SEA: Semantic Aligned Code Summary Generation with Contrastive Learning Framework

Anonymous ACL submission

Abstract

Code summary aims to generate a natural language description of a piece of code, which can help understand the program and increase development productivity. Although the performance of previous works has greatly improved, the in-depth exploration of inherent gaps between code semantics and natural language semantics still needs to be developed. To this end, we explore the role of data flow and summary in code semantics. In this paper, we propose the **SE**mantic-**A**ligned code summary generation framework (SEA), a semantic-aware method that conducts in-depth research into code semantics and converges on alignment with natural language semantics. Specifically, we use the data-flow-guided walking algorithm to capture co-occurrence nodes and utilize destruction-construction ideas to represent code semantics. We also design a semantic alignment loss to align code and natural language semantics in the same space. Extensive experiments on Java and Python datasets show the effectiveness and generalization of our SEA.

1 Introduction

Code summary has been acknowledged as a critical issue in software development and maintenance, which aims to generate intelligible natural language descriptions for source code segments. As shown in Figure 1(a), given a piece of code, the summary describes its main goal of "array calculation". A good summary can facilitate program comprehension and support various programming applications (e.g., code search), potentially increasing developers' productivity and significantly reducing their tedious workload (Ahmad et al., 2020; Lu et al., 2021; Wang et al., 2022).

The core problem of code summary is to understand and align both semantics. Along this line, several strategies have been investigated to model the codes using language models (Iyer et al., 2016; Allamanis et al., 2017; Hu et al., 2018a; Alon et al., 2018, 2019), including learning the sequential semantics by treating the code token-by-token and learning the topological semantics via establishing its abstract syntax tree (AST) (Figure 1(b) shows one AST example). Despite their success, most of them are still far from a comprehensive code understanding, since the following two key aspects are underexplored. First, they overlook the deep analysis of data flow. Specifically, the data flow of the code generally reflects the relation the of variable *where-the-value-comes-from*. In Figure 1(b), we can understand the loop statement "for value ..." by analyzing its dataflow edges: the element *value* is taken from the input *array*, and then the *value* is added by *sum*, and finally, the *sum* is returned. Though some work considers it as auxiliary information (Wu et al., 2020; Guo et al., 2020), they fail to extract sufficient semantics from the whole code process of "input-computation-return" reflected by the data flow from a dynamic perspective. Second, they generally follow a standard encoder-decoder framework, where the summary only works in the decoding stage. Therefore, they often suffer from the exposure bias problem (Ahmad et al., 2020; Wu et al., 2020) because the code and summary

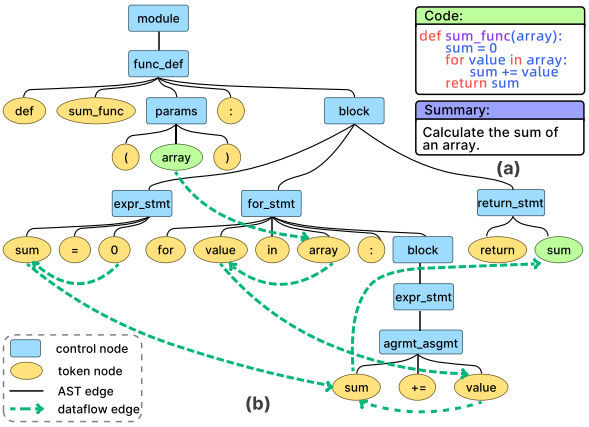

Figure 1: (a) An example of a code and summary. (b) The abstract syntax tree (AST) and data flow correspond to the code.

follow different grammar and exist in inconsistent semantic space. Thus, how to align both code and summary into consistency is important.

In this paper, we deeply focus on code summary from the above aspects. However, it is always challenging. First, how to exploit code intermediate forms like abstract syntax tree (AST) and data flow to explore code semantics has yet to be deeply explored. Sequentially modeling code lacks structural information (e.g., topology information in Figure 1(b)), while graph representation methods fail to capture the long-distance dependencies that often exist in code (e.g., the input *"array"* and the output *"sum"* distance is 5-hops in Figure 1(b)).

Second, since both the code and summary follow different grammars, which have different inherent properties, it is difficult to align them. For example, in Figure 1(a), if we replace the word "*array*" with the word "*var_1*" in summary, the meaning of the sentence changes dramatically. In contrast, nothing has changed if we arbitrarily replace the variable names in the code (e.g., replacing "*array*" with the word "*var_1*"). We make deeper data analysis in Figure 6 to demonstrate such differences. Specifically, the token imbalance issue in code is more severe than it is in summary, i.e., fewer token categories (20 words) occupy a higher proportion (38.4%) of the code corpus, making code representations more ambiguous. Thus, how to mine code semantics from the data flow and align it with the summary semantics has become a problem that needs to be solved.

To meet the above challenges, we propose a novel **SE**mantic **A**ligned code summary generation framework (SEA). The whole model includes two modules: Semantic Extraction Module (SEM) and Semantic Align Module (SAM). In the SEM, we try to model the structured long-distance dependencies to mine semantic information. Based on the source code, we build AST with data flow edges like Figure 1(b). Then, for each token node, we apply a novel data-flow-guided walking method to capture the semantically related nodes (see Figure 3). Next, we use a model inspired by the destruction-reconstruction process to depict the co-occurrence correlation in the walking path. In the SAM, the use of Bi-GRU first ensures that code and natural language are projected into the same space. After that, to get the semantic representation of code as close as feasible to that of natural language, while keeping the various meanings as far away as possible inside the same space, we build on the promising results of bilinear contrastive learning (Kong et al., 2019; Clark et al., 2020). After constructing negative samples with convex interpolation, contrastive learning not only optimizes semantic similarity function but also guides the subsequent decoding stage. In the decode stage, we integrate the semantic representation into the code representation and use the trained semantic similarity function to guide the generation process. We conduct extensive experiments on real-world datasets. The experimental results fully validate the effectiveness of SEA in semantic representation and semantic alignment.

2 Related Works

2.1 Code Summary

Based on the ways of source code representations, generally, we divide the previous work into three categories. The most basic is the *linearization* method. The general approach is to take the code sequence directly as input (Iyer et al., 2016; Hu et al., 2018b; Feng et al., 2020; Ahmad et al., 2020; Parvez et al., 2021; Wei et al., 2019). Some other methods consider that AST has rich structural information, and traverse the AST to get the linearized input, such as, Pre-Order Traversal (Guo et al., 2022; Tang et al., 2022) and Structure-Based Traversal (Hu et al., 2018a, 2020). Simply entering a sequence cannot capture the hierarchical relationship and structural information of the code. Whereas *path-aware* methods focus more on modeling paths in the AST. It regards AST as the backbone, which represents the code by integrating path information (Alon et al., 2018, 2019). Despite the strong interpretability, the rigid way of path selection leads to its poor performance. The *graph representation* method comprehensively considers the topological structure of the code graph. Most of the early works rely on convolutional neural network (CNN) to carry out convolution operations on AST (Mou et al., 2016) or rely on recurrent neural network (RNN) to represent the entire AST bottom-up (Wan et al., 2018; Zhang et al., 2019). Although the above approaches have achieved considerable success, AST is typical non-Euclidean data, which contains complex structural information. The emergence of graph representation algorithms such as graph convolutional networks (GCN) has effectively filled this gap (Kipf and Welling, 2016). The graph representation method is applied on the code

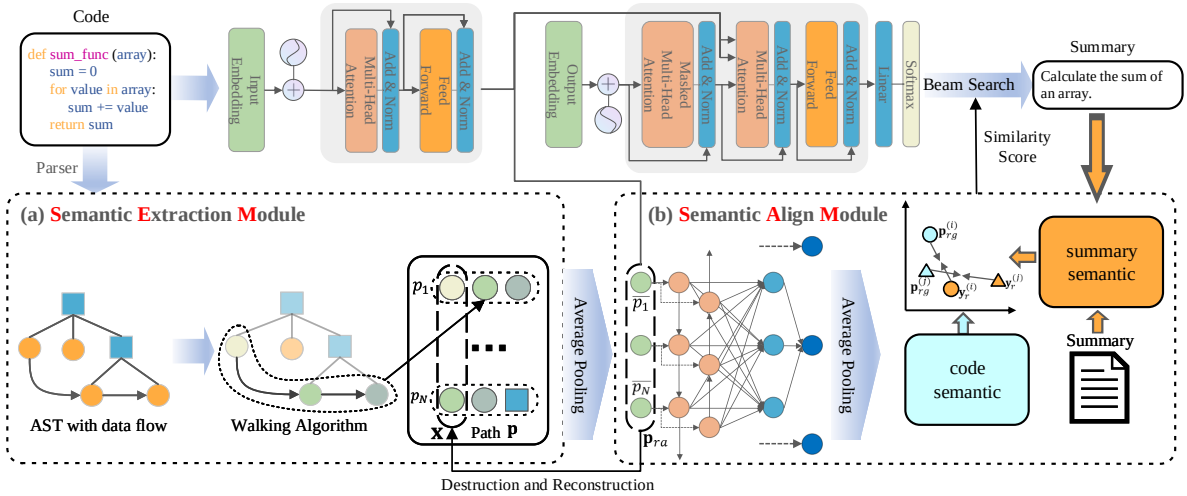

Figure 2: The overview of our proposed model SEA. The architecture of SEA is based on the Transformer as the backbone. We design a data flow-guided walk algorithm to capture the co-occurrence relationships of key variables. SEM and SAM are designed for semantic extraction and semantic alignment, respectively.

graph to get the code representation (Allamanis et al., 2017; Xu et al., 2018b; LeClair et al., 2020; Fernandes et al., 2018; Guo et al., 2020; Liu et al., 2020). The core idea of graph representation is the propagation and aggregation of neighbor information, which is difficult to capture the dependencies between multi-hop neighbors (Xu et al., 2018a) that exist in code graphs (e.g., the input *"array"* and the output *"sum"* distance is 5-hops in Figure 1(b)). For the reasons outlined above, prior arts can't fully exploit the data flow and disregard the gap between code and natural language.

2.2 Contrastive Learning

Contrastive learning, as one of the self-supervised learning methods, has the characteristics of obtaining prior knowledge distribution between sample pairs without relying on labeled data, which has been well received since it was proposed. Contrastive learning has gradually become the new paradigm in the CV and NLP field (Chen et al., 2020; He et al., 2020; Kong et al., 2019; Clark et al., 2020; Gao et al., 2021). Contrastive learning has also attracted increasing attention in the field of code. (Bui et al., 2021) uses five operations (e.g., Variable Renaming) to construct positive samples from the original code. The code representation is optimized by contrastive learning, which minimizes the distance between positive samples while maximizing the distance between negative samples. In our article, we draw on the experience of contrastive learning to pull code semantics and summary semantics closer.

3 Preliminary

3.1 Abstract Syntax Tree and Data Flow

In this section, we will introduce abstract syntax tree (AST) and data flow. A typical AST and data flow are shown in Figure 1. AST consists of two kinds of nodes including control nodes V_c and token nodes V_t. Control nodes represent certain construction (e.g., *for_statement* and *block*), while token nodes are composed of the lexical token, such as identifiers (*array*), keywords (*def*), numbers(*0*), etc. AST edges E_a reflect the topology structure between nodes and data flow edges E_d reflect the transfer of information between variables. For example, if there is an assignment statement *sum=0*, then a one-way edge is drawn from token node *0* to *sum*. Therefore we define a code graph as $G(V_c, V_t, E_a, E_d, W_e)$, where V_c, V_t, E_a, E_d represent the sets of the control node, token node, AST edge, and data flow edge. In addition, AST edges and data flow both affect the preference of the following walking algorithm. To emphasize the data flow guidance, we define the weights of the two types of edges W_e are inversely proportional to their numbers:

$$w_{ij} = \begin{cases} 1 & \text{if } e_{i,j} \in E_a, \\ \frac{|E_a|}{|E_d|} & \text{if } e_{i,j} \in E_d, \end{cases} \quad (1)$$

where $e_{i,j}$ means edge between node i and j.

3.2 Problem Definition

We denote $(\mathbf{x}, \mathbf{y}) \in (\mathcal{X}, \mathcal{Y})$ as a pair of $(code, summary)$, where $\mathbf{x} = \{x_1, x_2, ..., x_N\}$

is a source code with N tokens (e.g., "*int sum = 0*"), $\mathbf{y} = \{y_1, y_2, ..., y_{N'}\}$ is a target summary with N' tokens. Code summary usually employs a sequence-to-sequence model of which the purpose is to learn the transformation from the source space to the target space, that is $\mathcal{X} \rightarrow \mathcal{Y} : f(\mathbf{y} \mid \mathbf{x}; \Theta)$. Formally, given a set of k observed (code, summary) pairs, $\mathcal{S} = \{(\mathbf{x}, \mathbf{y})_1, (\mathbf{x}, \mathbf{y})_2, ..., (\mathbf{x}, \mathbf{y})_k\}$, the training objective is to minimize the following log-likelihood:

$$\begin{aligned} \mathcal{L}_{mle}(\Theta) &= \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{S}}(-\log P(\mathbf{y} \mid \mathbf{x}; \Theta)) \\ &= \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{S}} \sum_{t=1}^{N'} -\log P\left(\mathbf{y}_t \mid \mathbf{y}_{<t}, \mathbf{x}; \Theta\right), \end{aligned} \tag{2}$$

where $\mathbf{y}_{<t}$ is sequence of summary before time t and Θ is a trainable set of model parameters.

4 The Proposed Model

The overall architecture of SEA is shown in Figure 2. Specifically, we separate the code semantics into two phases and design two modules to correspond to the two phases. At first step, we create a novel Semantic Extraction Module (SEM), which aims to mine code semantics guided by the data flow. In the meantime, a Semantic Align Module (SAM) is proposed to solve the semantic gap between code and summary. The following two sections will describe these modules in detail.

4.1 Semantic Extraction Module

To extract specific semantic information from the code, we design a data flow-guided walking algorithm. To be specific, given a certain source code $\mathbf{x} = \{x_1, x_2, ..., x_N\}$, we output the semantic representation $\mathbf{p}_{ra}$ through SEM. To this end, we first extract the walking paths $\mathbf{p}$ from the code graph G, and then obtain the code semantic representation $\mathbf{p}_{ra}$ through the walking paths $\mathbf{p}$.

Unlike the previous path-aware method (Alon et al., 2018, 2019), we do not take a mandatory path from token to token but adopt a data flow-guided random walk strategy. Inspired by previous work (Perozzi et al., 2014; Tang et al., 2015; Grover and Leskovec, 2016), the walking process we take starts from all token nodes V_t and walks along the edges (i.e., the AST edges E_a and data flow edges E_d). Specifically, given a start token node $v \in V_t$, we simulate a path of fixed length l. We donate c_i is the i-th node in the path, starting with $c_0 = v$. The probability of accessing the node c_i from the former node c_{i-1} is:

$$P\left(c_i \mid c_{i-1}\right) = \begin{cases} \frac{e_{i-1,i}}{z} & \text{if } (c_{i-1}, c_i) \in E_a \cup E_d, \\ 0 & \text{otherwise}, \end{cases} \tag{3}$$

where z is the normalizing constant. We show the detailed walking algorithm in Appendix A.3.

The introduction of data flow edges makes it more likely that semantically related nodes will occur together, and this walking algorithm captures the relationship between semantically related nodes. Taking Figure 1 as an example, we calculate the expected hitting time (time to first reach the destination node) between the input parameters *array* and the final result *sum* (green node). However, if we remove data flow edges, the expected hitting time will rise significantly from 9.9 to 135. This indicates that the existence of data flow edges provides a shortcut to capture the co-occurrence relationship between variables. The existence of this shortcut ensures that the algorithm can capture nodes with close semantics with a small path length l, which improves efficiency and avoids noises. The details of the calculation of hitting time will be presented in Appendix A.2.

Figure 3 shows the specific walking process. We take node "*array*" as the starting point and use the walk algorithm above to capture co-occurrence nodes with close semantics. First we calculate the transition probability distribution, and transfer to node "*array*" with high probability (0.89). This step reflects that "*array*" is inherited from the input "*array*". Likewise, node "*array*" transfers to nodes "*value*" and "*sum*", reflecting the value of "*value*" being taken from "*array*" and flowing out to "*sum*".

In this way, we get paths of fixed length $\mathbf{p} = \{p_1, p_2, .., p_N\}$ for each start node $v \in V_t$, where $p_i = \{p_{i1}, p_{i2}, ..., p_{il}\}$ (e.g., *array->array->value->value->sum*) and l is a small path length. It should be noted that the sequence obtained by arranging the token nodes V_t in a pre-order traversal of the AST is exactly the input sequence $\mathbf{x}$, that is $\mathbf{x} = \{x_1, x_2, ..., x_N\} = \{p_{11}, p_{21}, ..., p_{N1}\}$. To better represent the path, we adopt the idea of destruction and reconstruction (Mikolov et al., 2013; Devlin et al., 2018). In the destruction step, we use average pooling at each path to destruct the representation of each token $\mathbf{p}_{ra} = \{\overline{p_1}, \overline{p_2}, ..., \overline{p_N}\}$. Then, we reconstruct the corrupted token using the input tokens in the reconstruction step by optimizing the loss function. This process can be

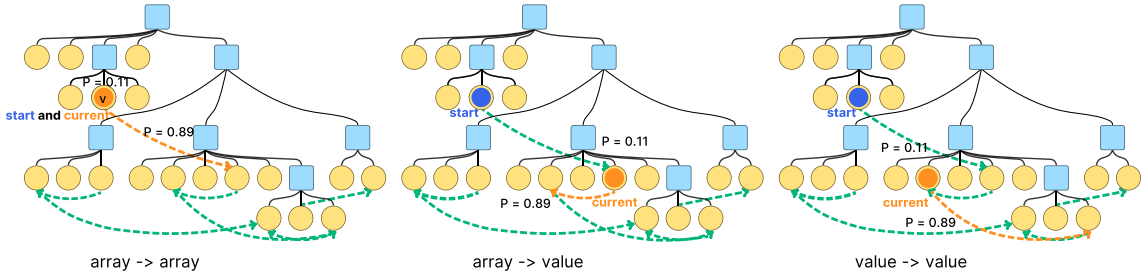

Figure 3: The process diagram of the walking algorithm. The example comes from Figure 1. We omit some nodes for clarity. Guided by data flow, the walking algorithm automatically captures co-occurrence relationships. Since the number of data flow edges is 7 (unidirectional) and the number of AST edges is 56 (bidirectional), the weights of the edge are 8 and 1. The starting point is *array*, and after being guided by the data flow, it reaches the *sum* node through the node *array* and *value* respectively.

formalized as follows:

$$\mathbf{x}^{'} = f(\mathbf{p}_{ra}), \tag{4}$$

$$\mathcal{L}_{drc}(\Theta) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{S}}(-\log P(\mathbf{x} \mid \mathbf{x}^{'}; \Theta)), \tag{5}$$

f represents the fitting function, such as a neural network. $\mathbf{x}^{'}$ is the reconstructed token. We optimize the semantic representation $\mathbf{p}_{ra}$ by optimizing the loss function $\mathcal{L}_{drc}$. Finally, we incorporate $\mathbf{p}_{ra}$ into the vanilla Transformer's representation as:

$$\mathbf{x_r} = \lambda \cdot \text{Transformer}(\mathbf{x}) + (1 - \lambda)\, \mathbf{p}_{ra}, \tag{6}$$

where λ is a trainable parameter controlling the weights of the representation.

4.2 Semantic Align Module

To alleviate the semantic gap between code and summary, we try to put forward the SAM using bilinear contrastive learning to align code semantics with summary semantics. Therefore in SAM, given a code semantic representation $\mathbf{p}_{ra}$ and summary $\mathbf{y} = \{y_1, y_2, ..., y_{N'}\}$, our goal is to train a semantic similarity function $s_\omega(\cdot, \cdot)$ that makes positive samples as close as possible and negative samples as far apart as possible. During this process, we mainly focus on three points: the projection of semantic representation, the construction of negative samples, and the use of contrastive learning.

The first is the projection of semantic representation. For natural language, we don't put much focus on the issue of how to represent its semantics, as this is not a simple problem and still merits more study. Therefore, we only use mean-pooled word embeddings as semantic representations for natural language $\mathbf{y}_r = \frac{1}{N'}\sum_{i=1}^{N'} y_i$. For code, a Bi-GRU encoder is applied to the semantic representation $\mathbf{p}_{ra}$ to project it into a vectorized semantic representation $\mathbf{p}_{rg}$:

$$\mathbf{p}_{rg} = \frac{1}{N}\sum_{i=1}^{N} \text{FC}\left(\text{Bi-GRU}\left(\mathbf{p}_{ra}\right)\right), \tag{7}$$

where $\text{FC}(\cdot)$ is the fully connected layer.

The second is the construction of negative samples. Instead of sampling directly from a mini-batch, we use convex interpolation to construct negative samples from other samples in a mini-batch (Wei et al., 2022). Specifically, we construct negative samples in the following way:

$$\begin{aligned} \mathbf{p}_{rg}^{\prime(j)} &= \mathbf{p}_{rg}^{(i)} + \lambda_{\mathbf{p}}\left(\mathbf{p}_{rg}^{(j)} - \mathbf{p}_{rg}^{(i)}\right), \lambda_{\mathbf{p}} \in \left(\frac{d}{d'_{\mathbf{p}}}, 1\right], \\ \mathbf{y}_{r}^{\prime(j)} &= \mathbf{y}_{r}^{(i)} + \lambda_{\mathbf{y}}\left(\mathbf{y}_{r}^{(j)} - \mathbf{y}_{r}^{(i)}\right), \lambda_{\mathbf{y}} \in \left(\frac{d}{d'_{\mathbf{y}}}, 1\right], \end{aligned} \tag{8}$$

where $d = ||\mathbf{p}_{rg}^{(i)} - \mathbf{y}_r^{(i)}||_2$, $d'_{\mathbf{p}} = ||\mathbf{p}_{rg}^{(i)} - \mathbf{p}_{rg}^{(j)}||_2$ and $d'_{\mathbf{y}} = ||\mathbf{y}_r^{(i)} - \mathbf{y}_r^{(j)}||_2$. The existence of $\lambda_{\mathbf{p}}$ ensures that once the distance between the positive sample pair $(\mathbf{p}_{rg}, \mathbf{y}_r)$ is larger than the negative sample pair $(\mathbf{p}_{rg}^{(i)}, \mathbf{p}_{rg}^{(j)})$, the constructed interpolated negative sample semantics will not be too far from the original sample semantics. The same goes for the existence of $\lambda_{\mathbf{y}}$.

Finally, we apply bilinear contrastive learning (Hou et al., 2022) to align the code semantic representation and natural language semantic representation. All samples are combined to optimize semantic similarity function $s_\omega(\cdot, \cdot)$ through minimizing contrastive learning loss:

$$\begin{aligned} \mathcal{L}_{clt}(\Theta) &= \mathbb{E}_{\left(\mathbf{p}_{rg}^{(i)}, \mathbf{y}_r^{(i)}\right)\sim\mathcal{B}}\left(-\log \frac{e^{s_\omega\left(\mathbf{p}_{rg}^{(i)}, \mathbf{y}_r^{(i)}\right)}}{e^{s_\omega\left(\mathbf{p}_{rg}^{(i)}, \mathbf{y}_r^{(i)}\right)} + \xi}\right), \\ \xi &= \sum_{j\&j\neq i}^{|\mathcal{B}|}\left(e^{s_\omega\left(\mathbf{y}_r^{(i)}, \mathbf{y}_r^{\prime(j)}\right)} + e^{s_\omega\left(\mathbf{p}_{rg}^{(i)}, \mathbf{p}_{rg}^{\prime(j)}\right)}\right), \end{aligned} \tag{9}$$

where $\mathcal{B}$ is a mini-batch sampled from training set $\mathcal{S}$ and $s_\omega(\cdot,\cdot)$ is the similarity function parameterized by ω, we define it in the form of bilinear:

$$s_\omega\left(\mathbf{p}_{rg}^{(i)}, \mathbf{y}_r^{(i)}\right) = \mathbf{p}_{rg}^{(i)\top} \cdot W \cdot \mathbf{y}_r^{(i)}, \tag{10}$$

where $W \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ is a trainable matrix. The final training objective is:

$$\mathcal{L} = \mathcal{L}_{mle}(\Theta) + \mathcal{L}_{drc}(\Theta) + \mathcal{L}_{clt}(\Theta). \tag{11}$$

4.3 Generation with Similarity Function

Contrasted with previous generate tasks, we incorporate the semantic similarity function learned in Eq. 10 into the decoding stage to guide the generated results. Given the code $\mathbf{x}$, we aim to generate optimal target $\mathbf{y}^*$. We first utilize beam search to autoregressively generate k candidates $\hat{\mathbf{y}} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_k\}$. Then, we employ the trained similarity function to evaluate the semantic similarity between the candidates and the code. Finally, we combine the likelihood and similarity scores to find the target sequence $\mathbf{y}^*$:

$$\begin{aligned}\mathbf{y}^* = \arg\max_{\hat{\mathbf{y}}} \{\alpha \cdot s_\omega\left(\mathbf{p}_{rg}, \hat{\mathbf{y}}_r\right) + \\ (1-\alpha) \prod_{t=0} P\left(\hat{\mathbf{y}}_t \mid \hat{\mathbf{y}}_{<t}, \mathbf{x}\right)\}.\end{aligned} \tag{12}$$

5 Experiments

5.1 Dataset

We conduct our experiments on two commonly used code summary generation datasets. One is Java (Hu et al., 2018b) and the other is Python (Wan et al., 2018). We filter summaries that are less than four characters to ensure quality. We give statistics of these two datasets in Appendix 4. We split CamelCase and snake_case, such as splitting *sum_func* into *sum* and *func*, which can greatly alleviate the Out-Of-Vocabulary problem.

5.2 Evaluation Metrics

We evaluate the performance with BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004). BLUE compares the result of the summary with its corresponding reference and calculates a composite score. The higher the score, the better the machine translates. Here, we use smoothed BLEU-4 as an evaluation metric and report the overall score. METEOR considers the accuracy rate and recall rate based on the whole corpus, sentence fluency, and the influence of synonyms on semantics. ROUGE-L computes the longest common subsequence used by summary and reference

5.3 Baselines

To fully prove the validity of our model, we compared SEA to nine different baseline models. As (Shi et al., 2022) mentioned, the metric varies with different BLEU calculation methods. Therefore for a fair comparison, in our experiment, we rerun the two best-performing baselines and use the same method to calculate BLEU scores. In general, we divide baselines into three categories: 1) **Linearization Methods:** CODE-NN (Iyer et al., 2016), DeepCom (Hu et al., 2018a), API+CODE (Hu et al., 2018b), Dual Model (Wei et al., 2019), TransBase (Ahmad et al., 2020); 2) **Path-Aware Methods:** Code2Seq (Alon et al., 2018), SiT (Wu et al., 2020); 3) **Graph Methods** Tree2Seq (Eriguchi et al., 2016), RL+Hybird2Seq (Wan et al., 2018).

5.4 Main Results

Our model results on Java and Python datasets are shown in Table 1, the following conclusions can be drawn from the results: 1) Our model outperforms all baseline models on the Java dataset. Specifically, our model outperforms the best baseline model by 28.3% in METEOR. Since we augment token nodes with co-occurrence nodes to enrich semantics, our model generates more synonyms. Therefore, as a metric for accurate measurement of each word, BLEU does not improve significantly, but METEOR comprehensively considers all synonyms, so it received a huge boost. 2) On the Python dataset, the improvement of BLEU value is relatively limited (although there is still a huge performance boost over the BaseTrans (Ahmad et al., 2020)), but METEOR and ROUGE-L have achieved 3.04 and 5.19 improvements respectively. Experimental results demonstrate the effectiveness of our model.

5.5 Ablation Study

Ablation on the Semantic Extraction Module. We introduce a variant of the model by removing the SEM. This variant directly uses the unmodified transformer encoder output as the input of the decoder. We named this variant as SEA_*w/o_SEM*.

Ablation on the Semantic Align Module. Likewise, a variant of the model is introduced by remov-

Category	Methods	Java			Python		
		BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
Linearization	CODE-NN (Iyer et al., 2016)	27.60	12.61	41.10	17.36	09.29	37.81
	DeepCom (Hu et al., 2018a)	39.75	23.06	52.67	20.78	09.98	37.35
	API+CODE (Hu et al., 2018b)	41.31	23.73	52.25	15.36	08.57	33.65
	Dual Model (Wei et al., 2019)	42.39	25.77	53.61	21.80	11.14	39.45
	BaseTrans* (Ahmad et al., 2020)	44.58	29.12	53.63	25.77	16.33	38.95
Graph	Tree2Seq (Eriguchi et al., 2016)	37.88	22.55	51.50	20.07	08.96	35.64
	RL+Hybrid2Seq (Wan et al., 2018)	38.22	22.75	51.91	19.28	09.75	39.34
Path-Aware	Code2Seq (Alon et al., 2018)	24.42	15.35	33.95	17.54	08.49	20.93
	SiT* (Wu et al., 2020)	44.98	26.97	55.18	**33.84**	20.94	48.26
Ours	**SEA**(our)	**45.23**	**37.37**	**56.01**	30.30	**23.98**	**53.45**

Table 1: Comparison of SEA with the baseline methods. * means we re-run and use a consistent BLEU calculation method. **Bold** means state of the art on this metric. Some results of the baseline methods are directly reported from (Ahmad et al., 2020).

(a) Java

Methods	BLEU	Δ(%)	METEOR	Δ(%)	ROUGE-L	Δ(%)
SEA	45.23	-	37.37	-	56.01	-
SEA_*w/o_SEM*	44.97	-0.26	37.30	-0.07	54.45	-1.56
SEA_*w/o_SAM*	44.26	-0.97	37.33	-0.04	54.06	-1.95
SEA_*w/o_dfg*	44.41	-0.82	36.95	-0.42	54.37	-1.95
SEA_*random*	44.26	-0.97	37.02	-0.35	54.35	-1.66

(b) Python

Methods	BLEU	Δ(%)	METEOR	Δ(%)	ROUGE-L	Δ(%)
SEA	30.30	-	23.98	-	53.45	-
SEA_*w/o_SEM*	27.43	-2.87	19.57	-4.41	40.57	-12.88
SEA_*w/o_SAM*	26.93	-3.37	19.00	-4.98	40.13	-13.32
SEA_*w/o_dfg*	27.83	-2.47	19.13	-4.85	40.65	-12.80
SEA_*random*	28.28	-2.02	19.53	-4.45	41.28	-12.17

Table 2: Ablation study on the effect of the different modules we designed.

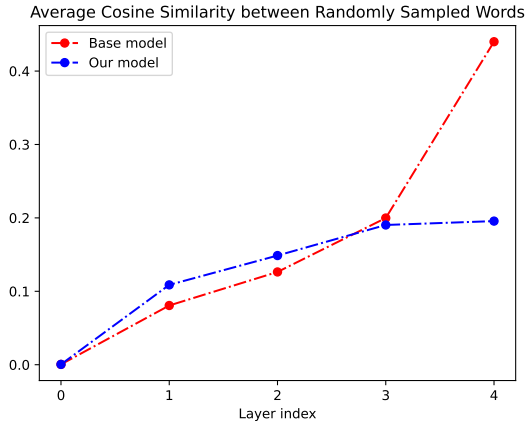

Figure 4: Comparison of cosine similarity between our model and vanilla Transformer. The higher the value, the stronger the anisotropy.

ing SAM. We named this variant as SEA_*w/o_SAM*. SEA_*w/o_SAM* removes the process of using Bi-GRU encoding path and bilinear contrastive learning. These components make code and natural language semantically close in the same space.

Ablation on the Walking Algorithm. We design two variants to verify the effectiveness of the walking algorithm. The first variant is to remove the guidance. Specifically, we perform the walking algorithm directly on AST instead of AST with data flow edges. We note this variant as SEA_*w/o_dfg*. The second variant is that we do not perform a walking algorithm to capture co-occurring nodes but directly use random selection. This variant is denoted as SEA_*random*.

The results are reported in Table 2. Experimental results demonstrate the effectiveness of each module. The major results are summarized as: 1) On the Java dataset, the variants of the model have an overall performance loss relative to the full model. For example, the performance degradation of methods that directly use random selection to capture co-occurrence nodes (such as SEA_*random*) is not particularly severe. We conjecture the reason is that long-distance variable dependencies exist in the code graph, although in the absence of guidance, random selection can capture some long-distance dependencies. 2) Compared with the Java dataset, the model variants' performance drops significantly under the Python dataset. Among them, the performance drop of variant SEA_*w/o_SAM* is the most obvious, indicating that the Semantic Align Module in the Python dataset can effectively align the code and the summary in the latent space, further filling the gap between the two.

5.6 Anisotropy Study

Previous methods have successfully created contextualized code token representations, which are sensitive to the context in which they appear (Ahmad et al., 2020; Wu et al., 2020). This kind of representation has achieved success, but there are also works pointing out that the context-sensitive method embeds words in a narrow cone space, rather than being uniform in all directions (Etha-

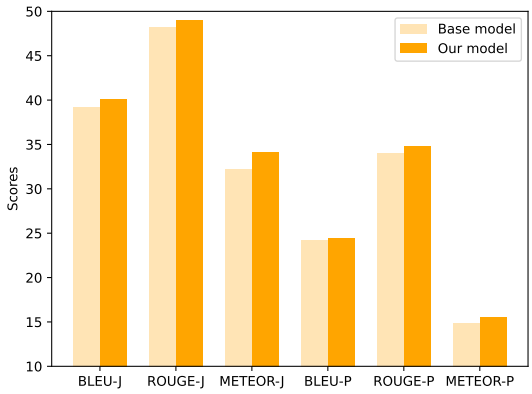

Figure 5: Results comparison with vanilla Transformer in inappropriate naming scenarios. The three columns on the left are metrics on the Java dataset, and the three columns on the right are metrics on the Python dataset.

yarajh, 2019; Cai et al., 2020). This phenomenon is called anisotropy. The greater the anisotropy, the narrower this cone will be (Mimno and Thompson, 2017). Contrarily, isotropy often increases the space's robustness and efficiency. (Cai et al., 2020).

Note that, Isotropy has theoretical (Arora et al., 2017) and empirical (Mu et al., 2017) benefits. Inspired by the above works, we further explore the anisotropy of our model. We follow their procedure: uniformly randomly sample 5K words and calculate the average cosine similarity:

$$S \triangleq \mathbb{E}_{i \neq j} \left[\cos \left(\phi \left(t_i \right), \phi \left(t_j \right) \right) \right], \qquad (13)$$

where $\phi(t_i)$ is one random sample's representation. The result of the cosine similarity calculation is shown in Figure 4. The results we observed are consistent with the conclusions of previous work: The higher the layer, the more anisotropic. But the difference is that our model does not significantly improve the anisotropy in the final output layer (layer index 4). This phenomenon may be explained by 1) In the SEM, strongly linked words are caught and displayed to bring similar words closer. 2) In the SAM, negative samples are pushed further apart so that representations can be evenly spread throughout the entire space. Based on these findings, it's clear that our model successfully alleviates the issue of anisotropy in representation.

5.7 Performance in Meaningless Identifiers

As we mentioned in Section 1, one difference between code and natural language is that we can replace identifiers in code at will, but natural language cannot. However, previous arts utilized the identifiers to learn the representation of code, while the identifiers are not convinced, and replacing them with meaningless identifiers will cause performance reduction. Therefore, to verify the robustness of our model, we replace all identifiers in the code with anonymous identifiers, e.g., replace *sum_func* with *a*. The result is shown in Figure 5. Although the performance of both has declined, our model still outperforms the vanilla Transformer in all metrics. Our method outperforms vanilla Transformer due to the introduction of a semantic understanding that does not vary with how identifiers are named, and the introduction of natural language as an additional semantic complement. This further proves the robustness of SEA in confusing scenarios.

5.8 Learned Similarity Function

To support whether our model learns semantic consistency between code and summary, we calculate semantic similarity of positive and negative samples and compare it with vanilla Transformer on the testing set. Specifically, the decreasing trend of semantic similarity is: positive samples of SEA (0.7463) > positive samples of the base model (0.0126) > negative samples of the base model (0.0125) > negative samples of SEA (0.0119). Compared with the vanilla Transformer, SEA learns the commonality of positive samples, which is shown as the similarity between positive samples is closer, and the similarity between negative samples is farther. This indicates that our model can close the semantic distance between code and summary.

6 Conclusion

In this paper, we proposed a semantically aligned code summary framework with contrastive learning. We designed two novel modules: Semantic Extraction Module and Semantic Align Module. Specifically, the Semantic Extraction Module performed a guided walking algorithm on the code graph to capture co-occurrence nodes and represent the code semantics. Semantic Align Module used contrastive learning to align code and summary semantics on the latent space. Extensive experiments on two benchmark datasets demonstrated the effectiveness of the proposed model, with good semantic comprehension insights.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A simple but tough-to-beat baseline for sentence embeddings. In *International conference on learning representations*.

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

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Xingyu Cai, Jiaji Huang, Yuchen Bian, and Kenneth Church. 2020. Isotropy in the contextual embedding space: Clusters and manifolds. In *International Conference on Learning Representations*.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. *arXiv preprint arXiv:1603.06075*.

Kawin Ethayarajh. 2019. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. *arXiv preprint arXiv:1909.00512*.

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

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824*.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcode-bert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738.

Min Hou, Chang Xu, Zhi Li, Yang Liu, Weiqing Liu, Enhong Chen, and Jiang Bian. 2022. Multi-granularity residual learning with confidence estimation for time series prediction. In *Proceedings of the ACM Web Conference 2022*, pages 112–121.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217.

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Lingpeng Kong, Cyprien de Masson d'Autume, Wang Ling, Lei Yu, Zihang Dai, and Dani Yogatama. 2019. A mutual information maximization perspective of language representation learning. *arXiv preprint arXiv:1910.08350*.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405*.

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

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

David Mimno and Laure Thompson. 2017. The strange geometry of skip-gram with negative sampling. In *Empirical Methods in Natural Language Processing*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*.

Jiaqi Mu, Suma Bhat, and Pramod Viswanath. 2017. All-but-the-top: Simple and effective postprocessing for word representations. *arXiv preprint arXiv:1702.01417*.

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

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

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710.

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1597–1608.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077.

Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zhelin Zhu, and Bin Luo. 2022. Ast-trans: Code summarization with efficient tree-structured attention. In *2022 IEEE/ACM 44th International Conference on Software Engineering(ICSE 2022)*. ICSE.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407.

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. *arXiv preprint arXiv:2205.02029*.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.

Xiangpeng Wei, Heng Yu, Yue Hu, Rongxiang Weng, Weihua Luo, Jun Xie, and Rong Jin. 2022. Learning to generalize to more: Continuous semantic augmentation for neural machine translation. *arXiv preprint arXiv:2204.06812*.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2020. Code summarization with structure-induced transformer. *arXiv preprint arXiv:2012.14710*.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018a. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*, pages 5453–5462. PMLR.

Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. 2018b. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE.

A Appendix

A.1 Gap Between Code and Natural Language

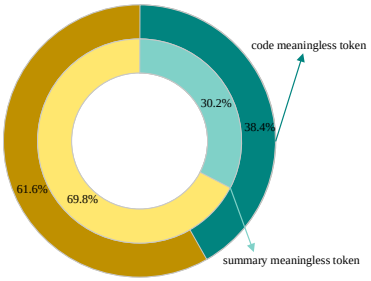

Figure 6: Gap between code and natural language. The outer circle is the corpus of codes, and the inner circle is the corpus of summaries.

We count the proportion of 20 completely meaningless words (e.g., '{' and '}') on the whole code corpus. Similarly, we count the proportion of 179 English stop words that are commonly offered by NLTK[1] on the whole summary. We put detailed trivial words list in Table 3. The results of the ratio of the two are shown in Figure 6. The outer circle is the corpus of codes, and the inner circle is the corpus of summaries. A little meaningless words account for a considerably higher proportion of the code corpus.

A.2 Detailed Procedure for Calculating Hitting Time

Given a connected AST with data flow edges G, we can get its corresponding probability transition matrix B, whose element B_{ij} represents the probability of transitioning from the i-th node to the j-th node. Meanwhile, probability transition matrix B satisfies the spectral radius less than 1 and is a primitive matrix.

[1]https://github.com/nltk

Summary

t	these	having	haven't	weren't	their	then	just	ma	been
which	ourselves	don't	to	nor	aren	what	yourself	ve	won't
no	we	it	is	m	needn	hasn't	whom	our	because
down	you've	needn't	by	themselves	with	d	there	myself	theirs
be	don	had	who	wasn't	are	weren	shouldn't	for	now
you're	when	but	own	you'll	at	hasn	haven	under	through
up	i	wouldn	did	out	over	yourselves	you'd	below	any
do	his	hadn't	off	above	she's	the	herself	than	into
wasn	shan't	and	until	each	ain	not	mightn't	against	can
before	too	an	very	as	on	or	those	other	here
them	should	about	he	so	isn	she	of	why	where
aren't	hadn	am	mustn't	itself	isn't	has	once	such	hers
while	does	it's	again	y	didn't	a	some	ours	after
you	from	re	o	shan	doesn	how	ll	will	they
both	won	during	if	didn	most	few	have	doesn't	couldn
mustn	doing	only	shouldn	being	all	mightn	me	in	were
more	wouldn't	yours	further	himself	my	between	its	her	this
same	couldn't	your	was	that	him	that'll	should've	s	

Code

(	)	.	{	}	#	$	[	]	\
;	:	"	'	?	,	_	`	\t	\n

Table 3: Detailed trivial words list in summary and code respectively.

The formula for calculating the expected of the hitting time H is:

$$(I - B)^{-1} \widetilde{\boldsymbol{f}}(1),$$

where $\widetilde{\boldsymbol{f}}(1) = [1, 1, ..., 1]^T$. The proof is as follows:

We denote the random variable $H_t^{(s)}$ to represent the hitting time from the starting point s to the target point t. First, let us consider the simplest case, the starting point s has only one adjacent node, and this node is the target node t, then the hitting time of the walk is equal to one, that is:

$$P\left(H_t^{(s)} = n\right) = \delta_{1,n}, n \geq 1.$$

where $\delta_{1,n}$ returns 1 if $n = 1$ else 0.

Now we consider the most general case: the starting point does not coincide with the target point, and the target node is not the only neighbor of the target node. For convenience, we record the starting node as i and the target node as t. Specify $i \neq t$. Note that the probability of starting from i and reaching t for the first time after $n \geq 1$ steps is $P(H_t^{(i)} = n)$. Due to the memoryless nature of walks, we can obtain a recursion relation:

$$P\left(H_t^{(i)} = n\right) = \sum_{\substack{i_\alpha = 1 \\ i_\alpha \neq t}}^{m} P(i \to i_\alpha) P\left(H_t^{(i_\alpha)} = n - 1\right), n \geq 2,$$

where i_α represents a neighboring node of node i, $P\left(i \to i_\alpha\right)$ represents the probability of moving from node i to its neighbor i_α, which is B_{ii_α}. Then the above equation can be rewritten as:

$$P\left(H_t^{(i)} = n\right) = \sum_{\substack{j = 1 \\ j \neq t}}^{|\mathbb{V}|} B_{ij} P\left(H_t^{(j)} = n - 1\right), n \geq 2,$$

where $\mathbb{V}$ represents the set of all nodes in the graph G. Since the probability transition matrix has a spectral radius of less than 1 and is a primitive matrix. In this way, we can recursively get the probability distribution of hitting times. Then define the vector

$$\boldsymbol{X}_n^i = P\left(H_t^{(i)} = n\right), n \geq 1,$$

which satisfy the difference equation:

$$\begin{aligned}\boldsymbol{X}_n &= B\boldsymbol{X}_{n-1}, n \geq 2 \\ &= B^{n-1}\boldsymbol{X}_1, n \geq 1.\end{aligned}$$

The probability density function of the random vector X_n is written as:

$$\boldsymbol{f}(x) = \sum_{n=1}^{\infty} \boldsymbol{X}_n \delta(x-n),$$

and its characteristic function is:

$$\begin{aligned}\hat{\boldsymbol{f}}(\omega) &= \int_{x\in\mathbb{R}} \boldsymbol{f}(x) e^{i\omega x} \\ &= \sum_{n=1}^{\infty} \boldsymbol{X}_n e^{i\omega n}, \omega \in \mathbb{R}.\end{aligned}$$

Note that $z = e^{i\omega}$, the characteristic function can be rewritten as the probability generating function as follows:

$$\begin{aligned}\widetilde{\boldsymbol{f}}(z) &= \hat{\boldsymbol{f}}(\omega) \\ &= \sum_{n=1}^{\infty} \boldsymbol{X}_n z^n \\ &= \sum_{n=1}^{\infty} z^n B^{n-1} \boldsymbol{X}_1 \\ &= \left(\sum_{n=1}^{\infty} z^n B^{n-1}\right) \boldsymbol{X}_1 \\ &= z\left(I - zB\right)^{-1} \boldsymbol{X}_1, z = e^{i\omega}, \omega \in \mathbb{R}.\end{aligned}$$

The higher-order differentiation of the probability generating function gives the higher-order moments of the probability distribution of hitting time, while the first-order moments correspond to the expectation of H, respectively namely:

$$\begin{aligned}\mathbb{E}(H) &= \hat{\boldsymbol{f}}'(0) \\ &= \widetilde{\boldsymbol{f}}'(1) \\ &= (I-B)^{-1}\left(B\widetilde{\boldsymbol{f}}(1) + \boldsymbol{X}_1\right) \\ &= (I-B)^{-1}\widetilde{\boldsymbol{f}}(1).\end{aligned}$$

Meanwhile, we used the Monte Carlo method to analogously demonstrate the hitting time, and the results are consistent with the calculation.

A.3 Walking Algorithm

The guided walk algorithm process is shown in Algorithm 1.

Algorithm 1 Co-occurrence Capture Algorithm

Input: Graph $G = (V_c, V_t, E_a, E_d)$, Start node set $S = V_t$, Walk length l
Output: Sampled paths P

```
P = []
π = ModifiedWeights(G) // Eq. 3
G' = (V, E, π)
for s ∈ S do
  path = Walk(G', s, l)
  P.append(path)
end for
return P
```

A.4 Data Statistics

The detailed data analysis results are shown in Table 4. We split CamelCase and snake_case, such as splitting *sum_func* into *sum* and *func*, which can greatly alleviate the OOV problem.

Perspectives	Java	Python
# of Train instances	69708	55538
# of Validation instances	8714	18505
# of Test instances	8714	18502
Avg. tokens per code	120.16	47.98
Avg. tokens per summary	17.73	9.48
Unique tokens of code in train set	73492	94325
Unique tokens of summary in train set	28047	29890
# of intersection of code and summary tokens in train set	9368	11173
Unique tokens of code in dev set	23877	50352
Unique tokens of summary in dev set	9555	15929
# of intersection of code and summary tokens in dev set	3894	6806
Unique tokens of code in test set	23606	49726
Unique tokens of summary in test set	9293	16046
# of intersection of code and summary tokens in test set	3858	6845
Unique tokens of code in data set	82621	123471
Unique tokens of summary in data set	31249	39824
# of intersection of code and summary tokens in data set	10217	13798

Table 4: Statistics of datase