Retrieval Augmented Code Generation and Summarization

Md Rizwan Parvez[§], Wasi Uddin Ahmad[§], Saikat Chakraborty[†]
Baishakhi Ray[†], Kai-Wei Chang[§]
[§]University of California, Los Angeles, [†]Columbia University
[§]{rizwan, wasiahmad, kwchang}@cs.ucla.edu, [†]{saikatc, rayb}@cs.columbia.edu

Abstract

Software developers write a lot of source code and documentation during software development. Intrinsically, developers often recall parts of source code or code summaries that they had written in the past while implementing software or documenting them. To mimic developers' code or summary generation behavior, we propose a retrieval augmented framework, REDCODER, that retrieves relevant code or summaries from a retrieval database and provides them as a supplement to code generation or summarization models. REDCODER has a couple of uniqueness. First, it extends the state-of-the-art dense retrieval technique to search for relevant code or summaries. Second, it can work with retrieval databases that include unimodal (only code or natural language description) or bimodal instances (code-description pairs). We conduct experiments and extensive analysis on two benchmark datasets of code generation and summarization in Java and Python, and the promising results endorse the effectiveness of our proposed retrieval augmented framework.

1 Introduction

In recent years, automating source code generation and summarization is receiving significant attention due to its potential in increasing programmers' productivity and reducing developers' tedious workload. Consequently, various approaches have been explored in the literature to facilitate code generation (Yin and Neubig, 2017; Gu et al., 2016) and code documentation/summarization (Ahmad et al., 2020; Wei et al., 2019; Allamanis et al., 2018). Despite initial success, most of the generated code still suffers from poor code quality (Xu et al., 2021). Therefore, the question remains—how to generate better code from a given summary and vice versa.

Source code generation and summarization, however, are intrinsically complex and challenging. They involve generating diverse token sequences such as different variables, operators, keywords, classes, and method names (Parvez et al., 2018), which requires understanding the programming languages at lexical, syntax, and semantics levels. To combat these issues, recent studies (*e.g.,* Ahmad et al. (2021); Guo et al. (2021); Xu et al. (2020); Feng et al. (2020a); Xu et al. (2020)) take a learning-based approach—they train representations of code and the associated text by leveraging existing high-quality source code and short text descriptions available in open-source repositories and question answering forums such as GitHub and Stack Overflow. Then fine-tune the representation models on the downstream tasks. Although these dataset contains high-quality human-written code and text, since the existing approaches do not directly leverage them during the generation process, the gain achieved by these approaches is still limited, especially when the source code is long.

To overcome this, we take advantage of the existing high-quality source code and their description by including them directly in the generation process that are retrieved via information retrieval technique. In this work, we present REDCODER, a **R**etrieval augment**ED** **COD**e g**E**neration and summa**R**ization framework. While designing REDCODER, we take motivation from how developers take advantage of existing resources. For example, developers often search for relevant code in the code repository, and if found, adapt the retrieved code in their own context. Similarly, when an API usage is unclear, they search in question answering forums (*e.g.,* StackOverflow) (Brandt et al., 2010; Sadowski et al., 2015). Such an additional resource helps developers to increase their development productivity (Li et al., 2013).

We design REDCODER as a two-step process (see Figure 1). In the first step, given the input (*nl* text for code generation, or *code snippet* for summarization) a *retriever* module retrieves relevant source code (for code generation) or summaries

Findings of the Association for Computational Linguistics: EMNLP 2021, pages 2719–2734
November 7–11, 2021. ©2021 Association for Computational Linguistics

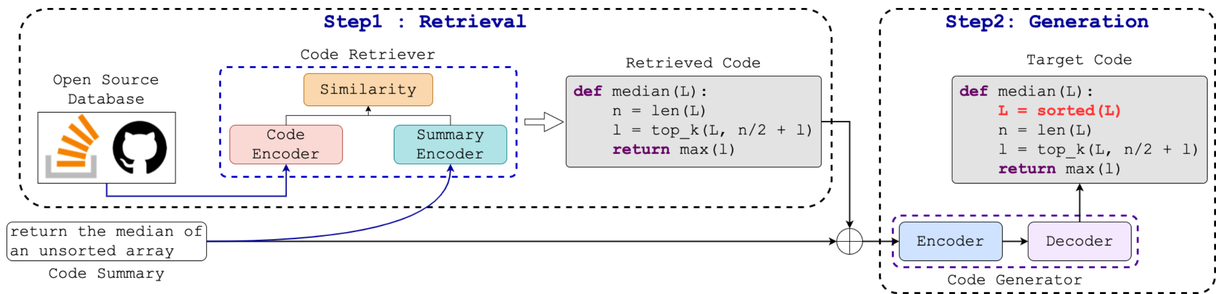

Figure 1: Illustration of our proposed framework REDCODER for code generation. Given an input summary, we first retrieve top-k candidate code (k=1 in this example). We then aggregate them and based on that a *generator* module generates the target sequence.

(for code summarization) from a database.[1] In the second step, a *generator* processes the retrieved code/summary along with the original input to generate the target output. In this way, REDCODER enhances the generation capability by augmenting the input through retrieval. The two-step process allows us to design a modular and configurable framework for source code and summary generation. Various designs of retriever and generator models can be incorporated into this framework.

Existing cross-encoder code retrievers being computationally expensive, their applicability to retrieve from a large database is limited (Humeau et al., 2020). A natural choice would be to use sparse term based retrievers such as TF-IDF or BM25 (Robertson and Zaragoza, 2009). However, the *retriever* module in REDCODER should exhibit a good understanding of source code and programmers' natural language, which is a non-trivial task due to the syntactic and semantic structure of the source code (Guo et al., 2021; Ahmad et al., 2021). Such an expectation of searching for semantically similar code and summary may not be attainable by a sparse token level code retriever (*e.g.,* BM25). To that end, we design the *retriever* module in REDCODER based on programming languages (PL) and natural languages (NL) understanding models (*e.g.,* GraphCodeBERT (Guo et al., 2021)). This *retriever* module extends the state-of-the-art dense retrieval technique (Karpukhin et al., 2020) using two different encoders for encoding the query and document.

As for the *generator*, REDCODER can handle retrieval databases consisting of both unimodal (only code or natural language description) and bimodal instances (code-description pairs) and makes the best usage of all the auxiliary information that

[1]The database could be open source repositories (*e.g.,* GitHub) or developers' forums (*e.g.,* Stack Overflow).

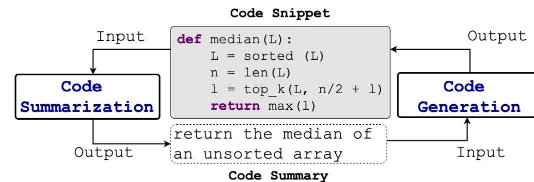

Figure 2: Example input/output for the code generation and summarization tasks.

are available. Yet, to incorporate information, we augment the retrieved information only in the input level. It does not modify the underlying architecture of the *generator* module —preserving its model agnostic characteristics.

We evaluate the effectiveness of REDCODER on two popular programming languages (Java and Python) on both code generation and code summarization tasks. The empirical results show that, REDCODER's concept of *retrieval augmented generation* elevates the state-of-the-art code generation from an Exact Match score of 18.6 to 23.4 and the summary generation BLEU-4 score from 18.45 to 22.95 even when we forcefully remove the target candidate from the retrieved code or summary. With further experiments, we establish the importance of both the retrieved code and retrieves summary in the generation process. The source code for reproducing our experiments are at `https://github.com/rizwan09/REDCODER`.

2 Background

We first introduce the problem formulation and discuss the fundamentals of the *retriever* and *generator* components that REDCODER is built upon.

2.1 Problem Formulation

Our goal is two folds: (i) code generation: Generating source code (C), given their natural language description, such as code summaries, code comments or code intents (S); (ii) code summarization: Generating natural language summaries S, given source code snippets C. Fig 2 shows an example.

Let X and Y denote a collection of input and output sequences ($X = S_1, \ldots, S_n$, $Y = C_1, \ldots, C_n$ in code generation, $X = C_1, \ldots, C_n$, $Y = S_1, \ldots, S_n$ in summary generation). We assume that we have access to a retrieval database consisting of an extensive collection of source code (*e.g.,* aggregated from GitHub or Stack Overflow) or summaries (*e.g.,* docstrings, code comments) (Y_R). Note that, target sequences (Y) may or may not be present in the retrieval database (Y_R). Now, given an input $x \in X$, a *retriever* retrieves the top-k relevant output sequences from the database: $\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_k \in Y_R$. Then the input sequence x is augmented with the retrieved sequences to form $x' = x \oplus \mathcal{Y}_1 \oplus \mathcal{Y}_2 \ldots \oplus \mathcal{Y}_k$, where $\oplus$ denote the concatenation operation. Finally, a *generator* generates the target output $y \in Y$ given x'. In the following, we first discuss the base *retriever* and *generator* modules used in REDCODER and then how we improve these components is in Section 3.

2.2 Retriever: DPR

Information retrieval (IR) systems or retriever models are designed to retrieve the top-k relevant documents that presumably best provide the desired information (Manning et al., 2008). Term-based retrieval methods, *a.k.a.* sparse retrieval models, such as TF-IDF or BM25 (Robertson and Zaragoza, 2009) use sparse vector representations to perform lexical matching and compute relevance scores to rank the documents based on a query.

On the other hand, dense retrieval methods encode documents into a fixed-size representations and retrieve documents via maximum inner product search (Sutskever et al., 2014; Guo et al., 2016). Particularly of interests, Karpukhin et al. (2020) propose a Dense Passage Retriever (DPR) model for open-domain question answering (QA). It consists of two encoders ($Q(.)$ and $P(.)$) that encode queries and passages, respectively. The similarity of a query q and a passage p is defined by the inner product of their encoded vectors $sim(p, q) = Q(q)^T \cdot P(p)$. Given a query q, a positive (relevant) passage p^+, and a set of n irrelevant passages p_i^-, DPR optimizes the classification loss:

$$L = -\log \frac{e^{sim(q,p^+)}}{e^{sim(q,p^+)} + \sum_{i=1}^{n} e^{sim(q,p_i^-)}}.$$

Karpukhin et al. (2020) propose to fine-tune DPR using *in-batch negatives* (Gillick et al., 2019; Yih et al., 2011) with curated "hard" negatives using BM25 (candidates with high BM25 scores but contain no sub-string that match the target). We refer to Karpukhin et al. (2020) for details.

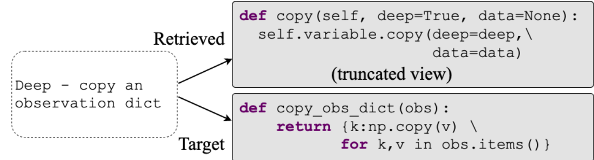

Figure 3: An example retrieved code that is relevant yet does not match the reference.

2.3 Generator: PLBART

PLBART (Ahmad et al., 2021) is a sequence-to-sequence Transformer model (Vaswani et al., 2017) that is pre-trained on a huge collection of source code and natural language descriptions via denoising autoencoding. PLBART has shown promise in several software engineering applications, including code generation and summarization. We adopt PLBART as the generator module in our proposed framework, REDCODER.

3 Proposed Framework: REDCODER

Our proposed code generation and summarization framework, REDCODER generates the target code or summary by augmenting the input x with relevant code snippets or summaries. We build our *retriever* module by training a DPR model differently from (Karpukhin et al., 2020). With an intelligent scheme, we then augment the retrieved candidates and their pairs (if available) to provide auxiliary supervision to the *generator*. We briefly describe the model components in this section.

3.1 Retriever: SCODE-R

Architecture The *retriever* module of REDCODER is built upon the DPR model (Karpukhin et al., 2020) and we call it SCODE-R (Summary and CODE Retriever). SCODE-R composed of two encoders that encode source code and natural language summary. We use bidirectional Transformer encoders (Vaswani et al., 2017) that are pre-trained on source code and natural language summaries. Specifically, we explore CodeBERT (Feng et al., 2020b) and GraphCodeBERT (Guo et al., 2021) as the code and summary encoders for SCODE-R.

Input/Output SCODE-R takes an input sequence x (code or summary) and retrieves a set of relevant documents from a database of output sequences Y (if the input is code, then the output

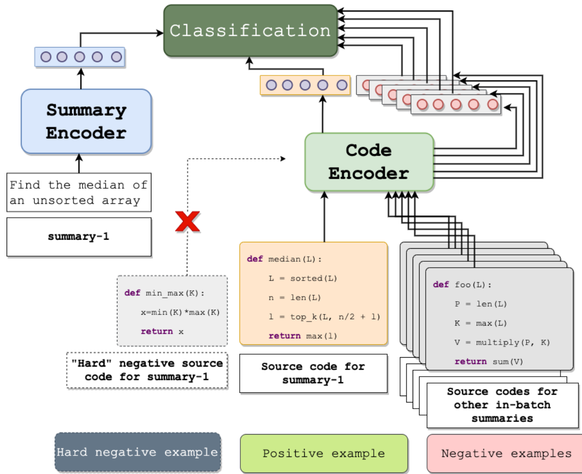

Figure 4: Training scheme of the *retriever* module (SCODE-R) of our proposed framework REDCODER for the code generation task. Unlike in open-domain QA (Karpukhin et al., 2020), we do not use "hard" negatives (*e.g.,* candidates retrieved by BM25 that do not exactly match the reference) during fine-tuning.

is summary and vice versa). SCODE-R returns the the top-k output sequences $\{\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_k\}$, where $sim(x, \mathcal{Y}_i) \geq sim(x, \mathcal{Y}_j) \forall j > i$.

Training We fine-tune SCODE-R using a set of parallel examples (x_i, y_i) of code and summaries. As mentioned in Section 2.2, DPR originally proposed to be fine-tuned using *in-batch negatives* and curated "hard" negatives from BM25 retrieved passages for open-domain QA. The key idea behind "hard" negatives is to fine-tune DPR to distinguish the target passage from relevant passages that do not contain the target answer. However, unlike open-domain QA, a retrieved code or summary that is not the target could still benefit code generation or summarization (verified in Section 6). We provide an example in Figure 3; although the retrieved code does not match the target one but can facilitate generating it. Therefore, we fine-tune SCODE-R without any "hard" negatives. Specifically, for each training instance (x_i, y_i), the corresponding output y_i is considered as positive and the other in-batch outputs (*i.e.,* the outputs of other instances in the same batch - $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{bsz}$) as negatives. Figure 4 shows an example of SCODE-R fine-tuning for code generation task.

3.2 Generator: SCODE-G

We adopt PLBART as discussed in Section 2.3 as the *generator* module of REDCODER and call it SCODE-G (Summary and CODE Generator). The input sequence x is concatenated with the top-k retrieved sequences to form the augmented input sequence, $x' = x \oplus \mathcal{Y}_1 \oplus \mathcal{Y}_2 \ldots \oplus \mathcal{Y}_k$. The augmented input x' is fed to PLBART to estimate $p_{gen}(y|x')$.

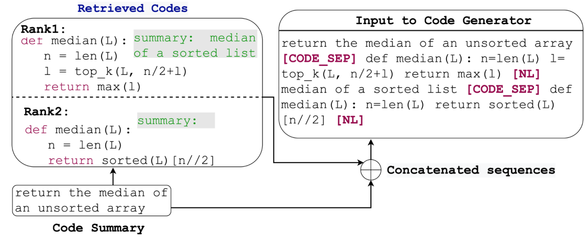

Figure 5: REDCODER-EXT input for code generation.

Note that a source code often consists of docstrings, comments that can be extracted to form code – summary pairs. In the retrieval databases, code and summaries are either singleton (*e.g.,* code without a description or a problem statement without any code) or parallel. Therefore, we consider two retrieval settings that require separate modeling consideration for the generator.

Case 1: Retrieve candidates are singleton In this case, we concatenate the original input sequence x and the top-k retrieved candidates with a special separator token.

$$x' = x \; [csep] \; \mathcal{Y}_1 \; [csep] \; \mathcal{Y}_2 \; \ldots \; [csep] \; \mathcal{Y}_k.$$

This is our default setting and we refer this as REDCODER in this work.

Case 2: Retrieve candidates are pairs In this case, retrieved candidates are pair of code and natural language (NL) summary. We augment the input sequence using both of them as follows.

$$x' = x \; [csep] \; \mathcal{Y}_1 \; [nsep] \; \mathcal{X}_1 \; [csep] \; \mathcal{Y}_2 \\ [nsep] \; \mathcal{X}_2 \; \ldots \; [csep] \; \mathcal{Y}_k \; [nsep] \; \mathcal{X}_k,$$

where $\mathcal{X}_j$ and $\mathcal{Y}_j$ are parallel sequences (*e.g.,* $\mathcal{Y}_j$ is a piece of code and $\mathcal{X}_j$ is its corresponding summary for the code generation task) retrieved from the database. We conjecture that the additional information $\mathcal{X}_j$ complements the input sequence x and verify its effectiveness in the experiments.

Note that retrieve candidates could be a mix of singleton and pairs. In case of a singleton candidate, we simply replace $\mathcal{X}_j$ or $\mathcal{Y}_j$ with an empty string. We refer this setting as REDCODER-EXT. Although, REDCODER-EXT is a more general setting which includes "Case 1", we study them separately to understand how these two retrieval settings benefit the target tasks. We illustrate an example on code generation in Figure 5. In both

Dataset	Gen.	Sum.	Lang.	Train	Valid	Test	|Code|	|Summary|
CodeXGLUE (Lu et al., 2021)	✓	✓	Java	164,923	5,183	10,955	97	12
			Python	251,820	13,914	14,918	99	14
Concode (Iyer et al., 2018)	✓	✗	Java	100,000	2,000	2,000	27	72

Table 1: Dataset Statistics. Gen., and Sum. refers to code generation and summarization tasks respectively. Summary denotes a natural language description paired with each code. For Concode, the input summary includes the corresponding environment variables and methods. All lengths are computed and averaged before tokenization.

cases, the augmented input $x^{'}$ is truncated to match PLBART's maximum input length 512.

4 Experiment Setup

In order to investigate the effectiveness of our framework, we perform a comprehensive study and analysis on code generation and summarization in two programming languages, Java and Python.

4.1 Datasets and Implementations

Datasets We perform evaluation on both the tasks using the code summarization dataset from CodeXGLUE (Lu et al., 2021). It is curated from CodeSearchNet (Husain et al., 2019) by filtering noisy examples. In addition, we conduct code generation experiments in Java using the Concode benchmark (Iyer et al., 2018). The dataset statistics are summarized in Table 1.

Retrieval Databases To generate a source code given its natural language description or a summary given the code, our proposed approach REDCODER first retrieves prospective candidates from an existing code or summary database. We form the code retrieval database using the deduplicated source code (on average 1.4M functions in Java and Python) that consists of both paired (59%) and monolingual code, released in CodeSearchNET (Husain et al., 2019). As for building the summary retrieval database, we extract the high quality natural language summaries from the paired instances in the training sets of CodeSearchNET. As many of the summaries are duplicated, we also consider the training sets in the other four available languages Ruby, Javascript, Go, and PHP. We then further enlarge it by aggregating the additional summaries from the CCSD corpus (Liu et al., 2021). After performing deduplication, we retain 1.1M unique code summaries and for evaluating REDCODER-EXT, 20% of them can be used as pairs with the corresponding Java and Python source code. We provide the statistics of the retrieval databases in Appendix. Note that the retrieval databases contain code and summaries that are curated from real developers' open sourced repositories on GitHub. By default, we exclude the target code/summary from the retrieval database.

Implementations As mentioned in Section 3, REDCODER has two disjoint components. First, the dense retriever SCODE-R is implemented adopting DPR (Karpukhin et al., 2020) and the encoders in DPR are initialized from GrpahCodeBERT available in the Huggingface API (Wolf et al., 2020). In addition, we implement a baseline BM25 retriever. We use the official codebase of PLBART (Ahmad et al., 2021) and set max epoch to 15, patience to 5, learning rate to 2×10^{-5}. We tune the batch size in {8, 16, 32, 64, 72} and the k value for top-k retrieval up to 10 for code generation and in range {10, 30, 50, 100} for code summarization. As some candidate code and summaries are short in length, we tune with this upper bound of k to accommodate as many candidates as possible within PLBART's maximum input length.

4.2 Evaluation Metrics

BLEU Following prior works (Ahmad et al., 2021; Feng et al., 2020a), we compute the corpus level BLEU (Papineni et al., 2002) and the smoothed BLEU-4 (Lin and Och, 2004) scores for code generation and summarization tasks.

CodeBLEU To demonstrate syntactic and semantic data flow correctness of code generation models, we report CodeBLEU (Ren et al., 2020). CodeBLEU is a weighted average of lexical, abstract syntax tree, and data flow match.

Exact Match (EM) indicates the percentage of output sequences that exactly match the references.

4.3 Baseline Methods

We compare REDCODER *w.r.t.* a number of state-of-the-art code models. We classify them into two categories: (i) retrieval based models and (ii) generative models. We study both generative models that are trained from scratch and are pre-trained on programming and natural languages.

Method		Java			Python		
Type	Name	EM	BLEU	CodeBLEU	EM	BLEU	CodeBLEU
Retrieval Based	BM25	0.00	4.90	16.00	0.00	6.63	13.49
	SCODE-R	0.00	25.34	26.68	0.00	22.75	23.92
Generative	CodeBERT	0.00	8.38	14.52	0.00	4.06	10.42
	GraphCodeBERT	0.00	7.86	14.53	0.00	3.97	10.55
	CodeGPT-adapted	0.00	7.10	14.90	0.01	3.11	11.31
	PLBART	0.00	10.10	14.96	0.00	4.89	12.01
Retrieval Augmented Generative	BM25 + PLBART	0.10	11.37	15.52	0.03	6.99	13.89
	REDCODER	8.95	26.92	31.15	8.88	22.74	28.93
	REDCODER-EXT	**10.21**	**28.98**	**33.18**	**9.61**	**24.43**	**30.21**

Table 2: Results on code generation on CodeXGLUE (Lu et al., 2021).

Methods	EM	BLEU	CodeBLEU
Retrieval based methods			
BM25	0.0	20.3	23.7
SCODE-R	0.0	32.6	36.5
Generative methods			
Seq2Seq	3.1	21.3	26.4
Guo et al. (2019)	10.1	24.4	29.5
Iyer et al. (2019)	12.2	26.6	-
GPT-2	17.4	25.4	29.7
CodeGPT-2	18.3	28.7	32.7
CodeGPT-adapted	20.1	32.8	36.0
CodeBERT	18.0	28.7	31.4
GraphCodeBERT	18.7	33.4	35.9
PLBART	18.6	36.7	38.5
Retrieval augmented generative methods			
BM25+PLBART	21.4	40.2	41.8
REDCODER	**23.4**	41.6	**43.4**
REDCODER-EXT	23.3	**42.5**	**43.4**

Table 3: Code generation results on Concode dataset. SCODE-R was initialized with CodeBERT. GraphCodeBERT initialized results are similar.

Methods	Python	Java
Retrieval based methods		
BM25	1.92	1.82
SCODE-R	14.98	15.87
Generative methods		
Seq2Seq	15.93	15.09
Transformer	15.81	16.26
RoBERTa	18.14	16.47
CodeBERT	19.06	17.65
GraphCodeBERT	17.98	17.85
PLBART	19.30	18.45
Retrieval augmented generative methods		
BM25 + PLBART	19.57	19.71
REDCODER	**21.01**	22.94
REDCODER-EXT	20.91	**22.95**

Table 4: Evaluation BLEU-4 score for code summarization on CodeXGLUE. Baseline results are reported from Ahmad et al. (2021).

Retrieval based models We examine two retriever baselines and consider the top-1 retrieved candidate as the prediction.

• **Dense Retriever** We consider DPR as the dense retriever baseline. We evaluate both the officially released models trained on the natural language open-domain QA task and a variant called DPR (code) that we fine-tune on the evaluation datasets.

• **Sparse Retriever** The second baseline is a sparse retriever that uses the BM25 algorithm to compute relevance scores.

Generative models The generative models work in a sequence-to-sequence (Seq2Seq) fashion.

• **RoBERTa, RoBERTa (code)** RoBERTa models (Liu et al., 2019) pre-trained on natural language corpora, and source code from CodeSearchNet (Husain et al., 2019) respectively.

• **CodeBERT** (Feng et al., 2020a) is pretrained with a hybrid objective incorporating masked language modeling (Devlin et al., 2019) and replaced token detection (Clark et al., 2020).

• **GraphCodeBERT** (Guo et al., 2021) is pretrained by modeling the data flow graph of source code. GraphCodeBERT holds the state-of-the-art results on code search using CodeSearchNet.

• **GPT-2, CodeGPT-2, and CodeGPT-adapted** are GPT-style models that are pre-trained on natural language (Radford et al., 2019) and code corpora CodeXGLUE (Lu et al., 2021).

• **PLBART** (Ahmad et al., 2021) is the generator module of our proposed framework.

In addition, we train an LSTM based Seq2Seq model with attention mechanism (Luong et al., 2015) and a Transformer model (Vaswani et al., 2017) on the benchmark datasets.

Methods	CodeXGLUE (Java)			CodeXGLUE (Python)			Concode (Java)		
	BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU
SCODE-R	36.6	21.0	37.9	35.6	19.2	35.1	70.3	61.7	72.0
REDCODER	36.3	29.4	41.4	32.1	27.5	38.0	76.7	67.5	76.5
REDCODER-EXT	**42.8**	**37.0**	**47.3**	**38.9**	**34.5**	**43.8**	**81.7**	**76.2**	**81.7**

Table 5: Results on code generation keeping the target code in the retrieval database.

Settings	Methods	Python	Java
Cross-Encoder	RoBERTa	0.587	0.599
	RoBERTa (code)	0.610	0.620
	CodeBERT	0.672	0.676
	GraphCodeBERT	**0.692**	**0.691**
Bi-Encoder	DPR	0.093	0.064
	DPR (code)	0.398	0.462
	SCODE-R	0.690	0.686

Table 6: MRR results on code retrieval from the validation and test set in CodeXGLUE. Our bi-encoder retriever SCODE-R is comparable with other cross-encoder models while it is much faster. DPR refers to Karpukhin et al. (2020) and DPR (code) is trained with BM25 "hard" negative training schema built upon our source code datasets.

5 Results

5.1 Code Generation

Table 2 and Table 3 show the evaluation results on code generation from summary descriptions on CodeXGLUE, and Concode datasets, respectively. First, we compare REDCODER with the state-of-the-art code generation models. They are transformers models pre-trained with different objectives using external resources of different sizes. Among them, the relatively strong baseline PLBART has an EM score of 18 on the Concode dataset while it rarely generates any code that matches the real target code in CodeXGLUE (See Table 2) (more discussion on this is in Appendix). The BLEU and CodeBLEU scores are also low. Such result indicates that automated code lacks quality and correctness without the proper supervision in the input to the generator.

Among the retriever-only models, SCODE-R significantly outperforms BM25 (more comparison is in § 6). As expected, the EM is zero as targets are filtered from the retrieval, and CodeBLEU scores are high as they are real code. However, although the retrieved code does not exactly match the target code, they are quite relevant (*e.g.,* Figure 3; more in Appendix). When comparing retrieval-only models to generative models, it is interesting to note that SCODE-R surpasses PLBART by a large margin on CodeXGLUE (Table 2), suggesting that retrieved code has high overlapping with target code that can benefit the generation.

Overall, the retrieval augmented generative models excel in code generation. Our proposed framework REDCODER outperforms PLBART by a large margin, validating the advantage of reusing existing codebases to help code generation. The REDCODER-EXT gains are even higher. For CodeXGLUE (Java, Python) and Concode, the gains in BLEU are 18.88, 19.54, and 5.8. Comparing REDCODER to REDCODER-EXT shows that BLEU scores on Concode and all metrics on CodeXGLUE are improved by ~1%. These results confirm our conjecture that complementing input with paired summaries of the retrieved code help code generation. We provide a qualitative example in the Appendix to explain how the retrieved information helps PLBART in generation.

5.2 Code Summarization

We compare REDCODER with three sets of baseline methods for code summarization, and Table 4 shows the results. Among the two retrieval base methods, SCODE-R performs significantly well, confirming the advantages of dense retrieval over its sparse counterpart. Out of the generative methods, PLBART excels on code summarization as it leverages an extensive collection of natural language descriptions during pre-training. As anticipated, retrieval augmented generative methods outperform the other two sets of models. We see that the "BM25 + PLBART" model improves over PLBART, confirming our conjecture that retrieval augmented techniques have the promise to improve code summarization. Our proposed framework REDCODER and its variant REDCODER-EXT outshine "BM25 + PLBART", surpassing its performance by ~1.5 and ~3.2 points for Python and Java languages, respectively.

6 Analysis

In this Section, we analyze REDCODER's performance on the following points.

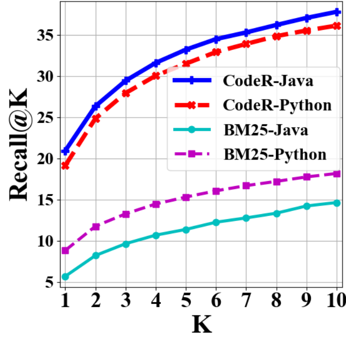

Figure 6: Recall@K for CodeR and BM25. CodeR refers to SCODE-R used for source code retrieval.

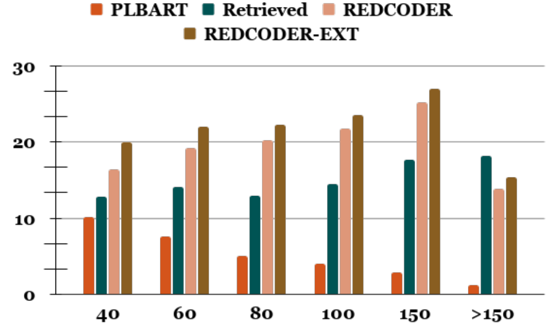

Figure 7: (Python) Code gen. BLEU vs target len.

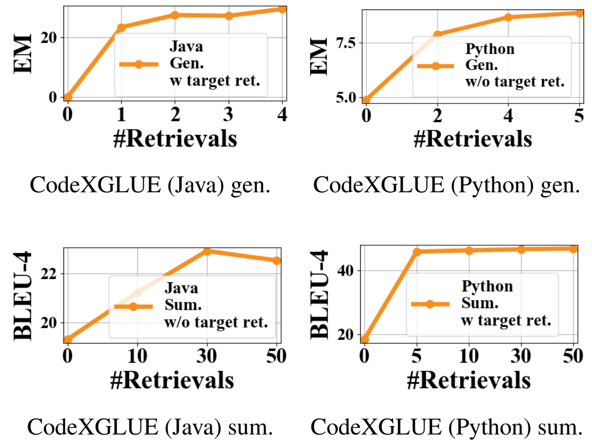

Figure 8: Code gen. and sum. performance vs #retrievals. In general performance improves with higher number of augmented candidates.

Retrieval database includes the target sequence Table 5 shows the code generation results when we did not filter the target from the retrieval (summarization results are in Appendix). As expected, SCODE-R performances are much better than those in Table 2, 3, and 4. In all cases, REDCODER gets more enhanced when target is present in the retrieval database. For the code generation task, we plot the recall@k curve for k upto 10 for both Java and Python on CodeXGLUE dataset when the retrieval contains the target in Figure 6. As we can see, SCODE-R significantly outperforms in both languages and for all k values.

Bi-encoder SCODE-R vs cross-encoder retrievers Table 6 shows the retrieval performance of different alternative retrieval techniques that we considered in REDCODER. SCODE-R performs comparably well with GraphCodeBERT while being significantly faster and scalable (Humeau et al., 2020). Note that, SCODE-R also uses GraphCodeBERT to initialize its encoders (see Figure 4). However, SCODE-R's design of using different encoders for query and documents enables pre-indexing of database and faster retrieval in practice.

Performance vs target length Figure 7 shows the code generation performances of different models *w.r.t.* the target code length for Python. While the generator model (PLBART)'s performance consistently decreases with increasing code size, the retriever (SCODE-R) performs consistently well. Such consistent performance from SCODE-R boosts performance of REDCODER (and also REDCODER-EXT) *significantly higher* than the generative model counterpart. For Java, we find similar results (details in Appendix).

Performance vs #retrievals Figure 8 shows that typically the performance improves more with more retrievals on both tasks. However, roughly 5 code and 30 summaries work sufficiently well.

Human evaluation Finally, we evaluate the quality of code generated by SCODE-G using human evaluation. In Table 7, we perform a human evaluation for code generation task on a subset of the test set in CodeXGLUE (Python). In this study, we compare REDCODER generated code with the code retrieved by SCODE-R. Note that both REDCODER and SCODE-R using the same retrievers, but REDCODER generates code using SCODE-G, while SCODE-R outputs code written by real programmers. We sample 30 instances where REDCODER generated code has a lower BLEU score than that of the SCODE-R and investigate whether the quality of code generated by them are significantly different on these cases.

As programming requires a specific skill, we do not evaluate the quality of the code generation using the mass crowd workers. We recruit 7 Ph.D. students studying in computer science as volunteers[2] to score (1 to 5) code based on three criteria

[2]Before participating in the evaluation process, all the participants are informed that it is a voluntary task and it may

Model	Human Evaluation			Automatic Metric		
	Similarity	Relevance	Compilability	BLEU	EM	CodeBLEU
SCODE-R	2.09	3.00	3.16	11.56	0.00	16.66
REDCODER	2.06	2.94	3.10	10.70	0.07	18.31

Table 7: Human evaluation on code generation (CodeXGLUE-Python). REDCODER (SCODE-R + SCODE-G) achieves similar scores as SCODE-R that directly retrieves developers' written code which suggests that the quality of the code generated by SCODE-G are competitive with real code from programmers' perspective.

(i) similarity, and (ii) relevance *w.r.t.* the target code; (iii) the compilability of the generated code.

The ratings show that both models receive similar scores, with a slightly higher score for SCODE-R in terms of similarity to the target code, relevancy, and compilability. This shows that the quality of the code generated by SCODE-G are competitive with real code from programmers' perspective. Interestingly, REDCODER achieves higher scores than SCODE-R in CodeBLEU and Exact Match even on the cases where its BLEU score is lower.

7 Related Works

Code Summarization. In recent years, source code summarization attracted a lot of attention (Iyer et al., 2016; Liang and Zhu, 2018; Allamanis et al., 2016; Hu et al., 2018b; Ahmad et al., 2020). Many of these works view code as a sequence of token. Other approaches leverage the structural properties of code using Tree based model (Shido et al., 2019; Harer et al., 2019; Hu et al., 2018a; LeClair et al., 2019). In literature, several retrieval-based methods were proposed that leverage retrieved information along with the input code. For example, Zhang et al. (2020) retrieves similar code snippet and use those as an auxiliary input for summarization. On the other hand, Hayati et al. (2018) retrieves related summaries for augmenting summarization input. Different from these approaches, REDCODER leverages both the retrieved code and its summary to augment the input.

Code Generation. Generating source code is a major stepping stone towards automated programming. Yin and Neubig (2017), and Rabinovich et al. (2017) proposed code generation as abstract syntax tree generation to ensure its syntactic correctness. Recent advancements in pre-training language models on unlabeled source code data (Lu et al., 2021; Ahmad et al., 2021) showed colossal promise towards learning code syntax and semantics, resulting in improved code generation models.

take roughly 30 minutes to perform the evaluation.

Code Retrieval and Others. Numerous software engineering applications require information retrieval. Sadowski et al. (2015); Xia et al. (2017); Stolee et al. (2014); Sim et al. (2011) show that developers search for related code, API examples for implementing or adapting new APIs. Design of REDCODER is inspired by developers' behavior while writing code. Developers use search engines for retrieving off-the-shelf libraries (Hucka and Graham, 2018), or "usable" source code (Rahman et al., 2018) for adapting in the development process (Nasehi et al., 2012; Arwan et al., 2015; Ponzanelli et al., 2014). Similarly, REDCODER retrieves existing code or summaries and adapts them to generate the target code or summary. In contrast, Hashimoto et al. (2018) optimizes a joint objective; Zhang et al. (2020); Liu et al. (2021) do not consider any decoder pre-training, Lewis et al. (2020) fine-tunes both of the retriever and the generator end-to-end. For open domain QA, Izacard and Grave (2021) propose a similar model of alternative generator (multi-encoder uni-decoder).

8 Conclusion

We propose REDCODER to automate developers' writing of code and documentation by reusing what they have written previously. We evaluate REDCODER on two benchmark datasets and the results demonstrate a significant performance boost with the help of the retrieved information. In the future, we want to extend REDCODER to support other code automation tasks such as code translation.

Acknowledgments

We thank anonymous reviewers for their helpful feedback. We also thank the UCLA NLP group for helpful discussions, comments, and participating voluntarily in the human evaluation. This work was supported in part by NSF OAC-1920462, SHF-2107405, SHF-1845893, IIS-2040961, IBM, and VMWare. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the US Government.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

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*.

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

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org.

Achmad Arwan, Siti Rochimah, and Rizky Januar Akbar. 2015. Source code retrieval on stackoverflow using lda. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 295–299. IEEE.

Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*.

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, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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

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

Daniel Gillick, Sayali Kulkarni, Larry Lansing, Alessandro Presta, Jason Baldridge, Eugene Ie, and Diego Garcia-Olano. 2019. Learning dense representations for entity retrieval. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 528–537, Hong Kong, China. Association for Computational Linguistics.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2019. Coupling retrieval and meta-learning for context-dependent semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 855–866, Florence, Italy. Association for Computational Linguistics.

Ruiqi Guo, Sanjiv Kumar, Krzysztof Choromanski, and David Simcha. 2016. Quantization based fast inner product search. In *Artificial Intelligence and Statistics*, pages 482–490. PMLR.

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930, Brussels, Belgium. Association for Computational Linguistics.

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*, page 200–210, New York, NY, USA. Association for Computing Machinery.

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-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization.

Michael Hucka and Matthew J Graham. 2018. Software search is not a science, even among scientists: A survey of how scientists and engineers find software. *Journal of Systems and Software*, 141:171–191.

Samuel Humeau, Kurt Shuster, Marie-Anne Lachaux, and Jason Weston. 2020. Poly-encoders: Architectures and pre-training strategies for fast and accurate multi-sentence scoring. In *International Conference on Learning Representations*.

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

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. 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)*, pages 5426–5435, Hong Kong, China. Association for Computational Linguistics.

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 (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Gautier Izacard and Edouard Grave. 2021. Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 874–880, Online. Association for Computational Linguistics.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, page 795–806. IEEE Press.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. 2013. What help do developers seek, when and how? In *2013 20th working conference on reverse engineering (WCRE)*, pages 142–151. IEEE.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, pages 5229–5236.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid {gnn}. In *International Conference on Learning Representations*.

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

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

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

CD Manning, P Raghavan, and H Schütze. 2008. Xml retrieval. In *Introduction to Information Retrieval*. Cambridze University Press.

Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming q&a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building language models for text with named entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics.

Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 465–475.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.

Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 191–201.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, pages 1–8. IEEE.

Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):1–25.

Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–45.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, volume 27, pages 3104–3112. Curran Associates, Inc.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 6563–6573. Curran Associates, Inc.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural

language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, Online. Association for Computational Linguistics.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-ide code generation from natural language: Promise and challenges. *arXiv preprint arXiv:2101.11149*.

Wen-tau Yih, Kristina Toutanova, John C Platt, and Christopher Meek. 2011. Learning discriminative projections for text similarity measures. In *Proceedings of the fifteenth conference on computational natural language learning*, pages 247–256.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE.

Supplementary Material: Appendices

A Qualitative Example

In Figure 11, we show an example of generated code by a baseline and different modules of REDCODER. The input summary asks to write a code (in Java) to get a MuxerStream given a position.

We show two of the corresponding retrieved code, their summaries (for *bimodal* instances), generated code of PLBART, REDCODER, and REDCODER-EXT. As can be seen, PLBART generates a basic but relevant code; both retrieved code (rank-1 and rank-3) contains the statements with variable cPtr one of them is of MuxerStream class, and another is from DeMuxerStream class. REDCODER generates a somewhat correct code of MuxerStream class and it takes the position argument too. Seemingly, while fusing the retrieved code, we suspect that as the tentative function name MuxerStream mentioned in the input summary does not match the function name DeMuxerStream of the rank-3 retrieved code, it only adapts one line containing cPtr from rank-3 retrieved code (line #3) and takes the rests including the function definition (i.e., line #1) from the rank-1 retrieved code. Now when REDCODER-EXT is allowed to leverage the summaries of the retrieved code, it can match the summary of the rank-3 retrieved code with the input, and that is why it produces the MuxerStream class object but with the throw exceptions from the rank-3 retrieved code.

B Performance Difference of PLBART on CodeXGLUE and Concode

Concode is a relatively easier dataset for code generation and retrieval due to several pre-processing steps taken by its authors. Along with additional contexts (environment variables and methods) in the input summary, Concode artifacts the target code by replacing the specific variable names with generic tokens.

```
1 void function(Element arg0,
2   Formula arg1) {
3   arg0.addElement(
4     "concode_string").setText(
5       arg1.getText());
6 }
```

Therefore, we suspect that due to this, PLBART achieves good EM score for Concode but not for the generation of real code in CodeXGLUE.

Analogously for the retrieval models, code retrieved by BM25 have also a large word overlapping with the targets in Concode in contrast to CodeXGLUE (1st row in Table 2 and 3). Consequently, BM25 retrieval boosts PLBART (i.e., BM25 + PLBART) more in Concode than that in CodeXGLUE (3rd row for the bottom in Table 2 and 3). Overall, we anticipate all these skewness in model performances are due to the dataset characteristics.

Dataset	Lang.	Task	Retrieval Database			|Size|	|Nonparallel|
			CSNet	CCSD	Concode		
CodeXGLUE	Python	Gen.	✓	✗	✗	1.2M	504K
		Sum.	✓	✓	✗	1.1M	833K
	Java	Gen.	✓	✗	✗	1.6M	543K
		Sum.	✓	✓	✗	1.1M	903K
Concode	Java	Gen.	✗	✗	✓	104K	0

Table 8: Retrieval database statistics. "Size" refers to both of parallel and nonparallel code or summaries. As Concode has a different data format, we only retrieve from itself. Nonparallel means the retrieval candidates are only code (for code gen.) and only summaries (for code sum.). CSNet (CodeSearchNet), CCSD refer to Husain et al. (2019) and Liu et al. (2021).

code retrieval	target present in retrieval	summary retrieval	CodeXGLUE (Java)			CodeXGLUE (Python)		
			BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU
✗	✗	✗	10.1	0.0	14.96	4.89	0.0	12.01
✓	✗	✗	26.92	8.95	31.15	22.74	8.88	28.93
		✓	**28.98**	**10.21**	**33.18**	**24.43**	**9.61**	**30.21**
	✓	✗	36.33	29.41	41.38	32.14	27.48	38.02
		✓	**42.82**	**36.99**	**47.25**	**38.87**	**34.51**	**43.78**

Table 9: Ablation results on source code generation using the retrieved code and its summary together when the reference target code is absent and present in the retrieval database respectively.

Methods	CodeXGLUE-Python		CodeXGLUE-Java	
	BLEU-4	ROUGE-L	BLEU-4	ROUGE-L
SCODE-R	46.6	53.8	48.0	55.7
REDCODER	47.0	55.4	**50.4**	**58.8**
REDCODER-EXT	**47.1**	**55.5**	**50.4**	58.7

Table 10: Evaluation results of code summarization keeping the target summary in the retrieval database.

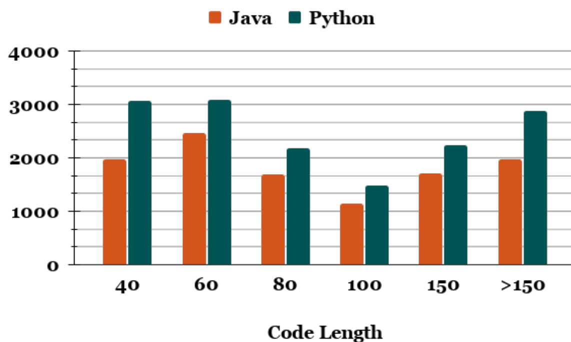

Figure 9: #Code per target length.

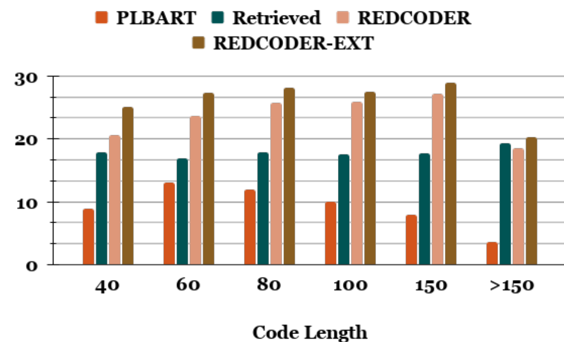

Figure 10: BLEU vs target len. (Java)

Input summary: Get the MuxerStream at the given position.

PLBART Prediction **[BLEU: 0.1439]**

```
1 public MuxerStream getMuxerStream (int position) {
2   if (muxerStream == null) {
3     muxerStream = new MuxerStream (this, position)
4   }
5   return muxerStream;
6 }
```

SCODER Retrieval (Rank-1) **[BLEU: 0.4298]**

```
1 public Muxer getMuxer () {
2   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this);
3   return (cPtr == 0) ? null : new Muxer (cPtr, false);
4 }
```

Paired summary: Get the Muxer that this stream belongs to

SCODER Retrieval (Rank-3) **[BLEU: 0.6740]**

```
1 public DemuxerStream getStream (int streamIndex)
2   throws java.lang.InterruptedException, java.io.IOException {
3   long cPtr = VideoJNI.Demuxer_getStream(swigCPtr, this, streamIndex);
4   return (cPtr == 0) ? null : new DemuxerStream (cPtr, false);
5 }
```

REDCODER Prediction **[BLEU: 0.6170]**

```
1 public MuxerStream getMuxerStream (int position) {
2   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this, position);
3   return (cPtr == 0) ? null : new MuxerStream (cPtr, false);
4 }
```

REDCODER-ext Prediction **[BLEU: 0.8062]**

```
1 public MuxerStream getMuxer (int streamIndex)
2   throws java.lang.InterruptedException, java.io.IOException {
3   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this, streamIndex);
4   return (cPtr == 0) ? null : new MuxerStream (cPtr, false);
5 }
```

Reference (Gold Output)

```
1 public MuxerStream getMuxer (int streamIndex)
2   throws java.lang.InterruptedException, java.io.IOException {
3   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this, streamIndex);
4   return (cPtr == 0) ? null : new MuxerStream (cPtr, false);
5 }
```

Figure 11: A qualitative example to show the effectiveness of retrieval-augmented generation as proposed in REDCODER framework