Can Neural Networks Understand Programs like Humans?

Anonymous ACL submission

Abstract

Program understanding is a fundamental task in program language processing. Despite the success, existing works fail to take human minds as reference in understanding programs. In this paper, we incorporate human minds and propose the PGNN-EK model that consists of two main components. On the one hand, inspired by the “divide-and-conquer” reading behaviours of humans, we present a partitioning-based graph neural network model PGNN on the upgraded AST of codes. On the other hand, to characterize human minds of resorting to other resources to help code comprehension, we transform raw codes with external knowledge and apply pre-training techniques for information extraction. Finally, we combine the two embeddings generated from the two components to output code embeddings. We conduct extensive experiments to show the superior performance of PGNN-EK on the code summarization and code clone detection tasks. In particular, to show the generalization ability of our model, we release a new dataset that is more challenging for code clone detection and could advance the development of the community. Our codes and data are publicly available at https://github.com/anonymousforpaper1997/PGNN-EK.

1 Introduction

The past decades have witnessed the prosperity of programming platforms, such as Github and Stack Overflow. These platforms generate massive open-source code data that is named as “Big Code” in (Allamanis et al., 2018a). To automate the software development and maintenance, based on the “Software Naturalness” hypothesis (Hindle et al., 2016), natural language processing (NLP) techniques have been applied in program understanding. After that, a series of downstream programming language processing (PLP) tasks can be performed, including code summarization (Zhang et al., 2020; Ahmad et al., 2020; Liu et al., 2021) and code clone detection (Zhang et al., 2019; Wang et al., 2020).

Existing works for understanding programs mainly utilize three types of information: code context, code structure and external knowledge. Specifically, code context refers to the token sequence in the code. For code structure, each code can be parsed into various types of intermediate representations, such as AST (Abstract Syntax Tree), CFG (Control Flow Graph) and PDG (Program Dependence Graph). These representations capture the structural information of codes. Further, there also exists external knowledge associated with codes, such as API documentation and other exemplary codes. Despite the success, all these models ignore human minds in reading programs. Recently, (Bengio et al., 2021) suggest the potential futures of deep learning by comparing current AI methods with human learning abilities. This further prompts us to revisit program understanding: Can we develop a model that understands programs like humans?

In the domain of programming education, how people understand codes is a topic that has been studied. For example, based on knowledge base including syntactical knowledge (e.g., programming basics) and semantic knowledge (e.g., API documentation), (Schulte et al., 2010) offer a bottom-up reading technique, which assumes that people begin with individual code lines and chunks, and then combine them into higher-level abstractions. Further, (Park et al., 2016) state that when people read codes, reasoning about the hierarchical relationship of blocks, statements, expressions and variables is necessary. Based on these studies, we conclude three key points for human understanding codes. First, the transition of defined variables has to be traced. Second, humans usually adopt a “divide-and-conquer” strategy, which divides codes based

---

1 We interchangeably use code and program in this paper.
on statements and then understands codes from a local-to-global view. Third, humans resort to external knowledge to comprehend codes, such as API documentation and code examples written by experts.

In this paper, inspired by human minds for code comprehension, we propose a novel Partitioning-based Graph Neural Network with External Knowledge (PGNN-EK). To capture code context and structure, PGNN-EK upgrades the traditional AST and defines a novel subtoken-based AST called S-AST. In S-AST, we add edges between variables to trace the variable transitions, edges between adjacent tree leaves from left to right to enrich the context and structure information, and edges between sub-nodes corresponding to subtokens tokenized from user-defined identifiers to handle the Out of Vocabulary (OOV) problem (Karampatsis et al., 2020). Details will be illustrated later. After that, we first apply graph neural network (GNN) models on the S-AST to derive a code embedding. To further implement the “divide-and-conquer” reading strategy, we partition the S-AST into multiple subgraphs, which follow the sequence of statements in the original code. For each subgraph, we use GNN models to generate the subgraph embedding. Then, these subgraph embeddings are fused to generate another code embedding. For these two code embeddings, since they are both derived from S-AST, we further aggregate them. On the other hand, to characterize the dependence on external knowledge for code comprehension, we traverse the AST of the original code to derive a sequence of tokens for syntactic knowledge and then add the API descriptions to the end for semantic knowledge. We then apply CodeBERT (Feng et al., 2020) on the token sequence to capture external knowledge. Finally, PGNN-EK generates the output code embedding by combining the embedding derived from S-AST and the one from external knowledge.

To evaluate the model performance, we conduct experiments on the code summarization task and code clone detection task, respectively. Before we apply PGNN-EK on the code clone detection benchmarks in CodeXGLUE (Shi et al., 2021) extracted from the BigCloneBench 2014 dataset (Svajlenko et al., 2014), we notice from the leaderboard that the results are incredibly high, where the minimum F1 score is 0.949. Then we dive into the characteristics of the dataset and find that the functionalities of codes in the test set have all appeared in the training set. Therefore, the dataset is very simple. To further test the model’s generalization ability, we construct a new dataset, where the test set contains codes whose functionality has never appeared in the training set. This new dataset provides an insightful reference for further research in the community.

Our main contributions are summarized as follows:

- We construct a new code structure representation S-AST that can be used to handle the OOV problem in PLP.
- We follow human minds in understanding codes and propose a novel model PGNN-EK that leverages code context, structure and external knowledge. Specifically, we put forward a novel partitioning-based graph neural network model that can effectively use code context and structure. We also present a code transformation method to utilize external knowledge in boosting comprehension.
- We conduct extensive experiments on code summarization and code clone detection tasks to demonstrate the effectiveness of our model. In particular, we identify the limitation of a benchmark dataset for code clone detection and release a new dataset that is more challenging.

2 Related Work

2.1 Program Understanding

Program understanding is a topic that has received wide attention. Early works use either code context or structure information. For example, taking codes as raw texts, some works use language models (Raychev et al., 2014; Allamanis et al., 2015), RNN-series (Zaremba and Sutskever, 2014; Dam et al., 2016) and attention (Iyer et al., 2016) to represent codes. However, different from natural language, programs are more structural, which can be parsed into intermediate graphs, such as AST. Many works for code analysis are then proposed based on AST, such as AST-based LSTM (Wei and Li, 2017), AST-based CNN (Yu et al., 2019), ASTNN (Zhang et al., 2019), code2vec (Alon et al.,
public int getLarger(int a, 
int b) {
         a = Math.abs(a);
         b = Math.abs(b);
         … }

Figure 1: An example of S-AST. To simplify the graph, we create a code snippet (top left), whose variables are defined with only one character, such as “a” and “b”. In real tasks, the codes are longer and user-defined identifiers are more semantically complex. This could add more subtoken nodes and edges. The figure is better viewed in color.

Despite the success, all these methods only consider the code context and structure information. There are also approaches that utilize the external knowledge associated with codes. For example, some methods apply pre-training techniques in NLP to boost comprehension, such as CodeBERT (Feng et al., 2020), GPT-C (Svyatkovskiy et al., 2020) and PLBART (Ahmad et al., 2021). There are also works that incorporate code characteristics into pre-training models, such as GraphCodeBERT (Peng et al., 2021), OSCAR (Peng et al., 2021) and InferCode (Bui et al., 2021). Further, API is another external source for program understanding, which has been introduced in many works (Hu et al., 2018; Xu et al., 2020). However, all these methods ignore human minds in program understanding.

2.2 Code Summarization and Code Clone Detection

In this paper, we focus on two program understanding downstream tasks: code summarization and code clone detection. For code summarization, some works (Iyer et al., 2016; Ahmad et al., 2020) use code context only, some methods (LeClair et al., 2019; Alon et al., 2019a) use code structure only, while there are also models (Hellendoorn et al., 2020; Zügner et al., 2021) that use both information. Further, Liu et al. (2021) introduce external knowledge for performance improvement. For code clone detection, existing works mainly employ code structure (Wei and Li, 2017; Zhang et al., 2019; Wang et al., 2020) and pre-training models (Feng et al., 2020; Ahmad et al., 2021).

3 S-AST Construction

In this section, we construct S-AST. The original AST has two main limitations:

- **Low connectivity.** The original AST is actually tree-structured, where every two nodes are minimally connected with only one path. This could lead to a long distance between leaf nodes. As pointed out in (Alon and Yahav, 2021), directly applying GNN models in tree-shaped graphs could cause the long-range problem.
• OOV problem. User-defined identifiers in
codes can be arbitrarily complex and most
of them are compound words, which could
induce a large vocabulary size. For exam-
ple, the training set size in the benchmark
dataset CodeXGLUE (Lu et al., 2021) for
code summarization is 164,814, while the
vocabulary size for AST nodes is 620,256.
After we split the nodes by camel case and
underscores (Cvitkovic et al., 2019), the
vocabulary size is still as high as 201,286. A
very large vocabulary could cause the OOV
problem (Jean et al., 2015) and thus adversely
affect the model performance.

To improve the connectivity of the AST, there
exist some works (Allamanis et al., 2018b; Wang
et al., 2020; Wang and Li, 2021) that add edges
to the AST. However, these methods cannot ad-
dress the OOV problem. Therefore, we propose a
new code intermediate graph S-AST, as shown in
Figure 1. Similar as in (Allamanis et al., 2018b;
Wang et al., 2020), we add data flow edges to trace
variable transitions and connect adjacent leaf nodes
to encourage learning from contexts. To solve the
OOV problem, we further reduce the vocabulary
size by using the tokenizer of RoBERTa (Liu et al.,
2019) to tokenize every leaf node in the AST. When
a leaf node can be tokenized into multiple subto-
kens, we keep the first subtree as the parent node
and take other subtrees as its children. For ex-
ample, the token “getLarger” is divided into the
parent node “get” and the children nodes “L” and
“arger”. These new parent-children connections are
defined as subtoken edges. With these three types
of edges added, we increase the number of edges
in the AST and improve the graph connectivity.
Further, the vocabulary size could be significantly
reduced. In our experiments, we use javalang\(^3\)
to generate Java AST and reduce the vocabulary
size to 50,336, where 50,265 is the size of original
RoBERTa vocabulary and 71 is the number of
keywords in non-leaf nodes defined by javalang.

4 Algorithm

In this section, we introduce the PGNN-EK model,
which is composed of two main components. On
the one hand, the partitioning-based graph neu-
nal network model (PGNN) is proposed to follow
the “divide-and-conquer” behaviours of humans to
understand programs. On the other hand, PGNN-
EK leverages external knowledge to enhance the
model’s capability. The overall architecture of
PGNN-EK is summarized in Figure 2.

\(\text{Figure 2: The overall architecture of PGNN-EK}\)

4.1 Partitioning-based Graph Neural
Networks

As illustrated in (Schulte et al., 2010) and (Park
et al., 2016), the bottom-up reasoning on the hierar-
chical relationship of statements plays an essential
role in human understanding. Therefore, we pro-
pose a statement-based partitioning algorithm to
divide S-AST into multiple subgraphs. Since S-
AST is no longer a tree, for convenience, we first
keep subtrees and their edges in-between in S-
AST, and remove edges linking variables and those
connecting adjacent leaf nodes, to derive a tree
structure. After that, we calculate the number of
nodes in each subtree of the root node and each
subtree corresponds to a statement of the raw code.
Then, we accumulate the number of nodes in sub-
trees from left to right. When the sum exceeds the
pre-defined threshold \(\lambda\), we group these subtrees
into one subgraph and reset the sum to zero. If
the current subgraph is not the first one, for each
variable node in it, we also add to the subgraph
the closest node indicating the same variable in
previous subgraphs to trace the variable transition.
After the subgraph is derived, we add edges be-
tween nodes that represent the same variable and
also connect adjacent leaf nodes as in the original
S-AST. We repeat this process until all subtrees are
visited. Note that if the node number of the last
subgraph is smaller than \(\lambda/2\), we merge the last
subgraph into the penultimate subgraph. Finally,
we summarize the pseudocodes of the partitioning
algorithm in Alg. 1.

After subgraphs are derived, as in (Hellendoorn
et al., 2020), we adopt GGNN (Li et al., 2016) as
the graph embedding model, which uses a multi-

\(^3\)https://github.com/c2nes/javalang
layer perceptron (MLP) and a gated recurrent unit (GRU) to perform message passing and embedding updating. Specifically, at the \((l+1)\)-th layer, to update the embedding \(h_i^{l+1}\) of node \(x_i\), we have:

\[
\begin{align*}
    m_i^{l+1} &= \sum_{j \in \mathcal{N}_i} \text{MLP}(h_j^l, e_{ij}), \\
    h_i^{l+1} &= \text{GRU}(m_i^{l+1}, h_i^l),
\end{align*}
\]

where \(\mathcal{N}_i\) is the neighbor set of \(x_i\) and \(e_{ij}\) is the feature vector of the edge between \(x_i\) and \(x_j\). After node embeddings are generated, we use a READOUT function to obtain the graph embedding \(G\):

\[
G = \text{READOUT}(\{h_i\}).
\]

We repeat the above process on each subgraph to derive a list of subgraph embeddings \(L = \{G_1, G_2, \ldots, G_n\}\), where \(n\) is the number of subgraphs. Next, we keep the order of the subgraph list and feed \(L\) into an unidirectional LSTM:

\[
O = \text{LSTM}(L).
\]

Inspired by the skip connection (He et al., 2016), we also perform GGNN on the whole S-AST graph to derive a code embedding \(C\). Finally, we concatenate \(C\) and the last output \(O[-1]\) of LSTM. We further feed the result into a fully connected layer to get the output code embedding \(E_p\):

\[
E_p = \text{FC}(\text{Concat}(C, O[-1])).
\]

### 4.2 External Knowledge

To help understand programs, people often resort to external knowledge. For example, humans usually learn from massive exemplary codes written by experts for better syntactic comprehension, which are in the format of programming language. Further, API documentation is written in natural language and provides semantic details on functions. Therefore, a research question arises: \textit{how to fuse these external syntactic and semantic knowledge into our model?}

To address the problem, we use pre-training techniques in programming language processing (PLP), which are trained on massive code corpus to learn programming basics. In particular, we adopt CodeBERT (Feng et al., 2020), which is a bimodal pre-trained model for both programming language and natural language.

Before CodeBERT is applied, we first combine the raw code and API descriptions. To enrich the syntactic information contained in the raw code, we perform pre-order traversal on the AST of the code to obtain a sequence of tokens and replace the raw code. This is because the AST includes extra code-related information, such as statements, variables and operations. Then we append the corresponding API description to the end. A toy example of transformation is shown in Figure 3. Finally, we feed the transformed context \(T\) into the pre-trained CodeBERT\(^4\) and obtain the embedding \(E_c\):

\[
E_c = \text{CodeBERT}(T).
\]

Finally, we concatenate the output embeddings of PGNN and CodeBERT, and feed the result into a fully connected layer to obtain the final embedding \(E_f\):

\[
E_f = \text{FC}(\text{Concat}(E_p, E_c)).
\]

### 5 Experiments

In this section, we evaluate the performance of PGNN-EK. We conduct experiments on two program understanding tasks: code summarization and code clone detection. For each task, we use two benchmark datasets, whose statistics are listed in Table 1.

#### 5.1 Implementation details

In our experiments, we use the AdamW optimizer and linear schedule from (Wolf et al., 2020) to update model parameters. For fair comparison, we run all experiments on 2 Tesla V100 with 32G memory. For PGNN, we set the number of GNN layers, the number of LSTM layers, the embedding size of GNN node, and the embedding size of LSTM hidden layer to 3, 2, 768 and 768, respectively. We choose the mean operator as the READOUT function. To avoid overfitting, we set the dropout rate to 0.2 in PGNN. We implement GNNs

\(^4\)https://huggingface.co/microsoft/codebert-base
Table 1: The statistics of datasets

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Training</th>
<th>Validation</th>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code summarization</td>
<td>CodeSearchNet-Java (CSN)</td>
<td>164,814</td>
<td>5,179</td>
<td>10,952</td>
<td>Provided by CodeXGLUE</td>
</tr>
<tr>
<td></td>
<td>TL-CodeSum (TLC)</td>
<td>69,708</td>
<td>8,714</td>
<td>8,714</td>
<td>Original</td>
</tr>
<tr>
<td>Code clone detection</td>
<td>BigCloneBench (BCB)</td>
<td>901,028</td>
<td>415,416</td>
<td>415,416</td>
<td>Provided by CodeXGLUE</td>
</tr>
<tr>
<td></td>
<td>BigCloneBench-Function (BCB-F)</td>
<td>398,110</td>
<td>78,602</td>
<td>81,202</td>
<td>Split by functionality</td>
</tr>
</tbody>
</table>

5.2 Code Summarization

Code summarization aims at generating natural language comments for codes. We evaluate the performance of PGNN-EK on two benchmark datasets, which are TL-CodeSum (shorted as TLC) (Hu et al., 2018) and the Java subset of CodeSearchNet (shorted as CSN) (Husain et al., 2019). For TLC, we use the original dataset. For CSN, we use the version provided by CodeXGLUE (Lu et al., 2021). For fair comparison, we use the smoothed BLEU-4 score (Lin and Och, 2004) as in CodeXGLUE. The larger the score, the better the model performance. We compare our model with five representative baselines, including CodeNN (Iyer et al., 2016), NCS (Ahmad et al., 2020), Rencos (Zhang et al., 2020), CodeBERT (Feng et al., 2020) and PLBART (Ahmad et al., 2021). Due to the space limitation, we move the details of these baselines to Appendix C.

Table 2 shows the code summarization results. Note that the results of CodeNN, NCS and Rencos are directly taken from (Shi et al., 2021). Also, the results of CodeBERT and PLBART on CSN are derived from the leaderboard of CodeXGLUE. For their results on TLC, we run the codes released by the authors of the paper and set hyper-parameters according to the original paper. From the table, we see that, due to the fusion of external knowledge, pre-training models CodeBERT, PLBART and PGNN-EK outperform other models on both datasets. Further, PGNN-EK performs the best. The gaps between PGNN-EK and the runner-up model PLBART on CSN and TLC are 0.5 and 1.05, respectively. This shows the importance of considering human minds for code comprehension. We also observe that scores on TLC are substantially larger than that on CSN. This is because codes in the training set and the test set of TLC are considerably more similar in functionalities, which will be elaborated in the next section.

Table 2: Code summarization results. We highlight the best results in bold.

<table>
<thead>
<tr>
<th>Model</th>
<th>CSN</th>
<th>TLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeNN</td>
<td>8.58</td>
<td>33.03</td>
</tr>
<tr>
<td>NCS</td>
<td>11.19</td>
<td>44.25</td>
</tr>
<tr>
<td>Rencos</td>
<td>11.80</td>
<td>46.81</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>17.65</td>
<td>48.53</td>
</tr>
<tr>
<td>PLBART</td>
<td>18.45</td>
<td>50.01</td>
</tr>
<tr>
<td>PGNN-EK</td>
<td>18.95</td>
<td>51.06</td>
</tr>
</tbody>
</table>

5.3 Code Clone Detection

The goal of code clone detection is to detect whether two code fragments implement the same functionality. Following (Zhang et al., 2019; Wang et al., 2020), we use the BigCloneBench 2014 dataset (Svajlenko et al., 2014) and adopt the version provided by CodeXGLUE. We short it as BCB. Before we apply PGNN-EK on BCB, we notice from the leaderboard of CodeXGLUE that the results on BCB are incredibly high, where the minimum F1 score is 0.949. Then we dive into the characteristics of the dataset and compare BCB with...
We find that the functionalities of codes in the test set have all appeared in the training set of BCB. Therefore, BCB is a very simple dataset. To test the model’s generalization ability, we construct a new dataset, named BCB-F, where the test set contains codes whose functionality has never appeared in the training set. We first extract codes from the new version benchmark (Svajlenko and Roy, 2015) that has more code fragments and code functionalities. We next split training/validation/test set based on code functionalities. Specifically, we construct training/validation/test set with 22/11/10 code functionalities. For details on the functionality splits of BCB and BCB-F, see Appendix D. We keep the same number of positive and negative samples in all the three sets. The comparison between BCB and BCB-F is given in Table 3.

In addition to the pre-training models CodeBERT and PLBART, we further compare our model with two representative methods in code clone detection, which are ASTNN (Zhang et al., 2019) and FA-AST (Wang et al., 2020) (For the details of these baselines, see Appendix C).

Table 4 shows the evaluation results on the two datasets. For BCB, we take the results of other baseline methods from CodeXGLUE. For BCB-F, we run the source codes released by their authors to obtain the results. From the table, we observe: 1) All models perform very well on BCB, indicating that the dataset is very simple. However, the best F1 score on BCB-F is only 0.724, which shows that this dataset is very challenging. 2) The non-pre-training models ASTNN and FA-AST predict all samples to be positive and perform poorly on BCB-F, while pre-training models perform better. This further demonstrates the importance of introducing external knowledge. 3) PGNN-EK achieves the best results on both datasets. This shows that incorporating human minds in program understanding enhances the generalization ability of PGNN-EK.

5.4 Ablation Study

We further conduct ablation study to verify the importance of its main components in PGNN-EK, including subtokens, the S-AST graph, the partitioning-based GNN and the external knowledge. Specifically, one variant employs only the S-AST graph without using external knowledge. This helps us realize the importance of external knowledge in program understanding. We call this variant PGNN only. Meanwhile, we define another variant that ignores the hierarchical relationships in code structure and uses only external knowledge. We call this variant EK only. To further show the significance of S-AST in code understanding, we replace S-AST with the original AST in the variant PGNN-EK with AST. We also implement a variant that does not use the subtoken tokenizer to generate extra subtoken nodes and edges. We call it PGNN-EK without subtoken. This variant can be used to show the importance of subtokens in addressing the OOV problem. To show the advantage of the partitioning strategy, we propose a variant GNN-EK that discards the partitioning step. Finally, we consider a variant that feeds the raw code into the pre-trained CodeBERT without transforming it with external knowledge. We call this variant PGNN-CodeBERT.

Table 5 summarizes the ablation study results. From the table, we see that: 1) S-AST contains richer information than AST and can serve as an effective code intermediate representation in program understanding. The introduction of subtokens nodes and edges alleviates the OOV problem and enhances the model performance. 2) External knowledge helps boost understanding codes. In particular, code transformation with external knowledge in program understanding.

---

The original benchmark (Svajlenko et al., 2014).

Table 3: Comparisons between BCB and BCB-F

<table>
<thead>
<tr>
<th>Code fragments</th>
<th>BCB</th>
<th>BCB-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionalities</td>
<td>9134</td>
<td>73182</td>
</tr>
<tr>
<td>Training/Test splitting</td>
<td>nearly 2:1</td>
<td>1:1</td>
</tr>
</tbody>
</table>

Table 4: Code clone detection results w.r.t. precision (P), recall (R) and F1 measures. We highlight the best results in bold.

<table>
<thead>
<tr>
<th>Model</th>
<th>BCB-P</th>
<th>BCB-R</th>
<th>BCB-F</th>
<th>BCB-F-P</th>
<th>BCB-F-R</th>
<th>BCB-F-F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTNN FA-AST</td>
<td>0.92</td>
<td>0.94</td>
<td>0.93</td>
<td>0.50</td>
<td>0.50</td>
<td>0.67</td>
</tr>
<tr>
<td>CodeBERT PLBART</td>
<td>0.96</td>
<td>0.94</td>
<td>0.95</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PGNN-EK</td>
<td>0.975</td>
<td>0.973</td>
<td>0.974</td>
<td>0.621</td>
<td>0.869</td>
<td>0.724</td>
</tr>
</tbody>
</table>

5.4 Ablation Study

We further conduct ablation study to verify the importance of its main components in PGNN-EK, including subtokens, the S-AST graph, the partitioning-based GNN and the external knowledge. Specifically, one variant employs only the S-AST graph without using external knowledge. This helps us realize the importance of external knowledge in program understanding. We call this variant PGNN only. Meanwhile, we define another variant that ignores the hierarchical relationships in code structure and uses only external knowledge. We call this variant EK only. To further show the significance of S-AST in code understanding, we replace S-AST with the original AST in the variant PGNN-EK with AST. We also implement a variant that does not use the subtoken tokenizer to generate extra subtoken nodes and edges. We call it PGNN-EK without subtoken. This variant can be used to show the importance of subtokens in addressing the OOV problem. To show the advantage of the partitioning strategy, we propose a variant GNN-EK that discards the partitioning step. Finally, we consider a variant that feeds the raw code into the pre-trained CodeBERT without transforming it with external knowledge. We call this variant PGNN-CodeBERT.

Table 5 summarizes the ablation study results. From the table, we see that: 1) S-AST contains richer information than AST and can serve as an effective code intermediate representation in program understanding. The introduction of subtokens nodes and edges alleviates the OOV problem and enhances the model performance. 2) External knowledge helps boost understanding codes. In particular, code transformation with external knowledge in program understanding.

---

\[\text{Ratio of positive-negative: nearly 2:1}\]

---

\[\text{Table 4 shows the evaluation results on the two datasets. For BCB, we take the results of other baseline methods from CodeXGLUE. For BCB-F, we run the source codes released by their authors to obtain the results. From the table, we observe: 1) All models perform very well on BCB, indicating that the dataset is very simple. However, the best F1 score on BCB-F is only 0.724, which shows that this dataset is very challenging. 2) The non-pre-training models ASTNN and FA-AST predict all samples to be positive and perform poorly on BCB-F, while pre-training models perform better. This further demonstrates the importance of introducing external knowledge. 3) PGNN-EK achieves the best results on both datasets. This shows that incorporating human minds in program understanding enhances the generalization ability of PGNN-EK.}\]
knowledge improves the expressiveness of the raw code. 3) The full model PGNN-EK outperforms other variants on all the datasets and tasks. This indicates the importance of every main component in PGNN-EK. It further shows that leveraging code context, code structure and external knowledge as humans is helpful for program understanding.

5.5 The Influence of Subgraph Size

We end this section with a hyper-parameter sensitivity analysis. In PGNN-EK there is a key hyper-parameter $\lambda$ that is used to control the size of subgraphs. Here, we investigate the sensitivity of $\lambda$. We vary the value of $\lambda$ from \{10, 30, 50, 70, 90, 110, 130, 150, 170, 190\}, and the final prediction results of PGNN-EK on 4 datasets are shown in the Figure 4.

![Figure 4: The influence of subgraph size on 4 datasets.](image)

Table 6: The average number of nodes in S-AST

<table>
<thead>
<tr>
<th>Datasets</th>
<th>CSN</th>
<th>TLC</th>
<th>BCB</th>
<th>BCB-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-AST size</td>
<td>137</td>
<td>140</td>
<td>372</td>
<td>348</td>
</tr>
</tbody>
</table>

The results indicate that 1) the model performance first increases and then drops, with the increase of the subgraph size. When the subgraph size is too small, each subgraph is a code fragment that no longer represents a code statement and thus contains less information. Further, when the subgraph is too large, each subgraph could be composed of statements that are of different semantic meanings, which thus degrades the model performance. 2) PGNN-EK performs the best at $\lambda = 30$ on CSN and TLC while it achieves the best results at $\lambda = 70$ on BCB and BCB-F. We further investigate the reason and show the average number of nodes in S-AST on the four datasets in Table 6. From the table, BCB and BCB-F contain $\sim 2.5$ times more nodes than that in CSN and TLC.

Table 5: Ablation study on PGNN-EK. We highlight the best results in bold.

<table>
<thead>
<tr>
<th>Method</th>
<th>CSN (Smoothed BLEU-4)</th>
<th>TLC (Smoothed BLEU-4)</th>
<th>BCB (F1)</th>
<th>BCB-F (F1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGNN only</td>
<td>14.05</td>
<td>47.71</td>
<td>0.951</td>
<td>0.667</td>
</tr>
<tr>
<td>EK only</td>
<td>17.95</td>
<td>49.66</td>
<td>0.965</td>
<td>0.711</td>
</tr>
<tr>
<td>PGNN-EK with AST</td>
<td>17.70</td>
<td>48.96</td>
<td>0.957</td>
<td>0.713</td>
</tr>
<tr>
<td>PGNN-EK without subtoken</td>
<td>17.82</td>
<td>49.01</td>
<td>0.958</td>
<td>0.712</td>
</tr>
<tr>
<td>GNN-EK</td>
<td>18.05</td>
<td>49.95</td>
<td>0.967</td>
<td>0.715</td>
</tr>
<tr>
<td>PGNN-CodeBERT</td>
<td>18.60</td>
<td>50.65</td>
<td>0.969</td>
<td>0.720</td>
</tr>
<tr>
<td>PGNN-EK (Full Model)</td>
<td><strong>18.95</strong></td>
<td><strong>51.06</strong></td>
<td><strong>0.974</strong></td>
<td><strong>0.724</strong></td>
</tr>
</tbody>
</table>

6 Conclusion

In this paper, we followed human understandings for programs and proposed the PGNN-EK model. To enrich the code structure information and alleviate the OOV problem, we presented the S-AST graph based on AST, which uses a subtoken tokenizer to generate subtoken nodes and edges between them. Inspired by the “divide-and-conquer” strategy, we proposed the partitioning-based graph neural network model on S-AST that employs code context and structure. To leverage the external knowledge to boost comprehension, we transformed the raw code to fuse syntactic and semantic knowledge and utilized pre-training techniques for information extraction. We performed extensive experiments to show the effectiveness of our model PGNN-EK on the code summarization and code clone detection tasks. In particular, to show the generalization ability of the model, we released a new benchmark that is more challenging.
References


Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In IJCAI 2017.


A  Partitioning S-AST Algorithm

Algorithm 1 Partitioning S-AST

**Input:** A S-AST $\mathcal{T}$ with node features $\mathcal{X}$, edge indexes $\mathcal{I}$ and edge features $\mathcal{E}$

**Parameter:** $\lambda$, which specifies the minimum number of nodes in the subgraph

**Output:** Nodes features list $\mathcal{L}_x$, edge indexes list $\mathcal{L}_i$, and edge features list $\mathcal{L}_e$ of subgraphs

1. Derive a tree structure $\mathcal{T}'$ by removing data flow edges and adjacent leaf edges in $\mathcal{T}$;
2. $\text{nodes}_\text{sum} \leftarrow 0, \text{nodes}_\text{set} \leftarrow \{\}$;
3. $\text{n}_f\_\text{list, e}_i\_\text{list, e}_f\_\text{list, L}_x, L_i, L_e \leftarrow \{\}$;
4. Obtain a subtree list $\{S\}$ based on subtrees of root nodes in $\mathcal{T}'$ from left to right;
5. for $S$ in $\{S\}$ do
6. $n \leftarrow$ the number of nodes in $S$;
7. $\text{nodes}_\text{sum} \leftarrow \text{nodes}_\text{sum} + n$;
8. Add nodes in $S$ to $\text{nodes}_\text{set}$;
9. if $\text{nodes}_\text{sum} \geq \lambda$ or $S$ is the last element of $\{S\}$ then
10. if $\mathcal{L}_x \neq \emptyset$ then
11. Add closest nodes that indicate the same variables in $\mathcal{L}_x$ to $\text{nodes}_\text{set}$ ;
12. end if
13. Assign $\text{n}_f\_\text{list, e}_i\_\text{list, e}_f\_\text{list}$ based on $\text{nodes}_\text{set}, \mathcal{X}, \mathcal{I}$ and $\mathcal{E}$;
14. Append $\text{n}_f\_\text{list, e}_i\_\text{list, e}_f\_\text{list}$ to $\mathcal{L}_x, \mathcal{L}_i, \mathcal{L}_e$ respectively;
15. $\text{nodes}_\text{sum} \leftarrow 0, \text{nodes}_\text{set} \leftarrow \{\}$;
16. end if
17. end for
18. $\text{A}[-i]$ denotes the $i$-th element from the bottom in $\mathcal{A}$.
19. if size of $\mathcal{L}_x[-1] < \lambda/2$ and size of $\mathcal{L}_x > 1$ then
20. Merge $\mathcal{L}_x[-1]$ and $\mathcal{L}_x[-2], \mathcal{L}_i[-1]$ and $\mathcal{L}_i[-2], \mathcal{L}_e[-1]$ and $\mathcal{L}_e[-2]$, respectively;
21. end if
22. return $\mathcal{L}_x, \mathcal{L}_i, \mathcal{L}_e$

B  Examples of API-Description Pairs

In the experiment, we obtain 51, 191 method description pairs after preprocessing, and Table 7 gives some examples.

C  Baselines Introduction

We compare our model with five representative models in code summarization task:

- CodeNN (Iyer et al., 2016) is the first method that applies deep neural networks in code summarization. It uses a classical attention-based encoder-decoder framework from Neural Machine Translation (NMT).
- NCS (Ahmad et al., 2020) applies Transformer (Vaswani et al., 2017) to model the pairwise relationship between code tokens and capture their long-term dependencies.
- Rencos (Zhang et al., 2020) proposes an attention-based encoder-decoder model and enhance it with the most similar code snippets retrieved from the training set.
- CodeBERT (Feng et al., 2020) is a bimodal pre-training model for programming and natural languages based on RoBERTa (Liu et al., 2019).
- PLBART (Ahmad et al., 2021) is a sequence-to-sequence pre-training model based on BART (Lewis et al., 2020).

In addition to the pre-training models CodeBERT and PLBART, we further compare our model with two representative model in code clone detection task:

- ASTNN (Zhang et al., 2019) proposes an AST-based neural network that splits AST into a sequence of statement trees and applies a bidirectional RNN model to produce source code representation. However, it ignores external knowledge associated with codes.
- FA-AST (Wang et al., 2020) augments original AST with explicit control and data flow edges, then introduces two different types of GNNs to detect code clones.

D  Functionalities Splits in BCB and BCB-F

For BCB, the functionalities in Train/Val/Test set are:

- **Train:** Web Download, Secure Hash(MD5), Copy a File, Decompress Zip, FTP Authenticated Login, Bubble Sort, Init. SGV with Model, SGV Selection Event Handler, Create Java Project(Eclipse), SQL Update and RollBACK.
- **Val:** Same to **Train**.
Table 7: Examples of API-Description Pairs

<table>
<thead>
<tr>
<th>APIs</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs</td>
<td>Returns the absolute value of an int value.</td>
</tr>
<tr>
<td>Arrays.hashcode</td>
<td>Returns a hash code based on the contents of the specified array.</td>
</tr>
<tr>
<td>Scanner.hasNext</td>
<td>Returns true if this scanner has another token in its input.</td>
</tr>
<tr>
<td>Color.getRGB</td>
<td>Returns the RGB value representing the color in the default sRGB ColorModel.</td>
</tr>
</tbody>
</table>

- **Test**: Same to **Train**.

For BCB-F, the functionalities in Train/Val/Test set are, where the emphasis discloses the whole 10 functionalities that exist in BCB:

- **Train**: Decompress Zip, Copy a File, Get Prime Factors, File Dialog, Resize Array, Get MAC Address String, Parse CSV File, Secure Hash(MD5), Send Email, Load Custom Font, Create Java Project(Eclipse), Extract Matches Using Regex, Open File in Desktop Application, Connect to Database, Load File to Byte Array, Call Method Using Reflection, Take Screenshot to File, Write PDF File, Delete Folder and Contents, Copy Directory, Binary Search, Delete Folder and Contents.

- **Val**: SQL Update and RollBACK, Bubble Sort, Execute External Process, XMPP Send Message, Zip Files, Convert Date String Format, Secure Hash, GCD, SGV Selection Event Handler, Init. SGV with Model, Play Sound.

- **Test**: Shuffle Array in Place, Create Encryption Key Files, Load Custom Font, Encrypt to File, Parse XML to DOM, CRC32 File Checksum, Transpose a Matrix, Test Palindrome, Web Download, FTP Authenticated Login.