Universal Representation for Code

Linfeng Liu[1⋆(✉)], Hoan Nguyen[2], George Karypis[2], and Srinivasan Sengamedu[2]

[1] Tufts University, Medford MA 02155, USA
linfeng.liu@tufts.edu
[2] Amazon Web Services, Seattle WA 98109, USA
{hoanamzn, gkarypis, sengamed}@amazon.com

Abstract. Learning from source code usually requires a large amount of labeled data. Despite the possible scarcity of labeled data, the trained model is highly task-specific and lacks transferability to different tasks. In this work, we present effective pre-training strategies on top of a novel graph-based code representation, to produce *universal* representations for code. Specifically, our graph-based representation captures important semantics between code elements (e.g., control flow and data flow). We pre-train graph neural networks on the representation to extract universal code properties. The pre-trained model then enables the possibility of fine-tuning to support various downstream applications. We evaluate our model on two real-world datasets – spanning over 30M Java methods and 770K Python methods. Through visualization, we reveal discriminative properties in our universal code representation. By comparing multiple benchmarks, we demonstrate that the proposed framework achieves state-of-the-art results on method name prediction and code graph link prediction.

Keywords: Code representation · Graph neural network · Pre-training.

1 Introduction

Analysis of software using machine learning approaches has several important applications such as identifying code defects [1], improving code search [2], and improving developer productivity [3]. One common aspect of any code-related application is that they learn code representations by following a two-step process. The first step takes code snippets and produces a *symbolic* code representation using program analysis techniques. The second step uses the symbolic code representation to generate *neural* code representations using deep learning techniques.

Symbolic representations need to capture both syntactic and semantic structures in code. Approaches to generating symbolic representations can be categorized as sequence-, tree-, and graph-based. Sequence-based approaches represent code as a sequence of tokens and only capture the shallow and textual structures of the code [4]. Tree-based approaches represent the code via abstract syntax

⋆ Work done while the author was an intern at Amazon Web Services.

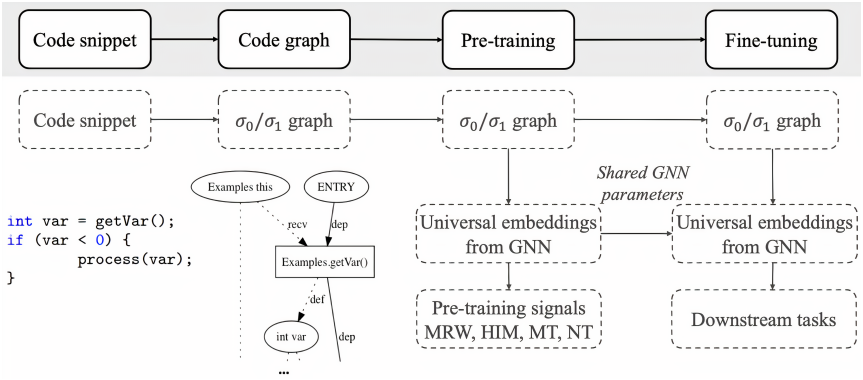

Fig. 1. Model pipeline. The σ_0/σ_1 graph is defined in Section 3, and the pre-training signals are defined in Section 4.

trees (ASTs) [5] that highlight structural and content-related details in code. However, some critical relations (e.g., control flow and data flow), which often impact machine learning models' success in abstracting code information, are not available in trees. Graph-based approaches augment ASTs with extra edges to partially represent the control flow and the data flow [1,4,6]. Depending on the type of symbolic representation used, the approaches for generating the neural code representations are either sequence-based [3,7] or graph-based [4,6] neural network models. However, these works are generally task-specific, making it hard to transfer the learned representations to other tasks. In addition, the scarcity of labeled data may cause insufficient training in deep learning models.

In this work, we touch upon all the three aspects of ML-based code analysis: symbolic code representation, task-independent neural code representation, and task-specific learning. Fig. 1 gives an overview of our approach. For symbolic code representation, we explore two alternatives and show that symbolic code representation (called σ_0/σ_1 graph) which captures richer relations leads to better performance in downstream tasks. For neural code representation, we specialize a recently proposed universal representation for graphs, PanRep [8], to code graphs. And, finally, we explore two tasks to demonstrate the effectiveness of the learned representations: method name prediction (for Python and Java) and link prediction (for Java). Our proposed method consistently improves the prediction accuracy across all experiments.

To summarize, the contributions of this work are as follows:

- We introduce a fine-grained symbolic graph representation for source code, and adapt to 29M Java methods collected from GitHub.
- We present a pre-training framework that leverages the graph-based code representations to produce universal code representations, supporting various downstream tasks via fine-tuning.
- We combine the graph-based representation and the pre-training strategies to go beyond code pre-training with sequence- and tree-based representations.

2 Preliminary

Notation Let $G = \{\mathcal{V}, \mathcal{E}\}$ denote a *heterogeneous* graph with $|\mathcal{T}|$ node types and $|\mathcal{R}|$ edge types. $\mathcal{V} = \{\{\mathcal{V}^t\}_{t\in\mathcal{T}}\}$ represents the node set, and $\mathcal{E} = \{\{\mathcal{E}^r\}_{r\in\mathcal{R}}\}$ represents the edge set. Each node $v_i^t \in \mathcal{V}^t$ is associated with a feature vector. Throughout the paper, we often use "representation" and "embedding" interchangeably unless there is any ambiguity.

2.1 Graph Neural Networks

Graph neural networks (GNNs) learn representations of graphs [9]. A GNN typically consists of a sequence of L graph convolutional layers. Each layer updates nodes' representation from their direct neighbors. By stacking multiple layers, each node receives messages from higher-order neighbors. In this work, we utilize the relational graph convolutional network (RGCN) [10] to model our heterogeneous code graphs. RGCN's update rule is given by

$$\mathbf{h}_i^{(l+1)} = \phi\left(\sum_{r\in\mathcal{R}}\sum_{n\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}\mathbf{h}_n^{(l)}\mathbf{W}_r^{(l)}\right),$$

where $\mathcal{N}_i^r$ is the neighbor set of node i under edge type r, $c_{i,r}$ is a normalizer (we use $c_{i,r} = |\mathcal{N}_i^r|$ as suggested in [10]), $\mathbf{h}_i^{(l)}$ is the hidden representation of node i at layer l, $\mathbf{W}_r^{(l)}$ are learnable parameters, and $\phi(\cdot)$ is any nonlinear activation function. Usually, $\mathbf{h}_i^{(0)}$ is initialized as node features, and $\mathbf{h}_i^{(L)}$ (the representation at the last layer) is used as the final representations.

2.2 Pre-training for GNNs and for Source Code

Recently, there is a rising interest in pre-training GNNs to model graph data [11,12]. To pre-train GNNs, most works encourage GNNs to capture graph structure information (e.g., graph motif) and graph node information (e.g., node feature). PanRep [8] further extends GNN pre-training to heterogeneous graphs.

Pre-training on source code has been studied in [13,14,15]. However, these models build upon sequence-based code representations and fail to encode code's structural information explicitly. We differ from these works, by pre-training on a novel graph-based code representation to capture code's structural information.

3 Code Graph

Previous Machine Learning (ML) models [4,6,16] are largely based on ASTs to reflect structural code information. Though ASTs are simple to create and use, they have tree-based structures and do not capture control flow and data flow relations. Here, the control flow represents the order of the execution and the data flow represents the flow of data along the computation. For example, to represent

a loop snippet, ASTs cannot naturally use an edge pointing from the end of the program statement to the beginning of the loop. In addition, the relation between the definition and uses of a variable is not captured in ASTs. In this work, we represent code as graphs to efficiently capture both control flow and data flow between program elements. We call our code graphs as σ_0 graphs and σ_1 graphs. The σ_0 graphs are related to classical Program Dependence Graphs (PDGs) [17]. The σ_1 graphs build upon the σ_0 graphs and include additional syntactic and semantic information (detailed in section 3.2). Our experiments show that ML models using the σ_1 graphs achieve better prediction accuracy than using the σ_0 graphs.

3.1 The σ_0 Graph

The σ_0 graphs, which relate to PDGs, are used for tasks such as detection of security vulnerabilities and identification of concurrency issues. In σ_0 graphs, nodes represent different kinds of program elements including data and operations; edges represent different kinds of control flow and data flow between program elements. We showcase a σ_0 graph in Fig. 2.

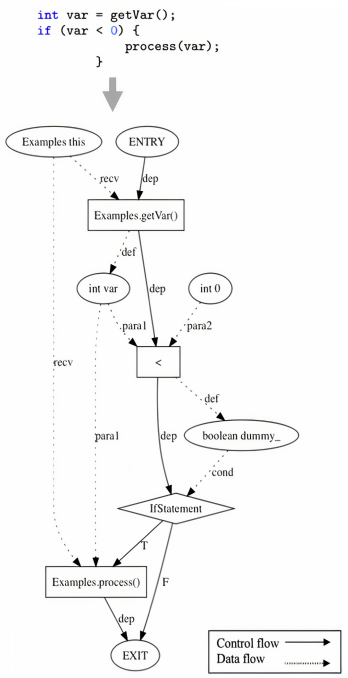

Fig. 2. An example of σ_0 graph.

Both nodes and edges in the σ_0 graph are typed. Specifically, we have five node types: entry, exit, data, action, and control nodes. Entry and exit nodes indicate the control flow entering and exiting the graph. Data nodes represent the data in programs such as variables, constants and literals. Action nodes represent the operations on the data such as method calls, constructor calls and arithmetic/logical operations, etc. Control nodes represent control points in the program such as branching, looping, or some special code blocks such as catch clauses and finally blocks. We have two edge types: control and data edges. Control edges represent the order of execution through the programs and data edges represent how data is created and used in the programs. Examples of edge types include parameter edges which indicate the data flow into operations and throw edges which represent the control flows when exceptions are thrown.

3.2 The σ_1 Graph

One limitation of σ_0 graph is that the downstream modules perform additional analysis such as control dependence and aliasing is not reflected in the σ_0 graph explicitly and requires machine learning models to infer it. We propose σ_1 graph as an augmentation of the σ_0 graph, which is enhanced by additional information.

Specifically, σ_1 graph attaches AST node types to nodes in the graph. AST node types capture syntactic information (e.g., InfixOperator) provided by the parser. Higher-order semantic relations such as variable usage (e.g., FirstUse/LastUse), node aliasing, and control dependence are also included as graph edges.

3.3 The Heterogeneous Code Graph

The proposed σ_0 and σ_1 graphs are heterogeneous graphs, i.e. nodes and edges have types and features. Nodes are categorized into five types: `entry`, `exit`, `data`, `action`, and `control`. Node features are attributed by their names. Edge types are the same as edge features, which are defined by their functionalities. Below we first describe node features, followed by edge features.

Entry nodes have feature `ENTRY` and exit nodes have feature `EXIT`. Features of control nodes are their corresponding keywords such as `if`, `while`, and `finally`. Features of variables are types, and the variable names are ignored. For example, `int.x` $\rightarrow$ `int`; `String.fileName` $\rightarrow$ `String`. Method names contain the method class, method name, and parameter class. For example, `Request.setConnectionKeepAlive#boolean#`. Here, `Request` is the method class, `setConnectionKeepAlive` is the method name, and `boolean` is the parameter class.

Features of control edges and data edges are defined separately. There are two kinds of control edges: normal control edges and exception control edges. A normal control edge is a directed edge that connects two control or action nodes; defined as `dep`. See Fig. 2. An exception control edge connects an action node to a control node to handle an exception that could be thrown by the action; defined as `throw`. Data edges have five features: `receiver`, `parameter`, `definition`, `condition`, and `qualifier`. Examples include `receiver.call()` as a receiver edge; `call(param)` as a parameter edge, and `foo=bar()` as a definition edge.

3.4 Corpus-level Graphs

In a typical ML application, the corpus consists of a collection of packages or repositories. Repositories contain multiple files or classes. Classes contain methods. The σ_0 and σ_1 graphs are at *method-level*. Corpus-level graphs are a collection of method-level σ_0/σ_1 graphs. Let σ refer to either σ_0 or σ_1 for notational ease.

4 Method

We propose a new model, Universal Code Representation via GNNs (UniCoRN), to produce universal code representations based on σ graphs. UniCoRN has two components. First, a GNN encoder that takes in σ graphs and generates node embeddings. Second, pre-training signals that train the GNN encoder in an unsupervised manner. By sharing the same GNN encoder across all σ graphs, the learned embeddings reveal universal code properties. Below, we show our design of pre-training signals to help UniCoRN efficiently distill universal code semantics. The instantiation of the GNN encoder is given in the experiment.

4.1 Pre-training Signals

Metapath Random Walk (MRW) signal. A MRW is a random path that follows a sequence of edge types. We assume node pairs in the same MRW are proximal to each other; accordingly, they should have similar embeddings. For example, for a MRW with nodes [`Collection.iterator()`, `Iterator`, `Iterator.hasNext()`] and edges [`definition`, `receiver`], nodes `Iterator` and `Iterator.hasNext()` are expected to have similar embeddings. Formally, the signal is defined as

$$\mathcal{L}_{MRW} = \sum_{v,v' \in \mathcal{V}} \log\left(1 + \exp\left(-y \times \mathbf{h}_v^t \mathbf{W}^{t,t'} \mathbf{h}_{v'}^{t'}\right)\right), \tag{1}$$

where $\mathbf{h}_v^t$ and $\mathbf{h}_{v'}^{t'}$ are embeddings for nodes v and v' with node types t and t'. $\mathbf{W}^{t,t'}$ is a diagonal matrix weighing the similarity between different node types. y equals 1 as positive pairs if v and v' are in the same MRW, otherwise y equals -1 as negative pairs. During training, we sample 5 negative pairs per positive pair.

Heterogeneous Information Maximization (HIM) signal. Nodes of the same type should reside in some shared embedding space, encoding their similarity. On the other hand, nodes of different types, such as control nodes and data nodes, ought to have discriminative embedding space as they are semantically different. However, standard GNNs fail to do so with only local message propagation. Following [8], we use a HIM signal to encode these properties:

$$\mathcal{L}_{HIM} = \sum_{t \in \mathcal{T}} \sum_{v \in \mathcal{V}^t} \Big(\log\left(\phi(\mathbf{h}_v^\top \mathbf{W} \mathbf{s}^t)\right) + \log\left(1 - \phi(\tilde{\mathbf{h}}_v^\top \mathbf{W} \mathbf{s}^t)\right) \Big). \tag{2}$$

Here, $\mathbf{s}^t = \frac{1}{|\mathcal{V}^t|} \sum_{v \in \mathcal{V}^t} \mathbf{h}_v$ is a global summary of nodes typed t, $\phi(\cdot)$ is a sigmoid function, and $\phi(\mathbf{h}_v^\top \mathbf{W} \mathbf{s}^t)$ quantifies the closeness between a node embedding $\mathbf{h}_v$ and a global summary $\mathbf{s}^t$. Negative samples $\tilde{\mathbf{h}}_v$ are obtained by first row-wise shuffling input node features then propagating through the GNN encoder [8].

Motif (MT) signal. Code graph has connectivity patterns. For example, node `ENTRY` has only one outgoing edge; node `IfStatement` has True and False branch. Such structural patterns can be captured in graph motifs, see Fig. 3. With this observation, we pre-train GNNs with MT signal to generate structure-aware node embeddings for code graphs. Formally, we aim to recover the ground truth motif around each node, $\boldsymbol{m}_v$, from the node embedding $\mathbf{h}_v$ using an approximator $f_{MT}(\cdot)$ (we use a two-layer MLP),

Fig. 3. Motifs sized 3 and 4.

$$\mathcal{L}_{MT} = \sum_{v \in \mathcal{V}} ||\boldsymbol{m}_v - f_{MT}(\mathbf{h}_v)||_2^2. \tag{3}$$

The ground truth $\boldsymbol{m}_v$ is obtained using a fast motif extraction method [18].

Node Tying (NT) signal. The corpus-level graph (Cf. section 3.4) contains many *duplicate nodes* that have the same feature (e.g., two `ENTRY` nodes will be

induced from two methods). These duplicate nodes serve as anchors to imply underlying relations among different graphs. We divide duplicate nodes into two categories: strict equality and weak equality. Strict equality refers to duplicate nodes whose semantic meaning should be invariant to their context, including keywords (`if`, `while`, `do` ...), operators (`=`, `*`, `<<` ...), entry and exit nodes. Duplicate nodes of strict equality will have the same embedding in all σ graphs. We keep a global embedding matrix to maintain their embeddings. Weak equality refers to other duplicate nodes whose semantic meaning can be affected by their context. For example, two `foo()` nodes in two methods, or two tied nodes due to the simple qualified types of one or two nodes[1]. We use NT signal to encourage duplicate node of weak equality to have similar embeddings:

$$\mathcal{L}_{NT} = \sum_{k \in \mathcal{K}} \text{ave}(\{||\mathbf{h}_v - \mathbf{g}^k||_2^2\}, v \text{ has feature } k), \tag{4}$$

where $\mathcal{K}$ is the set of distinct node features (exclude strict equality nodes), ave$(\cdot)$ is an average function, and $\mathbf{g}^k = \text{ave}(\{\mathbf{h}_v\}, v \text{ has label } k)$ is a global summary of nodes featured k. In (4), we first group nodes featured k, followed by computing the group centers $\mathbf{g}^k$, then minimize the distance between nodes to their group centers.

Pre-training objective. We combine the four pre-training signals to yield a final objective:

$$\mathcal{L} = \omega_1 \mathcal{L}_{MRW} + \omega_2 \mathcal{L}_{HIM} + \omega_3 \mathcal{L}_{MT} + \omega_4 \mathcal{L}_{NT}, \tag{5}$$

where $\omega_1, \ldots, \omega_4$ balance the importance of different signals. The objective resembles the objective in multi-task learning [19].

4.2 Data Pre-processing and Fine-tuning

Numeric node features. The initial node features are strings (Cf. section 3.3), which need to be cast into numeric forms before feeding into the GNN encoder. To this end, we first split each node's feature into subtokens based on the delimiter ".". Then, language models are used to get subtoken embeddings, in which we use FastText [20]. Finally, we use average subtoken embeddings as the node's numeric feature.

Inverse edges. We enrich our σ graphs with inverse edges. Recent work has proven improved performance by adding inverse edges to ASTs [6].

Fine-tuning. After pre-training, we can fine-tune on downstream tasks. Fine-tuning involves adding downstream classifiers on top of the pre-trained node embeddings, and predicting downstream labels. A graph pooling layer [9] might be needed if the downstream tasks are defined on the graph/method level.

[1] We use simple types instead of fully qualified types since we create graphs from source files and not builds. In this case, types are not fully resolvable.

Table 1. Dataset statistics. The σ_1 graph doubled the number of edges as the σ_0 graph, providing extra information for code graphs.

Dataset	Repository	Method	Node	Edge
Java (σ_0)	28K	29M	621M	1,887M
Java (σ_1)	28K	29M	529M	3,782M
Python	14K	450K	57M	156M

Table 2. Result for method name prediction on Java dataset. Higher value indicates better performance.

	F1	Precision	Recall
σ_0	21.7	26.1	19.9
σ_1	**23.6**	**27.5**	**22.0**

5 Experiment

5.1 Dataset

We tested on two real-world datasets, spanning over two programming languages Java and Python. Summary of the datasets is listed in Table 1.

Java The Java dataset is extracted from 27,581 GitHub packages. In total, these packages contain 29,024,142 Java methods. We convert each Java method into a σ_0 graph and a σ_1 graph. The data split is on package-level, with training (80%), validation (10%), and testing (10%).

Python The Python dataset is collected from Stanford Open Graph Benchmark (`ogbg-code`) [6]. The total number of Python methods is 452,741, with each method is represented as an AST. These ASTs are further augmented with next-token edges and inverse edges. The data split keeps in line with `ogbg-code`.

5.2 Experimental Setup

We tune hyperparameters on all models based on their validation performances. For Java dataset, we consider a two-layer RGCN with 300 hidden units. For Python dataset, we follow `ogbg-code` and use a five-layer RGCN with 300 hidden units. We use Adam [21] as optimizer, with learning rate ranges from 0.01 to 0.0001. Mini-batch training is adopted to enable training on very large graphs[2]. We apply dropout at rate 0.2, L2 regularization with parameter 0.0001. The model is first pre-trained on a maximum of 10 epochs, then fine-tuned up to 100 epochs on downstream tasks until convergence. The model is implemented using Deep Graph Library (DGL) [22]. Models have access to 4 Tesla V100 GPUs, 32 CPUs, and 244GB memory.

5.3 Analysis of Embeddings

We begin by analyzing code embeddings via t-SNE [23] visualization. We study two levels of embeddings: node-level embeddings and method-level embeddings. Here, a method-level embedding summarizes a method, computed by averaging node embeddings in its σ graph. For better visualization, we show results on 10 random Java packages (involving 4,107 Java methods) using σ_1 graphs.

[2] https://github.com/dmlc/dgl/tree/master/examples/pytorch/rgcn-hetero

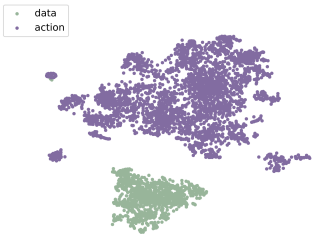

Fig. 4. Visualization of node-level embeddings with t-SNE.

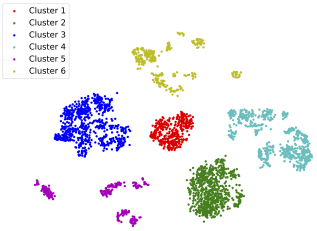

Fig. 5. Visualization of method-level embeddings with t-SNE. Colored with K-means.

In Fig. 4, we see that data nodes and action nodes are forming separate clusters, indicating our code embeddings preserve important node type information. Fig. 5 suggests method embeddings are forming discriminative clusters. By manually annotating each cluster, we discovered that cluster 4 contains 91% (out of all) `set` functions, cluster 3 contains 78% `find` functions, and cluster 1 contains 69% functions which end with `Exception`. This result suggests that our model has the potential to distinguish methods in terms of method functionalities.

5.4 Method Name Prediction

We use pre-trained UniCoRN model to initialize code embeddings. Then following [6,16], we predict method names as downstream tasks. The method name is treated as a sequence of subtokens (e.g. `getItemId` → [`get`, `item`, `id`]). As in [6], we use independent linear classifiers to predict each subtoken. The task is defined on the method-level: predict one name for one method (code graph). We use attention pooling [24] to generate a single embedding per method. We follow [6,16] to report F1, precision, recall for evaluation. Below we show results on Java and Python separately, as they are used for different testing purposes.

Java. We evaluate the performance gain achieved by switching from the σ_0 graph to the σ_1 graph. We truncate subtoken sequences to a maximal length of 5 to cover 95% of the method names. Vocabulary size is set to 1,000, covering 95% of tokens. Tokens not in the vocabulary are replaced by a special `unknown` token. Similar techniques have been adopted in [6]. We experiment on approximately 774,000 methods from randomly selected 1,000 packages.

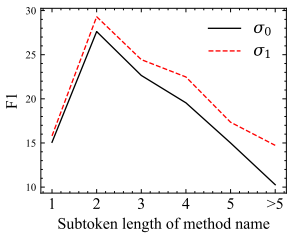

Fig. 6. F1 at name length.

The result is summarized in Table 2. We see that the σ_1 graph outperforms the σ_0 graph, indicating that the extra information provided by the σ_1 graph is beneficial for abstracting code snippets. Fig. 6 further supports this observation. The σ_1 graph consistently outperforms the σ_0 graph w.r.t. the F1 score for different method name lengths.

Table 3. Method name prediction for Python. Pooling: average[†], virtual node[§], and attention[‡]. GCN[†,§] and GIN[†,§]: Reported in [6].

	F1	Precision	Recall
GCN-NextTokenOnly[†]	29.77	31.09	29.18
GIN-NextTokenOnly[†]	29.00	30.98	28.13
GCN[†]	31.63	-	-
GIN[†]	31.63	-	-
UniCoRN w/o pretrain[†]	32.81	35.25	31.71
UniCoRN[†]	33.28	35.28	32.36
GCN[§]	32.63	-	-
GIN[§]	32.04	-	-
UniCoRN[§]	33.80	35.81	32.89
GCN[‡]	32.80	34.72	31.88
GIN[‡]	32.60	34.42	31.77
UniCoRN[‡]	**33.94**	**36.02**	**32.99**

Table 4. MRR and Hit@K(%) results for link prediction. Higher values are better. Superscripts $^{\mathcal{D}}$ and $^{\mathcal{M}}$ denote DistMult and MLP link predictors. Hit@K for random is computed as $K/(1+200)$, where 200 is the number of negative edges per testing edge.

	MRR	Hit@1	Hit@3	Hit@10
Random	-	0.5	1.5	5.0
FastText$^{\mathcal{D}}$	0.01	0.4	1.0	2.3
$\sigma_0^{\mathcal{D}}$	0.26	15.4	28.5	41.3
$\sigma_1^{\mathcal{D}}$	**0.32**	**18.1**	**38.4**	**61.4**
FastText$^{\mathcal{M}}$	0.05	1.9	4.0	8.0
$\sigma_0^{\mathcal{M}}$	0.51	46.1	49.8	58.4
$\sigma_1^{\mathcal{M}}$	**0.53**	**46.2**	**55.0**	**65.2**

Ground truth	Prediction
`get_config`	`get_config`
`create_collection`	`create_collection`
`get_aws_credentials`	`get_ec2`
`wait_for_task_ended`	`wait_job`
`add_role`	`create_permission`
`load_bytes`	`upload_file`

Fig. 7. Examples of method name prediction on Python in different degree of consensus. Each pair of results is demonstrated as ground truth name and predicted name.

```
def wait_for_task_ended(self):
    try:
        waiter = self.client.\
            get_waiter('job_execution_complete')
        # timeout is managed by airflow
        waiter.config.max_attempts = sys.maxsize
        waiter.wait(jobs=[self.jobId])
    except ValueError:
        # If waiter not available use expo
        retry ...
```

Prediction: `wait_job`.

Fig. 8. Reasonable prediction based on the code context is observed, though it is inexact match.

Note that the F1 score at method names of length 1 is low. We suspect that some names at this length are not semantically meaningful, such as `a` or `xyz`. Thus, these method names are hard to predict correctly.

Python. We compare the performances of UniCoRN with various baselines on Python. Our experiment setup closely follows `ogbg-code` [6]. For the baseline, `ogbg-code` considers GCN and GIN. Additionally, we introduce two baselines that run GCN and GIN with next-token edges only. We expect these two new baselines to mimic the performance of sequence-based models. In this task, we test three pooling methods: average, virtual node [6], and attention [24].

Results are given in Table 3. We list three observations. First, UniCoRN with attention pooling (UniCoRN[‡]) performs the best, endorsing UniCoRN's superior modeling capacity. Second, UniCoRN with pre-training shows performance gain over UniCoRN without pre-training, verifying the usefulness of our pre-training strategies. Third, GCN(GIN) improves GCN(GIN)-NextTokenOnly, confirming the importance of using structural information.

Example pairs of ground truth and prediction are shown in Fig. 7. The examples of prediction encompass exact matches, such as `get_config` pair,

context matches, such as `get_aws_credentials` pair, and mismatches, such as `load_bytes` pair. We showcase a prediction example in Fig. 8.

5.5 Link Prediction

In this task, we examine how UniCoRN recovers links in code graphs. We follow the same experimental setup as in [8]. We feed two node embeddings of a link to a predictor, a DistMult [25] or a two-layer MLP, to predict the existence of the link. Here, node embeddings are obtained by applying UniCoRN to σ_0/σ_1 graphs, or simply obtained from FastText embeddings. Note that the FastText is the initial embedding of UniCoRN. We freeze UniCoRN after pre-training. When fine-tuning link predictors, we ensure σ_0 and σ_1 have the same set of training edges. During inference, we sample 200 negative edges per testing edge (positive) and evaluate the rank of the testing edge. Evaluations are based on Mean Reciprocal Rank (MRR) and Hit@K (K=1, 3, 10).

Table 4 shows the results. UniCoRN outperforms FastText. The results suggest that node embeddings from UniCoRN capture neighborhood correlations. We see σ_1 graph again improves σ_0 graph. Fig. 9 demonstrates the histogram of scores for 1,000 positive and 1,000 negative edges. The score, which ranges from 0 to 1, indicates the plausibility of the link existence. Positive edges (0.58±0.33) score higher than negative edges (0.13±0.15), with p-value less than 0.00001 using t-test, suggesting that UniCoRN is capable to distinguish positive and negative links in code graphs.

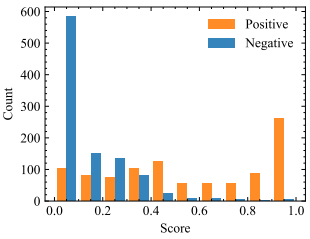

Fig. 9. Scores for positive and negative edges.

6 Conclusion

This paper presents a new model, UniCoRN, to provide a universal representation for code. Building blocks of UniCoRN include a novel σ graph to represent code as graphs, and four effective signals to pre-train GNNs. Our pre-training framework enables fine-tuning on various downstream tasks. Empirically, we show UniCoRN's superior ability to offer high-quality code representations.

There are several possibilities for future works. First, we are looking to enhance UniCoRN with additional code-specific signals. Second, the explainability of the learned code representation deserves further study. The explainability can in turn motivate additional signals to embed desired code properties. Third, more downstream applications are left to be explored, such as bug detection and duplicate code detection upon the availability of labeled data.

References

1. E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in ICLR, 2019.

2. J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in ESEC/FSE, pp. 964–974, 2019.
3. V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in PLDI, pp. 419–428, 2014.
4. M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in ICLR, 2018.
5. L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in AAAI, 2016.
6. W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," NeurIPS, 2020.
7. A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in ICSE, pp. 837–847, IEEE, 2012.
8. V. N. Ioannidis, D. Zheng, and G. Karypis, "Panrep: Universal node embeddings for heterogeneous graphs," DLG-KDD, 2020.
9. Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," TNNLS, 2020.
10. M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in ESWC, pp. 593–607, Springer, 2018.
11. W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, "Strategies for pre-training graph neural networks," ICLR, 2020.
12. W. Jin, T. Derr, H. Liu, Y. Wang, S. Wang, Z. Liu, and J. Tang, "Self-supervised learning on graphs: Deep insights and new direction," arXiv preprint, 2020.
13. A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," ICML, 2020.
14. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., "Codebert: A pre-trained model for programming and natural languages," EMNLP, 2020.
15. A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," ESEC/FSE, 2020.
16. U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," ICLR, 2019.
17. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," TOPLAS, 1987.
18. N. K. Ahmed, J. Neville, R. A. Rossi, N. G. Duffield, and T. L. Willke, "Graphlet decomposition: Framework, algorithms, and applications," KAIS, vol. 50, no. 3, pp. 689–722, 2017.
19. Y. Zhang and Q. Yang, "A survey on multi-task learning," CoRR, 2017.
20. P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," TACL, vol. 5, pp. 135–146, 2017.
21. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," ICLR, 2015.
22. 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, 2019.
23. L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," JMLR, vol. 9, no. Nov, pp. 2579–2605, 2008.
24. Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," ICLR, 2016.
25. B. Yang, W.-t. Yih, X. He, J. Gao, and L. Deng, "Embedding entities and relations for learning and inference in knowledge bases," ICLR, 2015.