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 bi-modal 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 Retrieval augmentED CODE generation and summarization 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
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.

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., doctstrings, 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: $Y_1, Y_2, \ldots, Y_k \in Y_R$. Then the input sequence $x$ is augmented with the retrieved sequences to form $x' = x \oplus Y_1 \oplus Y_2 \ldots \oplus Y_k$, 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 $\text{sim}(p, q) = Q(q)^T \cdot P(p)$. Given a query $q$, a positive (relevant) passage $p^+$, and a set of $n$ irrelevant passages $p_i$, DPR optimizes the classification loss:

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

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.

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
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 $p_{gen}(y|x')$.

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.

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

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.

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

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
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeXGLUE (Lu et al., 2021)</td>
<td>✓</td>
<td>✓</td>
<td>Java</td>
<td>164,923</td>
<td>5,183</td>
<td>10,955</td>
<td>97</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Python</td>
<td>251,820</td>
<td>13,914</td>
<td>14,918</td>
<td>99</td>
<td>14</td>
</tr>
<tr>
<td>Concode (Iyer et al., 2018)</td>
<td>✓</td>
<td>×</td>
<td>Java</td>
<td>100,000</td>
<td>2,000</td>
<td>2,000</td>
<td>27</td>
<td>72</td>
</tr>
</tbody>
</table>

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.

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 GraphCodeBERT 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 $2 \times 10^{-5}$. 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.
### Table 2: Results on code generation on CodeXGLUE (Lu et al., 2021).

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

### Table 3: Code generation results on Concode dataset.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Java EM</th>
<th>BLEU</th>
<th>CodeBLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieval based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM25</td>
<td>0.00</td>
<td>20.3</td>
<td>23.7</td>
</tr>
<tr>
<td>SCODE-R</td>
<td>0.00</td>
<td>32.6</td>
<td>36.5</td>
</tr>
<tr>
<td>Generative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq2Seq</td>
<td>3.1</td>
<td>21.3</td>
<td>26.4</td>
</tr>
<tr>
<td>Guo et al. (2019)</td>
<td>10.1</td>
<td>24.4</td>
<td>29.5</td>
</tr>
<tr>
<td>Iyer et al. (2019)</td>
<td>12.2</td>
<td>26.6</td>
<td>-</td>
</tr>
<tr>
<td>GPT-2</td>
<td>17.4</td>
<td>25.4</td>
<td>29.7</td>
</tr>
<tr>
<td>CodeGPT-2</td>
<td>18.3</td>
<td>28.7</td>
<td>32.7</td>
</tr>
<tr>
<td>CodeGPT-adapted</td>
<td>20.1</td>
<td>32.8</td>
<td>36.0</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>18.0</td>
<td>28.7</td>
<td>31.4</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>18.7</td>
<td>33.4</td>
<td>35.9</td>
</tr>
<tr>
<td>PLBART</td>
<td>18.6</td>
<td>36.7</td>
<td>38.5</td>
</tr>
<tr>
<td>Retrieval augmented</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM25 + PLBART</td>
<td>21.4</td>
<td>40.2</td>
<td>41.8</td>
</tr>
<tr>
<td>REDCODER</td>
<td>23.4</td>
<td>41.6</td>
<td>43.4</td>
</tr>
<tr>
<td>REDCODER-EXT</td>
<td>23.3</td>
<td>42.5</td>
<td>43.4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Methods</th>
<th>Python EM</th>
<th>BLEU</th>
<th>CodeBLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieval based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM25</td>
<td>1.92</td>
<td>1.82</td>
<td></td>
</tr>
<tr>
<td>SCODE-R</td>
<td>14.98</td>
<td>15.87</td>
<td></td>
</tr>
<tr>
<td>Generative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq2Seq</td>
<td>15.93</td>
<td>15.09</td>
<td></td>
</tr>
<tr>
<td>Transformer</td>
<td>15.81</td>
<td>16.26</td>
<td></td>
</tr>
<tr>
<td>RoBERTa</td>
<td>18.14</td>
<td>16.47</td>
<td></td>
</tr>
<tr>
<td>CodeBERT</td>
<td>19.06</td>
<td>17.65</td>
<td></td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>17.98</td>
<td>17.85</td>
<td></td>
</tr>
<tr>
<td>PLBART</td>
<td>19.30</td>
<td>18.45</td>
<td></td>
</tr>
<tr>
<td>Retrieval augmented</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM25 + PLBART</td>
<td>19.57</td>
<td>19.71</td>
<td></td>
</tr>
<tr>
<td>REDCODER</td>
<td>21.01</td>
<td>22.94</td>
<td></td>
</tr>
<tr>
<td>REDCODER-EXT</td>
<td>20.91</td>
<td>22.95</td>
<td></td>
</tr>
</tbody>
</table>

### 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 pre-trained 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 pre-trained 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.
Table 5: Results on code generation keeping the target code in the retrieval database.

<table>
<thead>
<tr>
<th>Methods</th>
<th>CodeXGLUE (Java)</th>
<th>CodeXGLUE (Python)</th>
<th>Concode (Java)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLEU</td>
<td>EM</td>
<td>CodeBLEU</td>
</tr>
<tr>
<td>SCODE-R</td>
<td>36.6</td>
<td>21.0</td>
<td>37.9</td>
</tr>
<tr>
<td>REDCODER</td>
<td>36.3</td>
<td>29.4</td>
<td>41.4</td>
</tr>
<tr>
<td>REDCODER-EXT</td>
<td>42.8</td>
<td>37.0</td>
<td>47.3</td>
</tr>
</tbody>
</table>

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.

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.

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

<table>
<thead>
<tr>
<th>Settings</th>
<th>Methods</th>
<th>Python</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-Encoder</td>
<td>RoBERTa</td>
<td>0.587</td>
<td>0.599</td>
</tr>
<tr>
<td></td>
<td>RoBERTa (code)</td>
<td>0.610</td>
<td>0.620</td>
</tr>
<tr>
<td></td>
<td>CodeBERT</td>
<td>0.672</td>
<td>0.676</td>
</tr>
<tr>
<td></td>
<td>GraphCodeBERT</td>
<td>0.692</td>
<td>0.691</td>
</tr>
<tr>
<td>Bi-Encoder</td>
<td>DPR</td>
<td>0.093</td>
<td>0.064</td>
</tr>
<tr>
<td></td>
<td>DPR (code)</td>
<td>0.398</td>
<td>0.462</td>
</tr>
<tr>
<td></td>
<td>SCODE-R</td>
<td>0.690</td>
<td>0.686</td>
</tr>
</tbody>
</table>

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.
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
<table>
<thead>
<tr>
<th>Model</th>
<th>Human Evaluation</th>
<th>Automatic Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Similarity</td>
<td>Relevance</td>
</tr>
<tr>
<td>SCODE-R</td>
<td>2.09</td>
<td>3.00</td>
</tr>
<tr>
<td>REDCODER</td>
<td>2.06</td>
<td>2.94</td>
</tr>
</tbody>
</table>

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 tokens. Other approaches leverage the structural properties of code using Tree based model (Shido et al., 2019; Harer et al., 2019; Hu et al., 2018; 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.

**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


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.


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.


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


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 adopts 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.  

```java
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.
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).

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

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.

<table>
<thead>
<tr>
<th>code retrieval</th>
<th>target present in retrieval</th>
<th>summary retrieval</th>
<th>CodeXGLUE (Java)</th>
<th>CodeXGLUE (Python)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>BLEU  EM CodeBLEU</td>
<td>BLEU  EM CodeBLEU</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>10.1 0.0 14.96</td>
<td>4.89 0.0 12.01</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>26.92 8.95 31.15</td>
<td>22.74 8.88 28.93</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>28.98 10.21 33.18</td>
<td>24.43 9.61 30.21</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>36.33 29.41 41.38</td>
<td>32.14 27.48 38.02</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>42.82 36.99 47.25</td>
<td>38.87 34.51 43.78</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Methods</th>
<th>CodeXGLUE-Python BLEU-4 ROUGE-L</th>
<th>CodeXGLUE-Java BLEU-4 ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCODE-R</td>
<td>46.6 53.8</td>
<td>48.0 55.7</td>
</tr>
<tr>
<td>REDCODER</td>
<td>47.0 55.4</td>
<td>50.4 58.8</td>
</tr>
<tr>
<td>REDCODER-EXT</td>
<td>47.1 55.5</td>
<td>50.4 58.7</td>
</tr>
</tbody>
</table>

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]

```java
public MuxerStream getMuxerStream (int position) {
    if (muxerStream == null) {
        muxerStream = new MuxerStream (this, position)
    }
    return muxerStream;
}
```

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

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

Paired summary: Get the Muxer that this stream belongs to

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

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

REDCODER Prediction [BLEU: 0.6170]

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

REDCODER-ext Prediction [BLEU: 0.8062]

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

Reference (Gold Output)

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

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