Reduce Redundancy then Rerank: Enhancing Code Summarization with a Novel Pipeline Framework

Xiaoyu Hu†, Xu Zhang†, Zexu Lin, Deyu Zhou‡
School of Computer Science and Engineering, Southeast University, Nanjing, China
Key Laboratory of New Generation Artificial Intelligence Technology and Its Interdisciplinary Applications (Southeast University), Ministry of Education, China
{xiaoyuhu, xuzhang123, zxlin, d.zhou}@seu.edu.cn

Abstract

Code summarization is the task of automatically generating natural language descriptions from source code. Recently, pre-trained language models have gained significant popularity in code summarization due to their capacity to capture richer semantic representations of both code and natural language. Nonetheless, contemporary code summarization models grapple with two fundamental limitations. (1) Some tokens in the code are irrelevant to the natural language description and damage the alignment of the representation spaces for code and language. (2) Most approaches are based on the encoder-decoder framework, which is often plagued by the exposure bias problem, hampering the effectiveness of their decoding sampling strategies. To address the two challenges, we propose a novel pipeline framework named Reduce Redundancy then Rerank (Re^3). Specifically, a redundancy reduction component is introduced to eliminate redundant information in code representation space. Moreover, a re-ranking model is incorporated to select more suitable summary candidates, alleviating the exposure bias problem. The experimental results show the effectiveness of Re^3 over some state-of-the-art approaches across six different datasets from the CodeSearchNet benchmark.

Keywords: Code Summarization, Reduce Redundancy, Rerank

1. Introduction

Code summarization is essential to program comprehension and software maintenance vital in the entire software life cycle. Due to the expense of writing these summaries manually, a holy grail of software engineering research has long been to generate these summaries automatically (Haque et al., 2023).

The state-of-the-art code summarization models tend to follow the encoder-decoder paradigm, which first encodes the code into a distributed vector by pre-trained language models (PLMs) for code intelligence and then decodes it into natural language summary (Wu et al., 2021; Son et al., 2022). However, these code summarization models grapple with two fundamental limitations. The first challenge is the presence of a surplus of redundant tokens within the code, creating a substantial gap between the concise natural language descriptions that need to be generated and the intricate code. Bridging this gap and aligning the representation spaces for code and language is formidable. Figure 1 shows two examples where redundant code tokens significantly affect the performance of the code summarization model. When the model begins decoding with a redundant token in the wrong direction, the model will end up generating a short and low-quality summary. Although some works (Hu et al., 2018b; Gao et al., 2021) implemented by function names or APIs attempt to use more refined information to avoid the impact of redundant tokens, the performance improvement brought by these methods is always limited. Because redundant tokens will appear not only in the function body but also in the function name, i.e., "safe" in the left code example. Therefore, we try to reduce redundancy in the code representation to deal with this challenge.

```
def safe_infer(
    node: astroid.node_classes.NodeNG,
    context=None
    )> Optional[astroid.node_classes.NodeNG]:
    try:
        inferit = node.infer(context=context)
        value = next(inferit)
    except astroid.InferenceError:
        return None
    try:
        next(inferit)
        return None
    except astroid.InferenceError:
        return None
    except StopIteration:
        return value
```

Golden:
Return the inferred value for the given node.

Generated:
Safely infer a node.

```
def execute(self, context):
    hook=GoogleCloudStorageHook(
        self.google_cloud_storage_conn_id,
        self.delegate_to
    )
    hook.upload(
        bucket_name=self.bucket
        object_name=self.dst
        mime_type=self.mime_type
        filename=self.src
        gzip=self.gzip
    )
```

Golden:
Uploads the file to Google cloud storage.

Generated:
Execute the hook.

Figure 1: Two examples of code summarization models ending up generating low-quality summaries due to redundant tokens. For the left example, the redundant token is **safe** in the function name; For the right example, the redundant token is the variable name **hook** in the function body.

† These authors contributed equally to this work.
‡ Corresponding author.

LREC-COLING 2024, pages 13722–13733

The second challenge concerns the exposure bias problem (Bengio et al., 2015; Ranzato et al., 2016) in encoder-decoder based generative models. This issue hampers the effectiveness of decoding sampling strategies employed by prevalent models, limiting their ability to generate accurate and coherent summaries. In Figure 2, we illustrate this phenomenon with the difference between top beam search scores (the score of the first summary generated directly by beam search) and oracle scores (the maximum score over all summary candidates generated by beam search) on the Ruby and Javascript dataset of CodeSearchNet (Husain et al., 2019) with a UniXcoder (Guo et al., 2022) model. As we can see, oracle scores are significantly higher than top beam search scores, and the gap further widens as the number of candidates increases. This result suggests that current encoder-decoder based code summarization models with PLMs are not exploited to their total capacity, calling for better methods to identify the best summary candidate.

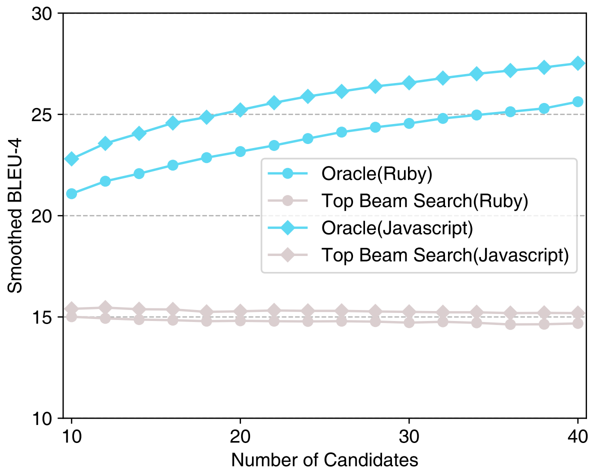

Figure 2: **Top Beam Search scores** (the score of the first summary generated directly by beam search) and **Oracle scores** (the maximum score over all summary candidates generated by beam search) with different numbers of candidates for UniXcoder on Ruby and Javascript dataset.

To address these persistent challenges, we present a novel pipeline framework named Reduce Redundancy then Rerank (Re^3). It consists of two essential stages. (1) Reduce Redundancy: it mainly involves implementing a covariance regularization strategy to reduce redundant information in the code representation space. By eliminating the effect of redundant tokens, this strategy paves the way for a better next stage of summary candidate selection. (2) Rerank: a re-ranking model is introduced to learn and implement more effective summary candidate selection strategies through metric learning. This approach addresses the exposure bias problem, enabling the model to make better-informed decisions when generating code summaries.

The main contribution of our work is the proposal of a new pipeline code summarization framework called Re^3. In the first stage, the redundancy reduction strategy removes unnecessary information from the code representation space, improving the quality of generated summary candidates. In the second stage, the re-ranking model is incorporated to choose better summarization candidates, thus mitigating the exposure bias problem. Experiments of the SOTA performance across six different programming language datasets of CodeSearchNet show the effectiveness of this framework. Our code is available at https://github.com/maebymaeby/Re3.

2. Related Work

2.1. Code Summarization

Automatic code summarization mainly deployed information retrieval techniques in the early stage research (Haiduc et al., 2010a,b). With the advancements in deep learning and neural machine translation, a shift occurred, and researchers began to explore code summarization using sequence-to-sequence neural networks, as demonstrated by (Iyer et al., 2016; Hu et al., 2018a; Wan et al., 2018; LeClair et al., 2020). Recently, the field has witnessed the rise of PLMs for code intelligence that has gained popularity, as highlighted in Feng et al. (2020); Guo et al. (2021); Ahmad et al. (2021); Wang et al. (2021); Guo et al. (2022). These pre-trained models are typically trained on extensive, multi-programming language datasets to capture the semantic nuances of code better. At the same time, there have also been some works with large language models such as Wang et al. (2023). Notably, the state-of-the-art code summarization models generally adhere to the encoder-decoder paradigm, where the code is first encoded into a distributed vector by a PLM and then decoded into a natural language summary, as exemplified by the recent work (Wu et al., 2021; Son et al., 2022).

Based on our observations, there are two main challenges in current code summarization models. First, code often contains redundant information, making it challenging to connect code and language. Second, these models struggle with exposure bias, affecting their decoding accuracy. To tackle these issues, we propose a new pipeline framework named Reduce Redundancy then Rerank.

2.2. Redundancy Reduction of Representation

Despite the significant progress that deep neural networks have achieved, recent studies (Gururangan et al., 2018; McCoy et al., 2019; Zhang et al., 2021, 2024) have found that these models often rely on spurious correlations between learned features and prediction labels, leading to instability and poor generalization to data with different distributions. For example, previous studies (Gururangan et al., 2018; McCoy et al., 2019) have demonstrated that specific linguistic phenomena or syntactic heuristics correlate highly with certain Natural Language Inference (NLI) inference classes. Zbontar et al. (2021) proposed an objective function that utilizes the cross-correlation matrix between the outputs of two identical networks that are fed with distorted versions of a sample to minimize redundancy while ensuring that the embedding vectors of distorted versions of a sample are similar. Ermolov et al. (2021) offered an alternative direction by introducing a new loss function based on whitening the latent space features, which avoids degenerate solutions where all the sample representations collapse to a single point. Unlike Zbontar et al. (2021), the method introduced by Ermolov et al. (2021) does not require asymmetric networks and is conceptually straightforward. To further prevent informational collapse, Bardes et al. (2022) proposed VICReg. This method employs two regularization terms to ensure that the variance of each embedding dimension remains above a threshold and that each pair of variables is de-correlated. Additionally, VICReg attracts covariances over a batch between every pair of centered embedding variables towards zero, effectively preventing variables from varying or highly correlated.

While redundancy reduction techniques have demonstrated notable success in various domains, scant attention has been devoted to this matter in code summarization undertakings. Our objective is to address the issue of code representation redundancy.

2.3. Re-ranking in Natural Language Generation

Re-ranking has long been adopted in several Natural Language Generation (NLG) branches. In neural machine translation, Bhattacharyya et al. (2021) uses an energy-based model on top of BERT (Devlin et al., 2019) to select translation candidates with higher BLEU scores. In text-to-SQL generation, Xiang et al. (2023) presents a knowledge-enhanced re-ranking mechanism proposed to introduce domain knowledge to choose the best SQL query from the beam output. Recently, two-stage pipeline approaches with a re-ranking model have been widely used in abstract summarization. These approaches work based on the generate-then-rerank framework, which generates some candidate texts with a first-stage generator and then reranks them with a second-stage reranker. SimCLS (Liu and Liu, 2021), RefSum (Liu et al., 2021) and SummaReranker (Ravaut et al., 2022) train re-ranking models separately to re-rank the outputs of summarization models such as BART (Lewis et al., 2020). Furthermore, some works tried to compress the two-stage pipeline to one single model using extra training objectives, such as Colo (An et al., 2022) and BRIO (Liu et al., 2022).

Although re-ranking techniques have been explored sufficiently in various domains of NLG, scant attention has been devoted to this matter in code summarization undertakings. We aim to mitigate the exposure bias in the encoder-decoder paradigm based on a second-stage re-ranking model.

3. Methodology

3.1. Overview

We propose a pipeline framework named Re^3 (Reduce Redundancy then Rerank), which addresses redundancy and exposure bias problems through the two-stage models. Figure 3 illustrates the overall architecture of Re^3, which consists of an encoder-decoder based code summarization model (the first stage model) and a re-ranking model (the second stage model). The first stage model is optimized during the training time by combining the neural code summarization framework of Maximum Likelihood Estimation (MLE) and redundancy reduction of code representation, then generating summary candidates in the inference time. The second stage model re-ranks the natural language summary candidates from the perspective of the relationship between source code and gold summary, which is trained through metric learning methods to learn and implement a more compelling candidate selection strategy.

3.2. Stage I: Reduce Redundancy

Neural Code Summarization Given a code snippet C and its corresponding golden summary $\hat{S}$, the neural code summarization model aims to train a model $\mathcal{G}_\theta$ parameterized by θ that takes a source code C and generates an appropriate summary S.

$$S \leftarrow \mathcal{G}_\theta(D) \tag{1}$$

In practice, the MLE algorithm is employed for the encoder-decoder based neural code summarization model to maximize the likelihood of the golden summary. For a specific training sample $\{C, \hat{S}\}$, the training objective is to minimize the sum

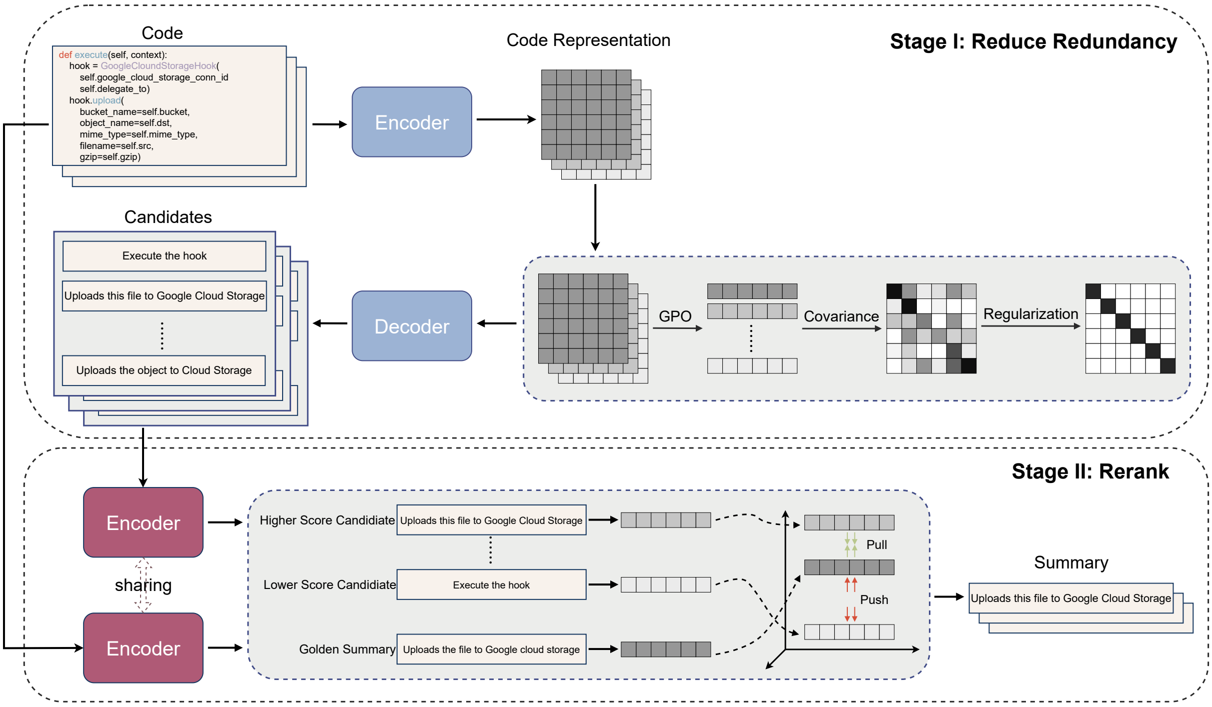

Figure 3: The architecture of Re^3.

of Negative Log-Likelihoods (NLL) of the golden summary tokens $\hat{S} = \{\hat{s}_1, \dots, \hat{s}_l\}$

$$\mathcal{L}_{NLL} = -\sum_{j=1}^{l}\sum_{i}^{l} P(s_i|\hat{S}_{<j}) log P_{\mathcal{G}_\theta}(s_i|C, \hat{S}_{<j}; \theta)$$

$$P(s_i|\hat{S}_{<j}, S) = \begin{cases} 1 & s_i = \hat{s}_j \\ 0 & s_i \neq \hat{s}_j \end{cases} \tag{2}$$

where $S = \{s_1, \dots, s_l\}$ is the generated summary tokens. $\hat{S}_{<j}$ denotes the partial previous ground truth from golden summary$\{\hat{s}_1, \dots, \hat{s}_j\}$

Redundancy Reduction of Code Representation Our target is to eliminate potential redundant information in the code representation, which also means that the feature vectors corresponding to different code fragments should have more considerable differences at the representation level. Inspired by previous work (Zbontar et al., 2021; Bardes et al., 2022), we encourage diversity and reduce redundancy by enforcing minimal covariance between different code representations.

Here, we describe a batch of code snippets $\mathbb{C} = \{C_1, \dots, C_m\}$ whose length is m. We use the encoder of the encoder-decoder based summarization model to obtain the code representation:

$$H_{C_i} = \underset{\tilde{C}_i \in \tilde{\mathbb{C}}}{Encoder}(C_i) \tag{3}$$

where $H_{C_i} \in \mathbb{R}^{N \times D}$ denotes the code representation feature matrix of code snippet C_i with length N and feature shape D.

To effectively aggregate the code representation feature matrix, instead of the traditional pooling approaches like maximum pooling or mean pooling, the Generalized Pooling Operator (GPO) (Chen et al., 2021) is employed to get feature vector

$$H_i = \underset{\tilde{C}_i \in \tilde{\mathbb{C}}}{GPO}(H_{C_i}) \tag{4}$$

where $H_i \in \mathbb{R}^{1 \times D}$ represents the dimensionally reduced feature vector after GPO aggregation.

We propose to define the covariance matrix of the batch of code feature vectors $\mathbb{H} = \{H_1, \dots, H_M\}$ as:

$$Cov(\mathbb{H}) = \frac{1}{M-1}\sum_{i=1}^{M}(H_i - \bar{H}_i)(H_i - \bar{H}_i)^T$$
$$\bar{H}_i = \frac{1}{M}\sum_{i=1}^{M}(H_i) \tag{5}$$

We then define the covariance regularization $c(\mathbb{H})$ as the sum of the squared off-diagonal coefficients of $Cov(\mathbb{H})$, and introduce the dimension scaling factor $\frac{1}{D}$ to construct the covariance loss:

$$c(\mathbb{H}) = \sum_{i \neq j}[Cov(\mathbb{H})]^2_{i,j}$$
$$\mathcal{L}_{COV} = \frac{c(\mathbb{H})}{D} \tag{6}$$

Minimizing the covariance loss encourages the off-diagonal coefficients of $c(\mathbb{H})$ to be close to 0, decorating the different feature vectors and preventing them from encoding similar information, ultimately having a decorrelation effect at the representation level.

The final training objective of the first stage model achieves a minimal two losses mentioned above:

$$\mathcal{L} = \mathcal{L}_{NLL} + \mathcal{L}_{COV} \tag{7}$$

Candidates Generation For generating a single summary, given a decode sampling strategy $\mathcal{D}$, the decoder maintains a list of top-k best summary candidates and outputs the best candidate based on $\mathcal{D}$ and the last $(k-1)$ candidates are discarded. To generate multiple summary candidates for the second stage model training, we modify the above strategy of the first stage model decoder and retain the complete list of top-k best candidates $S_1, \ldots, S_k$.

3.3. Stage II: Rerank

Metric Learning After Stage I, we get a pool of k summary candidates $\mathbb{S} = \{S_1, \ldots, S_k\}$. In the training time of the first model, the Teacher-Forcing algorithm (Williams and Zipser, 1989) is used for the encoder-decoder based code summarization model training and parallel loss calculation under the MLE framework. The model uses an autoregressive decoding sampling strategy (such as the beam search algorithm) to generate the output summary in the inference time. That means the model generates the token of step j according to $\hat{S}_{<j}$ in the training stage and generates the token of step j by autoregressive generation in the inference stage, which leads to an inherent discrepancy called exposure bias. It may result in the first summary S_1 generated by a simple decoding sampling strategy being far different from the golden summary $\hat{S}$.

Inspired by the previous work (Liu and Liu, 2021; Liu et al., 2021; Ravaut et al., 2022), we try to train a second stage model to re-rank summary candidates $\mathbb{S}$ from the perspective of the relationship between source code snippet C and golden summary $\hat{S}$. We introduce a PLM for code intelligence as a re-ranking model, as it shows good performance in code retrieval, which can help us coordinate the relationship between code and natural language summaries at the semantic level. However, code retrieval focuses on pairing between different code and natural language, and our target is to pair source code and natural language summaries corresponding to the same code. Therefore, we further fine-tune the second model by metric learning.

Given an evaluation metric $\mathcal{M}(\cdot)$, We get a new pool of k summary candidates $\tilde{\mathbb{S}} = \{\tilde{S}_1, \ldots, \tilde{S}_k\}$, where $\tilde{S}_i$ is sorted in descending order by $\mathcal{M}(S_i, \hat{S})$ and the position index i indicates the quality of the candidates. The intuitive idea is that a better candidate should be pulled closer to the source code in the representation space, and a worse candidate should be pushed further away. Specifically, in the training time, we use the re-ranking model to encode C, $\hat{S}$, and $\tilde{S}_i$ separately:

$$\begin{aligned} H_C &= Encoder(C) \\ H_{\hat{S}} &= Encoder(\hat{S}) \\ H_{\tilde{S}_i} &= \underset{\tilde{S}_i \in \tilde{\mathbb{S}}}{Encoder}(\tilde{S}_i) \end{aligned} \tag{8}$$

Here, we define a function $\mathcal{F}(\cdot)$ to obtain the similarity between the source code and the summary. We introduce a ranking loss with $\mathcal{F}(\cdot)$ to optimize the re-ranking model:

$$\begin{aligned} \mathcal{L} = &\sum_i max(\mathcal{F}(H_C, H_{\tilde{S}_i}) - \mathcal{F}(H_C, H_{\hat{S}}), 0) \\ &+ \sum_i \sum_{j>i} max(\mathcal{F}(H_C, H_{\tilde{S}_j}) - \mathcal{F}(H_C, H_{\hat{S}}) + \lambda_{ij}, 0) \end{aligned} \tag{9}$$

Where $\lambda_{ij} = \lambda \times s(j-i)$ is a margin hyper-parameter associated with the candidates' rank difference.

Candidate Selection In the inference time, given the source code snippet C, we use the re-ranking model to select the best candidate from the summary candidates pool $\mathbb{S}$ according to the function $\mathcal{F}(\cdot)$:

$$S^* = \underset{S_i \in \mathbb{S}}{argmax}\{\mathcal{F}(H_C, H_{S_1}), \ldots, \mathcal{F}(H_C, H_{S_k})\} \tag{10}$$

In practice, we instantiate $\mathcal{F}(\cdot)$ as the cosine similarity between the first tokens of source code embedding H_C and summary candidate embedding H_{S_i}.

4. Experiment and Analysis

4.1. Datasets

We chose the CodeSearchNet (Husain et al., 2019) benchmark dataset as our training and evaluation dataset. CodeSearchNet is a large-scale dataset mined from popular GitHub projects, which contains code-comment pairwise data from six programming languages, including Ruby, Javascript, Go, Python, Java, and PHP. The detailed statistics of each dataset are listed in Table 1. For each language, the table lists the number of examples in each category. We use the version provided by the CodeXGLUE team (Lu et al., 2021).

4.2. Baselines

We choose a variety of related PLMs with strong performance for code intelligence as baselines. Code-

Language	Training	Dev	Testing
Go	167,288	7,325	8,122
Java	164,923	5,183	10,955
Javascript	58,025	3,885	3,291
PHP	241,241	12,982	14,014
Python	251,820	13,914	14,918
Ruby	24,927	1,400	1,261

Table 1: Data statistics about CodeSearchNet.

BERT (Feng et al., 2020) is a bimodal PLM encoder for programming languages and natural language, which is pre-trained with Mask Language Modeling (MLM) and Replaced Token Detection (RTD). GraphCodeBERT (Guo et al., 2021) is a PLM encoder pre-trained with MLM, data flow edge prediction, and node alignment. PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2023) are sequence-to-sequence PLMs. The former is pre-trained through denoising autoencoding, and the latter is pre-trained through three identifier-aware pre-training tasks. CodeT5+ (Wang et al., 2023) is a large language model (LLM) for code intelligence initialized from the existing LLM and fine-tuned through a mixture of training target fine-tuning and instruction tuning. UniXcoder (Guo et al., 2022) is a multi-modal contrastive pre-training PTM, which is pre-trained with MLM, unidirectional language modeling, denoising autoencoder, and two contrastive learning-related tasks. Specifically, we use UniXcoder to implement our framework.

4.3. Evaluation Metric

We use the Smoothed BLEU-4 (Lin and Och, 2004) as the evaluation metric recommended by the CodeXGLUE team (Lu et al., 2021). BLEU (Bilingual Evaluation Understudy) is a percentage number between 0 and 100, calculated by matching the n-grams between the candidate and the golden summary. BLEU-4 is commonly used to evaluate the quality of the generated text. However, the original BLEU is designed for the corpus level. When any n-gram is zero, the final geometric mean will be zero. Smoothed BLEU indicator usually uses methods such as adding one smoothing or additive smoothing to adjust the count of n-gram matching so that the n-gram denominator is not zero, reducing uncertainty and errors in the evaluation results.

$$SmoothedBLEU = BP \times exp(\sum_{n=1}^{N} W_n log P_n)$$

$$P_n = \frac{Count(S) + 1}{Count(\hat{S}) + 1}$$

Where the N is the maximum base element of n-gram, W_n is the weight of n-gram, BP is the brevity penalty factor, and $Count(\cdot)$ is the minimum n-gram number in candidate or golden summary. In Smoothed BLEU-4, the N is 4, and W_n is $\frac{1}{N}$.

4.4. Experimental Setup

Hyperparameter We use the pre-trained language model UniXcoder to initialize all the encoders/decoders of the two-stage models. Therefore, we follow the same hyperparameter settings of UniXcoder as much as possible to produce the performance of Re^3. For Stage I, we train our model on batch 48 and learning rate 5e^{-5} (same as UniXcoder). For Stage II, we train our model on batch 24 and learning rate 2e^{-4}. Two-stage models are trained in 10 epochs using the AdamW optimizer. The max sequence length of code and summary is 256 and 128, respectively.

Device We conducted experiments on a workstation on Ubuntu 22.04 with four Nvidia RTX3090 GPUs (24GB). The version of CUDA and cuDNN for GPU usage are 11.7 and 8.5, respectively.

4.5. Experiment Results and Analysis

Effectiveness of Re^3 We implement Re^3 based on the baseline model UniXcoder, which further improved its performance, as shown in Table 2.

Table 2 shows the overall performance of the experimental methods. Re^3 raises the performance of UniXcoder with an average improvement of 38% on the CodeSearchNet of six programming language datasets. Notably, the performance improvement of Re^3 in UniXcoder has exceeded that of the current state-of-the-art LLM CodeT5+ (770B). Such a gap is more significant than the progress made by research in the whole field of code summarization over the five years. At the same time, even without the redundancy reduction strategy (-w/o Reduce Redundancy) and just using the second-stage model to re-rank the summary candidates generated from UniXcoder, its performance improvement will also reach 32%. It suggests that the potential of current code summarization models based on pre-trained models and encoder-decoder architectures is completely underutilized due to exposure bias problems, calling for better methods to identify the best summary candidates.

Effectiveness of Redundancy Reduction This section aims to explain the effectiveness of the redundancy reduction strategy. That is to say, why do we need to train the first stage model based on the redundancy reduction strategy before reranking in Stage II.

As shown in Table3, we illustrate this phenomenon with the difference in output on CodeSearchNet with a baseline model UniXcoder and

Model	Ruby	Javascript	Go	Python	Java	PHP	Overall
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	17.83
GraphCodeBERT	12.39	14.81	18.41	18.06	19.00	25.59	18.04
PLBART	14.11	15.56	18.91	19.30	18.45	23.58	18.32
CodeT5-base	15.24	16.16	19.56	20.01	20.31	26.03	19.55
CodeT5+ (220M)	15.51	16.27	19.60	20.16	20.53	26.78	19.81
CodeT5+ (770M)	15.63	17.93	19.64	20.47	20.83	26.39	20.15
UniXcoder	14.87	15.85	19.07	19.13	20.31	26.54	19.30
Re^3+UniXcoder	**21.03**	**21.28**	**26.66**	25.62	**27.96**	**37.36**	**26.65**
-w/o Reduce Redundancy	18.62	19.78	26.32	**26.45**	26.54	35.79	25.58
-w/o Rerank	14.67	15.71	19.14	18.95	20.13	26.50	19.18

Table 2: Code summarization results on CodeSearchNet. The best scores are in bold.

our first stage model, where "Oracle" denotes the maximum Smooth BLEU-4 scores over the pool of summary candidates. Notably, for the result that usually uses beam search for decoding sampling, the redundancy reduction strategy shows an average performance of 0.63% lower than the baseline model. However, for the result calculated in Oracle, the redundancy reduction strategy shows an average performance of 0.96% better than the baseline model. In other words, a redundancy reduction strategy improves the potential performance of the code summarization model. However, due to problems such as exposure bias, summaries that are directly generated based on existing decoding sampling strategies cannot reflect this potential increase in performance. Therefore, training a second-stage model is needed because it can refine this part of the performance improvement by rethinking the relationship between source code, golden summary, and summary candidates.

We hope to further discuss the necessity of redundancy reduction strategy through the results in Table 2 and Table 3. According to Table 2, after adding a re-ranking model to both Re^2+UniXcoder and UniXcoder (i.e., w/o Reduce Redundancy), the average difference of performance on the five datasets (except for the Python dataset) is 1.44. According to Table 3, the average difference of oracle scores on the same five datasets is 0.31, significantly lower than the difference between their performance. We suggest that the redundancy reduction module has two advantages: (1) The redundancy reduction module could help reduce redundancy at the representation level to obtain better code representation, reflected by better potential summary output quality (i.e., oracle score). (2) The redundancy reduction strategy increases the diversity of code representation feature vectors, allowing the re-ranking model to learn representation space more effectively.

To verify the second proposed advantage, we designed preliminary experiments for verification on the Ruby and Javascript datasets. We use UniXcoder as the encoder and calculate the average Euclidean distance between the summary candidates and the golden summary generated by Re^3+UniXcoder and UniXcoder respectively. As Table 4 shows, the average Euclidean distance between golden summary and summary candidates generated by our method has a higher mean and lower variance than UniXcoder. Evidence shows that reducing redundancy could increase the diversity of summary candidates and improve their distribution in the feature space, finally helping the rethinking module through metric learning.

Effectiveness of Different Pre-trained Models This section aims to show the effectiveness of transfer setting in Re^3 and explore whether it could be applied to different code intelligence PLMs.

Table 5 shows the outcomes of an experiment that evaluated the impact of three distinct PLMs on six programming language datasets of CodeSearchNet. Experiments are conducted on three PLMs: CodeBERT, GraphCodeBERT, and UniXcoder. When training the first-stage models, we follow the same hyperparameter settings of these models, which are used to fine-tune the code summarization downstream task. As expected, we found that Re^3 is a PTM-independent framework, which achieves an average performance of 35%, 31%, and 38% better with CodeBERT, GraphCodeBERT, and UniXcoder, respectively. The experimental results provide compelling evidence of the transfer ability of Re^3.

4.6. Ablation Study

Table 2 shows the performance difference of UniXcoder when using the redundancy reduction strategy (w/o Rethink) or the rethink model (w/o De-Redundancy). As we analyzed above, the performance of the first stage model trained with a redundancy reduction strategy is not ideal. It may be because the redundancy reduction strategy further increases the diversity of the code representation feature vectors encoded by the encoder, and current decoding sampling strategies are not good at

Model	Ruby	Javascript	Go	Python	Java	PHP	Overall
UniXcoder	14.87	15.85	19.07	19.13	20.31	26.54	19.30
UniXcoder (Oracle)	21.08	22.81	26.80	26.60	27.98	37.76	27.17
Re^2+UniXcoder	14.67	15.71	19.14	18.95	20.13	26.50	19.18
Re^2+UniXcoder (Oracle)	22.04	23.03	26.78	26.59	28.23	37.92	27.43
Re^3+UniXcoder	21.03	21.28	26.66	25.62	27.96	37.36	26.65

Table 3: Comparison of UniXcoder, Re^2+UniXcoder (the first stage model), and Re^3+UniXcoder (complete two-stage models). **Oracle** means the maximum score over all generated candidates.

Model	Ruby		Javascript	
	Mean	Std	Mean	Std
All candidates				
UniXcoder	23.28	7.51	22.28	7.55
Re^3+UniXcoder	23.57	7.30	23.82	6.91
Candidates except for top-1				
UniXcoder	23.53	7.37	22.61	7.37
Re^3+UniXcoder	23.79	7.18	24.02	6.81

Table 4: Euclidean-distance calculation results. **All candidates** means calculating the average Euclidean distance between the golden summary and all candidates. **Candidates except for top-1** means calculating the average Euclidean distance between the golden summary and filtered candidates, which removes the first candidate with the highest score after sorting.

searching within a more extensive solution space due to the exposure bias problem. The performance of only using the rethink model decreases compared to the Re^3, consistent with the changing trend of the Oracle score reported in Table 3.

In addition, we also conducted ablation experiments on the GPO strategy used in the de-redundancy method, as shown in Table 6. To explore the impact of GPO, we adopted two other different pooling strategies: maximum pooling (i.e., using the representation of $[CLS]$ token as the representation of the source code) and mean pooling. Experimental results on Ruby and Javascript datasets show that the performance of the Re^3+UniXcoder decreased when using other pooling strategies, proving GPO's effectiveness.

4.7. Qualitative Evaluation

Case Study We present the result of our case studies in Figure 4. Specifically, we demonstrate the significantly better quality of generated summaries of Re^3 over the baseline model UniXcoder. For the first case on the Ruby dataset, it may be that the first word of the golden summary, "Decrease" does not appear in the code. Both Re^3 and UniXcoder occur errors in the first step. Due to the exposure bias problem and the weakness of beam search, the mistakes made in the previous steps will accumulate. UniXcoder has no way of knowing that it should adjust the decoding and finally end by generating a short summary with three words. In contrast, Re^3 performs better while beginning decoding with a wrong step and ultimately generates a more extended summary, which captures the key sentence "the priority of one or more torrents." For the second case on the Javascript dataset, UniXcoder may be confused by "delete" and "remove" appearing in the code snippet, and an error occurred in the first step. Re^3 can correctly catch the critical information in the code through the redundancy reduction strategy.

<table>
<tr><th>Ruby</th><th>Javascript</th></tr>
<tr><td>def minimize_priority torrent_hashes
  torrent_hashes=Array(torrent_hashes)
  torrent_hashes=torrent_hashes.join('|')
  options={body:"hashes=#{torrent_hashes}"}
  self.class.post('/command/bottomPrio',options)
end</td><td>function remove(repoState,driver,branch){
  return driver.deleteBranch(
    branch)..then(()=>{
      return repoState.updateBranch(
        branch, null);
      });
    }</td></tr>
<tr><td>Golden:
Decrease the priority of one or more torrents
to the minimum value.</td><td>Golden:
Remove the given branch
from the repository.</td></tr>
<tr><td>UniXcoder Generated:
Minimizes torrent priorit.</td><td>UniXcoder Generated:
Delete a branch.</td></tr>
<tr><td>Re^2+UniXcoder Generated:
Minimizes the priority of one or more torrents.
Minimizes the priority of one or more torrent hashes.
Minimizes the priority of the torrents.
Minimizes the priority of a set of torrent hashes.
Minimizes the priority of a list of torrent hashes.</td><td>Re^2+UniXcoder Generated:
Remove a branch from the repository.
Removes a branch from a repository.
Remove a branch from a repo.
Remove a branch.
Removes a branch.</td></tr>
<tr><td>Re^3+UniXcoder Selected:
Minimizes the priority of one or more torrents.</td><td>Re^3+UniXcoder Selected:
Remove a branch from the repository.</td></tr>
</table>

Figure 4: Two examples for case study on Ruby and Javascript datasets, respectively. We compare the golden summary and summaries generated/selected by UniXcoder, Re^2+UniXcoder, and Re^3+UniXcoder.

Human Evaluation We also conduct a human evaluation. The metric reported here is the number of human evaluators' preference for summaries. We asked four human evaluators to evaluate 50 summaries randomly sampled from the test dataset for the Ruby and Javascript datasets of CodeSearchNet. Human evaluators are shown the source code, the top beam search summary from UniXcoder, and the corresponding summary candidate selected by Re^3. We filter samples where UniXcoder and Re^3 obtained Smoothed BLEU-4 scores above 80. Human evaluators are asked to choose which one they believe is more faithful and

Model	Ruby	Javascript	Go	Python	Java	PHP	Overall
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	17.83
Re^3+CodeBERT	17.13	19.93	22.94	24.74	25.55	34.04	24.06
GraphCodeBERT	12.39	14.81	18.41	18.06	19.00	25.59	18.04
Re^3+GraphCodeBERT	15.29	18.61	24.54	23.37	26.23	34.16	23.70
UniXcoder	14.87	15.85	19.07	19.13	20.31	26.54	19.30
Re^3+UniXcoder	21.03	21.28	26.66	25.62	27.96	37.36	26.65

Table 5: Code summarization results of different pre-trained models.

Model	Ruby	Javascript
Re^3+UniXcoder (GPO)	21.03	21.28
Re^3+UniXcoder (Max)	19.67	20.47
Re^3+UniXcoder (Mean)	19.84	20.88

Table 6: Ablation study results for GPO. **Max** represents the maximum pooling strategy and **Mean** represents the average pooling strategy.

Model	Ruby		Javascript	
	Mean	Std	Mean	Std
UniXcoder	8.75	0.83	10.25	2.49
Re^3+UniXcoder	27.25	3.34	23.75	2.77

Table 7: Human evaluation results.

can choose a tie cause we found that some summaries have lower Smoothed BLEU-4 scores but meet the requirements from the semantic perspective. As shown in Table 7, we see that, on average, human evaluators are likelier to pick the summary that Re^3 selects.

5. Conclusion

In this work, we introduce Re^3, a novel pipeline framework that aims to enhance the performance of the code summarization model. To tackle two main challenges in current code summarization, our framework involves two steps: The redundancy reduction strategy constrains the code representation, which is used to generate better summary candidates. A re-ranking model is incorporated to choose a better candidate, thus mitigating the exposure bias problem. The experimental results show the effectiveness of Re^3 over some state-of-the-art approaches across six distinct test datasets from the CodeSearchNet benchmark.

6. Acknowledgements

We thank anonymous reviewers for their valuable comments and helpful suggestions. The authors acknowledge financial support from the National Natural Science Foundation of China (62176053). This research work is also supported by the Big Data Computing Center of Southeast University.

7. Bibliographical References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

Chenxin An, Ming Zhong, Zhiyong Wu, Qin Zhu, Xuanjing Huang, and Xipeng Qiu. 2022. Colo: A contrastive learning based re-ranking framework for one-stage summarization. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, pages 5783–5793. International Committee on Computational Linguistics.

Adrien Bardes, Jean Ponce, and Yann LeCun. 2022. Vicreg: Variance-invariance-covariance regularization for self-supervised learning. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Sumanta Bhattacharyya, Amirmohammad Rooshenas, Subhajit Naskar, Simeng Sun, Mohit Iyyer, and Andrew McCallum. 2021. Energy-based reranking: Improving neural machine translation using energy-based models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4528–4537. Association for Computational Linguistics.

Jiacheng Chen, Hexiang Hu, Hao Wu, Yuning Jiang, and Changhu Wang. 2021. Learning the best pooling strategy for visual semantic embedding. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual,* June 19-25, 2021, pages 15789–15798. Computer Vision Foundation / IEEE.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Aleksandr Ermolov, Aliaksandr Siarohin, Enver Sangineto, and Nicu Sebe. 2021. Whitening for self-supervised representation learning. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 3015–3024. PMLR.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Xuejian Gao, Xue Jiang, Qiong Wu, Xiao Wang, Chen Lyu, and Lei Lyu. 2021. Multi-modal code summarization fusing local API dependency graph and AST. In *Neural Information Processing - 28th International Conference, ICONIP 2021, Sanur, Bali, Indonesia, December 8-12, 2021, Proceedings, Part V*, volume 1516 of *Communications in Computer and Information Science*, pages 290–297. Springer.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel R. Bowman, and Noah A. Smith. 2018. Annotation artifacts in natural language inference data. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 107–112. Association for Computational Linguistics.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010a. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 223–226. ACM.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010b. On the use of automated text summarization techniques for summarizing source code. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 35–44. IEEE Computer Society.

Sakib Haque, Aakash Bansal, and Collin McMillan. 2023. Label smoothing improves neural source code summarization. In *31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023*, pages 101–112. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210. ACM.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2269–2275. ijcai.org.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 184–195. ACM.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004, 20th International Conference on Computational Linguistics, Proceedings of the Conference, 23-27 August 2004, Geneva, Switzerland*.

Yixin Liu, Zi-Yi Dou, and Pengfei Liu. 2021. Refsum: Refactoring neural summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 1437–1448. Association for Computational Linguistics.

Yixin Liu and Pengfei Liu. 2021. Simcls: A simple framework for contrastive learning of abstractive summarization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021*, pages 1065–1072. Association for Computational Linguistics.

Yixin Liu, Pengfei Liu, Dragomir R. Radev, and Graham Neubig. 2022. BRIO: bringing order to abstractive summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 2890–2903. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Tom McCoy, Ellie Pavlick, and Tal Linzen. 2019. Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3428–3448. Association for Computational Linguistics.

Mathieu Ravaut, Shafiq R. Joty, and Nancy F. Chen. 2022. Summareranker: A multi-task mixture-of-experts re-ranking framework for abstractive summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 4504–4524. Association for Computational Linguistics.

Jikyoeng Son, Joonghyuk Hahn, HyeonTae Seo, and Yo-Sub Han. 2022. Boosting code summarization by embedding code structures. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, pages 5966–5977. International Committee on Computational Linguistics.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 397–407. ACM.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Ronald J. Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 1078–1090. Association for Computational Linguistics.

Yanzheng Xiang, Qian-Wen Zhang, Xu Zhang, Zejie Liu, Yunbo Cao, and Deyu Zhou. 2023. G^{3}r: A graph-guided generate-and-rerank framework for complex and cross-domain text-to-sql generation. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 338–352. Association for Computational Linguistics.

Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny. 2021. Barlow twins: Self-supervised learning via redundancy reduction. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 12310–12320. PMLR.

Xingxuan Zhang, Peng Cui, Renzhe Xu, Linjun Zhou, Yue He, and Zheyan Shen. 2021. Deep stable learning for out-of-distribution generalization. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 5372–5382. Computer Vision Foundation / IEEE.

Xu Zhang, Xiaoyu Hu, Zejie Liu, Yanzheng Xiang, and Deyu Zhou. 2024. Hfd: Hierarchical feature decoupling for sql generation from text. *Intelligent Data Analysis*, pages 1–15.