# ISAGWR: Iterative Self-augmented Generation with Reviewer

Anonymous ACL submission


## Abstract

Code generation plays a vital role in software development and has gained widespread attention. Some researchers prone to employ Retrieval-augmented Generation (RAG) and achieved impressive results. However, these methods ignore the real-world iterative code refining process as they solely reuse external retrieved code. To tackle this limitation, we propose a self-augmented generation method SAG, which iteratively constructs augmented datasets using Generator's output. The Generator refine its own code with the help of the datasets. Furthermore, inspired by the real-world role of programmer reviewers, we propose an iterative generator-review architectural method ISAGWR based on the SAG datasets. As its core, a Reviewer module is employed to detect and handle errors. These feedback are then feed into Generator for better coding output. We conduct extensive experiments on five benchmarks, and the results show that ISAGWR significantly surpasses all the baselines. The results also indicate that the SAG datasets and the Reviewer module respectively provides valuable insight to perform automatic data augmentation and integrate self-correct ability into a unified framework. [1]


## 1 Introduction

Programmers often make considerable efforts to manually write code. Code generation (Yin and Neubig, 2018; Sun et al., 2019; Wang et al., 2021, 2023) aims to automate this process and generate programming languages that meet specific natural language requirements.

Inspired by the code reuse behavior of programmers, some research (Hayati et al., 2018; Parvez et al., 2021; Lu et al., 2022; Shi et al., 2022; Li et al., 2023) have incorporated retrieval (Robertson and Zaragoza, 2009; Karpukhin et al., 2020) to enhance code generation, achieving promising results by leveraging existing code snippets. These Retrieval Augmented Generation methods (RAG) teach models how to utilize relevant retrieved code (see Figure 1 (a)). Typically, they adopt an *data augmentation* technique (Shorten and Khoshgoftaar, 2019), which concatenates the retrieved code with the input requirements to create an augmented training dataset (Song et al., 2016).

During the coding process, programmers not only refer to external code, but also iteratively refine their own code. However, the latter behavior is ignored by existing RAG method. To remedy this issue, we propose a novel Self-augmented Generation method, namely SAG, which automates the iterative coding refine process (see Figure 1 (b)). SAG leverages the Generator's output to construct augmented datasets at each epoch. The SAG datasets are then fed into the Generator to improve its training effectiveness. As shown in Case 1 from Figure 14 (a), compared to RAG, SAG can fix some obvious errors through iterative refinement.

SAG also exits some limitations. As illustrated in Case 2 form Figure 14 (b), the code error "Mana.Colorless Mana(1)" repeatedly occurs in each epoch. This issue deserves our attention and should be addressed. In the real world, such repeated errors are often observed by a role of programmer reviewers, whose responsibility is to identify code errors and improve code quality. Enlightened by this, we design a novel Review model and add it into SAG. This forms our **I**terative **S**elf-**a**ugmented **G**eneration **w**ith **R**eviewer method, ISAGWR (see Figure 1 (c)). This method comprises two key modules: Reviewer and Generator. The former automates the code review process. The latter iteratively generate higher-quality code with the help of reviewed code provided by the Reviewer. As illustrated in Case 3 from Figure 14 (c), the aforementioned repeated mistakes are successful identified and masked by the Reviewer module,

[1]We will release the code after the double-blind review period.

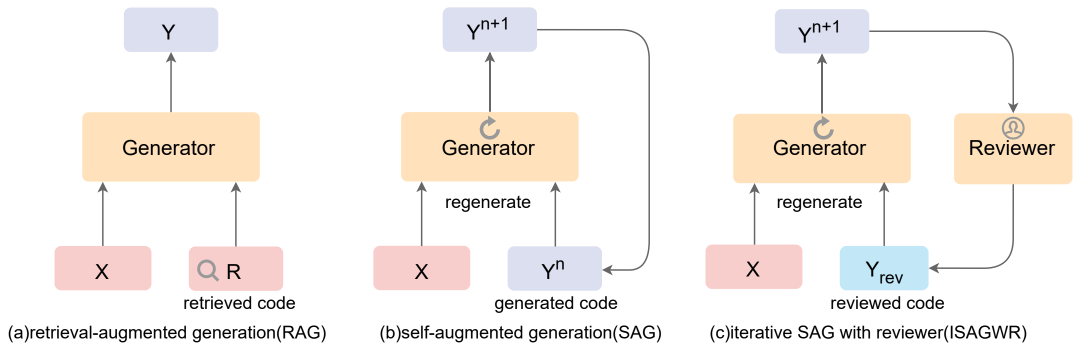


Figure 1: **Comparison of three generation methods.** **(a)** RAG mimics programmer's code reuse behavior. **(b)** SAG emphasizes iterative behavior of a hard-working programmer repeatedly optimizing their own code after refering the external code. **(c)** ISAGWR imitates the real-world code reviewer to check and ensure that the generated code meets the requirements. A Reviewer module is added in SAG to build ISAGWR.

then promoting the Generator to output correct code at the 2nd epoch. Our contributions of this work are summarized as follows:

- We propose a code generation method SAG which automatically constructs augmented datasets with Generator's output at each epoch. Due to its simplicity and no need for extra data, this method can be easily applied in other scenarios, such as RAG.
- Based on SAG, We propose ISAGWR which incorporates a Reviewer module to facilitate iterative generation from detecting and masking errors in Generator's output. In this way, harmful errors in current code cannot influence subsequent code generation.
- All of existing code translation datasests provide only one reference answer for the same task. To alleviate this limitation, we release a new code translation dataset AtCoder, which collected multiple high-quality coding solutions for the same task from AtCoder website.
- We conduct extensive experiments on five benchmarks. The results show that ISAGWR outperforms all the baselines. Further study demonstrate the effectiveness of SAG datasets and the Reviewer module for benefiting high-quality code generation.

## 2 Related Work

**Retrieval-augmented generation.** Inspired by programmers' code reuse behaviors, several studies have explored the RAG in code generation (Li et al., 2023), code summarization (Wei, 2019; Parvez et al., 2021; Shi et al., 2022), code completion (Lu et al., 2022; Zhang et al., 2023). In these fields, there exists a challenge: the retrieved data might be irrelevant. How to ensure it does not affect the model generation. Some research (Shi et al., 2022) such as SKCODER (Li et al., 2023) introduce Skeleton-based (Cai et al., 2018; Wu et al., 2018; Wei, 2019; Zan et al., 2022) approach to extract relevant part from the retrieved code. The SAG data augmentation proposed in this work contribute to solving this challenge, which we will discuss in subsequent sections.

**Iterative Generation.** Like human beings, language models do not always generate the best code through the first try. Some methods iteratively generate revise feedback to help the models optimize the outputs (Madaan et al., 2023), and some other methods need additional reviewer datasets to train a supervised reviewer (Schick et al., 2022; Welleck et al., 2022). To better reuse the generated code, some works just generate in a iterative style without reviewer-like structure. (Zhang et al., 2023).

**Pre-trained Model.** Pre-trained models are trained on data of code and fine-tuned on code generation tasks specifically to enhance code generation performance. Typically, code-based LLMs can be categorized into three architectures. Encoder-only model is mostly used in code comprehension like masked language modeling or code retrieving, including CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), etc. Decoder-only model is mainly used to predict following tokens based on the input context like GPT series which including CodeGPT (Lu et al., 2021)based on GPT-2 (Radford et al., 2019). Based on the fine-tuned GPT-Neo (Black et al., 2021), PyCodeGPT (Zan et al., 2022) generates codes by a user-defined generated sketch.

Encoder-decoder model can support both code comprehension and generation tasks including CodeT5 (Wang et al., 2021), or introduce text-code matching and contrastive learning to learn rich contextual representations like CodeT5+ (Wang et al., 2023), PLBART (Ahmad et al., 2021), SPT-Code (Niu et al., 2022), etc.

## 3 Self-Augumented Generation (SAG)

The training process of SAG is illustrated in Figure 2. Implementation details are given as follows:

**Static RAG Dataset.** We retain the static augmented training dataset used by RAG because it can help Generator convergence. RAG dataset concatenates the retrieved code $R$ and the input requirement $X$, which is denoted as $X + R \rightarrow Y$.

**Dynamic SAG Datasets.** SAG Datasets are dynamically updated using the Generator's output at each training epoch. Specifically, a queue $Q$ is designed to maintain the datasets. At the $n$-th epoch, the dataset concatenates the generated code $Y^n$ and $X$, which is denoted as $X + Y^n \rightarrow Y$. Here, $Y^n = \mathbf{G}(X)$. Then we push the dataset into $Q$. Note that the size of $Q$ is limited to $m$ to ensure that each epoch's re-constructed training data has equal chance to be trained.

**Generator.** Existing sequence to sequence models, such as CodeT5 and CodeT5+ (Wang et al., 2021, 2023) can be employed as SAG's Generator.

## 4 Iterative SAG with Reviewer(ISAGWR)

### 4.1 Background and Overview

Real-world Reviewers find code errors, handle them and utilize the reviewed code for next coding iteration. To mimic this role, we design a Reviewer module and integrate it into SAG. This forms ISAGWR.

ISAGWR includes two modules: Reviewer and Generator (Figure 3). Technically, Generator $\mathbf{G}$ outputs improved code $Y_{gen}^{(n+1)}$ based on reviewed code $Y_{rev}^{(n)}$ from the $n$-th iteration, which is denoted as $\mathbf{G} : (Y_{rev}^{(n)}, X) \rightarrow Y_{gen}^{(n+1)}$. Then, Reviewer $\mathbf{R}$ identifies potential errors in $Y_{gen}^{(n+1)}$ with $X$, and mask them to output $Y_{rev}^{(n+1)}$. This is denoted as $\mathbf{R} : (X, Y_{gen}^{(n+1)}) \rightarrow Y_{rev}^{(n+1)}$. By iterating in such a loop, the Reviewer module plays the role of a real-world code Reviewer, promoting ISAGWR to achieve high-quality code generation.

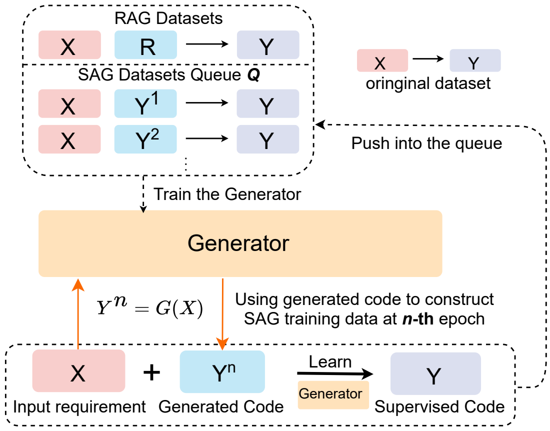


Figure 2: **Training process of SAG.** Compared to RAG dataset, SAG datasets dynamically updated using the output from the Generator. In each training epoch, both kinds of datasets are used to train the Generator. Note that $Y^n$ from different epochs have different code quality levels, ensuring diverse patterns in SAG datasets. Therefore, the Generator can learn more effectively by utilizing external code and its generated code.

### 4.2 Reviewer

Reviewer is the core module of ISAGWR and needs to be meticulously designed. We mainly face two challenges. First, how to detect and handle potential errors so as to assist the Generator in outputting better code? Second, how to automatically collect a high-quality dataset for the Reviewer?

Regarding the first challenge, we enable the Reviewer to calculate the validness probability for each token in the code. When the validness probability is less than a threshold $t$, the token will be judged as an error. Technically, $Y_{gen}$ is a list of token $(y_1, y_2, ..., y_n)$, to review whether $Y_{gen}$ meets $X$, we concatenate them into $X \oplus Y_{gen}$, and then fed it into the Encoder as follows:

$$[X' \oplus Y'_{gen}] = Encoder(X \oplus Y_{gen}) \quad (1)$$

$$Y'_{gen} = (y'_1, y'_2, ..., y'_n) \quad (2)$$

Both $Y'_{gen}$ and $X'$ are a list of vectors. Through Encoder, each token $y_i$ is transformed into a 256-dim vector $y'_i$. Then each $y'_i$ is input into the 256x1 linear layer with a sigmoid function:

$$p_i = sigmoid(W_m y'_i + b_m) \quad (3)$$

where $W_m$ and $b_m$ are learnable parameters, the output $p_i$ is the obtained validness probability for token $y_i$. Following that, we compare $p_i$ with $t$.

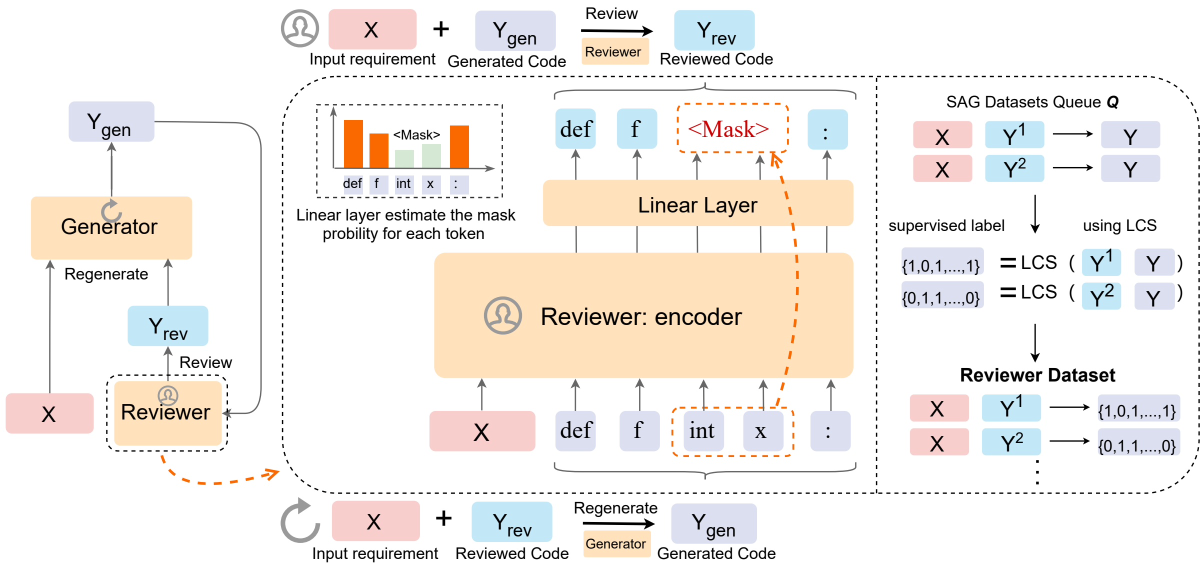


Figure 3: **The Zoom-in view of REVIEWER from ISAGWR.** The Reviewer contains a encoder and a linear layer which aims to review the code generated by Generator. For example, here is a generated code $Y_{gen}$ : "$def\ f\ int\ x\ :$". The encoder encodes it as "$X \oplus def\ f\ int\ x\ :$". Each encoded vector is then fed into the linear layer to output reviewed code $Y_{rev}$ in which potential errors "$int\ x$" are detected and masked. The right side of the figure illustrates how to convert SAG datasets into Reviewer dataset.

If the former has a smaller value, $y_i$ will be marked.

After identifying code errors, the next step is to handle them. We choose to mask them. Specifically, both a single marked token and consecutive marked tokens are replaced by a Mask token. Through these mask operations, we ultimately obtain the reviewed code $Y_{rev}$, which are then used to support Generator training.

Regarding the second challenge, we construct the Reviewer dataset based on the SAG datasets. Specifically, we employ the Longest Common Sub-sequence algorithm (LCS), which algorithm is shown in Appendix B, to annotate those Mask tokens as binary-classification supervised labels. Here, LCS helps the Reviewer extract the longest common tokens between the generated code and its corresponding supervised code. Common tokens are labeled as 1, while the remains are labeled as 0. Therefore the constructed Reviewer datasets have the form as $X + Y^n \rightarrow D$, where $D$ is a list of supervised label.

We train the Reviewer by minimizing the following loss function:

$$\mathcal{L}_{rev} = -\sum_{i=1}^{s}\sum_{j=1}^{h}[\mathcal{I}(D_{ij}=1)\cdot log(P_{ij}) + \mathcal{I}(D_{ij}=0)\cdot log(1-P_{ij})] \tag{4}$$

where $s$ denotes the size of the Reviewer dataset, $h$ denotes the length of $Y_{gen}$ which need to review. $\mathcal{I}$ is an indicator function that outputs 1 when the condition is true, 0 otherwise. $P$ denotes the validness probability of the Reviewer and $D$ denotes the supervised label.

### 4.3 Generator and Complete Training Process

Generator adopted in ISAGWR is the same as SAG. As introduced in subsection 4.1, we regenerate the code $Y_{gen}^{(n+1)}$ as follow.

$$Y_{gen}^{(n+1)} = \mathbf{G}(X \oplus Y_{rev}^{(n)}) \tag{5}$$

The complete training process of ISAGWR includes Generator training and Reviewer training, which are integrally given in Algorithm 1.

## 5 Experiments

Although we focus on code generation when describing ISAGWR, it can be easily applied to other generation scenarios, such as code translation. Accordingly, the Reviewer module reviews translated code. In this regard, we evaluate ISAGWR on code generation and translation tasks.

### 5.1 Datasets

We adopt three public datasets and construct a new AtCoder dataset for the experiments. The statistics of the datasets are given in Table 1.

| Dataset | Training | Validation | Test |
|---|---|---|---|
| Code Generation | | | |
| Hearthstone | 533 | 66 | 66 |
| Magic | 11,969 | 664 | 664 |
| AixBench-L | 190,000 | 10,000 | 175 |
| Code Translation | | | |
| CodeXGLUE(trans) | 10,300 | 500 | 1,000 |
| AtCoder | 564 | 36 | 57 |

Table 1: **Statistics of the Datasets.**

**HearthStone and Magic** (Ling et al., 2016). Both datasets automatically generate code for game cards. Each individual sample within these datasets comprises a semi-structural description accompanied by a human-authored program.
**AixBench-L** (Li et al., 2023). It is an augmented function-level code generation benchmark based on AixBench, containing preprocessed popular Java projects without test data from GitHub.
**CodeXGLUE** (Lu et al., 2021). This dataset collects both Java and C# codes from several public repos, including Lucene, POI, JGit and Antlr.
**AtCoder Dataset**. We collect various versions of correct code in different languages for the same task from AtCoder. This is the first dataset to provide multiple reference answers for the same coding task. Details are given in Appendix C.

### 5.2 Evaluation Metrics

We employ Exact match (EM), BLEU4, CodeBLEU and Pass@1 as the evaluation metrics. Higher values suggest higher performance. More details of these metrics are given in Appendix D.
**EM** assesses the accuracy of a model's output by measuring whether it exactly matches a reference or expected answer.
**BLEU4** (Papineni et al., 2002) measures the similarity between a machine-generated text and one or more reference texts in the context of tasks.
**CodeBLEU** score (Ren et al., 2020) is a variant of BLEU4, which considers syntactic and semantic matches based on the code structure.
**Pass@1** is an unit test metric which calculates the percentage of generated code that can pass the test. Value 1 stands for only 1 version of code is generated for each task.

### 5.3 Baselines

We compare ISAGWR with CodeT5 (Wang et al., 2021), CodeT5+ (Wang et al., 2023), SkCoder (Li et al., 2023), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020) and CodeGPT (Lu et al., 2021). In addition, RNN and Transformer are also selected as the baselines.

**Algorithm 1** The training process of ISAGWR

**Require:**
$\#N_{gen}$ : The training epoch of Generator
$\#N_{rev}$ : The training epoch of Reviewer
$\#queue$ : SAG datasets queue for Generator
$\#\mathbf{G}$ : Generator of the ISAGWR
$\#\mathbf{R}$ : Reviewer of the ISAGWR
$\#D$ : Original Tranining dataset
$\#D_{rev}$ : Tranining dataset of the Reviewer
**Ensure:** **G**, **R**

**for** $i$ in $range(N_{\text{gen}})$ **do**
    # Train the Generator
    **G**. train ( $queue$, $D$ )
    # SAG datasets queue for Generator
    $queue$. enqueue ($\{X : \mathbf{G}(X)\} \to Y$)
    **If** size ($queue$) $> M$ :
        $queue$. dequeue ()
    # SAG datasets for Reviewer
    $D_{rev}$. insert ($\{X : \mathbf{G}(X)\} \to Y$)
**end for**
# Using LCS to tansform the SAG dataset
$D_{rev}$= Transform($D_{rev}$)
**for** $i$ in $range(N_{\text{rev}})$ **do**
    # Train the Reviewer
    **R**. train ( $D_{rev}$ )
**end for**
**return** **G**, **R**

### 5.4 Retrieval

The retrieval adopted in our experiments is built upon the DPR architecture (Karpukhin et al., 2020). We use the training dataset as retrieval database, and fintune the retrieval with Moco-based *text-code contrastive learning* (He et al., 2019; Wang et al., 2023; Li et al., 2021). Please refer to Appendix A for the details, .

### 5.5 Experiment-1: Effectiveness of SAG

In the first set of experiments, we compare SAG with RAG to verify its effectiveness. Then, we conduct further explorations to figure out whether this improvement is achieved through its iterative process or through its data augmentation method. Specifically, for the latter, we try to answer the question "do the SAG datasets essentially improve code generation?". For fair comparisons, we restrict SAG from performing iterative generation

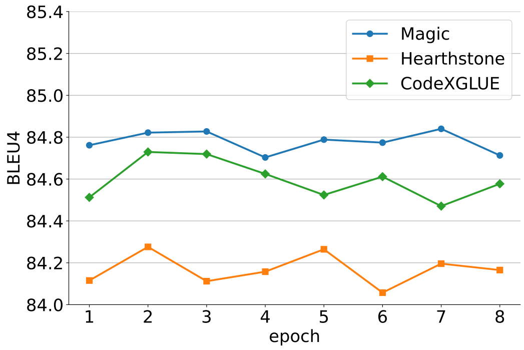


Figure 4: **Results of SAG in terms of BLEU4.**

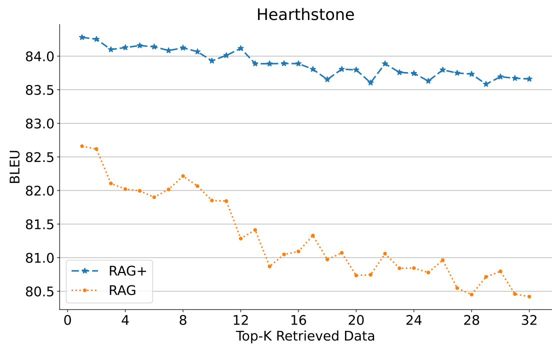


Figure 5: The result of RAG and RAG$^+$ evaluated by metric BLEU4 on Hearthstone datasets.

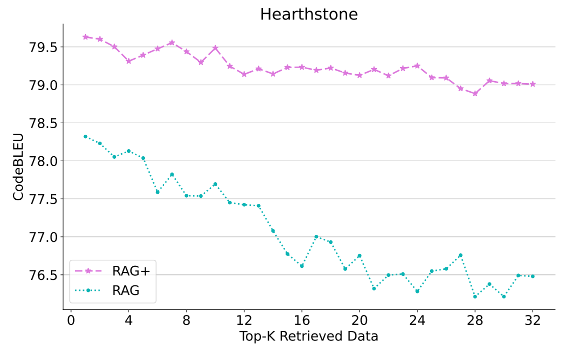


Figure 6: The result of RAG and RAG$^+$ evaluated by metric CodeBLEU on Hearthstone datasets.

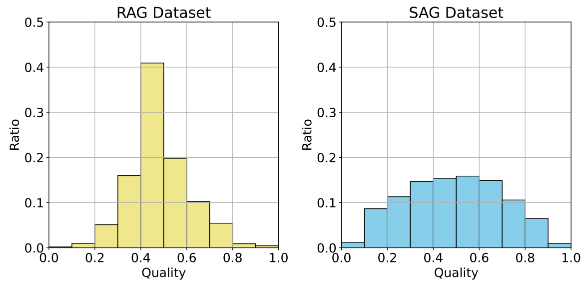


Figure 7: **Statistic for two types datasets.** Quality is the CodeBLEU score between augmented code and supervised code, which is divide averagely in 10 categories. Ratio is the probability density for the number of sample in each category.

and employ RAG's retrieved dataset. This forms a new generation method RAG$^+$. Therefore, the key to the answer is to compare RAG with RAG$^+$.

### 5.5.1 Experiment-1 Setup

The training methods of RAG, RAG$^+$, and SAG follow previous research. We train RAG$^+$ 50 epochs with batch size 16 and learning rate 5e-5 on Hearthstone. On the test dataset, we generate outputs using retrieved code from Top-1 to Top-32. Here Top-$k$ is obtained from ranking the calculated similarities between the retrieved code and the input requirements. A smaller $k$ means better retrieved results. Note that to perform a more accurate ranking, we use CodeBLEU to recalculate the similarity between retrieved code and the supervised code, and then re-rank them. For fairness, we take the average value of multiple experiments.

### 5.5.2 Experiment-1 Results

**SAG vs RAG.** As illustrated in Table 2, SAG outperforms RAG for all the metrics on the three datasets. This demonstrate that taking both external retrieved code and Generator's output code into account promotes code generation.

**Iterative process of SAG.** From Figure 4, we find that the BLEU4 scores of SAG fluctuated in a small range from 1st epoch to 8th epoch. A possible reason is that the Generator has difficulties to identify its own code errors, which limits the improvements at each epoch. However, this iterative process can not be overlooked. As illustrated in Case Study 1, SAG can indeed fix some obvious errors through this iterative process.

**RAG vs RAG$^+$.** Figure 5 and Figure 6 respectively shows the results of RAG and RAG$^+$ on BLEU4 and CodeBLEU metrics. We find that RAG$^+$ always significantly beats RAG. These results demonstrate that enhanced with SAG's augmented datasets, RAG$^+$ facilitate the Generator to effectively utilize external code. Another interesting finding is that as the $k$ increases, RAG$^+$ exhibits relatively stable performance, while RAG rapidly oscillates and decreases. In other words, under small-scale or low-quality retrieval code situation, RAG$^+$ can maintain much better and stable results than RAG, indicating that the SAG datasets endows RAG with robustness.

**Analysis of SAG and RAG datasets.** For further exploration, we also analyze statistics on RAG and

| Model | Hearthstone (Python) | | | Magic (Java) | | | AixBench-L (Java) |
|---|---|---|---|---|---|---|---|
| | EM | BLEU4 | CodeBLEU | EM | BLEU4 | CodeBLEU | Pass@1 |
| RNN∗ | 3.03 | 64.53 | 58.56 | 16.26 | 71.96 | 61.83 | 4.00 |
| Transformer∗ | 3.03 | 62.46 | 51.63 | 12.20 | 73.10 | 66.61 | 6.29 |
| CodeBERT∗ | 3.03 | 66.50 | 59.39 | 19.42 | 78.69 | 71.73 | 9.14 |
| GraphCodeBERT∗ | 3.03 | 66.32 | 58.87 | 27.41 | 82.33 | 74.76 | 10.86 |
| CodeGPT | 24.24 | 80.90 | 75.42 | 27.40 | 78.68 | 70.04 | 17.71 |
| CodeT5-base | 28.79 | 81.28 | 77.02 | 29.82 | 81.57 | 75.85 | 15.42 |
| SKCODER(CodeT5-base) | 31.81 | 84.12 | 79.45 | 35.39 | 85.39 | 80.62 | **20.00** |
| CodeT5+ 220M | 30.30 | 81.95 | 77.81 | 33.43 | 82.30 | 77.43 | 17.71 |
| RAG(CodeT5+ 220M) | 30.30 | 82.65 | 78.31 | 34.19 | 83.32 | 78.24 | 17.71 |
| SAG(CodeT5+ 220M) | 31.81(+5.0%) | 84.28(+2.8%) | 79.63(+2.3%) | 34.93(+4.5%) | 84.79(+3.0%) | 79.77(+3.0%) | 19.43(+9.7%) |
| ISAGWR(CodeT5-base) | 31.81(+5.0%) | 84.44(+3.0%) | 79.90(+2.7%) | 35.39(+5.9%) | 85.52(+3.9%) | 80.64(+4.1%) | 20.00(+12.9%) |
| ISAGWR(CodeT5+ 220M) | **31.81**(+5.0%) | **84.91**(+3.6%) | **80.16** (+3.0%) | **35.54**(+6.3%) | **85.80**(+4.3%) | **80.71**(+4.2%) | **20.00**(+12.9%) |

Table 2: **Results for code generation task**. Method name with "∗" indicates that its results are obtained from previous works. The "( )" next to the method name specifies the Generator. The improvement percentage compared to CodeT5+ 220M are displayed in green. Note that the last three methods output the same EM results on the Hearthstone dataset, this may attributes to the size of this dataset is too small.

| Model | CodeXGLUE(Java-to-C#) | | | AtCoder(Cpp-to-Python) | | AtCoder(Java-to-Python) | |
|---|---|---|---|---|---|---|---|
| | EM | BLEU4 | CodeBLEU | BLEU4 | CodeBLEU | BLEU4 | CodeBLEU |
| CodeBERT | 59.00 | 79.92 | 85.10 | 9.12 | 18.58 | 18.24 | 23.97 |
| CodeT5-base | 65.90 | 84.03 | 86.91 | 11.65 | 20.76 | 19.48 | 25.33 |
| CodeT5+ 220M | 66.20 | 84.25 | 87.36 | 12.83 | 21.61 | 20.89 | 26.25 |
| SAG(CodeT5+ 220M) | 67.10(+1.4%) | 85.35(+1.3%) | 88.23(+1.0%) | 13.45(+4.8%) | 22.50(+4.1%) | 21.45(+2.7%) | 27.01(+2.9%) |
| ISAGWR(CodeT5-base) | 67.00(+1.2%) | 85.27(+1.2%) | 88.31(+1.1%) | 13.52(+5.4%) | 22.54(+4.3%) | 21.15(+1.2%) | 26.70(+1.7%) |
| ISAGWR(CodeT5+ 220M) | **67.30**(+1.7%) | **85.52**(+1.5%) | **88.79**(+1.6%) | **13.88**(+8.2%) | **23.59**(+9.2%) | **22.13**(+5.9%) | **28.09**(+7.0%) |

Table 3: **Results for code translation task**. The improvement percentage compared to CodeT5+ 220M are displayed in green. The "()" next to the method name specifies the Generator.

SAG datasets, the results are shown in Figure 7. Compared to RAG dataset, SAG datasets exhibit a more uniform distribution. The reason is that RAG solely uses the Top-1 code as its augmented code, resulting in relatively homogeneous code patterns. In contrast, SAG datasets utilize Generator's output from different epochs, resulting in more diverse code patterns. This diverse characteristic ensures SAG's robustness.

In summary, we demonstrate the effectiveness and robustness of SAG for code generation. As its core, the SAG datasets are essentially helpful. Since SAG self-augments with Generators' output and no extra data is necessary, it can be easily applied to enhance existing models, such as RAG.

### 5.6 Experiment-2: Effectiveness of ISAGWR

In the second set of experiments, we compare ISAGWR with SAG, and also explore the iterative process of ISAGWR, to verify the advantanges of the Reviewer module. Then, we compare ISAGWR with other baselines to demonstrate its effctiveness.

#### 5.6.1 Experiment-2 Setup

ISAGWR trains the Generator similar as SAG. We train the Reviewer module 20 epochs with a batch size 16 and learning rate 2e-5.

#### 5.6.2 Experiment-2 Results

Table 2 and Table 3 present various metrics of baselines and our methods (SAG and ISAGWR).

**ISAGWR vs SAG.** (1) Code generation task (see Table 2), ISAGWR succeeds in all the Generator settings upon the three datasets compared to SAG; (2) Code translation task (see Table 3), ISAGWR always outperforms SAG when employing the same CodeT5+ Generator. These results not only demonstrate the superiority of ISAGWR over SAG for both generation tasks, but also verify the effectiveness of the Reviewer module adopted in ISAGWR.

**Iterative review process of ISAGWR**. ISAGWR achieves the best BLEU4 at the 3rd epoch on Magic and CodeXGLUE datasets, and at the 4th epoch on Hearthstone dataset (see Figure 8). As illustrated in Case 2, the iteration process of SAG can not identify some code errors that the Generator repeatedly outputs. This situation has largely changed since the Reviewer module is involved in ISAGWR. In each iteration, the Reviewer checks the output code of the Generator and masks error tokens, and then the Generator performs next-epoch training based on the masked code. Therefore, with the iterative collaboration between the Generator and the Reviewer, ISAGWR can better identify and handle code errors, thus improving generation performance. Case 3 also confirms this point.

**ISAGWR vs OTHER BASELINES .** ISAGWR(CodeT5+ 220M) achieves noticeable performance improvement over all baselines, showing its effectiveness in code generation and translation. We attribute this superiority to its iterative generation-review strategy. Take a close look at some interesting findings: (1) Code generation task (see Table 2). For the AixBench-L dataset, compared to CodeT5+ 220M, ISAGWR(CodeT5+ 220M) obtains the best improvements on Pass@1 (12.9%). A possible reason is that AixBench-L is a large-scale dataset, which can be used to build larger-scale SAG datasets, thereby promoting Generator and Reviewer to refine code. (2) Code translation task (see Table 3). ISAGWR shows better improvements on AtCoder compared to results on CodeXGLUE. One possible reason is that compared to AtCoder dataset, the results retrieved on CodeXGLUE have relatively lower relevance, which affects the retrieval quality.

### 5.7 Experiment-3: Effect of queue size $m$

In the third set of experiments, we vary queue size $m$ (1, 2, 3, $\infty$) of SAG datasets to explore its impact on performance. Due to limited space, we only report results on Hearthstone (see Figure 7). We find that the best performance is achieved at $m = 3$. In addition, we also explore the two extreme situations: (1) when $m = \infty$, ISAGWR performs the worst. The SAG dataset constructed at each epoch will be retained indefinitely. This will cause an issue of biasing augmented datasets, as the datasets constructed in early epochs will be trained more times than the later constructed datasets, leading to performance degradation. (2) when $m = 1$, ISAGWR is ranked as the second worst. The SAG dataset created for each epoch will be removed from the queue after one epoch. Due to insufficient training for Generator, ISAGWR fails to achieve satisfactory results. Therefore, to strike a balance between unbiased datasets and sufficient training, it's necessary to find a suitable $m$ value within a reasonable range.

## 6 Conclusion

SAG is an augmented generation method proposed in this work. Different from RAG, SAG iteratively reuse the Generator's output to build augmented datasets. This SAG datasets are then used to train the Generator for better output. Furthermore, enlightened by the real-world code reviewer role, we design a Reviewer module and integrate it into SAG, which forms an iterative generator-review architectural method ISAGWR. In each epoch, the Reviewer module identifies and eliminates error tokens based on the SAG datasets. After that, the reviewed code is feed into the Generator to ensure high-quality code generation. Extensive experiments verify that ISAGWR can effectively perform generation task with the help of Reviewer module and SAG datasets as well as outperform all the baselines. We also believe that the AtCoder Dataset we collected will facilitates relevant researches.

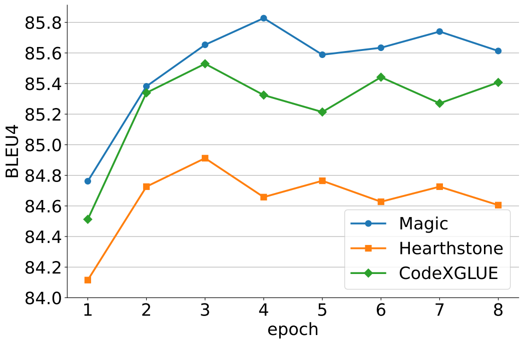


Figure 8: **Results of ISAGWR in terms of BLEU4.**

| | BLEU4 | CodeBLEU |
|---|---|---|
| ISAGWR (m=3) | **84.91** | **80.16** |
| Data Augmentation | | |
| - Fixed Size Queue (m=1) | 84.37 | 79.92 |
| - Fixed Size Queue (m=2) | 84.75 | 80.05 |
| - Fixed Size Queue (m=4) | 84.61 | 79.97 |
| - Fixed Size Queue (m=$\infty$) | 84.32 | 79.88 |

Table 4: **Results s on Hearthstone.** $m$ denotes the size of SAG datasets queue

## Limitations

The limitations of this paper are as follows:

*Time Complexity*. Constructing SAG datasets requires model generation at each epoch. Especially if the training dataset is too large, this can lead a bottleneck in terms of time complexity. In addition, iterative generation inevitably increases the cost of model inference. We will explore the trade-off between time complexity and performance in our future work.

*Limitation of the Reviewer module*. Although the proposed Reviewer module identifies and masks potential errors, it cannot correct them. We defer it as our future work.

*The size of the model's parameters*. This work focuses on validating the effectiveness of ISAGWR in small-parameter models. Due to the resource constraints, we does not investigate larger model parameters. Future research will explore this area.

## Ethical Statement

This research provides methods for generating and iteratively refining source code based on natural language descriptions. As with all AI techniques related to code, there exists potential for dual use and misuse. Our methods should only be applied to legal and ethical domains.

All datasets used in this paper are available publicly or were collected with appropriate permissions. The collection of the AtCoder dataset has already been approved.

All experiments in this work were conducted using public datasets.

## References


Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *ArXiv*, abs/2103.06333.

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *ArXiv*, abs/1905.13319.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, T. W. Hennigan, Saffron Huang, Lorenzo Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and L. Sifre. 2021. Improving language models by retrieving from trillions of tokens. In *International Conference on Machine Learning*.

Deng Cai, Yan Wang, Victoria Bi, Zhaopeng Tu, Xiaojiang Liu, Wai Lam, and Shuming Shi. 2018. Skeleton-to-response: Dialogue generation guided by retrieval memory. *ArXiv*, abs/1809.05296.

Nicola De Cao, Wilker Aziz, and Ivan Titov. 2021. Editing factual knowledge in language models. In *Conference on Empirical Methods in Natural Language Processing*.

Xin Cheng, Di Luo, Xiuying Chen, Lemao Liu, Dongyan Zhao, and Rui Yan. 2023. Lift yourself up: Retrieval-augmented text generation with self memory. *ArXiv*, abs/2305.02437.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: Multi-mode translation of natural language and python code with transformers. In *Conference on Empirical Methods in Natural Language Processing*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *ArXiv*, abs/2002.08155.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *ArXiv*, abs/2204.05999.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. 2020. Graphcodebert: Pre-training code representations with data flow. *ArXiv*, abs/2009.08366.

Shirley Anugrah Hayati, Raphaël Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *ArXiv*, abs/1808.10025.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross B. Girshick. 2019. Momentum contrast for unsupervised visual representation learning. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9726–9735.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2021. Unsupervised dense information retrieval with contrastive learning. *Trans. Mach. Learn. Res.*, 2022.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Yu Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Conference on Empirical Methods in Natural Language Processing*.

Jungo Kasai, Keisuke Sakaguchi, Yoichi Takahashi, Ronan Le Bras, Akari Asai, Xinyan Velocity Yu, Dragomir R. Radev, Noah A. Smith, Yejin Choi, and Kentaro Inui. 2022. Realtime qa: What's the answer right now? *ArXiv*, abs/2207.13332.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code prediction by feeding trees to transformers. *2021 IEEE/ACM 43rd International*

Conference on Software Engineering (ICSE)*, pages 150–162.

Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *ArXiv*, abs/2005.11401.

Jia Li, Yongming Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2124–2135.

Junnan Li, Ramprasaath R. Selvaraju, Akhilesh Deepak Gotmare, Shafiq R. Joty, Caiming Xiong, and Steven C. H. Hoi. 2021. Align before fuse: Vision and language representation learning with momentum distillation. In *Neural Information Processing Systems*.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *ArXiv*, abs/2203.07722.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *ArXiv*, abs/2303.17651.

Ansong Ni, Srini Iyer, Dragomir R. Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. *ArXiv*, abs/2302.08468.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*.

Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, LiGuo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 01–13.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Annual Meeting of the Association for Computational Linguistics*.

Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *ArXiv*, abs/2108.11601.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

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

Stephen E. Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3:333–389.

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning to retrieve prompts for in-context learning. *ArXiv*, abs/2112.08633.

Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. 2022. Peer: A collaborative language model. *ArXiv*, abs/2208.11663.

Jianhao Shen, Yichun Yin, Lin Li, Lifeng Shang, Xin Jiang, Ming Zhang, and Qun Liu. 2021. Generate & rank: A multi-task framework for math word problems. In *Conference on Empirical Methods in Natural Language Processing*.

Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Race: Retrieval-augmented commit message generation. In *Conference on Empirical Methods in Natural Language Processing*.

Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6:1–48.

Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. In *Conference on Empirical Methods in Natural Language Processing*.

Yiping Song, Rui Yan, Xiang Li, Dongyan Zhao, and Ming Zhang. 2016. Two are better than one: An ensemble of retrieval- and generation-based dialog systems. *ArXiv*, abs/1610.07149.

Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. 2023. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. *ArXiv*, abs/2310.12397.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2019. Treegen: A tree-based transformer architecture for code generation. *ArXiv*, abs/1911.09983.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Neural Information Processing Systems*.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. 2022. Code-mvp: Learning to represent source code from multiple views with contrastive pre-training. In *NAACL-HLT*.

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

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859.

Bolin Wei. 2019. Retrieve and refine: Exemplar-based neural comment generation. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1250–1252.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating sequences by learning to self-correct. *ArXiv*, abs/2211.00053.

Yu Wu, Furu Wei, Shaohan Huang, Zhoujun Li, and Ming Zhou. 2018. Response generation by context-aware prototype editing. In *AAAI Conference on Artificial Intelligence*.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Conference on Empirical Methods in Natural Language Processing*.

Ori Yoran, Tomer Wolfson, Ori Ram, and Jonathan Berant. 2023. Making retrieval-augmented language models robust to irrelevant context. *ArXiv*, abs/2310.01558.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. In *International Joint Conference on Artificial Intelligence*.

Fengji Zhang, B. Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *ArXiv*, abs/2303.12570.

# A Text-Code contrastive learning

Given a batch of positive pairs text $T$ and code $C$, we make the vector representations $E_t$ for text $T$ and $E_c$ for code $C$ by mapping $[CLS]$ embeddings to normalized lower-dimensional (256-d) from the encoder. We maintain two queues of size M to store the most recent vector representations $Q_t$ and $Q_c$ from the momentum encoders, and the element of which is not equal to the batch sample is denoted as negative samples.

We calculate the similarity of text-code $S^{t2c}(T)$ and code-text $S^{c2t}(C)$ as:

$$S^{t2c}(T) = E_t^\top Q_c, S^{c2t}(C) = E_c^\top Q_t \tag{6}$$

Then we softmax-normalized them as $p^{t2c}(T)$ and $p^{c2t}(C)$:

$$p_i^{t2c}(T) = \frac{exp(S^{t2c}(T)_i/\tau)}{\sum_{j=1}^{M} exp(S^{t2c}(T)_{i,j}/\tau)} \tag{7}$$

$$p_i^{c2t}(C) = \frac{exp(S^{c2t}(C)_i/\tau)}{\sum_{j=1}^{M} exp(S^{c2t}(C)_{i,j}/\tau)} \tag{8}$$

where $\tau$ is a learnable temperature parameter. Let $y^{t2c}(T)$ and $y^{c2t}(C)$ denote the ground-truth one-hot similarity, the text-code contrastive loss from a corpus $D$ is define as the cross-entropy $H$ between $y$ and $p$:

$$\begin{aligned}\mathcal{L}_{ce} = \frac{1}{2}E_{(T,C)\sim D}[H(y^{t2c}(T), p^{t2c}(T)) \\ + H(y^{c2t}(C), p^{c2t}(C))]\end{aligned} \tag{9}$$

# B LCS Algorithm

---

**Algorithm 2** Longest Common Subsequence (LCS)

---

**Require:**

$\#X$ : The first string of LCS

$\#Y$ : The second string of LCS

**Ensure:** LCS: $L[m, n]$

$m \leftarrow |X|$

$n \leftarrow |Y|$

**for** $i = 0$ to $m$ **do**

  **for** $j = 0$ to $n$ **do**

    **if** $i == 0$ or $j == 0$ **then**

      $L[i, j] \leftarrow 0$

    **else if** $X[i] == Y[j]$ **then**

      $L[i, j] \leftarrow L[i-1, j-1] + 1$

    **else**

      $L[i, j] \leftarrow \max(L[i-1, j], L[i, j-1])$

    **end if**

  **end for**

**end for**

**return** $L[m, n]$

---

# C AtCoder Dataset

Comparing to the text generation domain, code generation typically has various reference answers. It is challenging to collect multiple reference answers in the same programming languages that solve a specific task. The online judge platform like AtCoder holds competition regularly and requires participants submit codes to solve several problems, which provides a large amount of reference answers in multiple languages.

Inspired by this, the AtCoder dataset is constructed from the AtCoder platform which hosts weekly competitions. The data collection protocol is approved by an ethics review board and we have obtained the consent of the platform owner prior to use. We collect the code from 150-th to 233-th AtCoder Beginner Contest (ABC). We remove the unnecessary comments to simplify the content and provide an dataset which is specified to process tailored for code-to-code tasks. We filter out excessively long codes to ensure the model could fully process the input code, the filtered result is shown in Figure 9. To collect reference answers in a specific program language, we selected the same problem from AtCoder competition and gathered accepted code submissions in Java, C#, and Python. The codes can translate to each other since they fix the same task.

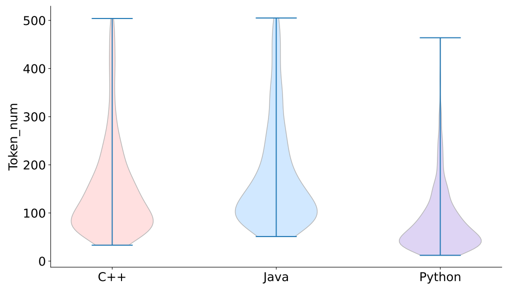


Figure 9: Statistic of token number for AtCoder datasets

# D Metrics

The calculating details of the metrics we use are shown below.

## D.1 BLEU4

The equation for BLEU4 is:

$$\text{BLEU4} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \cdot \log(p_n)\right) \quad (10)$$

where $w_n$ represents the weight assigned to the precision of $n$-grams. $\log(p_n)$ is the logarthm of the precision of $n$-grams.

## D.2 CodeBLEU

Unlike traditional BLEU, CodeBLEU aims to capture both syntactic and semantic correctness in code. It not only considers the lexical similarity but also the syntactic structure and semantic meaning of the generated code, which are crucial for assessing code quality.

First it calculates token-level BLEU using n-gram precision in a manner similar to traditional BLEU scores in natural language processing. The n-gram precision is computed as:

$$p_n = \frac{\sum_{C \in \text{Candidates}} \sum_{i=1}^{l} \mu_n^i \cdot \text{Count}_{\text{clip}}(C'(i, i+n))}{\sum_{C' \in \text{Candidates}} \sum_{i=1}^{l} \mu_n^i \cdot \text{Count}(C'(i, i+n))} \quad (11)$$

Then the syntactic matching involves comparing the abstract syntax trees (AST) of the generated and reference code. The syntactic matching score can be represented as:

$$\text{Syntactic Score} = \frac{\text{Number of matching nodes in AST}}{\text{Total number of nodes in reference AST}} \quad (12)$$

Then the data flow and control flow graph match involves comparing the data flow and control flow graphs. The semantic score can be similarly computed based on the proportion of matched elements in these graphs. Finally, these components are combined into the overall CodeBLEU score with weights indicating the importance of each component.

### D.3 Pass@1

The way to compute Pass@1 is to calculate the percentage of the number of sample passes the test once among the total number of the samples. It is calculated as:

$$\text{Pass@1} = \frac{1}{|I|} \sum_{i \in I} \mathbf{1}_{\{rank_i(y_i)=1\}} \tag{13}$$

where $|I|$ denotes the total number of instances for which predictions were made, $y_i$ is the true label or item for instance $i$, $rank_i(y_i)$ is the function that returns the rank of the true label in the list of predictions for instance $i$, with 1 being the top rank.

## E Supplementary Results of Experiment-1

The results of other datasets are shown below.

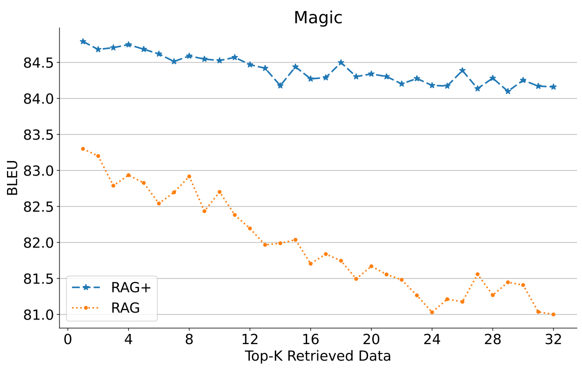


Figure 10: The result of RAG and RAG$^{+}$ evaluated by metric BLEU4 on Magic datasets.

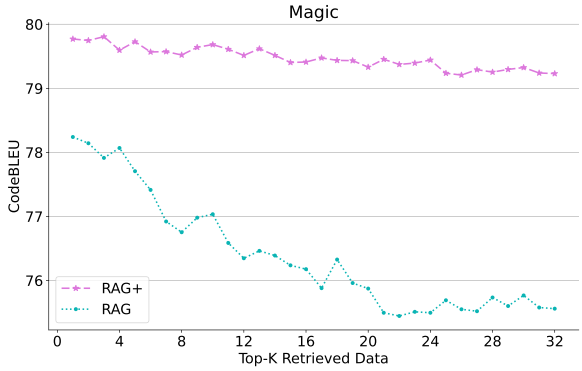


Figure 11: The result of RAG and RAG$^{+}$ evaluated by metric CodeBLEU on Magic datasets.

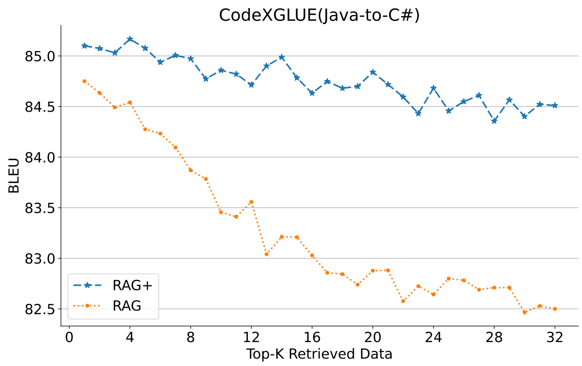


Figure 12: The result of RAG and RAG$^{+}$ evaluated by metric BLEU4 on CodeXGLUE(java-to-c#) datasets.

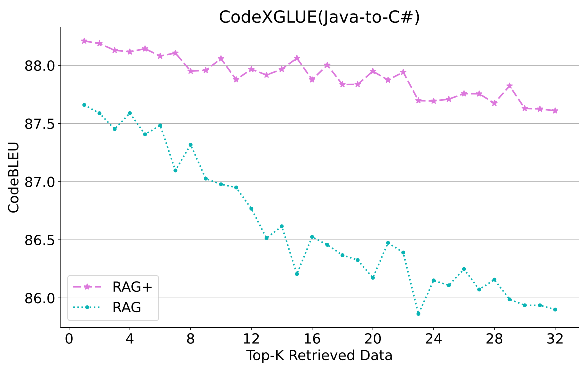


Figure 13: The result of RAG and RAG$^{+}$ evaluated by metric CodeBLEU on CodeXGLUE(java-to-c#) datasets.

## F Case Study

**Input requirement:** Wild Defiance NAME_END NIL ATK_END NIL DEF_END { 2 } { G } COST_END NIL DUR_END Enchantment TYPE_END Avacyn Restored PLAYER_CLS_END 203 RACE_END R RARITY_END Whenever a creature you control becomes the target of an instant or sorcery spell , that creature gets + 3 / + 3 until end of turn .

**Output of RAG**

```
public class Wild Defiance extends Card Impl {
   private static final Filter Spell filter = new Filter Spell (....);
   ...
      this . set Target Controller ( event . get Player Id ( ) ) ;
      return true ;
      }
   return false ; }
   @ Override public String get Rule ( ) {
      return ` Whenever a creature you control becomes the
      target of an instant or sorcery spell , that creature gets
      + 3 / + 3 until end of turn . ` ; }
```

**Output of epoch 1 (SAG)**

```
public class Wild Defiance extends Card Impl {
   private static final Filter Spell filter = new Filter Spell (....);  X
   ...
      this . set Target Controller ( event . get Player Id ( )) ;
      return true ;              X
      }
   return false ; }
   @ Override public String get Rule ( ) {
      return ` Whenever a creature you control becomes the
      target of an instant or sorcery spell , that creature gets
      + 3 / + 3 until end of turn . ` ; }
```

**Output of epoch 2 (SAG)**

```
public class Wild Defiance extends Card Impl {
   public Wild Defiance ( UUID owner Id ) {
   ...
      this . set Target Controller ( controller Id ) ;
      return true ; }
      }
   return false ; }
   @ Override public String get Rule ( ) {
      return ` Whenever a creature you control becomes the
      target of an instant or sorcery spell , that creature gets
      + 3 / + 3 until end of turn . ` ; }
```

A successful example of iterative generation using SAG

(a) **Case Study 1**. The grey background represents the repaired parts. SAG can fix some obvious errors through iteration.

**Input requirement:** Boseiju , Who Shelters All NAME_END NIL ATK_END NIL DEF_END NIL COST_END NIL DUR_END Legendary Land TYPE_END Champions of Kamigawa PLAYER_CLS_END 273 RACE_END R RARITY_END Boseiju , Who Shelters All enters the battlefield tapped . $ { tap } , Pay 2 life : Add { C } to your mana pool . If that mana is spent on an instant or sorcery spell ....

**Output of epoch 1 (SAG)**

```
public class Boseiju Who Shelters All extends Card Impl {
   public Boseiju Who Shelters All ( UUID owner Id ) {
      super ( owner Id , 273 , ` Boseiju , Who Shelters All ` ,
      Rarity . RARE , new Card Type [ ] { .... ) ;
      this . expansion Set Code = ` CHK ` ;
      this . supertype . add ( ` Legendary ` ) ;
      this . add Ability ( .... Ability ( ) ) ;
      Mana mana = Mana . Colorless Mana ( 1 ) ;
      mana . set Flag ( true ) ;              X
      ...
```

**Output of epoch 2 (SAG)**

```
public class Boseiju Who Shelters All extends Card Impl {
   public Boseiju Who Shelters All ( UUID owner Id ) {
      super ( owner Id , 273 , ` Boseiju , Who Shelters All ` ,
      Rarity . RARE , new Card Type [ ] { .... ) ;
      this . expansion Set Code = ` CHK ` ;
      this . supertype . add ( ` Legendary ` ) ;
      this . add Ability ( .... Ability ( ) ) ;
      Mana mana = Mana . Colorless Mana ( 1 ) ;
      mana . set Flag ( true ) ;              X
      ...
```

**Output of epoch 3 (SAG)**

```
public class Boseiju Who Shelters All extends Card Impl {
   public Boseiju Who Shelters All ( UUID owner Id ) {
      super ( owner Id , 273 , ` Boseiju , Who Shelters All ` ,
      Rarity . RARE , new Card Type [ ] { .... ) ;
      this . expansion Set Code = ` CHK ` ;
      this . supertype . add ( ` Legendary ` ) ;
      this . add Ability ( .... Ability ( ) ) ;
      Mana mana = Mana . Colorless Mana ( 1 ) ;
      mana . set Flag ( true ) ;              X
      ...                               repeated mistakes
```

SAG: the phenomenon of repeated mistakes

(b) **Case Study 2**. The grey background represents the repeated mistakes. SAG is unable to recognize and fix some errors, leading to a phenomenon of repeated mistakes.

**Input requirement:** Boseiju , Who Shelters All NAME_END NIL ATK_END NIL DEF_END NIL COST_END NIL DUR_END Legendary Land TYPE_END Champions of Kamigawa PLAYER_CLS_END 273 RACE_END R RARITY_END Boseiju , Who Shelters All enters the battlefield tapped . $ { tap } , Pay 2 life : Add { C } to your mana pool . If that mana is spent on an instant or sorcery spell ....

**Output of epoch 1 (ISAGWR)**

```
public class Boseiju Who Shelters All extends Card Impl {
   public Boseiju Who Shelters All ( UUID owner Id ) {
      super ( owner Id , 273 , ` Boseiju , Who Shelters All ` ,
      Rarity . RARE , new Card Type [ ] { .... ) ;
      this . expansion Set Code = ` CHK ` ;
      this . supertype . add ( ` Legendary ` ) ;
      this . add Ability ( .... Ability ( ) ) ;
      Mana mana = Mana . Colorless Mana ( 1 ) ;
      mana . set Flag ( true ) ;              X
      ...                               repeated mistakes
```

**After Review**

```
public class Boseiju Who Shelters All extends Card Impl {
   public Boseiju Who Shelters All ( UUID owner Id ) {
      super ( owner Id , 273 , ` Boseiju , Who Shelters All ` ,
      Rarity . RARE , new Card Type [ ] { .... ) ;
      this . expansion Set Code = ` CHK ` ;
      this . supertype . add ( ` Legendary ` ) ;
      this . add Ability ( .... Ability ( ) ) ;
      Mana mana = <Mask>
      mana . set Flag ( true ) ;
      ...
```

**Output of epoch 2 (ISAGWR)**

```
public class Boseiju Who Shelters All extends Card Impl {
   public Boseiju Who Shelters All ( UUID owner Id ) {
      super ( owner Id , 273 , ` Boseiju , Who Shelters All ` ,
      Rarity . RARE , new Card Type [ ] { .... ) ;
      this . expansion Set Code = ` CHK ` ;
      this . supertype . add ( ` Legendary ` ) ;
      this . add Ability ( .... Ability ( ) ) ;
      Mana mana = new Mana ( 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 ) ;
      mana . set Flag ( true ) ;
      ...
```

ISAGWR: mask the repeated mistakes

(c) **Case Study 3**. The grey background on output of epoch 1 represents the repeated mistakes. The grey background on output of epoch 2 represents the repaired parts.

Figure 14: Case Study