and text to study whether the representation spaces are aligned. We randomly sample 500 pairs from the test dataset and employ the encoder trained by the relevance detection task to embed them, respectively. We visualize them by the t-SNE algorithm [34] and compare the representation distributions before and after the representation alignment stage.

Finally, we investigate the overall effectiveness of the representation alignment stage by removing it from the *HedgeCode*.

[Experiment Results (RQ-3.1)]: The performances of the relevance detection task are shown in Table IV. Among them, the $CoCoSoDa_{HCL}$ achieves the highest accuracy. For each encoder (CodeBERT, UniXcoder, and CoCoSoDa), the detection accuracies of the models trained by the HCL method are better than those trained by the CE, which shows the effectiveness of the HCL.

The visualizations of the representation spaces are shown in Fig. 5. It can be seen that after representation space alignment, the distributions of code and text are more uniform and resemblant.

The ablation experiment results of the representation alignment stage are shown in Table V. We can see that the MRR of *HedgeCode* decreased by 0.6 compared to the complete *HedgeCode* on average without representation alignment. When only optimizing *HedgeCode* with representation alignment, the MRR of *HedgeCode* improved by 0.5 compared to the original model. These results illustrate the effectiveness of the representation alignment stage.

[Experiment Design (RQ-3.2)]: We answer RQ-3.2 by studying the effectiveness of the CTC task or CTRD task. Specifically, we train the CS task with the CTC task or CTRD task, respectively.

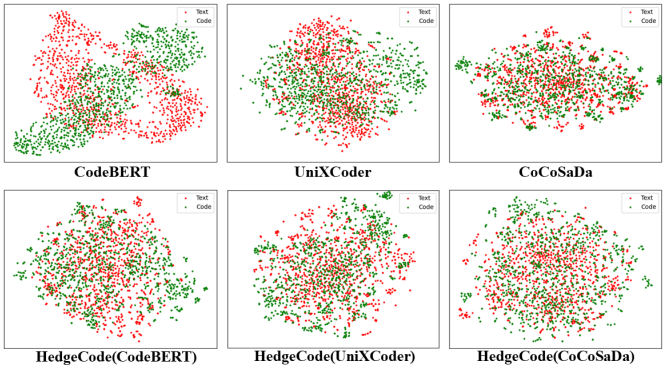

Fig. 5. The visualizations of *HedgeCodes* and original models.

TABLE V
THE RESULTS OF ABLATION STUDIES.

Language	Ruby	Javascript	Go	Python	Java	PHP	Avg.
$CoCoSoDa$	81.8	76.4	92.1	75.7	76.3	70.3	78.8
$+CTC$	81.9	76.5	92.3	76.4	76.6	70.9	79.1
$+CTRD$	82.1	76.5	92.2	76.7	76.9	71.4	79.3
$+CTRD+CTC$	82.3	76.7	92.4	77.1	77.5	72.1	79.7
$+RA$	82.0	76.6	92.2	76.8	76.9	71.5	79.3
$+RA+CTC$	82.2	76.7	92.5	77.0	77.9	72.6	79.9
$+RA+CTRD$	82.3	76.8	92.5	77.3	78.2	72.9	80.0
HedgeCode	**82.5**	**77.1**	**92.7**	**77.6**	**78.5**	**73.8**	**80.3**

***Notes:** RA is short for Representation Alignment Stage. CTC is short for Code-Text Contrast Learning Task. CTRD is short for Code-Text Relevance Detection Task.*

[Experiment Results (RQ-3.2)]: The results of ablation studies are shown in Table V. The results show that the complete *HedgeCode* achieved optimal results. When the CTC and CTRD are used alone without representation alignment, the performance of *HedgeCode* is improved compared with the original model. When adopting the CTC and CTRD alone after representation alignment, the performance of the code representation model can be further improved compared with the performance of *HedgeCode* only optimized with representation alignment. These results show that the CTC and CTRD are complementary, and the performance of the *HedgeCode* can be further improved when they are used together. Even after the representation alignment stage, the code representation model can still benefit from these two tasks.

✍ **Summary.** Both training stages of *HedgeCode* are valid. The representation spaces of code and text are aligned after the representation alignment stage. Each task of the multi-task joint learning stage has a positive effect on the code search task.

D. Effectiveness of HedgeCode in Special Code Search Scenarios

[Experiment Goal]: In order to investigate *HedgeCode*'s effectiveness in special code search scenarios, we explore the effectiveness of *HedgeCode* under zero-shot and few-shot settings. Specifically, we study the following two research questions:

TABLE VI
THE MRR SCORES OF HEDGECODE IN ZERO-SHOT CODE SEARCH SCENARIO.

Language	Ruby	Javascript	Go	Python	Java	PHP	Avg.
$HedgeCode_{CodeBERT}$	69.8	63.5	88.9	68.8	68.7	63.5	70.5(1.2↑)
$HedgeCode_{UniXcoder}$	74.9	69.5	91.5	72.3	73.3	68.5	75.0(0.6↑)
$HedgeCode_{CoCoSoDa}$	82.0	76.6	92.2	76.8	76.9	71.5	79.3(0.5↑)

TABLE VII
THE MRR SCORES OF HEDGECODES IN FEW-SHOT CODE SEARCH SCENARIO.

Language	Ruby	Javascript	Go	Python	Java	PHP	Avg.
$HedgeCode_{CodeBERT}$	71.6	65.6	89.2	69.8	69.8	64.9	71.8(2.5↑)
$HedgeCode_{UniXcoder}$	75.6	71.6	91.6	72.9	74.4	69.8	76.0(1.5↑)
$HedgeCode_{CoCoSoDa}$	82.1	76.8	92.1	77.2	77.5	72.6	79.7(0.9↑)

- **RQ-4.1: How effective is *HedgeCode* in the zero-shot code search scenario?**
- **RQ-4.2: How effective is *HedgeCode* in the few-shot code search scenario?**

[Experiment Design (RQ-4.1)]: In this study, the zero-shot code search scenario refers to code search using the model trained after the representation space alignment stage. Therefore, in this RQ, we aim to study: ***How well does the detector perform on the code search task?***

When using the detector to search, the detection pairs are unable to be input like search task in matrix form. It will cause an excessively large search space problem. To reduce the search space and improve the search efficiency, we introduced the Ball-Tree [40], which is an efficient high-dimensional vector matching algorithm. We first use the trained encoder to represent queries and codes as high-dimensional vectors. Then the Ball-Tree is used to recall N codes from the codebase before detection by comparing the similarity between the vectors of queries and codes. In this RQ, we set $N = 1000$.

Then, we combine the query and each recalled code as the detection pair. These pairs are input to the detector for detection. The detector outputs the relevance detection result (0 or 1) and the relevance score ($0 \sim 1$). We collect pairs whose detection result is 1 and rank them by the relevance score. The corresponding codes are returned as the code search result. We adopt MRR to evaluate the performance of the detector on the code search.

It is worth noting that the previous models search the codes by calculating the cosine similarity between code and query vectors. Unlike these models, we first recall the top N codes by calculating the spatial distance of vectors, and then performed more fine-grained matching and ranking through the detector to improve the accuracy of code search.

[Experiment Results (RQ-4.1)]: The experiment results of *HedgeCode* in the zero-shot code search scenario are shown in Table VI. We can find that after the representation alignment stage, the code search performance of each integrated encoder is improved. Among them, $HedgeCode_{CoCoSoDa}$ achieves the best result, whose MRR is 0.5 higher than CoCoSoDa.

[Experiment Design (RQ-4.2)]: In this study, the few-shot code search scenario refers to training the model only with a small amount of data during the multi-task joint learning stage. Therefore, in this RQ, we aim to investigate: ***How well does the HedgeCode perform on the code search when only trained with a few data?*** Specifically, we explore the performance of *HedgeCode* under the few-shot scenario by randomly sampling 20% of code-text pairs from the training data to further train the model.

[Experiment Results (RQ-4.2)]: The experiment results of the few-shot code search are shown in Table VII. We can find that the performance of *HedgeCode* can further improve after multi-task joint training with a few data. Among them, $HedgeCode_{CoCoSoDa}$ achieves the best result, whose MRR is 0.9 higher than CoCoSoDa and 0.4 higher than $HedgeCode_{CoCoSoDa}$ in the zero-shot scenario.

✍ **Summary.** Through representation alignment, the original model can optimize code and text representations, which enables accurate code search in the zero-shot scenario. *HedgeCode* can learn the fine-grained relevance between code and text from a few data by the multi-task joint learning, which enables accurate code search in the few-shot scenario.

VI. THREATS TO VALIDITY

Although *HedgeCode* has the overall effectiveness, there are several threats:

Constructed Threat. *HedgeCode* is model-agnostic, we evaluate its portability by integrating three typical models, including CodeBERT [20], UniXcoder [23] and CoCoSoDa [24]. Whether the proposed method is applicable to other pre-trained source code models such as CodeT5 [22], has not been explored.

Internal Threat. Firstly, following previous works [23], [24], *HedgeCode* samples negative samples from the mini-batch, so its effectiveness is influenced by the batch size. Secondly, in the existing benchmark, the paired code is the correct result of a given query. In fact, there may be multiple codes that match the intent of the given query. However, this is not fully considered in the existing benchmark.

External Threat. Although we have evaluated the effectiveness of *HedgeCode* in six widely used programming languages. However, limited by available datasets, many programming languages are still not evaluated. In order to evaluate the universality of *HedgeCode*, we need to evaluate it in more programming languages in the future.

VII. RELATED WORK

In this section, we describe the related work to highlight the relevance and the novelty of our work.

A. Code search

The existing code search methods can be divided into IR-based and DL-based methods.

IR-based methods focus on lexical matching between natural language texts and codes. McMillan et al. [10] proposed

Portfolio, which searches API sequences by keyword matching and PageRank algorithm. Linstead et al. [12] proposed Sourcerer, which combines the textual content and structural information of programs to achieve code search. Lv et al. [11] proposed CodeHow, which extends queries with APIs and considers the effect of textual similarity and latent APIs. However, these methods are susceptible to tokens of codes and queries and cannot understand the deep semantics.

To achieve semantic matching, DL-based methods have been introduced for code search. Gu et al. [6] proposed the first DL-based code search method DeepCS, which employs the Long Short-Term Memory Network (LSTM) [41] and multilayer perceptron (MLP) [42] to represent queries and codes. Cambronero et al. [13] proposed UNIF, which adopts fastText [43] and attention mechanism [44] to achieve better performance. Zhu et al. [45] proposed OCoR, which constructs an overlap matrix to capture the character-level overlap between tokens of code and text. Yang et al. [14] proposed TabCS, which introduces structural features to better represent code semantics. Cheng et al. [16] proposed CSRS, which takes advantage of IR-based methods and DL-based methods to capture semantic and lexical correlation. Cheng et al. [17] proposed TranCS, which translates codes into Intermediate representation provided in natural language to alleviate the insufficient semantic understanding of code and embedding differences between text and code. Chai et al. [15] proposed CDCS, which introduces meta-learning [46] to code search and extends the pre-train and fine-tune paradigm with a transfer learning framework.

HedgeCode is a DL-based code search method, which improves the performance of code search by optimizing the code and text representation space and capturing their fine-grained semantic relevance through meticulously designed multi-task hedging contrastive learning.

B. Code Large Language Model

Inspired by the successful application of large language models (LLMs) such as BERT [35] and GPT [47] in natural language processing (NLP). Recently, some researchers have introduced the LLMs into the programming language processing domain to improve code understanding and achieve significant improvements in code search tasks. Feng et al. [20] present CodeBERT, which is the first bimodal pre-trained code LLM. Guo et al. [21] proposed GraphCodeBERT, which combines code tokens and data flow information during pre-training to improve the semantic understanding of the code. Wang et al. [22] present CodeT5, which is a unified pre-trained encoder-decoder transformer model and leverages developer-assigned identifiers to learn the code semantics relevance. Wang et al. [32] proposed CodeT5+, which is a family of encoder-decoder LLM for code and suit a wide range of downstream code tasks.

HedgeCode is a model-agnostic framework and can easily integrate existing code LLMs and improve their performance on code search.

C. Contrastive Learning

There are two contrastive learning (CL) paradigms: self-supervised and supervised contrastive learning. Self-supervised CL is a popular representation learning approach and has been successfully applied in CV [25], [26] and NLP [27], [48]–[50]. The basic idea of self-supervised CL is that good representations should be able to recognize the same object while distinguishing themselves from other objects.

Recently, self-supervised CL has been widely used in programming language processing. To address the long tail problem and the challenge of cross-language representation, Li et al. [28] proposed CodeRetriever, which learns function-level code semantic representation through unimodal and bimodal CL. Bui et al. [30] used the program transformation technique to construct positive samples and used self-supervised CL method to identify equivalent or unequivalent codes with similar tokens. Ding et al. [29] proposed BOOST, which adopts the bug injection method to construct negative samples and designs a pre-training task to learn code structure. Wang et al. [31] proposed SYNCOBERT, which obtains a more comprehensive code representation from three modalities via grammar-guided multi-modal self-supervised CL. Li et al. [24] propose CoCoSoDa, which adopts multi-modal self-supervised CL and soft data augmentation method for code search and achieves SOTA performance.

The basic idea of supervised contrastive learning (SCL) is maximizing the distance of training samples with the same labels while minimizing the distance between different labels. Khosla et al. [33] extended the CL to the supervised task. SCL treats samples with the same label as positive pairs and samples with different labels as negative pairs and achieves significant improvements in the image classification task. Gunel et al. [51] extend supervised CL to the NLP domain. Chen et al. [36] propose dual contrastive learning (DualCL) to improve text classification tasks by using label information and sample features to optimize the text representation and classifier via contrastive learning, simultaneously.

Unlike existing CL-based code search methods, we propose *HedgeCode*, which is a multi-task hedging contrastive learning framework integrated with self-supervised and supervised CL. We designed hedging contrastive learning (HCL) to align the representation spaces between code and text and capture fine-grained semantics differences between codes. DualCL is a pivotal element of HCL. Due to differences in data characteristics, the original DualCL cannot be directly migrated to the CTRD task. We enhance the original DualCL with the designed data augmentation strategy.

VIII. Conclusion & Future Work

To tackle the challenges posed by the disparities in representation spaces between code and natural language descriptions, as well as the limitations faced by current code search techniques in capturing fine-grained semantic distinctions, we introduced *HedgeCode*. Our empirical findings demonstrate that *HedgeCode* is capable of effectively synchronizing the

representation spaces of code and natural language, thereby markedly enhancing the efficacy of code search task.

Despite the overarching benefits of *HedgeCode*, it occasionally delivers imprecise outcomes, particularly with code involving third-party library APIs or custom functions. As a direction for future research, we plan to augment our framework by incorporating a broader spectrum of related external information. This enhancement aims to further refine the accuracy and performance of *HedgeCode*.

IX. Acknowledgments

This work was supported by National Natural Science Foundation of China (Grant No. 62250610224) and the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant No. 949014).

References

[1] K. Kim, S. Ghatpande, D. Kim, X. Zhou, K. Liu, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Big code search: A bibliography," *ACM Computing Surveys*, vol. 56, no. 1, August 2023.

[2] Y. Xie, J. Lin, H. Dong, L. Zhang, and Z. Wu, "Survey of code search based on deep learning," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, p. 1–42, December 2023.

[3] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2020, p. 196–207.

[4] J. Li, F. Liu, J. Li, Y. Zhao, G. Li, and Z. Jin, "MCodeSearcher: Multi-view contrastive learning for code search," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. New York, NY, USA: Association for Computing Machinery, October 2023, p. 270–280.

[5] Z. Li, G. Yin, T. Wang, Y. Zhang, Y. Yu, and H. Wang, "Correlation-based software search by leveraging software term database," *Frontiers of Computer Science*, vol. 12, no. 5, pp. 923–938, October 2018.

[6] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 933–944.

[7] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys*, vol. 54, no. 9, October 2021.

[8] L. Di Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *ACM Computing Surveys*, vol. 55, no. 11, February 2023.

[9] Y. Hu, H. Jiang, and Z. Hu, "Measuring code maintainability with deep neural networks," *Frontiers of Computer Science*, vol. 17, no. 6, p. 176214, January 2023.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *2011 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[11] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "CodeHow: Effective code search based on api understanding and extended boolean model," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 260–270.

[12] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: Mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, p. 300–336, April 2009.

[13] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019, p. 964–974.

[14] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, "Two-stage attention-based model for code search with textual and structural features," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021, pp. 342–353.

[15] Y. Chai, H. Zhang, B. Shen, and X. Gu, "Cross-domain deep code search with meta learning," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022, p. 487–498.

[16] Y. Cheng and L. Kuang, "CSRS: Code search with relevance matching and semantic matching," in *2022 IEEE/ACM 30th International Conference on Program Comprehension*, 2022, pp. 533–542.

[17] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, and Q. Zhang, "Code search based on context-aware code translation," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, July 2022, p. 388–400.

[18] Y. Shi, Y. Yin, Z. Wang, D. Lo, T. Zhang, X. Xia, Y. Zhao, and B. Xu, "How to better utilize code graphs in semantic code search?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022, p. 722–733.

[19] S. Fang, Y.-S. Tan, T. Zhang, and Y. Liu, "Self-attention networks for code search," *Information and Software Technology*, vol. 134, p. 106542, 2021.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics*. Association for Computational Linguistics, November 2020, pp. 1536–1547.

[21] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[22] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, November 2021, pp. 8696–8708.

[23] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, vol. 1. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225.

[24] 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 *Proceedings of the 45th International Conference on Software Engineering*. IEEE Press, 2023, p. 2198–2210.

[25] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 37th International Conference on Machine Learning*, vol. 119. PMLR, July 2020, pp. 1597–1607.

[26] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9726–9735.

[27] Y. Yan, R. Li, S. Wang, F. Zhang, W. Wu, and W. Xu, "ConSERT: A contrastive framework for self-supervised sentence representation transfer," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, vol. 1. Association for Computational Linguistics, August 2021, pp. 5065–5075.

[28] X. Li, Y. Gong, Y. Shen, X. Qiu, H. Zhang, B. Yao, W. Qi, D. Jiang, W. Chen, and N. Duan, "CodeRetriever: A large scale contrastive pre-training method for code search," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, December 2022, pp. 2898–2910.

[29] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Contrastive learning for source code with structural and functional properties," *CoRR*, vol. abs/2110.03868, 2021.

[30] N. D. Q. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2021, p. 511–521.

[31] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation," *CoRR*, vol. abs/2108.04556, 2021.

[32] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, December 2023, pp. 1069–1088.

[33] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., December 2020, pp. 18 661–18 673.

[34] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, vol. 1. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186.

[36] Q. Chen, R. Zhang, Y. Zheng, and Y. Mao, "Dual contrastive learning: Text classification via label-aware data augmentation," *CoRR*, vol. abs/2201.08702, 2022.

[37] R. D. Hjelm, A. Fedorov, S. Lavoie-Marchildon, K. Grewal, P. Bachman, A. Trischler, and Y. Bengio, "Learning deep representations by mutual information estimation and maximization," in *International Conference on Learning Representations*, 2019.

[38] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodesearchNet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[39] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized bert pretraining approach," *CoRR*, vol. abs/1907.11692, 2019.

[40] T. Liu, A. W. Moore, and A. Gray, "New algorithms for efficient high-dimensional nonparametric classification," *J. Mach. Learn. Res.*, vol. 7, p. 1135–1158, December 2006.

[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] M. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric Environment*, vol. 32, no. 14, pp. 2627–2636, 1998.

[43] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, vol. 2. Valencia, Spain: Association for Computational Linguistics, April 2017, pp. 427–431.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[45] Q. Zhu, Z. Sun, X. Liang, Y. Xiong, and L. Zhang, "OCoR: An overlapping-aware code retriever," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 883–894.

[46] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70. PMLR, August 2017, pp. 1126–1135.

[47] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[48] H. Fang, S. Wang, M. Zhou, J. Ding, and P. Xie, "CERT: Contrastive self-supervised learning for language understanding," *CoRR*, vol. abs/2005.12766, 2020.

[49] T. Gao, X. Yao, and D. Chen, "SimCSE: Simple contrastive learning of sentence embeddings," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, November 2021, pp. 6894–6910.

[50] J. Giorgi, O. Nitski, B. Wang, and G. Bader, "DeCLUTR: Deep contrastive learning for unsupervised textual representations," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, vol. 1. Association for Computational Linguistics, August 2021, pp. 879–895.

[51] B. Gunel, J. Du, A. Conneau, and V. Stoyanov, "Supervised contrastive learning for pre-trained language model fine-tuning," in *International Conference on Learning Representations*, 2021.