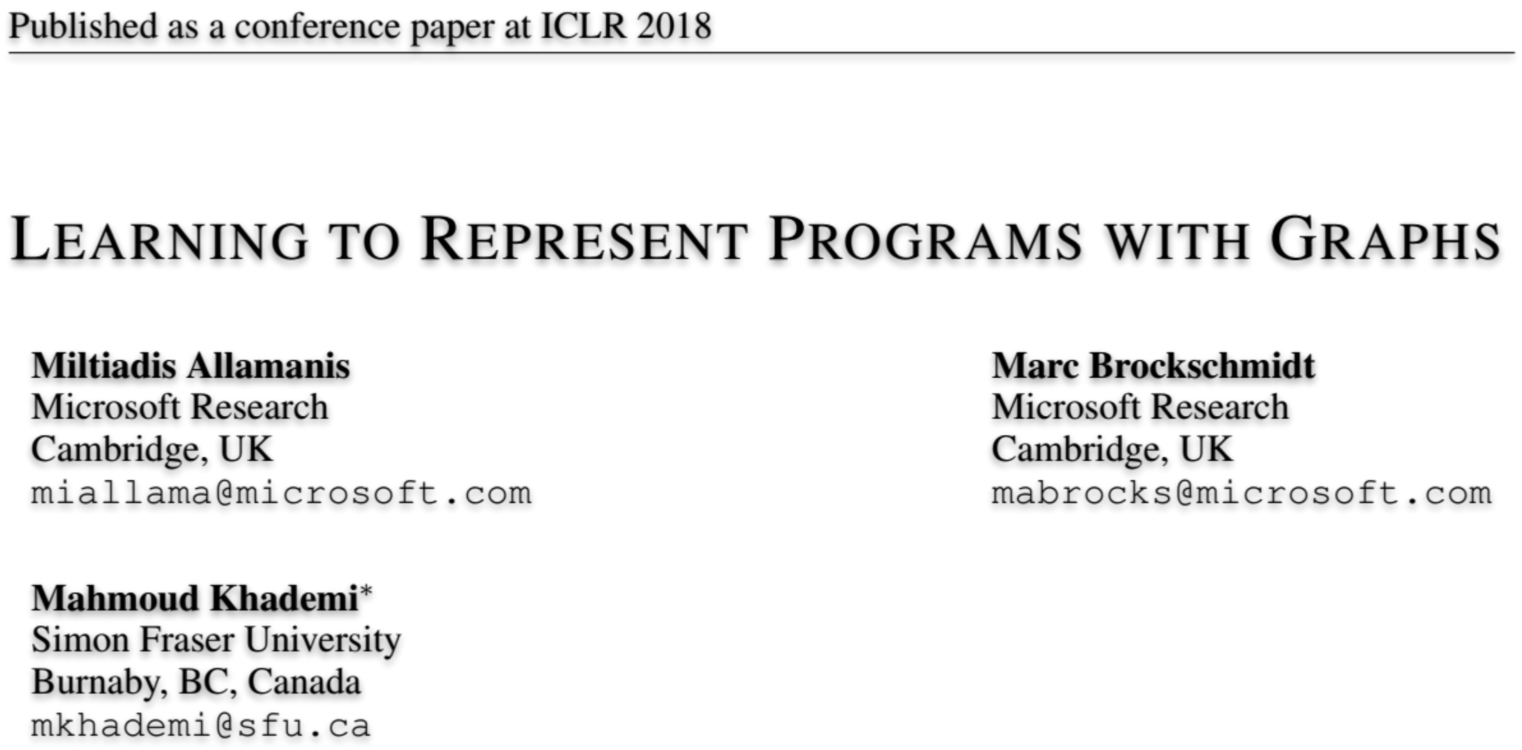
LEARNING TO REPRESENT PROGRAMS WITH GRAPHS

Toronto Deep Learning Series
June 25, 2018
Amir Feizpour



Published as a conference paper at ICLR 2018

Learning to Represent Programs with Graphs

Miltiadis Allamanis
Microsoft Research
Cambridge, UK
miallama@microsoft.com

Marc Brockschmidt
Microsoft Research
Cambridge, UK
mabrocks@microsoft.com

Mahmoud Khademi*
Simon Fraser University
Burnaby, BC, Canada
mkhademi@sfu.ca

https://arxiv.org/pdf/1711.00740.pdf

sema

Home | Product | About | What We Believe | Lokakṣema | Blog

Fixing the past so that you can change the future

Automated code maintenance through smart systems

Member of

NVIDIA

INCEPTION PROGRAM

Want more info?

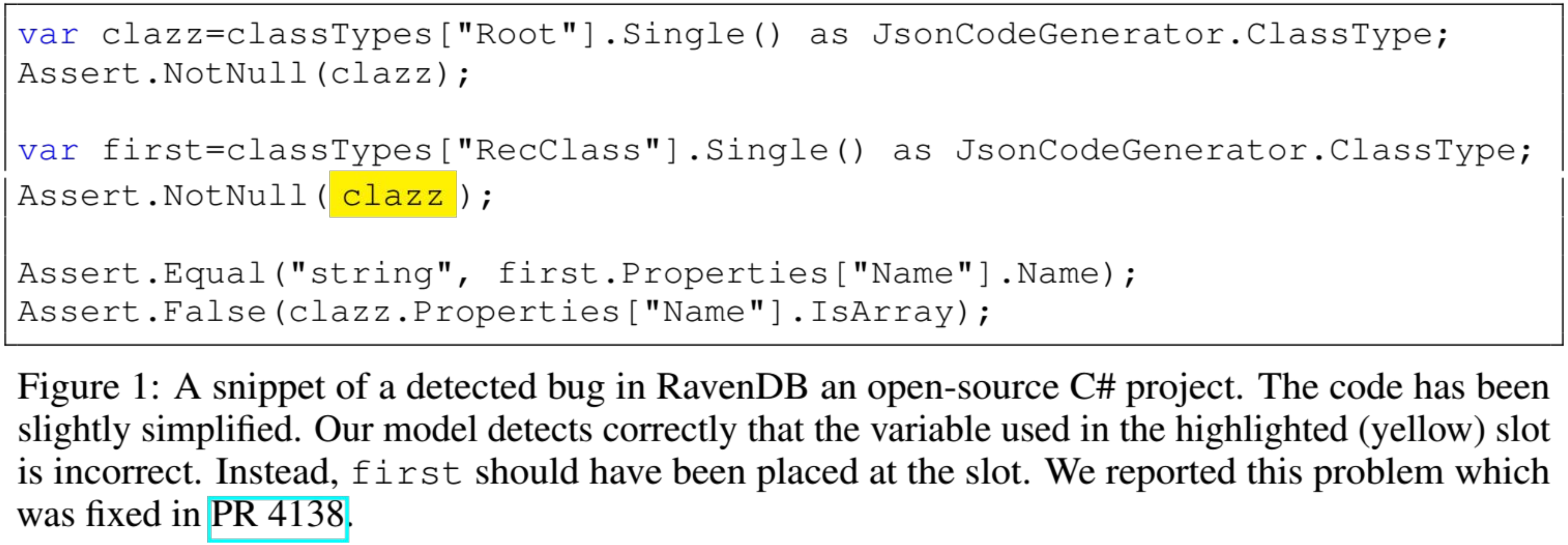
Email Address	Contact Me

ABSTRACT

Learning tasks on source code (*i.e.*, formal languages) have been considered recently, but most work has tried to transfer natural language methods and does not capitalize on the unique opportunities offered by code's known sematics. For example, long-range dependencies induced by using the same variable or function in distant locations are often not considered. We propose to use graphs to represent both the syntactic and semantic structure of code and use graph-based deep learning methods to learn to reason over program structures.

In this work, we present how to construct graphs from source code and how to scale Gated Graph Neural Networks training to such large graphs. We evaluate our method on two tasks: VARNAMING, in which a network attempts to predict the name of a variable given its usage, and VARMISUSE, in which the network learns to reason about selecting the correct variable that should be used at a given program location. Our comparison to methods that use less structured program representations shows the advantages of modeling known structure, and suggests that our models learn to infer meaningful names and to solve the VARMISUSE task in many cases. Additionally, our testing showed that VARMISUSE identifies a number of bugs in mature open-source projects.

example



```
var clazz=classTypes["Root"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

var first=classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull( clazz );

Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

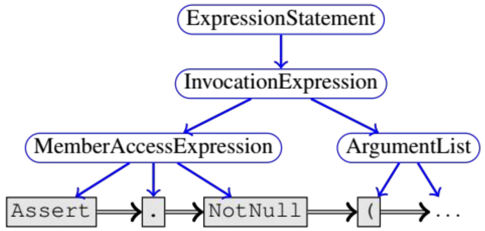
Figure 1: A snippet of a detected bug in RavenDB an open-source C# project. The code has been slightly simplified. Our model detects correctly that the variable used in the highlighted (yellow) slot is incorrect. Instead, `first` should have been placed at the slot. We reported this problem which was fixed in PR 4138.

Tasks

- VarNaming
  - Given a program and a variable *v*
  - Replace all occurrences of *v* with a placeholder: <SLOT>
  - Then predict the string corresponding to *v* using a generated sequence of sub-tokens
- VarMisuse
  - Given a program and a variable *v*
  - Replace *v* at a particular location with a placeholder: <SLOT>
  - Predict the correct *v* within the set of type-correct variables within the scope

Representing programs as graphs

- Why?
  - To capture both syntactic and semantic nature of the programming language
- How?
  - Abstract Syntax Tree (AST)
  - Data flow

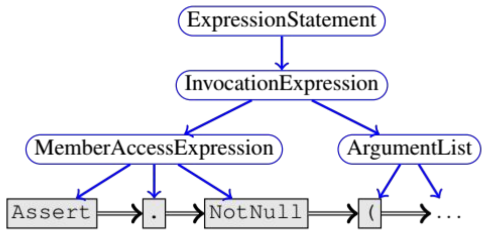


(a) Simplified syntax graph for line 2 of Fig. 1, where blue rounded boxes are syntax nodes, black rectangular boxes syntax tokens, blue edges Child edges and double black edges NextToken edges.

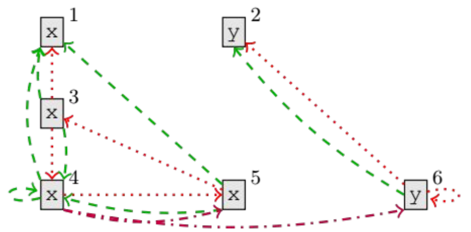
Figure 2: Examples of graph edges used in program representation.

Representing programs as graphs

- Why?
  - To capture both syntactic and semantic nature of the programming language
- How?
  - Abstract Syntax Tree (AST)
  - Data flow



(a) Simplified syntax graph for line 2 of Fig. 1, where blue rounded boxes are syntax nodes, black rectangular boxes syntax tokens, blue edges Child edges and double black edges NextToken edges.



(b) Data flow edges for $(\boxed{x}^1, \boxed{y}^2)$ `= Foo(); while` $(\boxed{x}^3$ `> 0)` $\boxed{x}^4 = \boxed{x}^5 + \boxed{y}^6$ (indices added for clarity), with red dotted LastUse edges, green dashed LastWrite edges and dashdotted purple ComputedFrom edges.

Figure 2: Examples of graph edges used in program representation.

https://astexplorer.net/

Side note: data flow edges

LastRead

LastWrite

ComputedFrom

LastLexicalUse

ReturnsTo

FormalArgName

GuardedBy

GuardedByNegation

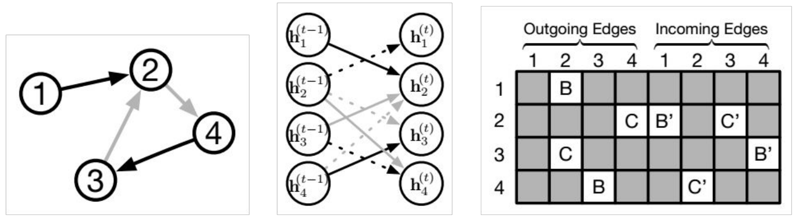
Graph Neural Network

Objective is to learn a faithful representation of the information encoded in the graph to be used for ML tasks where the inputs are graphs

Examples: obtaining properties of molecules, analysing source code, reasoning on knowledge graph, ...

https://arxiv.org/pdf/1511.05493.pdf

Graph Neural Network



(a) (b) (c) $\mathbf{A} = \left[\mathbf{A}^{(\text{out})}, \mathbf{A}^{(\text{in})}\right]$

Figure 1: (a) Example graph. Color denotes edge types. (b) Unrolled one timestep. (c) Parameter tying and sparsity in recurrent matrix. Letters denote edge types with B' corresponding to the reverse edge of type B. B and B' denote distinct parameters.

$$\mathbf{h}_v^{(t)} = f^*(l_v, l_{\text{CO}(v)}, l_{\text{NBR}(v)}, \mathbf{h}_{\text{NBR}(v)}^{(t-1)}).$$

$$f^*(l_v, l_{\text{CO}(v)}, l_{\text{NBR}(v)}, \mathbf{h}_{\text{NBR}(v)}^{(t)}) = \sum_{v' \in \text{IN}(v)} f(l_v, l_{(v',v)}, l_{v'}, \mathbf{h}_{v'}^{(t-1)}) + \sum_{v' \in \text{OUT}(v)} f(l_v, l_{(v,v')}, l_{v'}, \mathbf{h}_{v'}^{(t-1)});$$

$$f(l_v, l_{(v',v)}, l_{v'}, \mathbf{h}_{v'}^{(t)}) = \mathbf{A}^{(l_v, l_{(v',v)}, l_{v'})} \mathbf{h}_{v'}^{(t-1)} + \mathbf{b}^{(l_v, l_{(v',v)}, l_{v'})}.$$

Uses Almeida-Pineda algorithm to train

https://arxiv.org/pdf/1511.05493.pdf

Gated Graph Neural Network

- Use Gated Recurrent Units
- Unroll the recurrence for a fixed number of steps
- Use backpropagation through time in order to compute gradients
  - Requires more memory than the Almeida-Pineda algorithm, but it removes the need to constrain parameters to ensure convergence

$$\mathbf{h}_v^{(1)} = [\boldsymbol{x}_v^\top, \mathbf{0}]^\top \tag{1}$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{v:}^\top \left[\mathbf{h}_1^{(t-1)\top} \dots \mathbf{h}_{|\mathcal{V}|}^{(t-1)\top}\right]^\top + \mathbf{b} \tag{2}$$

$$\mathbf{z}_v^t = \sigma\left(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}\right) \tag{3}$$

$$\mathbf{r}_v^t = \sigma\left(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}\right) \tag{4}$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \tag{5}$$

$$\mathbf{h}_v^{(t)} = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \widetilde{\mathbf{h}_v^{(t)}}. \tag{6}$$

https://arxiv.org/pdf/1511.05493.pdf

Gated Graph Sequence Neural Network

- GG-NN’s are used for single output tasks
- In order to output sequences, one can concat GG-NN’s

$$\begin{tikzcd}
 & & \boxed{\boldsymbol{o}^{(1)}} & & & \boxed{\boldsymbol{o}^{(2)}} & \\
\boxed{\boldsymbol{\mathcal{X}}^{(1)}} \arrow[r, "\text{Init}"] & \boxed{\boldsymbol{\mathcal{H}}^{(1,1)}} \arrow[ur, "\mathcal{F}_{\boldsymbol{o}}^{(1)}"] \arrow[r, "\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(1)}"] & \boxed{\boldsymbol{\mathcal{X}}^{(2)}} \arrow[r, "\text{Init}"] & \boxed{\boldsymbol{\mathcal{H}}^{(2,1)}} \arrow[ur, "\mathcal{F}_{\boldsymbol{o}}^{(2)}"] \arrow[r, "\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(2)}"] & \boxed{\boldsymbol{\mathcal{X}}^{(3)}} \arrow[r] & \cdots
\end{tikzcd}$$

Figure 2: Architecture of GGS-NN models.

https://arxiv.org/pdf/1511.05493.pdf

Setting up the problem

Initial Node representation: text of the node + type information

Program Graphs for VARNAMING:

- Given a program and an existing variable v, build a program graph and then replace the variable name in all corresponding variable tokens by a special <SLOT> token.
- A one-layer GRU predicts the target name as a sequence of subtokens (e.g., the name inputStreamBuffer is treated as the sequence [input, stream, buffer]).
- This graph2seq architecture is trained using a maximum likelihood objective

Setting up the problem

Program Graphs for VARMISUSE:

- Compute a **context representation c(t):**
  - Insert a new node v_<SLOT> at the position of t,
  - Connect it to the remaining graph using edges that do not depend on the chosen variable at the slot (i.e., everything but LastUse, LastWrite, LastLexicalUse, and GuardedBy edges)
- Compute the **usage representation u(t; v):**
  - Insert a “candidate” node v_{t;v} for all type-correct candidate variables
  - Connect it to the graph by inserting the edges that would be used if the variable were to be used at this slot.
  - Use the initial node representations, concatenated with an extra bit set to one for the candidate
- The context and usage representation are then the final node states of the nodes
- Finally, the correct variable usage at the location is computed as $\arg\max_v W[\boldsymbol{c}(t), \mathbf{u}(t, v)]$

W is a linear layer that uses the concatenation of c(t) and u(t; v). We train using a max-margin objective.

Dataset (VARMISUSE)

Open source C# projects on GitHub

Final dataset contains 29 projects from a diverse set of domains (compilers, databases, . . .) with about 2.9 million non-empty lines of code

At each slot there are on average 3.8 type-correct alternative variables

Selected two projects as development set

Selected three projects for UNSEENPROJTEST to allow testing on projects with completely unknown structure and types.

Split the remaining 23 projects into train/validation/test sets in the proportion
60-10-30, splitting along files: SEENPROJTEST.

Baseline

Local Model (LOC): two-layer bidirectional GRU run over the tokens before and after the target location

AVGBIRNN: an extension to LOC, where the usage representation u(t; v) is computed using another two-layer bidirectional RNN run over the tokens before/after each usage, and then averaging over the computed representations at the variable token

AVGLBL: uses a log-bilinear model for 4 left and 4 right context tokens of each variable usage, and then averages over these context representations

Results

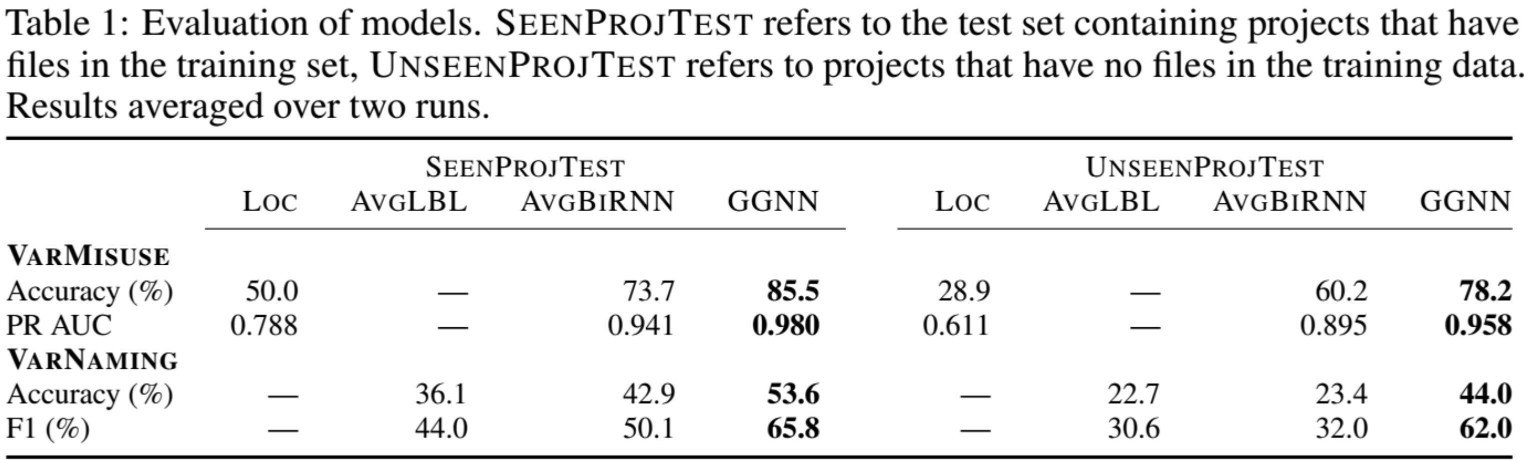


Table 1: Evaluation of models. SeenProjTest refers to the test set containing projects that have files in the training set, UnseenProjTest refers to projects that have no files in the training data. Results averaged over two runs.

	SeenProjTest				UnseenProjTest			
	Loc	AvgLBL	AvgBiRNN	GGNN	Loc	AvgLBL	AvgBiRNN	GGNN
VarMisuse								
Accuracy (%)	50.0	—	73.7	**85.5**	28.9	—	60.2	**78.2**
PR AUC	0.788	—	0.941	**0.980**	0.611	—	0.895	**0.958**
VarNaming								
Accuracy (%)	—	36.1	42.9	**53.6**	—	22.7	23.4	**44.0**
F1 (%)	—	44.0	50.1	**65.8**	—	30.6	32.0	**62.0**

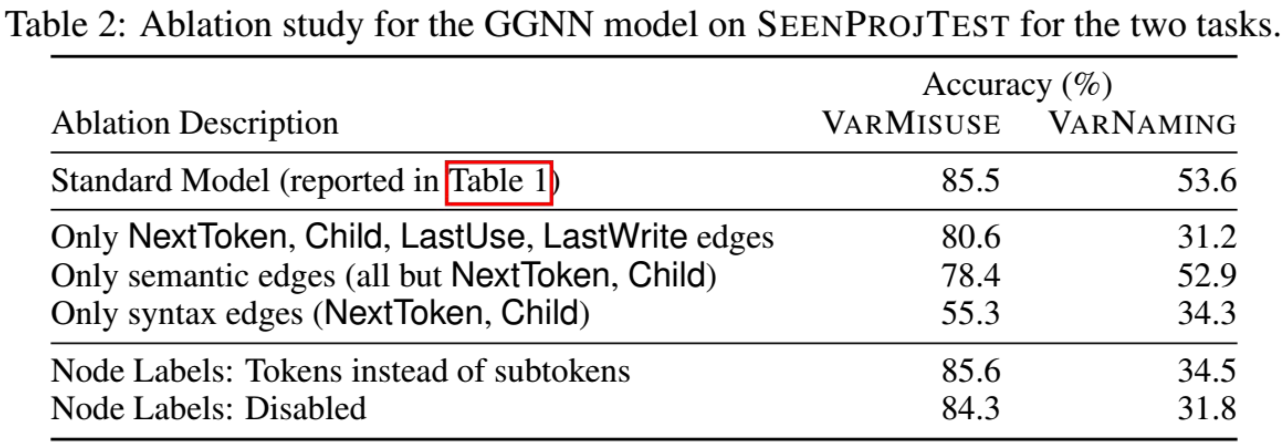
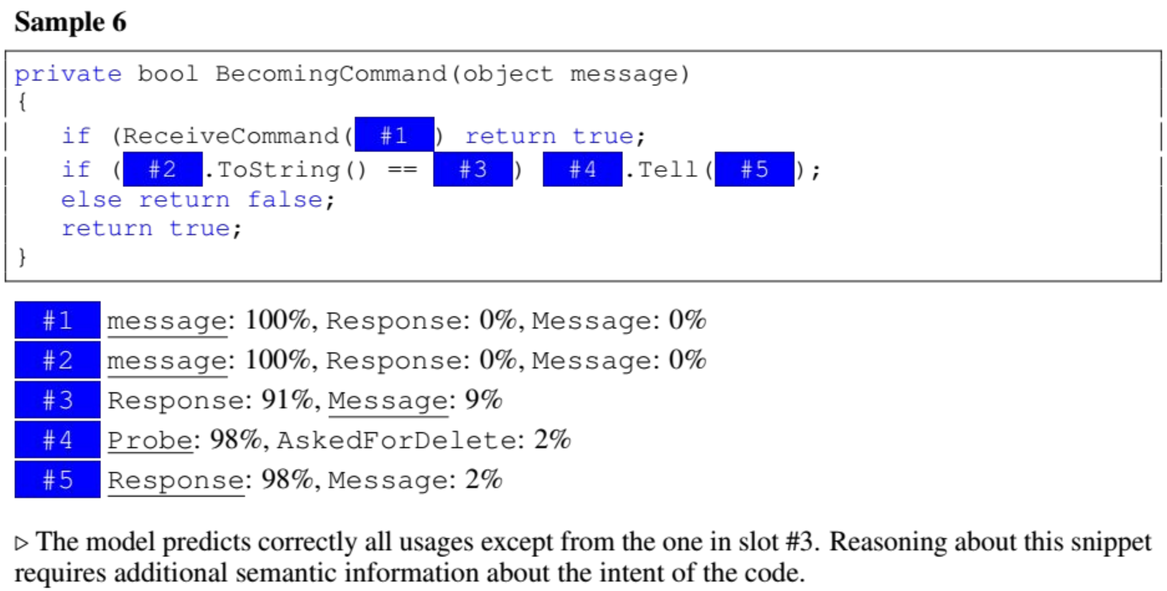


Table 2: Ablation study for the GGNN model on SeenProjTest for the two tasks.

	Accuracy (%)	
Ablation Description	VarMisuse	VarNaming
Standard Model (reported in Table 1)	85.5	53.6
Only NextToken, Child, LastUse, LastWrite edges	80.6	31.2
Only semantic edges (all but NextToken, Child)	78.4	52.9
Only syntax edges (NextToken, Child)	55.3	34.3
Node Labels: Tokens instead of subtokens	85.6	34.5
Node Labels: Disabled	84.3	31.8

Samples



Sample 6

```
private bool BecomingCommand(object message)
{
    if (ReceiveCommand( #1 ) return true;
    if ( #2 .ToString() == #3 ) #4 .Tell( #5 );
    else return false;
    return true;
}
```

#1	message: 100%, Response: 0%, Message: 0%
#2	message: 100%, Response: 0%, Message: 0%
#3	Response: 91%, Message: 9%
#4	Probe: 98%, AskedForDelete: 2%
#5	Response: 98%, Message: 2%

▷ The model predicts correctly all usages except from the one in slot #3. Reasoning about this snippet requires additional semantic information about the intent of the code.

code/data

Dataset: https://www.microsoft.com/en-us/download/details.aspx?id=56844

Github: https://github.com/Microsoft/gated-graph-neural-network-samples