PinNet: Pinpoint Instructive Information for Retrieval Augmented Code-to-Text Generation

Han Fu [1] **Jian Tan** [1] **Pinhan Zhang** [2] **Feifei Li** [1] **Jianling Sun** [2]

Abstract

Automatically generating high quality code descriptions greatly improves the readability and maintainability of the codebase. Recently, retrieval augmented code-to-text approaches have proven to be an effective solution, and have achieved the state-of-the-art results on various benchmarks. It brings out the potential to leverage large unlabeled code descriptions to further improve the generation quality. In spite of the promising performance, retrieval-augmented models however suffer from being deluded by inconducive retrieved references, due to irrelevant or even misleading information contained therein. To this end, we design *PinNet*, a new framework for code-to-text generation. *PinNet* relies on a discriminator to measure how well the retrievals match the semantics of the input code. Remarkably, the hidden representation of the reference from the last layer of the discriminator can be leveraged to significantly improve the code-to-text generation through modifying the attention weights. It essentially pays high attention to valuable information and eliminates misleading part. To effectively execute this idea, we also propose a novel contrastive learning method to quantify the semantical similarities between unlabeled references. Using extensive experiments on code summarization and SQL-to-text generation, we demonstrate that the proposed method can significantly outperform all of the baselines.

1. Introduction

Generating accurate descriptions automatically for given code snippets (code-to-text) can bring tremendous value for readability and maintainability of the codebase, and has become an emerging capability of AI coding assistant tools.

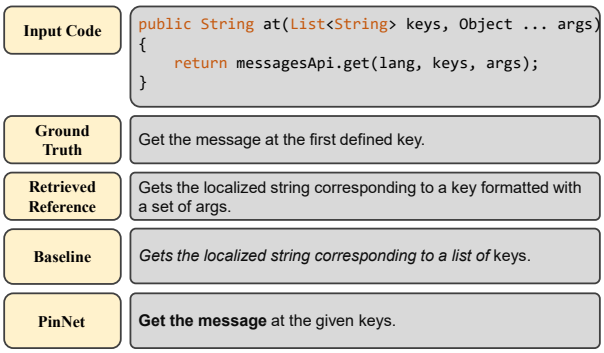

Figure 1. The misleading information in the retrieved reference deludes the baseline model in generating a wrong description in *italics*. The accurate description by PinNet is shown in **bold**.

Recently, various methods based on natural language processing have been developed for code-to-text generation. One prominent approach based on deep-learning (Iyer et al., 2016; Ahmad et al., 2020) formulates it as a translation task. However, applying vanilla translation models, *e.g.*, Seq2Seq (Sutskever et al., 2014; Bahdanau et al., 2014) and Transformer (Vaswani et al., 2017), to code-to-text is challenging due to the syntactic and semantic complexity of programming languages. Observing that many similar code snippets are frequently reused in various scenarios, some works (Zhang et al., 2020; Wei et al., 2020; Parvez et al., 2021) propose to use reference samples collected from annotated codebases that may contain relevant descriptions to further improve the generation quality.

Though effective, existing retrieval augmented code-to-text methods suffer from the fact that the model could be seriously deluded if the retrieved description is irrelevant or even misleading. The reason is that all existing models do not consider how well the retrieved descriptions match with the semantics of the input code. For example in Figure 1, the retrieved reference is semantically different from the ground truth, and the baseline model is negatively influenced to generate an inaccurate description.

Moreover, the accuracy of the retriever also plays a critical

[1]Alibaba Group, Hangzhou, China [2]College of Computer Science and Technology, Zhejiang University, Hangzhou, China. Correspondence to: Han Fu <fuhan.fh@alibaba-inc.com>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

role to the overall performance. Most existing retrievers (Parvez et al., 2021; Zhang et al., 2020) are typically based on contrastive learning. During training, these methods use the ground truth texts as the positive instances while regard all others, even when they contain relevant information, as negatives. In addition, the truly relevant references may not be selected by the retriever during inference. This gap between training and inference phases significantly limits the performance of retrieval models.

To mitigate the limitations of existing methods, we propose *PinNet*. It is designed to pinpoint the most relevant information from the retrieved references, by focusing on the valuable part and eliminating the misleading semantics. Specifically, we introduce a discriminator to predict the correlation between the retrieved description and the ground truth. The hidden representation of the reference from the last layer of the discriminator, which captures the critical semantics of the retrieved description, is directly leveraged to improve the accuracy of code-to-text generation. To sufficiently utilize the discriminative representations, we propose a new attention mechanism, namely *PinAttention*, which boosts the Multi-Head Attention (Vaswani et al., 2017) to use extra knowledge. Moreover, we introduce a new contrastive learning model, called *PinNet-Ret*, for code-text retrieval. The overall architecture of the proposed framework is shown in Figure 2.

To verify the effectiveness of the proposed framework, we conduct extensive experiments on code summarization and SQL-to-text generation tasks using four datasets. The experiments demonstrate that *PinNet* achieves the state-of-the-art results on all test sets and significantly outperforms all of the existing models. Ablation studies show that *PinNet* can adequately leverage the retrieved information. We also compare with modern large language models (*e.g.*, ChatGPT). The results further confirm the advantages of *PinNet*, which has great potential to enhance the performance of LLMs for code-to-text tasks.

The major contributions of this work are summarized as follows:

- A new framework *PinNet* for code-to-text generation is designed, which can better exploit the valuable information from the retrieved references and eliminate the misleading information.
- A new contrastive learning objective function for code-to-text retrieval is formulated. It quantifies the semantical similarities between the retrieved sample and the input code, and bridges the gap between the training and inference when only negative samples are present for inference.
- Experimental results show that the proposed method significantly outperforms compared approaches on code summarization and SQL-to-text tasks.
- Even when compared with the large language models, *PinNet* still achieves superior performance on code summarization tasks. The proposed method raises potential to obtain further improvement over the large language models.

Table 1. BLEU scores using different cutoff thresholds to remove outliers. The performance of REDCODER is evaluated on CodeXGLEU-Python. The similarity score of REDCODER ranges from 60 to 130.

Threshold	0	80	90	100	110	120
BLEU	19.64	19.61	19.47	18.67	18.46	18.40

2. The Design of PinNet

2.1. Overview

Task. The goal is to predict a natural language description for a given code snippet, *e.g.*, a Python function or a SQL query. Formally, for a code snippet of n tokens $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, a code-to-text generation model produces a natural language description, *e.g.*, a docstring, a code comment, or a natural language query, denoted by $\mathbf{y} = (y_1, y_2, \cdots, y_m)$ with m being the sequence length.

Motivation. Existing retrieval augmented code-to-text methods have two major limitations: 1) the generation could be misguided due to the semantic difference between the retrieved description and the ground truth description; 2) there exists a discrepancy between the training and inference phases for the dense retriever. During training, the embedding of ground truth description is made close to the input code, but all other descriptions, even relevant, are unanimously used as negatives. However, during inference, a most matched description may not be selected by the retriever, and even worse, may not exist in the retrieval database. This incapability of judging the relevance and worth of the reference samples could cause a serious performance degradation.

A naive approach to eliminate misleading information is to simply remove the outliers from the retrieved samples. Table 1 shows the performance of REDCODER (Parvez et al., 2021) with different cutoff thresholds for identifying outliers. Specifically, if a retrieved sample has a similarity score below a threshold, we remove the description from the prompt. From the results, we observe that the performance of REDCODER keeps decreasing as the threshold increases. This is because the similarity score does not fully reflect whether a retrieved sample is misleading or

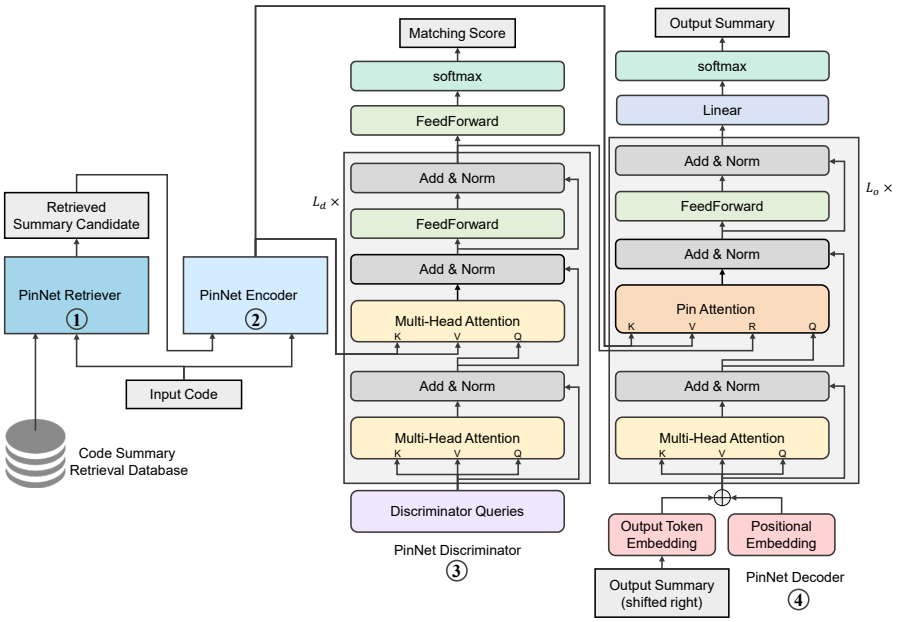

Figure 2. PinNet consists of 4 components: 1) given the input code snippet, the PinNet Retriever ① searches for the relevant description samples (without the corresponding code) from a code summary database; 2) the PinNet Encoder ② takes both the code and a selected reference description, which may not match with the code, as input and extracts the semantic features; 3) the PinNet Discriminator ③ estimates the similarity between the description and the input code, and provides hidden representations that contain the discriminative information; 4) using this information, the PinNet Decoder ④ generates the final description through output tokens sequentially.

not. Removing outliers thus completely discard the potentially valuable information provided in the sample. To this end, we propose *PinNet* to quantitively *pinpoint* the relevant semantic information from the retrieved reference.

Method. As illustrated in Figure 2, the *PinNet* architecture consists of four major components: Retriever, Encoder, Decoder, and Discriminator. The Retriever (*PinNet-Ret*) computes the similarity scores between the input code snippet and all code descriptions in the retrieval database. For example, code summarization tasks use the summary as the description. The sample with the highest score is selected as the reference. The retrieved reference and the code are then concatenated and fed to the Encoder (*PinNet-Enc*), which computes hidden states as the encoded representations. The Discriminator (*PinNet-Dec*) takes the encoder hidden states as the inputs and generates discriminative representations, which capture the characteristic semantic features between the retrieved reference and the code. Based on all the above information, the Decoder (*PinNet-Dec*) generates the target natural language tokens sequentially using three inputs: a code snippet, a retrieved reference and the discriminative representation.

Compared with typical retrieval augmented code-to-text methods (Zhang et al., 2020; Wei et al., 2020; Parvez et al., 2021; Zhu et al., 2022), *PinNet* exhibits two advantages. First, *PinNet* introduces a discriminator, which captures the semantic relevance of different parts of the reference. This facilitates the model to focus on the important semantics and ignore the misleading information. Second, during retrieval training, the relevant descriptions are explicitly leveraged through a new retrieval objective to bridge the gap between the training and inference. Notably, some recent works (Parvez et al., 2021; Zhang et al., 2020) use multiple description references, which introduce considerable computational overhead. In this work, we only consider the top one retrieved reference at runtime, which already achieves significant performance improvement.

2.2. PinNet Retriever

Applying code-code retrieval to code-to-text generation is challenging due to the difficulty of preparing a large set of high-quality code-text pairs. A natural solution is to build a collection of descriptions, and use language models through contrastive learning to retrieve the relevant descriptions for the input code. However, due to the above-mentioned discrepancy between the training and inference phases, the retriever may miss its goal. To this end, we propose *PinNet-Ret* to fully exploit the relevant semantics from the reference description.

Specifically, we use two independent pre-trained language models (*i.e.*, RoBERTa (Liu et al., 2019) and GraphCode-

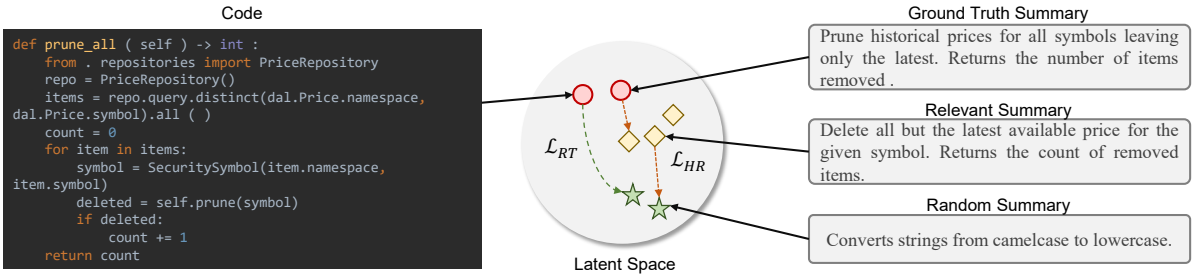

Figure 3. Illustration of *PinNet-Ret* Objective. In the semantics space, the embedding of a relevant description (yellow diamond) is pulled closer to the input code than a random summary (green star), but is pushed farther than the ground truth (red circle).

BERT (Guo et al., 2020)) to encode the natural language description and the code, respectively. Then, by sharing a last layer and using a novel contrastive learning loss function, we align these two embedding spaces so that their similarities can be directly computed.

To obtain the final representations for both the description and the code snippet, their encoder hidden states are fed to two seperate *Attention Pooling* layers, respectively. Let us use the code snippet as an example to illustrate the process, which also applies to the description input. The encoder hidden states are denoted by $\mathbf{H}(\mathbf{x}) = (\mathbf{h}_{x_1}, \mathbf{h}_{x_2}, \cdots, \mathbf{h}_{x_n})$ (special tokens are omitted). The attention pooling layer calculates the weighted summation of $\mathbf{H}(\mathbf{x})$ as:

$$\mathbf{E}_r(\mathbf{x}) = \mathrm{FFN}(\sum_{i=1}^{n} \alpha_i \mathbf{h}_{x_i}), \tag{1}$$

where FFN denotes a feed-forward network layer, and α_i is the weight score of x_i calculated by:

$$\alpha_i = \mathrm{softmax}(\mathbf{v}_\alpha^\top \tanh(\mathbf{W}_\alpha \mathbf{h}_q(\mathbf{x}) + \mathbf{U}_\alpha \mathbf{h}_{x_i})),$$

where $\mathbf{W}_\alpha, \mathbf{U}_\alpha, \mathbf{v}_\alpha$ are trainable matrices and vectors. The query vector $\mathbf{h}_q(\mathbf{x})$ captures the global information of the input. In this work, we use the hidden state embedding of `[CLS]` as the query vector. `[CLS]` is a special token of BERT (Devlin et al., 2019) and is typicall used (Liu et al., 2019; Feng et al., 2020; Guo et al., 2020) as a sentence representation for text classification tasks.

To bridge the gap between training and inference of retrieval, we propose a hierarchical triplet objective function. As shown in Figure 3, the intuition is to maintain a distance ranking order among the target description, a relevant description, and the input code. Specifically, we want to make the embedding of a relevant description closer to the input code than a random text in the latent space, but not closer than the target description. Specifically, given the input code $\mathbf{x}$ and target description $\mathbf{y}$, we randomly sample a batch of descriptions denoted by $\mathcal{B}$ and a set of relevant code summaries $\mathcal{Y}^+$ using BM25 search (Robertson et al., 2009) during training. The hierarchical retrieval loss $\mathcal{L}_{HR}$ is computed as:

$$\begin{aligned}\mathcal{L}_{HR} = \sum_{\mathbf{y}^+ \in \mathcal{Y}^+} & \mathrm{ReLU}(\mathrm{s}(\mathbf{E}_r(\mathbf{x}), \mathbf{E}_r(\mathbf{y}^+)) \\ & - \mathrm{s}(\mathbf{E}_r(\mathbf{x}), \mathbf{E}_r(\mathbf{y})) + m_1) \\ + \sum_{\mathbf{y}^- \in \mathcal{B}} \sum_{\mathbf{y}^+ \in \mathcal{Y}^+} & \mathrm{ReLU}(\mathrm{s}(\mathbf{E}_r(\mathbf{x}), \mathbf{E}_r(\mathbf{y}^-)) \\ & - \mathrm{s}(\mathbf{E}_r(\mathbf{x}), \mathbf{E}_r(\mathbf{y}^+)) + m_1),\end{aligned}$$

where m_1 is a manually defined marginal error and ReLU is the linear rectification function (Nair & Hinton, 2010) which always takes the positive part of a value. $\mathrm{s}(\cdot)$ is the similarity score function and we use cosine in this work. The conventional retrieval objective is also considered:

$$\begin{aligned}\mathcal{L}_{RT} = \sum_{\mathbf{y}^- \in \mathcal{B}} & \mathrm{ReLU}(\mathrm{s}(\mathbf{E}_r(\mathbf{x}), \mathbf{E}_r(\mathbf{y}^-)) - \mathrm{s}(\mathbf{E}_r(\mathbf{x}), \mathbf{E}_r(\mathbf{y})) \\ & + m_2),\end{aligned}$$

Intuitively, to maintain the consistency of the two objectives, the hyperparameter setting should satisfy $m_1 < m_2$. The final training loss for the retrieval model is:

$$\mathcal{L}_r = \mathcal{L}_{RT} + \gamma \mathcal{L}_{HR},$$

where γ is a hyperparameter to balance the losses. We observe that the performance is not sensitive to γ and thus simply set it to 1. During inference, *PinNet-Ret* finds the top-1 description reference $\mathbf{y}^*$ using cosine similarity.

2.3. PinNet Encoder

The encoder module is built upon a bi-directional Transformer encoder (Vaswani et al., 2017) pre-trained on a corpus of source code and natural language descriptions, *i.e.*, CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020). Given the input code $\mathbf{x}$ and a retrieved description reference $\mathbf{y}^*$, the input of *PinNet-Enc* follows the

RoBERTa cross-encoding format (Liu et al., 2019):

$$[\texttt{CLS}], x_1, x_2, \cdots, x_n, [\texttt{SEP}], [\texttt{SEP}], y_1^*, y_2^*, \cdots, y_t^*, [\texttt{SEP}],$$

where `[SEP]` is the separator token. The encoder generates the hidden state of each token, denoted by $\mathbf{E}(\mathbf{x}, \mathbf{y}^*)$, which will be used by the decoder and discriminator.

2.4. PinNet Discriminator

The major novelty of PinNet is the design of *PinNet-Dis*, which captures the discriminative information of the retrieved reference. It consists of multiple Transformer decoder layers. The output hidden states of the l-th layer are:

$$\begin{aligned}
\mathbf{O}_1^l &= \text{LayerNorm}\left(\text{MultiHead}(\mathbf{O}^{l-1}, \mathbf{O}^{l-1}, \mathbf{O}^{l-1}) \right. \\
&\qquad \left. + \mathbf{O}^{l-1}\right), \\
\mathbf{O}_2^l &= \text{LayerNorm}(\text{MultiHead}(\mathbf{O}_1^l, \mathbf{E}(\mathbf{x}, \mathbf{y}^*), \mathbf{E}(\mathbf{x}, \mathbf{y}^*)) \\
&\qquad + \mathbf{O}_1^l), \\
\mathbf{O}^l &= \text{LayerNorm}(\mathbf{O}_2^l + \text{FFN}(\text{ReLU}(\text{FFN}(\mathbf{O}_2^l))))
\end{aligned}$$

where $\mathbf{O}^{l-1}$ is the output of the $(l-1)$-th layer. LayerNorm represents Layer Normalization (Ba et al., 2016). $\text{MultiHead}(Q, K, V)$ is the Multi-Head Attention Layer (Vaswani et al., 2017), with Q, K, and V being the hidden states for *Query*, *Key* and *Value*, respectively. The fixed input of the discriminator network $\mathbf{O}^0$ is a sequence of n_d trainable vectors. The final representation is:

$$\mathbf{D}(\mathbf{x}, \mathbf{y}^*) = \text{FFN}(\mathbf{O}^{L_d}) \in \mathbb{R}^{n_d \times d_d},$$

where L_d denotes the last transformer layer. n_d and d_d are the length and the dimension of the discriminative vectors, respectively. Essentially, $\mathbf{D}(\mathbf{x}, \mathbf{y}^*)$ captures the relevant information of $\mathbf{y}^*$ and mitigate the misleading part.

To this end, we formulate the training objective of the discriminator as predicting the matching score between the retrieved description reference and the input code. Specifically, we categorize the set of descriptions $\mathcal{Y}^*$ into g groups according to their BLEU scores (Papineni et al., 2002) with respect to the ground truth description. Specifically, we divide the BLEU score range into g disjoint intervals, and each interval represents one class. Thus, each BLEU score, which is a float number, will be assigned with one label from g classes. The discriminator is trained to predict the class label of the BLEU score for each retrieved reference. With this novel method, *PinNet-Dis* is facilitated to capture the contrastive semantics between the relevant and irrelevant parts of $\mathbf{y}^*$. In addition, we use another attention pooling layer (Equation 1) to transform $\mathbf{D}(\mathbf{x}, \mathbf{y}^*)$ to a single vector $\mathbf{d}$. The query of the attention pooling layer is the average of all n_d discriminative vectors. The classification objective is estimated with a cross-entropy loss:

$$\begin{aligned}
\mathbf{P}_d &= \text{softmax}(\mathbf{d}) = \{P_1^d, \cdots, P_g^d\} \\
\mathcal{L}_d &= -\sum_j^g y_j^d \log(P_j^d)
\end{aligned}$$

where $\mathbf{p}_j^c$ is prediction probability of the j-th class and y_i^c is the corresponding label.

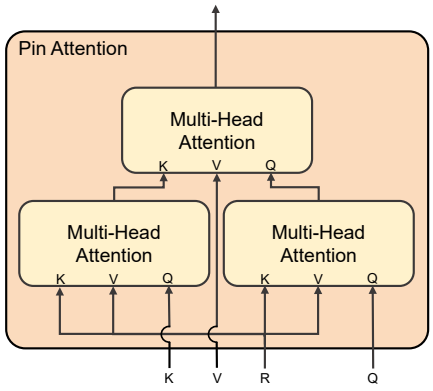

Figure 4. Illustration of *PinAttention*

2.5. PinNet Decoder

To integrate the discriminative representation $\mathbf{D}(\mathbf{x}, \mathbf{y}^*)$ into the decoder, we design *PinAttention*, which is an extension of the Multi-Head Cross-Attention mechanism (Vaswani et al., 2017). It can effectively leverage extra knowledge from the reference.

As illustrated in Figure 4, the input $\text{PinAttn}(Q, K, V, R)$ consists of four elements, including queries (Q), keys (K), values (V) and representations of the reference (R). We apply two independent Multi-Head Attention layers to integrate the reference knowledge into queries and keys, formulated as follows:

$$\begin{aligned}
\mathbf{Q}^r &= \text{LayerNorm}\left(\text{MultiHead}(\mathbf{Q}, \mathbf{R}, \mathbf{R}) + \mathbf{Q}\right) \\
\mathbf{K}^r &= \text{LayerNorm}\left(\text{MultiHead}(\mathbf{K}, \mathbf{R}, \mathbf{R}) + \mathbf{K}\right).
\end{aligned}$$

The output of *PinAttention* is calculated with another Multi-Head Attention:

$$\mathbf{O}^r = \text{MultiHead}(\mathbf{Q}^r, \mathbf{K}^r, \mathbf{V}).$$

With *PinAttention*, the discriminative information helps the model to dynamically control the attention to different parts of the encoder outputs. Similar to a transformer layer, the hidden states $\mathbf{D}^l$ of the l-th *PinNet-Dec* layer is calculated

as follows:

$$\mathbf{D}_1^l = \text{LayerNorm}(\text{MultiHead}(\mathbf{D}^{l-1}, \mathbf{D}^{l-1}, \mathbf{D}^{l-1}) + \mathbf{D}^{l-1}),$$
$$\mathbf{D}_2^l = \text{LayerNorm}(\text{PinAttn}(\mathbf{D}_1^l, \mathbf{E}(\mathbf{x}, \mathbf{y}^*), \mathbf{E}(\mathbf{x}, \mathbf{y}^*), \mathbf{D}(\mathbf{x}, \mathbf{y}^*)) + \mathbf{O}_1^l),$$
$$\mathbf{D}^l = \text{LayerNorm}(\mathbf{D}_2^l + \text{FF}(\text{ReLU}(\text{FF}(\mathbf{D}_2^l)))).$$

The training objective function for generation is formulated as an auto-regressive conditional factorization $\mathcal{L}_g = \sum_i^t \log P(y_i|\mathbf{y}_{<i}, \mathbf{x}, \mathbf{y}^*)$, where $P(y_i = t|\mathbf{y}_{<i}, \mathbf{x}, \mathbf{y}^*) = \text{softmax}(\mathbf{W}_o\mathbf{D}^{L_o})_t$ is the prediction probability of the i-th token in the output sequence where $\mathbf{W}_o$ is the trainable matrix of the output layer. Its dimension size changes from that of the final hidden state to the vocabulary size. L_o is the number of transformer layers in the decoder network.

The final loss of *PinNet* is $\mathcal{L} = \alpha\mathcal{L}_d + \beta\mathcal{L}_g$ where α and β are hyper-parameters to balance the two objectives.

3. Experiment Setup

3.1. Datasets

For code summarization task, we use a widely adopted benchmark, CodeXGLUE (Lu et al., 2021), for evaluation, and we follow (Parvez et al., 2021) to construct the retrieval database. For SQL-to-text generation task, we conduct experiments on two datasets, WikiSQL (Zhong et al., 2017) and StackOverflow (Iyer et al., 2016). The details of preprocessing and data statistics are provided in Appendix A.1.

3.2. Comparing baselines

For code summarization, we compare *PinNet* with the state-of-the-art approaches from the following four categories:

Generative Models: LSTM-based encoder-decoder model (Seq2Seq) (Luong et al., 2015) and Transformer (Vaswani et al., 2017).

Pre-trained Language Models: RoBERTa (code) (Husain et al., 2019), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021), UniXcoder (Guo et al., 2022), and CodeT5+ (Wang et al., 2023).

Retrieval Augmented Methods: *k*NN-Transformer (Zhu et al., 2022), BM25 + PLBART (Ahmad et al., 2021; Robertson et al., 2009), REDCODER (Parvez et al., 2021), and REDCODER-ext (Parvez et al., 2021).

Large Language Models: *PinNet* is also compared with large language models (LLM) including PolyCoder (Xu et al., 2022), CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023) and ChatGPT (OpenAI, 2023). We follow (Sun et al., 2023) to apply few-shot prompt and fine-tuning to the open-sourced LLMs. For ChatGPT, we augment the prompt with reference samples retrieved by *PinNet-Ret* to enable in-context learning. The implementation details are listed in Appendix A.3.

For SQL-to-Text, we categorize the baseline models into three groups.

Sequence-to-Sequence Models: RNN-based sequence-to-sequence model (seq2seq) (Bahdanau et al., 2014) with copy mechanism (cp) (Gu et al., 2016) or latent variable (lv) (Guo et al., 2018).

Pre-trained Models: RoBERTa (Liu et al., 2019) and PreQR2Seq (Tang et al., 2022).

Structural Information Augmented Models: Tree-to-sequence model (Tree2Seq) (Eriguchi et al., 2016) and Graph-to-sequence model Graph2Seq (Xu et al., 2018).

3.3. Implementations

Both *PinNet-Enc* and *PinNet-Ret* models are initialized with GraphCodeBERT (Guo et al., 2020). the *PinNet-Dec* network consists of 6 stacked *PinNet-Dec* layers which shares the same embedding shape with the encoder. To train the models sufficiently, we set the learning rates of *PinNet-Enc* and *PinNet-Dec* as 1×10^{-5} and 1×10^{-4} respectively. For the retrieval task, we set m_1 and m_2 to 0.2 and 0.4, respectively. The hyper-parameters of the generation loss, α and β, are both set to 1.0. More implementation details are provided in Appendix A.2.

4. Results on Code Summarization

4.1. Main Results

The main results on code summarization are listed in Table 2. Observe that the BLEU scores of all models are relatively low (< 25) due to the difficulty of this task. Retrieval augmented code-to-text methods give better performance than generative models, which shows the effectiveness of leveraging large-scale retrieval databases. Notably, *PinNet* outperforms all baseline models on all datasets. Compared with REDCODER (Parvez et al., 2021), which is the state-of-the-art model, *PinNet* achieves improvements of 0.92 points and 0.53 points on Python and Java datasets, respectively. It should be noted that REDCODER is based on the PLBART backbone (Ahmad et al., 2021) which actually outperforms the GraphCodeBERT (Guo et al., 2020) used by *PinNet*. However, *PinNet* achieves a better result, indicating the effectiveness of the proposed technique.

4.2. Ablation Study

The ablation study is provided in Table 3. Compared with GraphCodeBERT, *PinNet* achieves significant improve-

Table 2. BLEU-4 scores for code summarization on CodeXGLEU

Methods	Python	Java
Seq2Seq	15.93	15.09
Transformer	15.81	16.26
RoBERTa (code)	18.14	16.47
CodeBERT	19.06	17.65
GraphCodeBERT	17.98	17.85
CodeT5	20.01	20.31
UniXcoder	19.13	20.31
PLBART	19.30	18.45
CodeT5+	20.47	20.83
*k*NN-Transformer	17.61	16.73
BM25 + PLBART	19.57	19.71
REDCODER	21.01	22.94
REDCODER-ext	20.91	22.95
PinNet	**21.83**	**23.48**

Table 3. Ablation studies on code summarization

Methods	# Para	Python	Java
GraphCodeBERT	125M	17.98	17.85
PinNet	230M	**21.83**	**23.48**
w/o *PinNet-Dis*	182M	18.52	21.65
w/o $\mathcal{L}_{HR}$	230M	19.93	21.78
w/o *PinNet-Dis* + $\mathcal{L}_{HR}$	182M	17.88	20.31

ments of 3.85 and 4.63 points on Python and Java, respectively. Moreover, *PinNet-Dis* contributes to the final performance with 3.31 and 1.83 on Python and Java, respectively. Though less significant than *PinNet-Dis*, the proposed hierarchical retrieval loss $\mathcal{L}_{HR}$ also achieves around 2 points improvements. Another interesting observation is that the performance of the vanilla retrieval augmented method (the last row in Table 3) is even worse than GraphCodeBERT. This is because many retrieved summaries on Python are irrelevant to the ground truth and the model is seriously misguided. That explains why the improvements on Python brought by *PinNet* are more significant than that on Java.

4.3. Analysis

We conduct experiments to analyze the performance of *PinNet* on different sizes of retrieval sets. We sample multiple subsets from the original retrieval databases with different sampling rates from 0.1% to 10%. The comparing results are shown in Table 4. Note that *PinNet* only achieves comparable performance with REDCODER on small-scale retrieval databases (1% and 0.1%). This is because REDCODER uses 10 retrieved references while *PinNet* only uses the top-1 reference. As the size of the retrieval database increases, the superiority of *PinNet* over baselines becomes more clear. This result shows that *PinNet* can leverage large-scale unlabeled unimodal datasets.

Table 4. Code summarization. The subsets are randomly sampled from the database with different rates. The notation *w/o both* refers to *PinNet* without *PinNet-Dis* and $\mathcal{L}_{HR}$. The results of REDCODER* (Parvez et al., 2021) are from re-implementation using CodeGraphBERT (Guo et al., 2020) as the backbone.

Methods	Sampling Rate				Code
	100%	10%	1%	0.1%	
REDCODER*	19.64	18.69	**18.50**	**18.39**	**Python**
PinNet	**21.83**	**19.30**	18.49	18.16	**Python**
w/o *both*	17.88	17.25	17.24	16.71	**Python**
REDCODER*	21.33	19.31	**18.83**	**18.86**	**Java**
PinNet	**23.48**	**19.92**	18.82	18.42	**Java**
w/o *both*	20.31	18.69	18.55	18.34	**Java**

Table 5. BLEU-4 scores for code-summary retrieval on CodeXGLEU. The ground truth summaries are removed. Multiple implementations of BM25 and DPR using various index keys and values are included.

Methods	Python	Java	Search Index	
			Key	Value
Sparse Retrieval				
BM25-c2c	8.18	6.58	Code	Text
BM25-c2t	1.92	1.82	Text	Text
IR-baseline	12.62	13.46	Code	Text
Dense Retrieval				
DPR-c2c	7.88	7.71	Code	Text
DPR-c2ct	8.12	7.79	Code + Text	Text
SCORE-R	14.98	15.87	Code or Text	Code or Text
PinNet-Ret	**16.15**	**17.07**	Text	Text
w/o $\mathcal{L}_{HR}$	15.13	15.88	Text	Text

The evaluation of *PinNet-Ret* on the CodeXGLEU test sets is shown in Table 5. We compare with both sparse retrievers (e.g., BM25 search index (Robertson et al., 2009) and Apache IR-baseline (Gros et al., 2020)), and dense retrieval models (e.g, DPR (Karpukhin et al., 2020) and SCORE-R (Parvez et al., 2021)).

The results show that the code-to-text search index based on BM25 (Robertson et al., 2009) only achieves an improvement of 2 points due to the semantics and syntactic complexity of source code. For code-to-code search, the performance of the dense retriever (*i.e.*, DPR-c2c) and

Table 6. Comparison with Large Language Models (LLMs) on CodeXGLEU-Java. $\mathcal{B}$: BLEU, $\mathcal{M}$: METEOR; $\mathcal{R}$: ROUGE-L; $\mathcal{S}$: SentenceBERT.

Methods	# Para	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$
PolyCoder	2.7B	7.7	2.3	12.2	12.9
+ Few-shot		13.8	8.8	27.5	49.3
+ Fine-tuning		19.3	13.7	38.0	60.7
CodeGen	2B	8.0	3.1	13.2	15.2
+ Few-shot		15.0	11.8	31.1	55.3
+ Fine-tuning		19.4	14.2	38.5	60.6
StarCoder	3B	9.6	8.6	20.7	33.9
+ Few-shot		15.1	11.6	29.8	55.9
+ Fine-tuning		20.4	14.7	39.5	61.9
ChatGPT	-	10.7	15.4	25.0	57.1
+ *PinNet-Ret*		13.1	**16.2**	28.7	58.8
PinNet	0.2B	**23.5**	16.1	**40.6**	**61.9**

the sparse retriever (*i.e.*, BM25-c2c) are similar, significantly outperformed by code-to-text retrievers. This result indicates that the matching of two code snippets does not guarantee the semantic relevance of the corresponding descriptions. Compared with the state-of-the-art dense retrieval model SCORE-R (Parvez et al., 2021), *PinNet-Ret* outperforms SCORE-R with 1.17 points and 1.20 points on Python and Java respectively. The improvement contributed by the proposed hierarchical retrieval loss $\mathcal{L}_{HR}$ is 1.02 points and 1.19 points on the two test sets, respectively. This study validates that our method can fully leverage the sample information during training, and can explicitly characterize the relevance and inconsistency between code and description. The analysis on the embeddings is presented in Appendix B.1.

4.4. Comparison with Large Language Models.

We also conduct a performance comparison with LLMs. In order to fully quantify the performance from different aspects, we use both lexical-based metrics (BLEU (Papineni et al., 2002), ROUGE-L (LIN, 2004)) and semantic-based metrics (METEOR (Banerjee & Lavie, 2005), SentenceBERT (Reimers & Gurevych, 2019)).

The comparing results are shown in Table 6. As we can see, *PinNet* consistently outperforms all of the compared large language models for almost all metrics, even though our model size is much smaller. When using BERT Score as the criterion, our model has smaller advantages than other metrics. This is because the language style of LLMs is quite different from the ground truth summaries and SentenceBERT measures the semantic correlation rather than lexical matching. To further improve the performance ChatGPT, we augment the prompt using the top-3 code summaries retrieved by *PinNet-Ret*. Notably, the performance of ChatGPT is significantly improved with the augmentation of *PinNet-Ret*. This result shows the promising potential of using the proposed idea to improve augmented methods for large language models. We also use detailed samples from the CodeXGLEU-Python testing set to demonstrate the performance in Table 12 of Appendix B.2.

Table 7. Evaluation of SQL-to-text using BLEU metrics.

Methods	WikiSQL	StackOverflow
Seq2Seq	20.9	13.3
Seq2Seq + cp	24.1	16.6
Seq2Seq + cp + lv	26.3	18.4
Tree2Seq	26.7	17.0
Graph2Seq	29.3	19.9
PreQR2Seq	32.1	21.1
PinNet	**38.4**	**21.9**

Table 8. Ablation Studies on SQL-to-text using BLEU metrics. W: WikiSQL, S: StackOverflow

Methods	# Para	W	S
RoBERTa	182M	36.8	19.1
PinNet	230M	**38.4**	**21.9**
w/o *PinNet-Dis*	182M	37.8	20.4

5. Results on Sql-to-Text Generation

5.1. Main Results

The main results on Sql-to-Text generation is provided in Table 7. One can observe that the *PinNet* achieves the new state-of-the-art results on WikiSQL (Zhong et al., 2017) and StackOverFlow (Iyer et al., 2016). Specifically, *PinNet* obtains BLEU 38.4 and 21.9, respectively, which outperform the existing state-of-the-art approach (Tang et al., 2022) with 6.3 points and 0.8 points, respectively. It is because the size of the retrieval database on WikiSQL is much larger than the one on StackOverFlow.

5.2. Ablation Study

The ablation result on sql-to-text is summarized in Table 8. Compared to RoBERTa, *PinNet* achieves a significant improvement on both datasets. Specifically, *PinNet* outperforms RoBERTa by 1.6 and 2.8 points, respectively. The improvements of the proposed discriminator on the two datasets are 0.6 and 1.5 points, respectively. This result is consistent with the observation on the code summariza-

tion task that the improvement by *PinNet-Dis* is less significance on relatively easy tasks (Java and WikiSQL) than challenging problems (Python and StackOverFlow).

6. Related Work

6.1. Code-to-Text Generation

Using information retrieval techniques to find relevant descriptions to improve the quality of code summarization (Eddy et al., 2013; Haiduc et al., 2010) has been shown to be an effective method. Recently, due to the success of neural networks, many methods exploit deep-learning techniques on this task, based on, e.g., Seq2Seq framework (Iyer et al., 2016; Hu et al., 2018; 2020) and Transformers (Ahmad et al., 2020). These approaches significantly improve the performance on code-to-text summarization, which leverage code syntactic information (Hu et al., 2018; Peng et al., 2021b; Cai et al., 2020) by integrating new network modules (Shi et al., 2021; Peng et al., 2021a).

Observing that code snippets could be reused, recent studies use information retrieval techniques for code-to-text systems (Zhang et al., 2020; Wei et al., 2020; Parvez et al., 2021). Most existing methods (Zhang et al., 2020; Wei et al., 2020) use code-to-code search to retrieve syntactically relevant programs with the corresponding summaries. This method potentially requires a large collection of handcrafted code-text pairs, which is a challenge in real-world scenarios. A more practical way is to only use code-to-text retrieval, e.g., REDCODER (Parvez et al., 2021). Compared with our method, REDCODER can not fully utilize the semantics of relevant descriptions and may be misguided by inaccurate retrieved references.

6.2. SQL-to-Text Generation

SQL-to-Text is very important for database applications because it helps non-experts to understand SQL queries. SQL-to-text can be viewed as a reverse task of text-to-SQL parsing (Zhong et al., 2017; Guo et al., 2018). With the recent public large datasets (Zhong et al., 2017; Iyer et al., 2016), different approaches have been proposed from both the natural language and database communities. Typical text-to-SQL models integrate a tree-based or graph-based encoder into the Seq2Seq framework (Iyer et al., 2016; Eriguchi et al., 2016; Xu et al., 2018). In this work, we apply retrieval augmented code-to-text methods and achieve significant improvements.

7. Conclusion

Code-to-text generation is a practical and challenging problem. The state-of-the-art methods take advantage of information retrieval techniques to improve the performance significantly. However, all of the existing models could be deluded by inconducive retrieved references, due to the possible mismatch between the inputs and the retrievals. To address this limitation, we propose *PinNet* by introducing discriminative representations, which can effectively capture the critical information of retrieved descriptions and eliminate the misleading semantics. To better execute this idea, we also propose a new retrieval objective to effectively leverage the large retrieval database. Extensive experiments on code summarization and SQL-to-text tasks show that the proposed model can significantly outperform the existing models.

For future work, we plan to investigate how to further facilitate this model to understand the potential semantic correlations between code and retrieved texts. We are also interested in applying our method to large language models to obtain further improvements.

Impact Statement

Limitations. PinNet assumes a retrieval pool and only uses the top-1 reference. However, even the top-1 example could be irrelevant, especially when there exist long-distance dependencies that extend beyond the input snippet. In such a case, PinNet would still suffer from the irrelevant retrievals and generate incorrect code summaries. Besides, the newly proposed modules introduce additional parameters and incur 30% runtime overhead.

Societal Implications. The potential societal consequences of our work are summarized below.

- **Positive.** PinNet can assist non-technical people in understanding code. It may inspire more work on the problem of language models being deluded by irrelevant retrievals in RAG.
- **Negative.** PinNet introduces new parameters, which leads to increased energy cost and consequently results in more carbon emissions. Moreover, PinNet could be potentially misused to facilitate network attacks and reverse engineering, which could be a privacy threat.

References

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. 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*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics.

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W.

A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Banerjee, S. and Lavie, A. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72, 2005.

Cai, R., Liang, Z., Xu, B., Li, Z., Hao, Y., and Chen, Y. Tag: Type auxiliary guiding for code comment generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 291–301, 2020.

Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. 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, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

Eddy, B. P., Robinson, J. A., Kraft, N. A., and Carver, J. C. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 13–22. IEEE, 2013.

Eriguchi, A., Hashimoto, K., and Tsuruoka, Y. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 823–833, 2016.

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

Gros, D., Sezhiyan, H., Devanbu, P., and Yu, Z. Code to comment" translation" data, metrics, baselining & evaluation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 746–757, 2020.

Gu, J., Lu, Z., Li, H., and Li, V. O. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1631–1640, 2016.

Guo, D., Sun, Y., Tang, D., Duan, N., Yin, J., Chi, H., Cao, J., Chen, P., and Zhou, M. Question generation from sql queries improves neural semantic parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1597–1607, 2018.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. 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)*, pp. 7212–7225, 2022.

Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pp. 35–44. IEEE, 2010.

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pp. 200–210, 2018.

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3): 2179–2217, 2020.

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

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083, 2016.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

LIN, C. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop, Barcelona, Spain*, pp. 74–81, 2004.

Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., and Chen, W. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

Liu, S., Chen, Y., Xie, X., Siow, J., and Liu, Y. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405*, 2020.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

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

Luong, M.-T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421, 2015.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2022.

OpenAI. Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2023. Last accessed on 2023-07-24.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pp. 311–318. Association for Computational Linguistics, 2002.

Parvez, M. R., Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*, 2021.

Peng, H., Li, G., Wang, W., Zhao, Y., and Jin, Z. Integrating tree path in transformer for code representation. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 9343–9354. Curran Associates, Inc., 2021a.

Peng, H., Li, G., Wang, W., Zhao, Y., and Jin, Z. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems*, 34: 9343–9354, 2021b.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019.

Robertson, S., Zaragoza, H., et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

Rubin, O., Herzig, J., and Berant, J. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2671, 2022.

Shi, E., Wang, Y., Du, L., Zhang, H., Han, S., Zhang, D., and Sun, H. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv preprint arXiv:2108.12987*, 2021.

Sun, W., Fang, C., You, Y., Chen, Y., Liu, Y., Wang, C., Zhang, J., Zhang, Q., Qian, H., Zhao, W., et al. A prompt learning framework for source code summarization. *arXiv preprint arXiv:2312.16066*, 2023.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

Tang, X., Wu, S., Song, M., Ying, S., Li, F., and Chen, G. Preqr: Pre-training representation for sql understanding. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 204–216, 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *arXiv*, 2017.

Wang, T. and Isola, P. Understanding contrastive representation learning through alignment and uniformity on the hypersphere. In *International Conference on Machine Learning*, pp. 9929–9939. PMLR, 2020.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. 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*, pp. 8696–8708, 2021.

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D., Li, J., and Hoi, S. C. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Wei, B., Li, Y., Li, G., Xia, X., and Jin, Z. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 349–360. IEEE, 2020.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Xu, K., Wu, L., Wang, Z., Feng, Y., and Sheinin, V. Sql-to-text generation with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 931–936, 2018.

Zhang, J., Wang, X., Zhang, H., Sun, H., and Liu, X. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1385–1397. IEEE, 2020.

Zhong, V., Xiong, C., and Socher, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

Zhu, X., Sha, C., and Niu, J. A simple retrieval-based method for code comment generation. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1089–1100. IEEE, 2022.

A. Implementation Details

A.1. Datasets

For code summarization task, we use a widely adopted benchmark, CodeXGLUE (Lu et al., 2021), to produce descriptive summaries for given code snippets. The code snippets are in two programming languages, Java and Python. To obtain the reference summaries, we follow (Parvez et al., 2021) to construct the retrieval database on CodeSearchNET (Husain et al., 2019) and CCSD corpus (Liu et al., 2020). The retrieval dataset contains the code summaries of six programming languages (Java, Python, Ruby, Javascript, Go and PHP). Note that all target code summaries in test/developing partitions are excluded from the retrieval database.

For SQL-to-text generation task, the goal is to automatically generate a natural language description to explain an input SQL query. This task is important in real-world database applications, which helps non-expert users to understand complex SQL queries. We conduct experiments on two datasets, WikiSQL (Zhong et al., 2017) and StackOverflow (Iyer et al., 2016). WikiSQL is the largest dataset for SQL understanding tasks with 87,726 handcrafted SQL-text pairs. StackOverflow consists of 32,337 pairs of SQL queries and title from the posted questions in Stack Overflow. For both datasets, we use the training set as the retrieval database.

The detailed statistics of the datasets and retrieval samples are provided in Table 9.

Table 9. Statistics of training/developing/testing samples and retrieval databases of different datasets.

Dataset	**Train**	**Dev**	**Test**	**Retrieval**
Code Summarization				
CodeXGLEU-Java	164,923	5,183	10,955	1,070,229
CodeXGLEU-Python	251,820	13,914	14,918	150,007
SQL-to-Text Generation				
WikiSQL	56,355	8,421	15,878	56,355
StackOverflow	25,671	111	100	25,671

A.2. Model Setup

For both CodeXGLEU-Java and CodeXGLEU-Python datasets, we truncate the input and output sequence to maximum 512 and 80 tokens, respectively. For SQL-to-text generation, the input and output lengths are limited to 256 and 32, respectively.

Both *PinNet-Enc* and *PinNet-Ret* models are initialized with GraphCodeBERT (Guo et al., 2020) and RoBERTa-base (Liu et al., 2019) for code summarization and SQL-to-Text, respectively. Both pre-trained models consist of 12 stacked 8-head transformer encoder layers with the dimension of hidden states being 768. For all tasks, the *PinNet-Dec* network consists of 6 stacked *PinNet-Dec* layers which shares the same embedding shape with the encoder.

The *PinNet-Ret* model is trained with Adam (Kingma & Ba, 2014) using batch size 128. The learning rate is 2×10^{-5}. To train the generation model sufficiently, we set the learning rates of *PinNet-Enc* and *PinNet-Dec* as 1×10^{-5} and 1×10^{-4} , respectively. The generation model is trained with batch size 64. For all models, we use the corpus-level BLEU score (Papineni et al., 2002) on the developing set for validation.

For the retrieval task, we set m_1 and m_2 to 0.2 and 0.4, respectively. The hyper-parameters of the generation loss, α and β, are both set to 1.0.

During training, we use *PinNet-Ret* to select the top-10 code summaries by cosine score. Then, we randomly feed one of them to *PinNet-Enc* as input to prevent over-fitting. To train *PinNet-Dis* sufficiently, we categorize the retrieved descriptions into 6 classes according to the BLEU score: [0,5), [5,10), [10,20), [20,40), [40,60), and [60,100). These intervals are based on the distribution of the description BLEU scores as shown in Figure 5. At runtime, we use beam search and the beam size is set to 4.

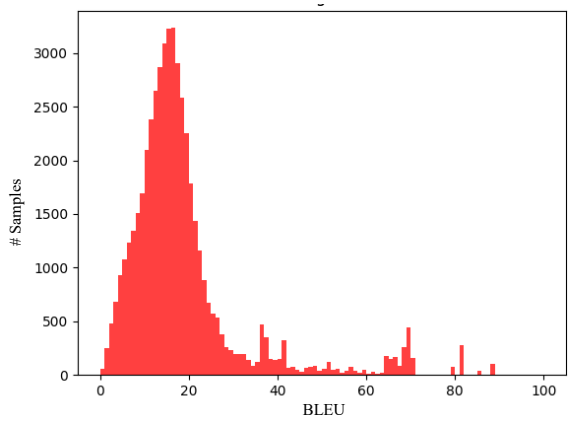

Figure 5. Number of retrieved references vs. BLEU scores.

A.3. Large Language Models Setup

We compare *PinNet* with three popular open-sourced LLMs (PolyCoder, CodeGen, StarCoder). The model size of PolyCoder, CodeGen and StarCoder is 2.7B, 2B, and 3B respectively. For the few-shot setting, we randomly select ten example pairs of (*code snnipet, summary*) and append the examples to the input code snippet. Besides, we also choose GPT-turbo-3.5-16K[1] for comparison. Recent work (Liu et al., 2021; Rubin et al., 2022) has demonstrated that the performance of large language models could be improved when few-shot examples are provided. Inspired by these results, we also augment the prompt with retrieved examples to enable few-shot in-context learning of ChatGPT. Figure 6 shows the prompt template we use for ChatGPT.

> The task is to generate a brief description of a piece of code.
> Please generate a one-sentence comment for the folllowing code snippet:
>
> *<input code>*
>
> Followings are some relevant code comments:
>
> *<code summary* 1*>*
> *<code summary* 2*>*
> *<code summary* 3*>*

Figure 6. Prompt template for ChatGPT to generate code comments. Three retrieved code summaries are provided for few-shot in-context learning.

A.4. Details of the Compared Models

For code summarization, we compare *PinNet* with the state-of-the-art approaches from multiple categories: generative methods, pre-trained models (PLM), and retrieval augmented methods.

A.4.1. Generative Models

The generative models follow the LSTM-based encoder-decoder framework (Seq2Seq) (Luong et al., 2015) or the Transformer architecture (Vaswani et al., 2017), and formulate the code summarization problem as a sequence-to-sequence task.

A.4.2. Pre-trained Language Models

We compare the results of multiple pre-trained models.

[1]https://platform.openai.com/docs/models/gpt-3-5

- RoBERTa (code) (Husain et al., 2019) is a variant of the original RoBERTa model (Liu et al., 2019). RoBERTa (code) is pre-trained on the CodeSearchNet dataset (Husain et al., 2019).
- CodeBERT (Feng et al., 2020) extends BERT (Devlin et al., 2019) with replaced token detection objective (Clark et al., 2020).
- GraphCodeBERT (Guo et al., 2020) is an extension of CodeBERT which encodes the data flow edges between code tokens.
- PLBART (Ahmad et al., 2021) is a sequence-to-sequence model pre-trained on an extensive collection of JAVA and Python functions through denoising autoencoding.
- CodeT5 (Wang et al., 2021) is a variant of T5 (Raffel et al., 2019) which leverages s the code semantics conveyed from the developer-assigned identifiers for pre-training.
- UniXcoder (Guo et al., 2022) exploits the structural information of AST to the performance of code representation.

A.4.3. **Retrieval Augmented Methods**

We also compare *PinNet* with the state-of-the-art methods which enhance the performance of generative models with retrieval augmented references.

- *k*NN-Transformer (Zhu et al., 2022) combines a nearest neighbor retrieval module and a transformer generation model.
- BM25 + PLBART (Ahmad et al., 2021; Robertson et al., 2009) leverages BM25 to retrieve the candidate summary from a code snippet and feed the retrieved information and the input code to a PLBART for summary generation.
- REDCODER (Parvez et al., 2021) trains a dense retriever to find multiple candidate summaries related to a input code fragment. REDCODER uses bimoal retrieved reference information (including both summaries and codes) with a PLBART model (Ahmad et al., 2021).
- REDCODER-ext (Parvez et al., 2021) is a variant of REDCODER which leverages unimoal retrieved candidates (either only summaries or codes).

For Code-to-Text Retrieval, we incorporate the following methods for comparison.

A.4.4. **Sparse Retrieval Methods**

Conventional information retrieval methods represents code or text with sparse vectors to perform lexical matching.

- BM25-c2c (Robertson et al., 2009) is a search index contains code as key and associated text descriptions as values. BM25 is used as the ranking score.
- BM25-c2t (Robertson et al., 2009) contains the code summary as both the key and value. BM25-c2t can use singleton corpus (*e.g.* problem states without any code).
- IR-baseline (Gros et al., 2020) is a search index over the code parts based on Apache Solr[2].

A.4.5. **Dense Retrieval Methods**

Deep learning-based retrievers encode the query into fixed-sized representations and retrieves the relevant key via maximum inner product search.

- DPR-c2c (Karpukhin et al., 2020) is a encoder-only network which encodes both the query and the code in the retrieval database. The similarity of a query and a key is defined by the inner product.

[2]https://solr.apache.org/

Table 10. Training speed (samples/second), inference speed (samples/second) and GPU memory usage (GB).

Method	Training Speed	Inference Speed	GPU Memory Usage
GraphCodeBERT + *PinNet-Ret*	45.35	1.83	1.85
PinNet	39.51	1.39	2.17

Table 11. Evaluation on code-summary retrieval when keeping the target code in the retrieval database. Evaluated on BLEU, Recall at top-K (R@K), Mean reciprocal rank (MRR), *alignment* (ℓ_a) and *uniformity* (ℓ_u) (Wang & Isola, 2020).

Methods	Retrieval Metrics							Code
	BLEU	R@1	R@5	R@10	MRR	ℓ_a	ℓ_u	
SCODE-R (Parvez et al., 2021)	45.71	37.29	56.62	63.79	0.463	0.737	-1.860	
PinNet-Ret	**50.86**	**42.67**	**61.72**	**68.25**	**0.507**	**0.228**	-3.788	**Python**
w/o $\mathcal{L}_{HR}$	47.25	38.39	57.01	63.52	0.473	0.292	**-3.825**	
SCODE-R (Parvez et al., 2021)	48.3	40.48	59.82	66.45	0.495	0.745	-1.853	
PinNet-Ret	**52.23**	**43.8**	**61.0**	**67.8**	**0.517**	**0.207**	**-3.835**	**Java**
w/o $\mathcal{L}_{HR}$	50.77	41.99	59.73	66.28	0.498	0.263	-3.828	

- DPR-c2ct (Robertson et al., 2009) shares the same architecture with DPR-c2c but encodes the concatenation of both code and text.
- SCORE-R is the retrieval module of REDCODER (Parvez et al., 2021). SCORE-R is a cross-encoder which estimates the semantic similarity between code snippets and textual summaries. At runtime, SCORE-R considers both paired data and singletons (*e.g.* code without a description or a problem statement without any code)

For SQL-to-Text evaluation, we categorize the baseline models into three classes.

A.4.6. **Sequence-to-Sequence Models**

These methods follow the RNN-based sequence-to-sequence architecture (seq2seq) (Bahdanau et al., 2014) and also introduce copy mechanism (cp) (Gu et al., 2016) and latent variable (lv) (Guo et al., 2018).

A.4.7. **Structural Information Augmented Models**

We compare with methods that exploit the structural information of SQL queries.

- Tree2Seq (Eriguchi et al., 2016) employs a tree-LSTM to model the interactive information of SQL tokens in a SQL parsing tree. The output text is predicted with an auto-regressive decoder based on LSTM layers.
- Graph2Seq (Xu et al., 2018) exploits a graph neural network to capture the structural information of SQL query. The method represents a SQL query as a directed graph and encodes the graph information with node embedding vectors.

A.5. Training and Inference

The models are trained on a single Nvidia A100 GPU. The batch size is set to 32 for training and 1 for inference respectively. The beam size is set to 4. The training speed, inference speed, and GPU memory usage are provided in Table 10.

B. Evaluation on Code Sumarization

B.1. Analysis of Retrieval Embedding

The retrieval results keeping the target summaries in the retrieval databases are summarized in Table 11. This evaluation is conducted to provide a comprehensive analysis of the retrieval models. Results on BLEU, recall, and MRR show the

proposed retriever significantly outperforms the baseline SCORE-R. Besides we also follow (Wang & Isola, 2020) to use *alignment* and *uniformity* to measure the quality of learned embeddings. *alignment* calculates expected distance between paired instances while *uniformity* measures how well the embeddings are uniformly distributed. From the results, the superiority of *PinNet-Ret* to REDCODER can be observed on representation quality. One can also observe that the proposed hierarchical retrieval loss improves the *alignment* but contributes nothing to *uniformity*. It is interesting to investigate how to improve it via leveraging the semantics of a retrieval database, and we leave it for future study.

Table 12. Cases on CodeXGLEU-Python testing set. The generation errors are highlighted with *italic* type and correct ones are **bolded**. Due to space limit, only the top-1 retrieved results are listed.

1	Code	`def rep1(parser: Union[Parser, Sequence[Input] ]) -> RepeatedOnceParser: if isinstance(parser, str): parser = lit(parser) return RepeatedOnceParser(parser)`
	Summary	Match a parser *one* or more times repeatedly.
	REDCODER	**Retrieval:** Match a parser *zero* or more times repeatedly ; **Generation:** Match a parser *zero* or more times repeatedly .
	PinNet	**Retrieval:** Match a parser *zero* or more times repeatedly . **Generation:** match a parser *one* or more times repeatedly .
	w/o *PinNet-Dis*	**Generation**: parses one or more times .
2	Code	`def get_settings(cls, show_hidden=False): settings = Integration.objects.get_settings(cls.ID) if not show_hidden: for field in cls.HIDDEN_FIELDS: settings.pop(field, None) return settings`
	Summary	Retrieves the settings for this integration as a **dictionary** .
	REDCODER	**Retrieval:** Return settings for given integration as a **dictionary** . **Generation:** Return settings for given integration .
	PinNet	**Retrieval:** return settings for given integration as a **dictionary** . **Generation:** return the settings for this integration as a **dictionary** .
	w/o *PinNet-Dis*	**Generation**: returns the settings for the given integration .
3	Code	`def _log_vector_matrix (vs, ms): return tf.reduce_logsumexp(input_tensor = vs[..., tf.newaxis] + ms, axis=- 2)`
	Summary	Multiply **tensor** of vectors by matrices assuming values stored are **logs** .
	REDCODER	**Retrieval: Multiply tensor** of matrices by vectors **assuming values stored are logs**. **Generation:** *Calculate* the *log matrix* .
	PinNet	**Retrieval: Multiply tensor** of matrices by vectors **assuming values stored are logs**. **Generation:** *combine* **tensors** of **matrices** into log .
	w/o *PinNet-Dis*	**Generation**: *calculate* log vectors

B.2. Case Studies

We use some detailed samples from the CodeXGLEU-Python testing set to demonstrate the performance in Table 12. For this first case, the information *zero* provided by the retrieved reference is inaccurate. All the comparing models except *PinNet* are misguided by the retrieved reference. With the discriminative representations, *PinNet* can focus on the critical semantics and successfully generates the accurate description. For the second case, the retrieved summary contains the critical information *dictionary*. However, both REDCODER and the variant of *PinNet* fail to utilize it. It shows that *PinNet* can better leverage the semantics from the retrieved samples.

The third is a failed case. Though the retrieved reference provides conducive semantics, both REDCODER and *PinNet* fail to leverage the information. This failure is due to the semantical complexity of the code snippet. It is challenging for models to understand the potential correlation between codes and retrieved knowledge. We leave the problem of addressing the semantic obscurities for the future work.