Utilizing Abstract Syntax Tree Embedding to Improve the Quality of GNN-based Class Name Estimation

Hiroto MAMBA†, Yasuhiro HAYASE†, and Toshiyuki AMAGASA†

† University of Tsukuba
1–1–1 Tennodai, Tsukuba, Ibaraki 305–8577, Japan
E-mail: †mamba@kde.cs.tsukuba.ac.jp, ††{hayase,amagasa}@cs.tsukuba.ac.jp

Abstract While giving comprehensible names to identifiers is essential in software development, it is sometimes difficult since it requires development experience and knowledge of the application domain. Among work to support the developer's identifier naming, a GNN-based class name estimation approach learns a graph of relationships between program elements, i.e., classes, methods, and fields, but it ignores information within the methods. This study proposes an approach that exploits information from method bodies, which can help estimate correct class names. The proposed approach extends the existing GNN-based approach to use embeddings of the corresponding ASTs for method nodes. An evaluation experiment measures how correctly the proposed approach can estimate class names in large datasets of open-source Java projects. The experimental result shows that the proposed approach improves the estimation correctness compared to the existing approach.

Key words software engineering, graph neural network, abstract syntax tree, class name estimation

1 Introduction

In software development, giving comprehensible names to identifiers is essential for developers' program understanding. When developers understand programs written before, they often try to obtain application domain knowledge from the identifier names of program elements such as variables, methods, and classes. [1] Especially when a program has complex source codes or whose documentation is outdated or not helpful, the names are important sources of knowledge. [2] If an identifier has a meaningless name or one identical name is used for different concepts, those names cause difficulties in program understanding. [3] Since a study indicates that developers spend 60% of their working time on program understanding [4], the names should be easy to understand the functions and roles.

However, sometimes it is difficult to give a good name to a program element. Good naming requires knowledge about the element's role in the project [3] or what names appear elsewhere [5]. Moreover, a name may become inappropriate as the system evolves. [3]

Among work to support the developer's identifier naming, an approach [6] for class name estimation learns the relationship between program elements. They modeled these relationships into a graph with nodes representing classes, fields, and methods. Although this graph can exploit complex classes' nature, it ignores most of the information that each node itself has, which may be helpful to the estimation.

This study proposes an approach to estimate class names using relationships and the information within method nodes. Since machine-learning work for software programs often utilizes AST [7–9], it may be helpful to represent the nodes' properties. While this approach is based on the existing GNN-based approach [10], it extends the existing approach to utilize embeddings of methods' ASTs.

An experiment investigates the contribution of the proposed approach by measuring the estimation correctness using large datasets of open-source Java projects. The experimental result that the proposed approach can estimate the class names more correctly compared to the existing one shows that the AST helps to represent the information within the method nodes.

The remaining part of this paper consists as follows. Section 2 describes prior work in machine learning for program code and class name estimation. Section 3 describes preliminary used as components of the proposed method. Section 4 describes the proposed method. Section 5 describes the experiments for the proposed method. Finally, Section 6 concludes this paper.

2 Related work

This section introduces prior work in machine learning for program code. Section 2. 1 describes the general approaches for various tasks; then, Section 2. 2 describes the existing approaches to learning class entities.

2. 1 Machine learning approaches for software programs

Recently, various studies have applied machine learning to source code for tasks such as method naming [7–9, 11–14], code summarization [15–19], method name consistency checking [20], type inference [21], and code translation [22,23].

Approaches that follow architectures for neural machine translation [24,25] learn source code as a sequence of tokens to generate method name [14] or method summary [15].

On the other hand, various approaches exploit specific features of source code, such as *abstract syntax trees* (AST) or method call dependencies. Alon et al. [7,11,26] and Peng et al. [27] treat an AST as a set of paths between two leaf nodes. Zhang et al. [28] and Lin et al. [29] treat it as a set of subtrees. Hu et al. [16,17] treats it as one sequence by structure-based traversal. GNN-based approaches [8,9,30] treat an AST as a graph with additional edges and learn it with graph neural networks. Besides, some approaches learn dependencies between elements that appear in source code, such as methods' call dependencies [5,31,32] or data flow [9,23,33].

2.2 Approaches for class entities

As for class name estimation, Kurimoto et al. [6] proposed an approach that learns the relationship between program elements to estimate class names by graph embedding. This relationship is represented as a graph with nodes of classes, methods, and fields. This approach can estimate a class name even before the class is used from other code, in contrast to another approach [34] that learns the context where the class is used.

Subsequently, a work [10] proposed an approach that learns the same graph using GNN. In contrast to Kurimoto et al., where memory usage depends on the graph size, this approach can learn from a large dataset because it calculates embedding representations for each word rather than the whole graph node.

As for utilizing ASTs, Compton et al. [35] obtained an embedding representation of the whole class. This approach first calculates embedding for each method in a Java file using code2vec [11], and then calculates entire class embedding by applying an aggregation function, such as summation and median. The author uses this obtained class embedding for code classification tasks.

3 Preliminary

This section describes methods used in the proposed method described in Section 4.

3.1 code2seq

code2seq [7] is an approach for method name estimation, which exploits the abstract syntax tree (AST) of methods. This estimation model takes a source code of method definitions as input and outputs candidates of methods' names.

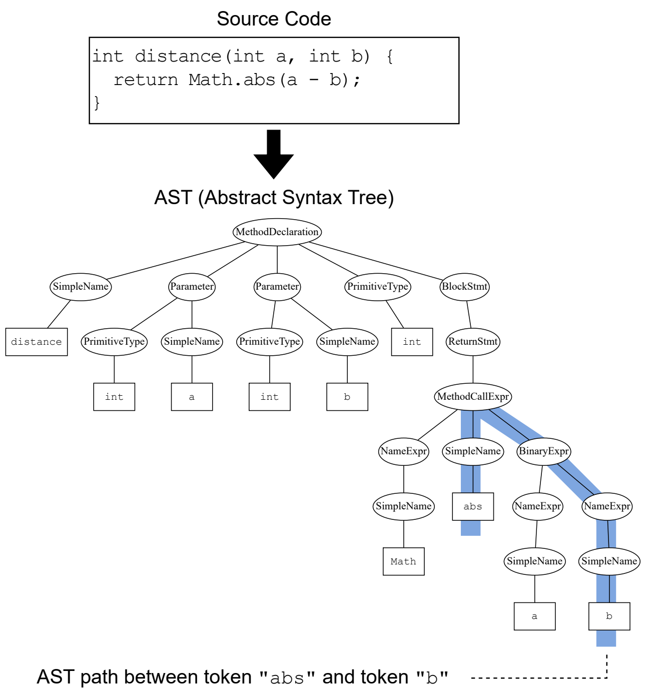
The code2seq treats an input method as a set of paths called *AST path* at the beginning of the estimation. AST paths are considered between any two leaf nodes in the AST, and each AST path provides information about a sequence of nodes on the path and tokens corresponding to two leaf nodes (illustrated in Figure 1).

The code2seq calculates feature vectors for AST paths as follows. Suppose that an AST path is represented as $\text{AST-Path}_i = \langle s_{i1}s_{i2}\cdots, v_{i1}v_{i2}\cdots, e_{i1}e_{i2}\cdots\rangle$. Here, $v_{i1}v_{i2}\cdots$ is a sequence of nodes on the AST path, and $s_{i1}s_{i2}\cdots$ and $e_{i1}e_{i2}\cdots$ indicate words that make up tokens corresponding to two leaf nodes, respectively, called *subtokens*. The code2seq model calculates a feature vector $\boldsymbol{z}_i$ for AST-Path_i using embeddings for the subtokens $E_{\text{subtoken}}(s_{i1}), E_{\text{subtoken}}(s_{i2}), \cdots$, $E_{\text{subtoken}}(e_{i1}), E_{\text{subtoken}}(e_{i2}), \cdots$ and embeddings for the nodes $E_{\text{node}}(v_{i1})$, $E_{\text{node}}(v_{i2}), \cdots$. Here, $E_{\text{subtoken}}(\cdot)$ and $E_{\text{node}}(\cdot)$ are operations that look up a subtoken embedding vector and a node embedding vector, respectively. Then, the model averages the feature vectors of all AST paths, which is used for the subsequent generation of a method name. The explanation in Section 4.1.2 denotes the operation up to this point as code2seq_encoder(AST), that is, the operation from inputting an AST of a method to calculating this averaged vector.

The code2seq has an advantage in that it can easily consider code snippets with different syntactic structures but similar semantics as similar ones. This is because the code2seq holds embedding vectors for each node in an AST path and calculates the feature vector of the AST path from them, in contrast to having embedding vectors for only the entire AST path [11].

3.2 Graph neural network

Graph neural networks (GNN) are neural networks that take graph structures as input and learn them. As a one of GNN, graph convolutional network (GCN) [36] convolutes features of neighboring nodes of each node and generates new features. By repeating the convolution multiple times, it is possible to convolute features of nodes that are more than one step away from each other. When $H^{(0)} = \left(\boldsymbol{h}_1^{(0)}, \boldsymbol{h}_2^{(0)}, \cdots, \boldsymbol{h}_N^{(0)}\right)$ and $A \in \mathbb{R}^{N\times N}$ represent initial node features and the adjacency matrix of a graph with N nodes, respec-



```
int distance(int a, int b) {
  return Math.abs(a - b);
}
```

Figure 1 An example of an abstract syntax tree (AST) and an AST path.

tively, this convolution is formulated as follows:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

where $I_N \in \mathbb{R}^{N\times N}$ is an identity matrix, $\tilde{A} = A + I_N, \tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, and $W^{(l)}$ is a weight matrix for l th convolution step.

3.3 GNN-based class name estimation

A work [10] has proposed an approach to estimate class names, which learns relationships between program elements by GNN. Figure 2 illustrates this approach. This approach takes a class's source code as input from a developer, then estimates candidates for the name of the class. This estimation model first constructs a graph representing the program elements' relationships. Then a GNN calculates the feature vector for each node in the graph, considering the relationships between nodes. Finally, the model generates a name from the calculated feature vector.

In the following explanation, a class whose name is to be estimated is called *target class*.

a) Relation graph

This approach represents the relationships between program elements such as classes, methods, and fields as a graph used by Kurimoto et al. [6] (later referred to as a *relation graph*). This graph contains edges for the following types of relationships between the program elements:

- Class inheritance, where one class extends another class.
- Ownership from a class to a method or a field.
- A relationship where one method calls another.
- An access relationship from a method to a field.
- A relationship from a method to the class of its return type.
- A relationship from a field to the class of the field's type.

When a developer inputs a code of a target class, the estimation model first constructs this relation graph. To include graph nodes for program elements outside the code of the target class, the model takes not only the code of the target class but also codes referenced by the target class.

In the following explanation, let $G = (V, E)$ be the constructed relation graph, where $V = \{v_1, v_2, \cdots\}$ is a set of nodes corresponding to the program elements, and $E = \{e_1, e_2, \cdots\}$ is a set of edges corresponding to the relations. $v_t \in V$ is the node corresponding to the target class. Besides, denote the number of nodes in G by $|V|$.

b) Obtaining initial node features

Before applying GNN, the model assigns a vector to each node, called *initial node feature*. For each node v_i, the model calculates an initial feature $\boldsymbol{h}_i^{(0)}$ based on the word sequence of the class name of v_i. The initial features make it possible for GNN to distinguish nodes.

c) Generation of the target class name from initial node features

Given the initial node features of each graph node, GNN obtains each node's new feature, and then an RNN-based generator outputs candidates for the target class name as word sequences. Formally, the GCN (described in Section 3.2) first receives the initial feature $\boldsymbol{h}_1^{(0)}, \boldsymbol{h}_2^{(0)}, \cdots, \boldsymbol{h}_{|V|}^{(0)}$ of each node $v_1, v_2, \cdots, v_{|V|}$, and then outputs the new feature $\boldsymbol{h}_1^{(l)}, \boldsymbol{h}_2^{(l)}, \cdots, \boldsymbol{h}_{|V|}^{(l)}$ after l times convolutions. Then LSTM [37] receives the feature $\boldsymbol{h}_t'$ of the target class node v_t and outputs sequentially the probabilities of the next words of the target class name.

d) Training of the neural network

In the training phase, the model learns to minimize the cross-entropy loss between the output probabilities and the ground-truth class name for each word. Adam [38] optimizer is used to update the model parameters.

4 Proposed method

This section describes an approach to estimate class names with a graph neural network using the ASTs of methods. This approach takes the source code of a class as input from a developer and then estimates candidates for the class name. The key idea of this approach is to learn information within methods along with relationships between program elements. While the relation graph can model the class's complex nature, the graph ignores the internal properties in each node. On the other hand, methods' ASTs have rich structural information to represent the properties of the methods. By combining the relation graph and the method AST, this approach can estimate the class name while considering each node's internal properties and its neighboring nodes' properties.

This approach is based on the existing GNN-based estimation approach [10] and extends it to include information from the AST of each method. This approach gives GNN an embedding containing the AST information as the initial node feature. This embedding is computed using the vector generated in the middle of code2seq [7]. Given the initial feature with AST information, GNN can obtain each node's new feature, capturing the neighborhood method's syntactic structure. Figure 3 illustrates this proposed method.

As another way to integrate AST information into the existing GNN-based approach, there may be an approach to treat the AST as a subgraph and add an edge between the root node of the AST and the relation graph. However, since the existing approach estimates the class name by considering the information of the neighboring nodes of a target class, this alternative approach can only use the information of the part near the root node of the AST and ignores most of its information.

The rest of this section describes embeddings of program element information used as initial node features in GNN.

4.1 Embedding of program elements' information

This section describes embedding generations of node information, especially ASTs for methods' nodes, to be used as initial node features.

Depending on the program element type, this estimation model embeds each graph node in two ways: *name embedding* and *AST*

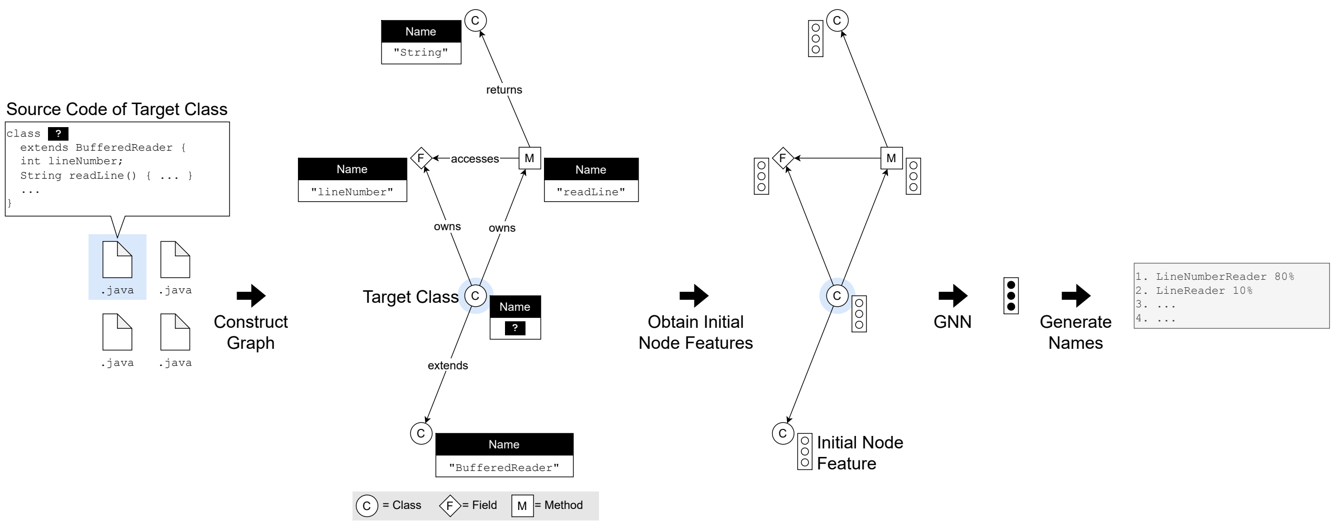


Figure 2 An illustration of the existing approach. [10]

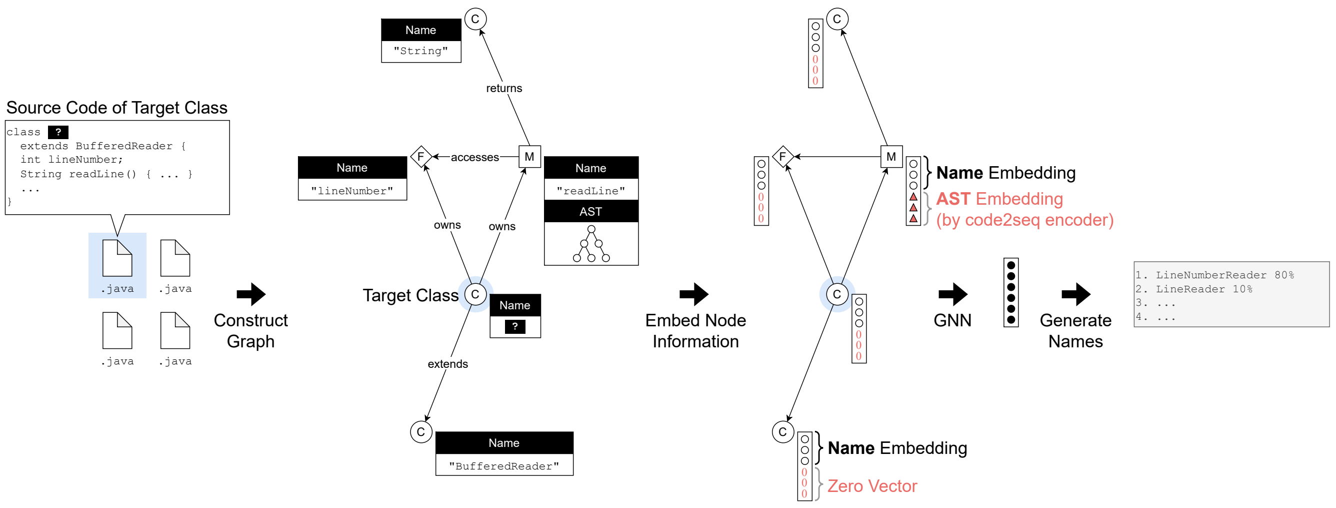


Figure 3 An illustration of the proposed method.

embedding. This approach newly employs the AST embedding in addition to the name embedding, while the name embedding is the only feature in the existing approach [10].

While the name embedding is calculated based on the name of a program element for all graph nodes, the AST embedding is calculated for nodes corresponding to a method with a body. In other words, a class's or field's embedding is computed only from the class name or field name. And the embedding of a method node is calculated from both the method name and its AST if it is not an abstract method; otherwise, it is calculated only from the method name.

Name embedding and AST embedding are calculated as described in Section 4.1.1 and 4.1.2, respectively. In the following explanation, let $\boldsymbol{h}_i^{\text{name}}, \boldsymbol{h}_i^{\text{AST}}$ denote name embedding and AST embedding of a node v_i, respectively.

After obtaining the name embedding and the AST embedding, two embeddings are concatenated. That is, the embedding of a node v_i is $\boldsymbol{h}_i^{(0)} = \text{concat}\left(\boldsymbol{h}_i^{\text{name}}, \boldsymbol{h}_i^{\text{AST}}\right)$

4.1.1 Name embedding

The model computes the name embedding by feeding the word sequence in an element name to bi-directional LSTM. The input element name is split into words by *camelCase* or *snake_case*: e.g., a name *LineNumberReader* is split into three words, *line*, *number*, and *reader*; a name *MAX_EXPONENT* is split into two words *max* and *exponent*. Then the model sequentially feeds trainable embeddings for each word. After feeding, the final hidden state of the bi-directional LSTM represents the name embedding of the element. Formally, the name embedding $\boldsymbol{h}_i^{name}$ of an element v_i with a name $w_{i1}, w_{i2}, \cdots, w_{il}$ is calculated as follows:

$$\boldsymbol{h}_{i1}^{\text{name}}, \boldsymbol{h}_{i2}^{\text{name}}, \cdots, \boldsymbol{h}_{il}^{\text{name}} = \text{LSTM}\left(E(w_{i1}), E(w_{i2}), \cdots, E(w_{il})\right)$$
$$\boldsymbol{h}_i^{\text{name}} = \text{concat}\left(\overrightarrow{\boldsymbol{h}}_{il}^{\text{name}}, \overleftarrow{\boldsymbol{h}}_{i1}^{\text{name}}\right)$$

where $E(\cdot)$ represents an operation to look up the embedding vector of a word, and $\text{LSTM}(\cdots)$ represents an operation to feed vectors to the LSTM sequentially.

As for the target class, the model calculates the name embedding similarly for other elements mentioned above but treats its name as a dummy since the name of the target class is to be estimated itself. Here, the dummy name is a sequence of masking tokens: $\langle SLOT \rangle, \cdots, \langle SLOT \rangle$.

4.1.2 AST embedding

The code2seq [7] encoder mentioned in Section 3.1 calculates the AST embedding of a method if the method has a body.

Each subtoken embedding used in the code2seq encoder shares weights the same as the word embedding used in the name embedding. This is because the subtoken embedding and the word embedding are both for tokens that appear in the code.

Besides, zero vectors act as substitutes for the AST embedding for elements without a method body. The dimension of this zero vector is the same as the AST embedding for the methods with a body.

Below shows the AST embedding $\boldsymbol{h}_i^{\mathrm{AST}}$ depending on whether the element corresponding to a node v_i has a method body or not:

$$\boldsymbol{h}_i^{\mathrm{AST}} = \begin{cases} \text{code2seq\_encoder}(\mathrm{AST}_i) & \text{if } v_i \text{ has method body} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

where AST_i represents the AST of the method corresponding to node v_i.

a) Utilizing weights pre-trained in code2seq encoder

Since the code2seq encoder is a part of a model designed for a different task, method name recommendation, utilizing pre-trained weights in the original task may boost the learning compared to training from scratch. If utilizing the pre-trained pre-trained weights, the model applies the weights at the beginning of the training process, and then the weights are fine-tuned during the training.

5 Evaluation

This section describes an experiment to evaluate the proposed method. The main goal of this experiment is to assess whether the AST embedding of the proposed method improves the correctness of the class name estimation compared to the existing approach. This correctness is measured and compared between the existing and proposed approaches using several metrics (described in Section 5.1) when estimating the class names of open-source Java projects. Each approach's estimation model is trained and hyperparameter-optimized (described in Section 5.3) on the datasets described in Section 5.2, and then its metrics are computed.

In addition, this experiment compares the metrics between the model with the pre-trained weights in the code2seq encoder, which is a component of the proposed approach, and the model without the weights to show whether these weights affect the estimation. In the following descriptions, the model with the pre-trained weights is referred to as *Proposed (PT)*, and the model without the weights is referred to as *Proposed (LFS)*.

5.1 Metrics

In this experiment, the estimation model takes as input a set of source codes where the name of the target class is hidden, and should then output the original name of the target class. The following metrics, between the estimated name and the original class name, indicate the quality of the estimation: precision, recall, f1, $partial(k)$, and $exact(k)$. These metrics are not case-sensitive.

Precision, recall, and f1 values are calculated by counting true positives, false positives, and false negatives between the estimated name at the first top of the candidates and the original. For example, suppose two estimated names are *Reader* and *LinkedArrayList*, while two original names are *LineNumberReader* and *LinkedList*, respectively. Then the true positives are 3 (*Reader*, *Linked*, *List*), the false positives are 1 (*Array*), and the false negatives are 2 (*Line*, *Number*). Therefore, precision is $3/4 = 0.75$, recall is $3/5 = 0.6$, and f1 is about 0.67.

$partial(k)$ and $exact(k)$ indicate the proportion of the estimated names that are partially or exactly the same as the original name in the top k candidates. The k values used in this experiment are 1 and 10. For $partial(k)$, when an estimated name contains any word of the original name, it is counted as a partial match.

5.1.1 Metrics for the estimation of trivial/non-trivial words

To investigate whether there is any difference in the correlation between the existing and proposed approaches in terms of the ease of estimating trivial and non-trivial words, this experiment also measures the metrics using only the following words:

- Only the most frequent words in the class names of each dataset, and conversely, only words other than those.
- Only the words other than those that make up the name of the class that the target class extends, if the target class extends another class.

This experiment considers two types of words that occur most frequently: a word in the top 1 and words in the top 10. The reason for considering these two types is that the term *test* occurs most frequently in the two datasets used in this experiment (described in Section 5.2) and occurs significantly more frequently than the words in the top 2 and below. Considering the two types of the most frequent words, the metrics are calculated by considering only the words in the top 1, only the words in the top 10, only the words in the top 2 or below (i.e., words other than *test*), and only the words in the top 11 or below.

When the metrics are calculated using only a portion of the total words, the metrics are calculated as follows. For example, suppose that when the metrics are calculated using only words other than the term *test*, an estimated name is *PyIndexingTest*, while an original name is *PyEmacsTabTest*. In this case, the true positives are 1 (*Py*), the false positives are 1 (*Indexing*), and the false negatives are 2 (*Emacs* and *Tab*). Therefore the precision is 0.5, the recall is about 0.33, and the f1 is 0.4.

5.2 Datasets

As real software project codes, this experiment uses two widely-used [20] existing source code corpus, both collected by Alon et al. [7], called *java-med* and *java-large*. java-med and java-large

Table 1 Hyperparameters after optimization for each estimation model.

	Exising [10]	Proposed (LFS)	Proposed (PT)
Embedding dim	118	505	512
Hidden dim	716	1018	1020
Batch size	96	24	84
Fanout	15	11	9
Learning rate	2.8×10^{-5}	1.81×10^{-5}	7.24×10^{-5}

are a set of top-starred projects on GitHub, containing 1,000 and 9,500 projects, respectively. Names in these datasets seem to have a certain level of quality since the top-starred projects have many users and contributors. The dataset is repartitioned into $\mathit{Train} : \mathit{Val1} : \mathit{Val2} : \mathit{Test} = 7 : 1 : 1 : 1$, independent of the partitioning by Alon et al.

5.3 Hyperparameter

The hyperparameters are set as shown in Table 1 by optimization with Optuna [39]. Throughout the optimization, the hyperparameters are optimized to maximize the f1 on the Val1. After the optimization, the estimation model is re-trained from scratch using the Val2 for validation. The following describes each hyperparameter, except for the learning rate.

Embedding dim. The dimension of the two types of embeddings: one where the name embedding and the subtoken embedding share weights; the other to embed nodes in AST.

Hidden dim. The dimension of the components in the neural network other than the embedding above, e.g., the dimension of the hidden state in LSTM or GCN.

Batch size. The number of target classes to learn in a single iteration.

Fanout. The number of neighbors to gather information from in GCN.

5.4 Environments

This experiment employs uses a Tesla V100-PCIE-32GB as the GPU. The neural network implementation uses Pytorch [40] and the Deep Graph Library [41] for GNN. An existing code2seq implementation(*1) is utilized for AST encoder and pre-training. The extraction of element relations uses JavaParser(*2). GCN in the proposed and existing approach has two layers.

5.5 Experimental result

This section describes the experimental results using the above experimental settings. Table 2 shows the results of the metrics measured on all words for each combination of model and dataset. In addition, Table 3 and Table 4 show the metrics measured on only a subset of all words, as described in Section 5.1.1. Table 3 shows the metrics measured only on the most frequent words or only on words other than such words. Furthermore, Table 4 shows the metrics measured without the words that make up the name of the class that the target class extends (if any).

(*1) : `https://github.com/m3yrin/code2seq`

(*2) : `https://javaparser.org/`

As shown in Table 2, the proposed approach outperforms the existing approach in almost all metrics, except for the $\mathit{exact}(k = 1)$ and $\mathit{exact}(k = 10)$ of the Proposed (PT). This indicates that the AST embedding of the proposed approach is helpful for correctly estimating class names.

Although the Proposed (PT) can use the pre-trained weights from the beginning of the training, its precision falls below that of the Proposed (LFS). This decrease indicates that the Proposed (PT) is more likely to estimate redundant words than the Proposed (LFS). This may be due to the difference in the target task between the proposed approach and the pre-training, i.e., the proposed approach targets class name estimation, while the code2seq encoder, whose weights are pre-trained, is designed for method name estimation. This difference may cause the pre-trained weights to interfere with the class name estimation.

The metrics measured only on trivial words and those measured only on non-trivial words show different trends, as shown in Table 3 and Table 4. When measured only on a top 1 word or top 2-10 words, the precision scores of the Proposed (PT) on the java-med dataset and the Proposed (LFS) on the java-large dataset are lower than those of the existing approach. This indicates that there are cases where the proposed approach is likely to estimate trivial words redundantly. On the other hand, the metrics measured without such trivial words have the same tendency as those in Table 2. In other words, all the values of precision, recall, and f1 of the proposed approach are better than those of the existing approach. Therefore, the proposed approach can estimate a wider variety of words, incredibly non-trivial words, rather than trivial words compared to the existing approach.

6 Conclusion

This paper proposes an approach that aims to exploit the information within method nodes in GNN-based class name estimation. The key idea of this approach is to learn the information within methods along with the relationships between program elements. The proposed approach extends an existing GNN-based approach to utilize the AST embedding generated in a method name estimation approach based on AST.

An experiment measures the estimation correctness using datasets from open-source projects, which aims to investigate the effectiveness of the AST embedding of the proposed approach. An experimental result shows that the proposed approach can estimate class names better than the existing one.

Future work includes the following. The first is to investigate other ways to embed node information well. While this work uses AST to represent node information, machine-learning work often uses other representations, such as dataflow. Depending on how the estimation model represents node information, it is valuable to investigate how the estimation performs. The second is to evaluate the contributions of the components in the approach on which this

Table 2 Metrics for each estimation model, measured on all words. Bold values represent the results scores on the java-med and java-large datasets, respectively.

					partial(k)		*exact*(k)	
	dataset	precision	recall	f1	$k=1$	$k=10$	$k=1$	$k=10$
Existing [10]	java-med	0.2267	0.1643	0.1905	0.4429	0.5805	0.0076	0.0239
Proposed (LFS)	java-med	**0.2769**	0.1968	**0.2301**	0.4951	**0.6591**	**0.0092**	**0.0341**
Proposed (PT)	java-med	0.2407	**0.1975**	0.2170	**0.5016**	0.5991	0.0065	0.0184
Existing [10]	java-large	0.2652	0.2197	0.2403	0.5228	0.6590	0.0085	0.0364
Proposed (LFS)	java-large	**0.3051**	**0.2336**	**0.2646**	**0.5368**	**0.6965**	**0.0149**	**0.0480**

Table 3 Metrics measured on only a subset of words based on their frequency.

						partial(k)	
Metrics are measured on		dataset	precision	recall	f1	$k=1$	$k=10$
top 2- words only	Existing [10]	java-med	0.1843	0.1278	0.1509	0.3393	0.4892
	Proposed (LFS)	java-med	**0.2359**	**0.1613**	**0.1916**	0.4062	**0.5895**
	Proposed (PT)	java-med	0.2101	0.1608	0.1822	**0.4086**	0.5200
	Existing [10]	java-large	0.2317	0.1880	0.2076	0.4440	0.5917
	Proposed (LFS)	java-large	**0.2900**	**0.1993**	**0.2362**	**0.4607**	**0.6451**
top 1 word only	Existing [10]	java-med	0.6699	0.8863	0.7630	0.8948	0.9407
	Proposed (LFS)	java-med	**0.7163**	0.8857	**0.7920**	0.8947	0.9427
	Proposed (PT)	java-med	0.4698	**0.8974**	0.6167	**0.9045**	**0.9462**
	Existing [10]	java-large	**0.7388**	0.8326	**0.7829**	0.8376	0.9154
	Proposed (LFS)	java-large	0.3912	**0.8838**	0.5424	**0.8885**	**0.9664**
top 11- words only	Existing [10]	java-med	0.1729	0.1162	0.1390	0.3009	0.4451
	Proposed (LFS)	java-med	**0.2239**	**0.1499**	**0.1796**	0.3671	**0.5465**
	Proposed (PT)	java-med	0.2026	0.1481	0.1711	**0.3677**	0.4731
	Existing [10]	java-large	0.2197	0.1756	0.1952	0.4040	0.5499
	Proposed (LFS)	java-large	**0.2885**	**0.1870**	**0.2269**	**0.4201**	**0.6036**
top 2-10 words only	Existing [10]	java-med	0.3218	0.3399	0.3306	0.3608	0.4954
	Proposed (LFS)	java-med	**0.3864**	0.3547	**0.3698**	0.3746	**0.5745**
	Proposed (PT)	java-med	0.2783	**0.3777**	0.3204	**0.3974**	0.5064
	Existing [10]	java-large	**0.3703**	0.3877	**0.3788**	0.4098	0.5962
	Proposed (LFS)	java-large	0.2992	**0.4061**	0.3446	**0.4312**	**0.6709**

Table 4 Metrics measured excluding the words of the class from which each class extends.

	dataset	precision	recall	f1
Existing [10]	java-med	0.1567	0.1193	0.1355
Proposed (LFS)	java-med	**0.2012**	0.1536	**0.1742**
Proposed (PT)	java-med	0.1734	**0.1549**	0.1636
Existing [10]	java-large	0.1930	0.1616	0.1759
Proposed (LFS)	java-large	**0.2316**	**0.1872**	**0.2070**

work is based. For example, evaluating how the estimation performs when some graph edges are removed is interesting. The third is to use some mechanism to account for the importance of graph nodes or edges. Although the graph seems to have many edges and nodes that are not helpful for the estimation, the proposed approach uses information similarly for all nodes and edges. Mechanisms such as attention may improve estimation performance by efficiently ignoring unimportant information.

Acknowledgments

This paper is based on results obtained from a project, JPNP20006, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, Jun. 2006, pp. 3–12.

[2] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, Jun. 2012, pp. 193–202.

[3] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[4] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A Large-Scale field study with professionals," *IEEE Trans. Software Eng.*, vol. 44, no. 10, pp. 951–976, Oct. 2018.

[5] 米内裕史, 早瀬康裕, and 北川博之, "ソースコード構文木とコールグラフの統合的な埋め込みに基づくメソッド名の推定," **研究報告ソフトウェア工学** *(SE)*, vol. 2020, no. 10, pp. 1–8, 2020.

[6] S. Kurimoto, Y. Hayase, H. Yonai, H. Ito, and H. Kitagawa, "Class name recommendation based on graph embedding of program elements," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2019-Decem, 2019, pp. 498–505.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International*

Conference on Learning Representations, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[8] F. Ge and L. Kuang, "Keywords guided method name generation," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, vol. 0, May 2021, pp. 196–206.

[9] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Nov. 2018.

[10] 萬場大登, 早瀬康裕, 天笠俊之, and 北川博之, "オブジェクト指向プログラムの要素関係グラフを用いたクラス名の end-to-end 学習," **情報処理学会第** *83* **回全国大会**, vol. 4, p. 01, 2021.

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, no. POPL, pp. 1–29, Jan. 2019.

[12] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin, "Learning to recommend method names with global context," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1294–1306.

[13] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon, "GraphCode2Vec: generic code embedding via lexical and program dependence analyses," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 524–536.

[14] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 2016, pp. 2091–2100.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "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)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083.

[16] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 200–210.

[17] ——, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, May 2020.

[18] W. U. Ahmad, S. Chakraborty, B. Ray, and K. W. Chang, "A transformer-based approach for source code summarization," *arXiv*, 2020.

[19] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=Xh5eMZVONGF

[20] S. Wang, M. Wen, B. Lin, and X. Mao, "Lightweight global and local contexts guided method name recommendation with prior knowledge," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 741–753.

[21] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 91–105.

[22] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation Pre-Training objective for programming languages," https://openreview.net/pdf?id=3ez9BSHTNT, accessed: 2022-12-26.

[23] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[24] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN Encoder–Decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[25] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421. [Online]. Available: https://aclanthology.org/D15-1166

[26] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, Jun. 2018.

[27] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, "Integrating tree path in transformer for code representation," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 9343–9354. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/4e0223a87610176ef0d24ef6d2dcde3a-Paper.pdf

[28] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. ieeexplore.ieee.org, May 2019, pp. 783–794.

[29] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, May 2021.

[30] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 378–389.

[31] B. Liu, T. Wang, X. Zhang, Q. Fan, G. Yin, and J. Deng, "A neural-network based code summarization approach by using source code and its call dependencies," in *ACM International Conference Proceeding Series*. Association for Computing Machinery, Oct. 2019.

[32] H. Yonai, Y. Hayase, and H. Kitagawa, "Mercem: Method name recommendation based on call graph embedding," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 134–141.

[33] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, X. Xia, and M. R. Lyu, "Code structure guided transformer for source code summarization," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2022.

[34] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, 2015, pp. 38–49.

[35] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Msr 2020*. ACM, 2020, p. 11.

[36] T. N. Kipf and M. Welling, "Semi-Supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[38] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–15, 2015.

[39] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna:

A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 2623–2631.

[40] Paszke, Gross, Massa, Lerer, and others, "Pytorch: An imperative style, high-performance deep learning library," *Adv. Neural Inf. Process. Syst.*, 2019.

[41] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.