Learning Scalable Representation for Source Code

Anonymous ACL submission

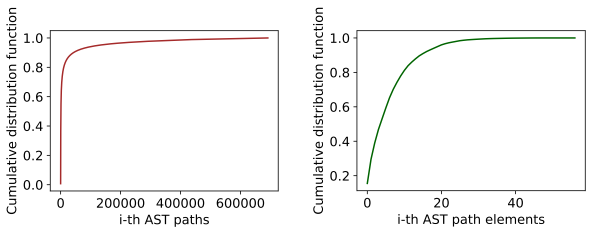
Abstract

This paper presents a **s**calable **d**istributed **c**ode **r**epresentation (SDCR) learning technique, which addresses the most common sparsity and out-of-vocabulary (OoV) concerns simultaneously. We introduce abstract syntax tree (AST) to reflect the structural information of code snippet and adopt the well-recognized 'bag of AST paths' as its intermediate representation, so that the unique structural and syntactic information of programs can be captured. Our proposed SDCR is supported by two core pillars. First, we provide comprehensive empirical study showing that only 1% of the AST paths can account for approximately 75% of the AST path occurrences. That is, dropping most of unnecessary AST paths still allows SDCR to perform well. Second, all AST paths (without leaf nodes in AST) are made up of a limited number of descriptive path elements, for which a lightweight encoder may produce a good embedding of any AST path. Incorporating these two pillars enables us to represent code snippets with better generalizability and scalability. Based on extensive experiments on two real-world datasets, we show that our SDCR have superior performance against the state-of-the-art with nearly 40% reduction in the number of model parameters.

1 Introduction

Code representation learning, as a fundamental technique to support various software engineering tasks, has attracted many recent interests. In this issue, a code snippet is learned to be represented as a low-dimensional distributed vector (a.k.a. *embedding*), and thus the development of machine learning-based software tools can be facilitated.

It is well-known that programming languages are highly structured in contrast to natural languages. So far, approaches to learning distributed code representations are mainly through tokens in source program (Iyer et al., 2016; Feng et al., 2020; Zhang et al., 2021), AST structure (Mou et al., 2016; Allamanis et al., 2018; Chen et al., 2018; Zhang et al., 2019; Wang et al., 2021; Bui et al., 2021; Guo et al., 2021), and paths in AST (Alon et al., 2018, 2019b,a; Zhang et al., 2021). Indeed, substantial empirical study has showed that a family of AST path-based (e.g. root-to-leaf paths and leaf-to-leaf paths) representations are more effective in many program prediction tasks (Alon et al., 2018). Compared with plain text and simple AST structure, the AST paths can dig deeper into program syntax and reveal more semantic information.

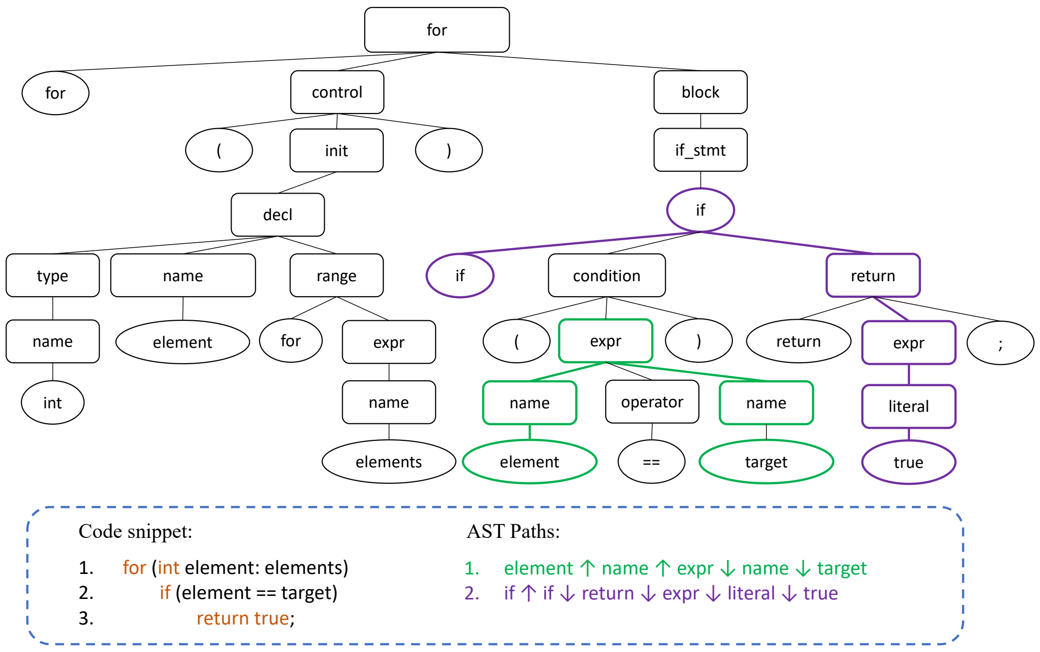


(a) AST paths (b) AST path elements

Figure 1: Cumulative distribution function (CDF) of AST path occurrences and AST path element occurrences.

However, while offering those promising characteristics, the prior AST path-based learning techniques still encounter two major obstacles that hinder their generalizability and scalability:

(1) Unlike natural languages where out-of-vocabulary (OoV) words are relatively limited, programming languages have increasing number of OoV AST paths as the data size grows, leading to serious noise problem to the representation models. Additionally, AST path occurrences are immensely sparse at most cases. Figure 1(a) illustrates this by showing that the CDF of AST path occurrences goes up sharply at first and then quickly converges. Nevertheless, those OoV and sparse AST paths are always not trivial and carry unique information of



Code snippet:

```
1.    for (int element: elements)
2.        if (element == target)
3.            return true;
```

AST Paths:

1. element ↑ name ↑ expr ↓ name ↓ target
2. if ↑ if ↓ return ↓ expr ↓ literal ↓ true

Figure 2: Example of a code snippet, and its AST as well as AST paths.

a code snippet intuitively.

(2) The number of paths in AST are explosive. For example, an AST with n leaf nodes will have $O(n^2)$ leaf-to-leaf paths, resulting in infeasible traversal for the previous AST path-based models. Although abstraction and downsampling are proposed to limit the number of AST paths (Alon et al., 2018), the signals of many useful AST paths will still not be captured.

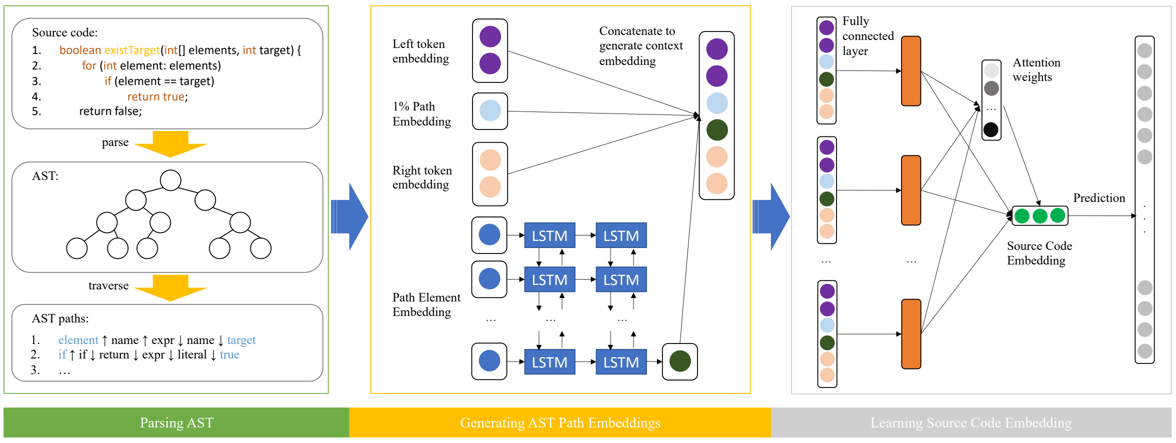
Therefore, similar to research issues in the other machine learning fields (He et al., 2021), a scalable learner for programming languages is also very much needed. In order to precisely understand and represent the source code, it is necessary to utilize the lost information caused by undesirable sparse and OoV AST paths. Figure 1(a) also shows our findings that only 1% of AST paths account for around 75% of occurrences, which is far more than 80/20 rule. This indicates that only 1% of AST paths may enable the representation model to perform well, resolving the problem of exploding number of AST paths. Furthermore, it is worth noting that AST (without leaf nodes in AST) is only composed by limited number of descriptive *path elements* (usually less than 100), such as <if_stmt>, <block_content>, <break> and <return>. An appropriate use of those path elements may mitigate the OoV AST path problem, and boost the quality of code embeddings. Besides, AST path element occurrences are not as sparse as AST paths, as depicted in Figure 1(b). As a matter of fact, the minimum number of AST path element occurrences in most cases are still more than 100, which can greatly enhance the quality of model training. This is partially inspired by the similar sparsity problem in information retrieval, where the factorization machine divides the second-order factors into two dense vectors so that their model can be well-trained (Rendle, 2010).

Based on the key ideas, we present SDCR, a *straightforward*, *effective* and *scalable* code representation learning technique. SDCR can rise to the challenge of the most common sparsity and OoV problems when generating code embeddings, and meanwhile enjoy a significant reduction in the number of model parameters. We evaluate our proposed model design on two real-world datasets and the experiment results show its superiority in effectiveness and efficiency over the state-of-the-art.

2 Representing Code as AST Paths

Programming languages are highly structured, requiring a good data structure to reflect their meaning. The abstract syntax tree (AST) is a well-recognized intermediate representation of the source code, and has shown to be useful in many previous works (Raychev et al., 2015; Gupta et al., 2019).

Figure 2 elaborates an example of a Java code snippet and its corresponding AST. The abstract syntax tree (AST) is a structure to represent the abstract syntactic structure of code in a specific



Source code:

```
boolean existTarget(int[] elements, int target) {
    for (int element: elements)
        if (element == target)
            return true;
    return false;
```

AST paths:

1. element ↑ name ↑ expr ↓ name ↓ target
2. if ↑ if ↓ return ↓ expr ↓ literal ↓ true
3. …

Figure 3: Overview of proposed SDCR.

programming language. In AST, the leaf nodes represent constant, identifiers and names, often provided by programmers, while the non-leaf nodes represent the grammar or the structure of the programming language. All non-leaf nodes except root node have their descendants and ancestors, portraying the syntax of the code snippet. The mutual conversion between a code snippet and its AST is supported, indicating that there is no information loss during the conversion. By learning representations from AST, the syntax and semantics of code snippets can be easily captured, which unveils the deeper information beyond the plain text.

However, either graph neural networks (GNNs) methods, such as GCN (Kipf and Welling, 2016) and GraphSAGE (Hamilton et al., 2017), or tree-based learning techniques, such as TBCNN (Mou et al., 2015, 2016) and TBLSTM (Chen et al., 2018) does not work well for learning AST. AST has unique structural information where its leaf nodes are tokens and non-leaf nodes are descriptive path elements, weakening their capability to represent program languages. Thus, AST leaf-to-leaf path-based methods are introduced in this paper and an example AST path of the code snippet in Figure 2 can be formulated as follows:

$$\text{element} \uparrow \text{name} \uparrow \text{expr} \downarrow \text{name} \downarrow \text{target} \tag{1}$$

where the first and last items are leaf nodes in AST, representing tokens, and in-between items are non-leaf nodes in AST, representing descriptive path elements. $\uparrow$ and $\downarrow$ are the direction of traversal. More formally, a leaf-to-leaf AST paths can be represented as a triplet:

$$< x^s, p, x^t > \tag{2}$$

where x^s and x^t are the start and end leaf nodes of AST paths, respectively, and p is all non-leaf nodes in the traversal path from x^s to x^t.

3 Proposed Method

Figure 3 shows the overview of our proposed SDCR. At first, a source code snippet is converted into AST, and a bag of AST paths are sampled randomly to capture the information of AST sufficiently. This is different from previous code2vec (Alon et al., 2019b), where AST paths are sampled in the limitation of their maximum length and width. Then, the initial AST path embeddings are queried from the embedding layers. To deal with sparsity and OoV problem, we synthesize path elements by a neural network to produce an additional AST path embedding. Next, we conduct concatenation and linear transformation to generate a bag of AST path embeddings and feed them into an attention network to obtain the overall code snippet representation. Finally, a prediction task of function name is performed so that our proposed SDCR can be trained.

3.1 Code as a Bag of AST Paths

Before learning the distributed source code representation, we introduce 'bag of AST paths' model as the intermediate representation of code snippets.

We denote $\mathcal{C}$ as a code snippet, $\mathcal{T}(\mathcal{C})$ as its AST and $\mathcal{R}(\mathcal{C})$ as its intermediate representation. Formally, given a code snippet $\mathcal{C}$, we utilize an AST parser to transform $\mathcal{C}$ to $\mathcal{T}(\mathcal{C})$. Further, the set of all pairs of the leaf nodes is denoted as:

$$\mathcal{TP}(\mathcal{C}) = \left\{ (u, v) \;\middle|\; \begin{array}{l} u, v \in \mathcal{T}(\mathcal{C})\text{'s} \\ \text{leaves} \wedge i \neq j \end{array} \right\} \tag{3}$$

With $\mathcal{TP}(\mathcal{C})$, the intermediate representation of $\mathcal{C}$ that can be derived from it:

$$\mathcal{R}(\mathcal{C}) = \{(x^s, p, x^t) \mid \exists (x^s, x^t) \in \mathcal{TP}(\mathcal{C})\} \quad (4)$$

where we use $\exists$ in $\mathcal{R}(\mathcal{C})$, in which case the random downsampling is applied to construct $\mathcal{R}(\mathcal{C})$ so that the maximum number of AST paths can be limited. Therefore, a code snippet is transformed into a bag of AST paths, which is a mathematical object that can be used in a learning model.

3.2 Path Embedding Generation

Once we obtain the intermediate representation of a code snippet, we embed them into a low-dimensional distributed space. Since an AST path can be represented as a triplet, we divide the embedding module into three parts - start token embedding, in-between path embedding and end token embedding.

For start and end token embedding, we define a learnable matrix $W^{token} \in \mathbb{R}^{(|X^{token}|+1)\times d^{token}}$ as the shared token vocabulary, where X^{token} are the set of tokens appeared in training corpus and '+1' represents OoV tokens. By querying W^{token}, the start and end token embedding of i-th AST path can be initialized as $h_i^s, h_i^t \in \mathbb{R}^{d^{token}}$ directly. Formally, the start and end token embedding of i-th AST path are calculated as:

$$\begin{aligned} h_i^s &= W_{j_s}^{token}, \quad h_i^s \in \mathbb{R}^{d^{token}} \\ h_i^t &= W_{j_t}^{token}, \quad h_i^t \in \mathbb{R}^{d^{token}} \end{aligned} \quad (5)$$

where j_s and j_t are the dictionary index of start token and end token.

For in-between path embedding, we should consider sparsity and OoV problem caused by various data size and countless AST paths. As our empirical findings mentioned in Section 1, we have two pillars to support the solution of problems.

The first pillar is that only 1% of AST paths make up approximately 75% of AST path occurrences. Followed by this, we introduce a learnable matrix $W^{path} \in \mathbb{R}^{(|X^{path}|+1)\times d^{path}}$ where X^{path} denotes the set of AST paths in the top 1% of occurrences and '+1' represents OoV AST path. As a result, not only is the sparsity problem solved, but also the model enjoys a large reduction in the number of model parameters. The top 1% based in-between path embedding of i-th AST path thus can be calculated as:

$$h_i^{1\%} = W_{j_p}^{path}, \quad h_i^{1\%} \in \mathbb{R}^{d^{path}} \quad (6)$$

where j_p is the dictionary index of in-between path.

The second pillar is that all of in-between paths are composed by very limited number of descriptive path elements. Consequently, by learning a combination model, the embedding of OoV and abandoned in-between path can be generated. As in-between path is a kind of typical sequential data, we use LSTM (Hochreiter and Schmidhuber, 1997) to produce path element based in-between path embedding of i-th path as:

$$\begin{aligned} &h_i^{pe} = LSTM(\{pe_{i1}, pe_{i2}, \ldots, pe_{im}\}), \\ &h_i^{pe} \in \mathbb{R}^{d^{pe}} \end{aligned} \quad (7)$$

where m is the length of path elements, pe_{ik} is the k-th path element of i-th AST path and $LSTM()$ outputs its final hidden state.

Then, the four embeddings of i-th AST path are concatenated to a single vector $h_{i,a} \in \mathbb{R}^{2d^{token}+d^{path}+d^{pe}}$ that represents that AST path:

$$\begin{aligned} \tilde{h_i^a} &= embedding(< x_i^s, p_i, x_i^t >) \\ &= [h_i^s, h_i^{1\%}, h_i^{pe}, h_i^t], \\ \tilde{h_i^a} &\in \mathbb{R}^{2d^{token}+d^{path}+d^{pe}} \end{aligned} \quad (8)$$

Since every AST path vector $\tilde{h_i^a}$ is formed by a concatenation of four independent embeddings, we adopt a fully connected layer to combine its components. This is done separately for each AST path and the computation can be described as:

$$h_i^a = tanh(W^{linear} \cdot \tilde{h_i^a}) \quad (9)$$

where $W^{linear} \in \mathbb{R}^{d\times(2d^{token}+d^{path}+d^{pe})}$ is a learnable weight and hyperbolic tangent function $tanh$ is used as the activation function.

3.3 Attention Aggregation Model

Given a bag of AST path embeddings, we apply attention mechanism to aggregate them since the length of bag is varying. The attention mechanism computes a scalar weight over each AST path embeddings, indicating which AST path is relatively important compared to the other. It is calculated as the normalized inner product between the combined path embeddings and the global attention vector $\boldsymbol{a} \in \mathbb{R}^d$:

$$\alpha_i = \frac{exp(h_i^{aT} \cdot \boldsymbol{a})}{\sum_{k=1}^{q} exp(h_k^{aT} \cdot \boldsymbol{a})} \quad (10)$$

where q is the number of AST paths. The exponential components are used to make the attention weights positive, and they are divided by their sum to have a sum of 1.

The aggregated vector $h^c \in \mathbb{R}^d$ represents a code snippet. It is a linear combination of the bag of AST path embeddings $\{h^a_1, h^a_2, ..., h^a_q\}$ factored by their attention weights:

$$h^c = \sum_{k=1}^{q} \alpha_k \cdot \tilde{h^a_k} \tag{11}$$

In this way, the low-dimensional distributed code representation can be acquired.

3.4 Optimization

Prediction of function names is performed using the code vector. This is because function names are often descriptive and provide a high-level summary of its purpose. Choosing good names are especially critical for functions in public project APIs, as poor function names can doom a project to irrelevance (Høst and Østvold, 2009).

We define a learnable matrix $W^{tag} \in \mathbb{R}^{|Y| \times d}$ as the tag vocabulary, where $|Y|$ is the set of tag values found in the training corpus. Indeed, each row vector in W^{tag} can be regarded as the tag embedding, which is jointly trained with code vectors. The predicted probability distribution is computed as the (softmax-normalized) dot product between the code vector h^c and each of the tag vectors:

$$q(y_j) = \frac{exp(W^{tag}_j \cdot h^c)}{\sum_{y_k \in Y} exp(W^{tag}_k \cdot h^c)} \tag{12}$$

We use cross-entropy loss between the predicted probability distribution p and the ground truth distribution p. It can be expressed as:

$$\mathcal{H}(p||q) = -\sum_{y \in Y} p(y) \log q(y) \tag{13}$$

Finally, we need to learn these parameters of SDCR: W^{token}, W^{path}, W^{tag}, W^{linear}, $\boldsymbol{a}$ and parameters in $LSTM$.

4 Experiment

In this section, we aim to address the following research questions: (1) *Will SDCR indeed outperform the AST-path based baselines in effectiveness and efficiency?* (2) *What is the quality of code vectors learned by SDCR?* (3) *Why does SDCR perform well in predicting the semantics of programs?*

Dataset	Java Small	Java Large
Training	14,998	141,308
Validation	5,930	51,529
Testing	6,030	50,548
Num. paths	243,297	911,417
Num. path elems	54	61
Avg. path length	22.50	22.81
Max. path length	93	155

Table 1: Statistics of Java Small and Java Large dataset.

4.1 Setup

Datasets We evaluate our model on two real-world Java datasets - a small dataset containing only 1 Java project (intra-project) and a large dataset containing 10 Java projects (inter-project). All of projects are collected from the most starred Java projects in GitHub. We extract functions from those projects and randomly divide them into three sets in the ratio of 6:2:2 for training, validation and testing, respectively. Table 1 shows the statistics of the experiment datasets. Besides, there are many AST parsers for Java, such as JavaParser[1] (Hosseini and Brusilovsky, 2013) and srcML[2] (Collard et al., 2013). In this work, we use srcML to convert source code to AST.

Evaluation Metrics Ideally, we prefer to evaluate the results manually since programming languages have deep semantic information. However, given that manual evaluation is hard to scale, we adopt the measures used in the previous works (Allamanis et al., 2015, 2016; Alon et al., 2018, 2019b), in which precision, recall and F1 score are measured in subtoken level and case insensitive manner. The core idea is that the quality of a function name prediction relies mainly on the subtokens used to compose it. For instance, for a function name called $is\_contain\_element$, we also consider a prediction of $is\_element\_contain$ are the exact match in subtoken level measures, $contain\_element$ has full precision but low recall, and $is\_integer\_element\_contain$ has full recall but low precision.

Competing Methods We compare our *SDCR* with the most recognized AST path-based model, *code2vec* (Alon et al., 2019b), as well as its variant *code2vec_most* that only uses the AST paths in the top 1% of occurrences. We also evaluate the

[1] http://javaparser.org/
[2] https://www.srcml.org/

Approach	Java Small			Java Large		
	Precision	Recall	F1 score	Precision	Recall	F1 score
code2vec	0.3315	0.3566	0.3436	0.5441	0.5615	0.5526
code2vec_most	0.3346	0.3611	0.3474	0.5468	0.5596	0.5531
pe_mean	0.3290	0.3558	0.3418	0.5415	0.5545	0.5479
pe_attention	0.3319	0.3554	0.3432	0.5548	0.5677	0.5612
pe_lstm	0.3481	0.3755	0.3613	0.5453	0.5612	0.5531
SDCR_mean	0.3403	0.3645	0.3520	0.5413	0.5562	0.5487
SDCR_attention	0.3362	0.3601	0.3478	0.5592	0.5699	0.5645
SDCR_lstm	0.3561	0.3821	0.3687	0.5583	0.5712	0.5647

Table 2: Performance comparison between the competing methods. The cell marked with dark blue achieves the **highest** performance in that column and the cell marked with light blue achieves a **better** performance in that column compared with the most recognized AST path-based baseline *code2vec*.

variants of SDCR, that is, *SDCR_mean* by replacing LSTM module with mean computation unit, *SDCR_attention* by replacing LSTM module with attention mechanism. Furthermore, we conduct ablation experiments on SDCR similarly - we remove the embeddings of top 1% of AST paths $h_i^{1\%}$ and only utilize the embeddings generated by the AST path elements h_i^{pe}. They are named as *pe_mean*, *pe_attention* and *pe_lstm*, respectively. Note that we remove the directional path elements $\uparrow$ and $\downarrow$ for LSTM module as it is a sequence-aware model. The hyperparameters for each model are tuned on validation set for maximizing F1 score and the results on the unseen test set are reported.

4.2 Performance Comparison and Ablation Study (RQ1)

Table 2 shows the overall performance of the models under two real-world datasets. From this table, we observe that our proposed SDCR_lstm significantly outperforms code2vec in F1 score, by 0.0251 on Java Small dataset and by 0.0121 on Java Large dataset. It is noteworthy that SDCR_mean and SDCR_attention achieve promising results in Java Small and Java Large dataset, respectively. A possible reason is that SDCR_attention may trap into the overfitting problem when the training corpus is small. Their advantages in F1 score are also significant, outperforming code2vec by 0.0084 from 0.3436 to SDCR_mean's 0.3520 on Java Small dataset and by 0.0119 from 0.5526 to SDCR_attention's 0.5645 on Java Large dataset.

Besides, the ablation experiments are also interesting. For the ablation model of code2vec, code2vec_most outperforms code2vec in all metrics except recall on Java Large dataset. This is

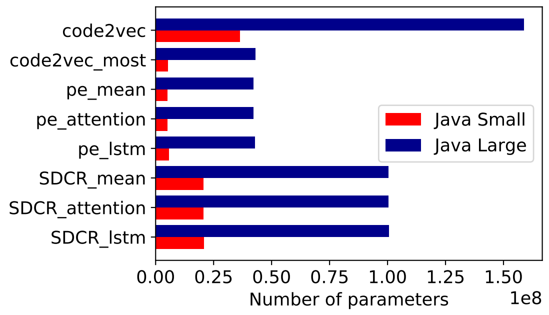
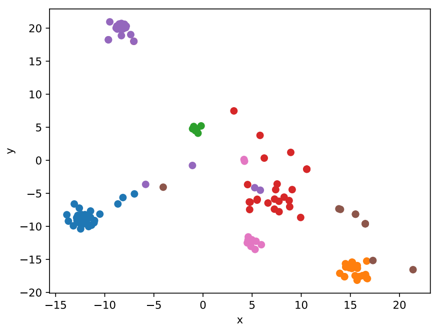


Figure 4: Number of parameters used in each model.

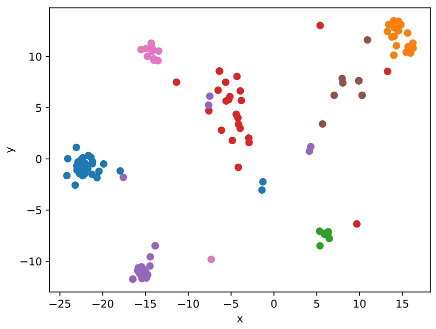
because code2vec_most only uses the AST paths in top 1% of occurrences, easing the sparsity problem that exists in code2vec. For the ablation models of SDCR, an interesting phenomenon is that the advantage of pe_lstm is not significant over code2vec on Java Large dataset, while SDCR_lstm achieves the highest F1 score compared with the others. This indicates that all modules in the core design of SDCR are important. They work together to address the common sparsity and OoV problems.

Figure 4 shows the number of parameters used in each model, which can be roughly divided into three levels - code2vec with the most number of model parameters, SDCR_mean, SDCR_attention and SDCR_lstm with moderate number of model parameters (around 40% reduction), as well as code2vec_most, pe_mean, pe_attention and pe_lstm with the least number of model parameters (around 85% reduction). It can be observed that the number of model parameters are significantly reduced by SDCR compared with the baseline on either Java Small or Java Large dataset.

Therefore, our first research question can be ad-



(a) code2vec



(b) SDCR

Figure 5: Visualization of code vectors from 7 classes on Java Small dataset produced by code2vec and SDCR.

dressed: SDCR has superiority in generalizability and scalability - it achieves the highest F1 score while using relatively fewer model parameters.

4.3 Visualization of Code Representations (RQ2)

To help understand why the code embeddings produced by SDCR are better than the code embeddings produced by the baseline, we visualize the code embeddings from 7 random classes on Java Small dataset. We adopt t-SNE (Van der Maaten and Hinton, 2008) to embed code embeddings into two-dimensional space and visualize them in that space. As shown in Figure 5, it points out that (1) The vectors produced by SDCR of the same class stick closer than the vectors produced by code2vec. For example, the code vectors marked with red color locate more tightly in SDCR while diffuse in code2vec. The code vectors marked by brown color in SDCR have clearer boundary and are easier to distinguish than in code2vec. (2) Compared to code2vec, SDCR takes advantage of a larger vector distribution area. That is, the code vectors of SDCR are distributed more evenly in the space, yet code2vec only utilizes the lower triangular space. However, we observe that some code vectors marked in the same color (e.g. pink) are somewhat far away from each other in SDCR. This is partially because the handcrafted function names cannot accurately represent the semantics of program, which could indicate further improvement to SDCR and can be made in the future work.

As a result, our second research question can be addressed: the code vectors learned by SDCR are distinguishable and have good expressive quality.

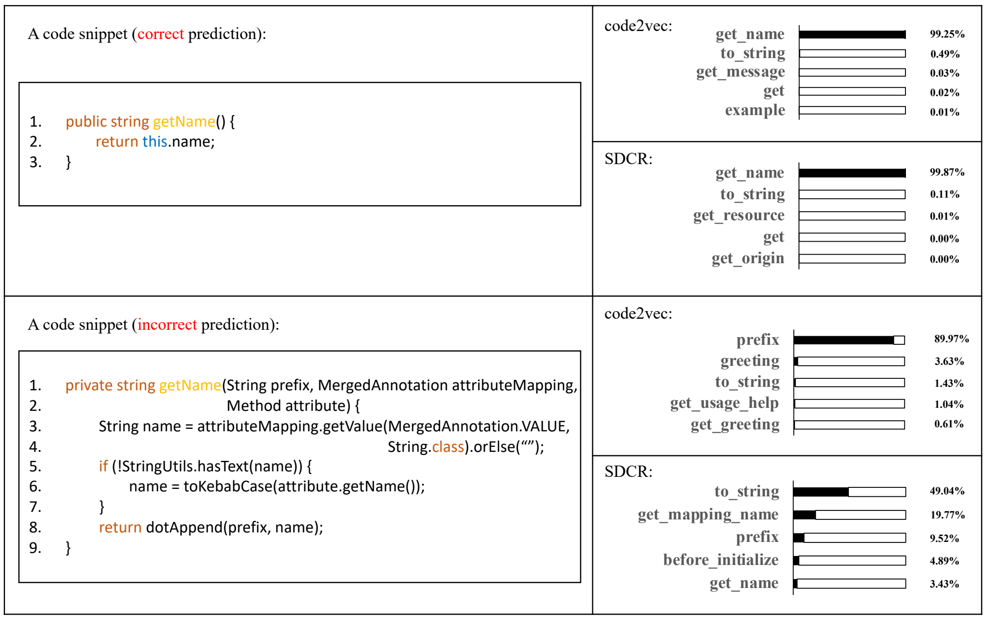
4.4 Qualitative Analysis (RQ3)

We conduct case study to further investigate how do the models behavior in function name prediction task. As shown in Figure 6, we randomly pick a function name (get_name in this experiment) to predict, along with an example of a correct prediction and an example of an incorrect prediction. We standardize all the function names to case-insensitive underscore nomenclature, and five names with highest predicted probability are reported. From *Case 1*, it is clear to see that both SDCR and code2vec predict correctly for the simple code snippet. However, compared to code2vec, SDCR has a higher confidence in the correct answer get_name while lower probability in the incorrect answers. Besides, all five function names predicted by SDCR are related to the correct answer, yet code2vec has a completely unrelated function name $example$. In *Case 2*, both SDCR and code2vec give an incorrect prediction. However, code2vec is very confident (with 89.97% probability) of an incorrect prediction of $prefix$, but SDCR behaviors not confidently (with 49.05% probability) in predicting an incorrect function name to_string. Additionally, SDCR generates two very related function names $get\_mapping\_name$ and get_name compared to code2vec. The reason why SDCR favors $get\_mapping\_name$ is that a token named 'attributeMapping' exists in the code snippet. In fact, $get\_mapping\_name$ is to some extent a better function name than get_name, which suggests an possible application of SDCR in helping programmers to name a function's code snippet.

Based on the above results, our third research question can be addressed: SDCR has better understanding and digs deeper into program semantics.

5 Related Work

With the development of deep learning techniques, researches for source code representations have attracted much attention. In general, existing code


A code snippet (correct prediction):

```
public string getName() {
    return this.name;
}
```

A code snippet (incorrect prediction):

```
private string getName(String prefix, MergedAnnotation attributeMapping,
                       Method attribute) {
    String name = attributeMapping.getValue(MergedAnnotation.VALUE,
                                            String.class).orElse("");
    if (!StringUtils.hasText(name)) {
        name = toKebabCase(attribute.getName());
    }
    return dotAppend(prefix, name);
}
```

Figure 6: Example of prediction results produced by code2vec and SDCR.

representation works are mainly in three ways:

Token-based Embedding This line of work simply regards the source code snippet as plain text and tokenize it into token sequences. (Iyer et al., 2016) firstly defined code summarization task and proposed a RNN model to translate the source code snippet to summary. Followed by this work, (Allamanis et al., 2016) described an attentional CNN model to summarize source code. (Feng et al., 2020) trained CodeBERT on a tremendous code corpus, aiming to advance code embedding by the pre-trained model. However, it is obvious that the syntax of source code can not be captured by token-based representation models.

AST-based Embedding To take advantage of highly structured features of source code, AST is introduced as the code snippet's intermediate representation. It can be directly represented via Tree-CNN (Mou et al., 2015, 2016), Tree-LSTM (Chen et al., 2018) and ASTNN (Zhang et al., 2019). Then, (Allamanis et al., 2018) proposed to add edges to AST, and use GNN to generate code embedding. (Wang et al., 2021) extended the type of edge so that their intermediate representation is represented as a heterogeneous graph. (Bui et al., 2021) adopted self-supervised learning technique to predict the subtree of AST. (Guo et al., 2021) incorporated the data-flow information among variables into pre-training and enhance their previous CodeBERT to GraphCodeBERT. Nevertheless, simply learning code representations by AST structures will ignore the unique syntax that all leaf nodes of AST are constants, handcrafted tokens, or identifiers, and all non-leaf nodes are predefined descriptive elements.

AST path-based Embedding Therefore, the third line of work are becoming very hot topic recently. (Alon et al., 2018) were the first to validate the superior performance of AST paths based on abundant empirical experiments. Then, he further extended his work to well-known code2vec (Alon et al., 2019b) and code2seq (Alon et al., 2019a), which are used as substantial baseline in this paper. (Hu et al., 2018) described a comment generation method by linearizing the AST into a sequence of nodes via traversing. However, all of them less considered the challenge in sparsity and OoV problem, which is worth further exploration and practice.

6 Conclusion

In this paper, we propose SDCR, a novel scalable representation learning technique for source codes. We adopt a bag of AST paths as the intermediate representation of code snippet, and couple the information of descriptive path elements and AST paths in the top 1% of occurrences, mitigating the common sparsity and OoV problems in AST path based model. An comprehensive empirical study is reported to support our core design. By conducting extensive experiments on two real-world datasets, we verify that our SDCR outperforms the state-of-the-art in effectiveness and efficiency.

References

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of ESEC/FSE*, pages 38–49.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Proceedings of ICLR*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of ICML*, pages 2091–2100.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *Proceedings of ICLR*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of PLDI*, pages 404–419.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: Learning distributed representations of code. In *Proceedings of POPL*, pages 1–29.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *Proceedings of ICSE*, pages 1186–1197.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of NeurIPS*.

Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of ICSM*, pages 516–519.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Min Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of EMNLP*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of ICLR*.

Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural attribution for semantic bug-localization in student programs. In *Proceedings of NeurIPS*.

William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of NeurIPS*, pages 1025–1035.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Doll'ar, and Ross Girshick. 2021. Masked autoencoders are scalable vision learners. *arXiv preprint arXiv:2111.06377*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Roya Hosseini and Peter Brusilovsky. 2013. Javaparser: A fine-grain concept indexing tool for java problems. In *Proceedings of CEUR-WS*, volume 1009, pages 60–63.

Einar W Høst and Bjarte M Østvold. 2009. Debugging method names. In *Proceedings of ECOOP*, pages 294–317.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of ICPC*, pages 200–20010.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of ACL-IJCNLP*, pages 2073–2083.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. In *Proceedings of ICLR*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of AAAI*.

Lili Mou, Hao Peng, Ge Li, Yan Xu, Lu Zhang, and Zhi Jin. 2015. Discriminative neural sentence modeling by tree-based convolution. In *Proceedings of EMNLP*.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "big code". In *Proceedings of POPL*, pages 111–124.

Steffen Rendle. 2010. Factorization machines. In *Proceedings of ICDM*, pages 995–1000.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(11).

Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. 2021. Learning to represent programs with heterogeneous graphs. In *Proceedings of ICLR*.

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 *Proceedings of ICSE*, pages 783–794.

Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu, and Yulei Sui. 2021. Disentangled code representation learning for multiple programming languages. In *Proceedings of ACL-IJCNLP*, pages 4454–4466.