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 syntax. 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.

1 INTRODUCTION

The advent of large repositories of source code as well as scalable machine learning methods naturally leads to the idea of “big code”, i.e., largely unsupervised methods that support software engineers by generalizing from existing source code (Allamanis et al., 2017). Currently, existing machine learning models of source code capture its shallow, textual structure, e.g. as a sequence of tokens (Hindle et al., 2012; Raychev et al., 2014; Allamanis et al., 2016), as parse trees (Maddison & Tarlow, 2014; Bielik et al., 2016), or as a flat dependency networks of variables (Raychev et al., 2015). Such models miss out on the opportunity to capitalize on the rich and well-defined semantics of source code. In this work, we take a step to alleviate this by including two additional signal sources in source code: data flow and type hierarchies. We do this by encoding programs as graphs, in which edges represent syntactic relationships (e.g. “token before/after”) as well as semantic relationships (“variable last used/written here”, “formal parameter for argument is called stream”, etc.). Our key insight is that exposing these semantics explicitly as structured input to a machine learning model lessens the requirements on amounts of training data, model capacity and training regime and allows us to solve tasks that are beyond the current state of the art.

We explore two tasks to illustrate the advantages of exposing more semantic structure of programs. First, we consider the VARNAMING task (Allamanis et al., 2014; Raychev et al., 2015), in which given some source code, the “correct” variable name is inferred as a sequence of subtokens. This requires some understanding of how a variable is used, i.e., requires reasoning about lines of code far apart in the source file. Secondly, we introduce the variable misuse prediction task (VARMISUSE), in which the network aims to infer which variable should be used in a program location. To illustrate the
```javascript
var clazz = classTypes["Root"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

var first = classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(first);
Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

Figure 1: A snippet of a detected bug in an open-source project (details anonymized). 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.

To achieve high accuracy on these tasks, we need to learn representations of program semantics. For both tasks, we need to learn the semantic role of a variable (e.g., “is it a counter?”; “is it a filename?”). Additionally, for VARMISUSE, learning variable usage semantics (e.g., “a filename is needed here”) is required. This “fill the blank element” task is related to methods for learning distributed representations of natural language words, such as Word2Vec (Mikolov et al., 2013) and GLoVe (Pennington et al., 2014). However, we can learn from a much richer structure such as data flow information. This work is a step towards learning program representations, and we expect them to be valuable in a wide range of other tasks, such as code completion (“this is the variable you are looking for”) and more advanced bug finding (“you should lock before using this object”).

To summarize, our contributions are: (i) We define the VARMISUSE task as a challenge for machine learning modeling of source code, that requires to learn (some) semantics of programs (cf. section 3). (ii) We present deep learning models for solving the VARNAMING and VARMISUSE tasks by modeling the code’s graph structure and learning program representations over those graphs (cf. section 4). (iii) We evaluate our models on a large dataset of 2.9 million lines of real-world source code, showing that our best model achieves 30.7% accuracy on the VARNAMING task and 82.1% accuracy on the VARMISUSE task, beating simpler baselines (cf. section 5). (iv) We document practical relevance of VARMISUSE by summarizing some bugs that we found in mature open-source software projects (cf. subsection 5.3). We provide all data at [anonymized].

2 RELATED WORK

Our work builds upon the recent field of using machine learning for source code artifacts (Allamanis et al., 2017). For example, Hindle et al. (2012); Bhoopchand et al. (2016) model the code as a sequence of tokens, while Maddison & Tarlow (2014); Raychev et al. (2016) model the syntax tree structure of code. All works on language models of code find that predicting variable and method identifiers is one of biggest challenges in the task.

Closest to our work is the work of Allamanis et al. (2015) who learn distributed representations of variables using all their usages to predict their names. However, they do not use data flow information and we are not aware of any model that does so. The work of Raychev et al. (2015) is also relevant, as it uses a CRF over the dependency network between variables — without considering data flow — to predict their types and names. However, all variable usages are deterministically known beforehand (as the code is complete and remains unmodified), as in Allamanis et al. (2014; 2015).

Our work is remotely related to work on program synthesis using sketches (Solar-Lezama, 2008) and automated code transplantation (Barr et al., 2015). However, these approaches require a set of specifications (e.g. input-output examples, test suites) to complete the gaps, rather than statistics learned from big code. These approaches can be thought as complementary to ours, since we learn to statistically complete the gaps without any need for specifications, by learning common dataflow structure from code.
Neural networks on graphs (Gori et al., 2005; Li et al., 2015; Defferrard et al., 2016; Kipf & Welling, 2016; Gilmer et al., 2017) adapt a variety of deep learning methods to graph-structured input. They have been used in a series of applications, such as link prediction and classification (Grover & Leskovec, 2016) and semantic role labeling in NLP (Marcheggiani & Titov, 2017). Somewhat related to source code is the work of Wang et al. (2017) who learn graph-based representations of mathematical formulas for premise selection in theorem proving.

3 THE VAR Misuse TASK

Detecting variable misuses in code is a task that requires understanding and reasoning about program semantics. To successfully tackle the task one needs to infer the role and function of the program elements and understand how they relate. For example, given a program such as Fig. 1 the task is to automatically detect that the marked use of clazz is a mistake and that first should be used instead. While this task resembles standard code completion, it differs significantly in its scope and purpose, by considering only variable identifiers and a mostly complete program.

Task Description We view a source code file as a sequence of tokens $t_0 \ldots t_N = T$, in which some tokens $t_{\lambda_0}, t_{\lambda_1}, \ldots$ are variables. Furthermore, let $\forall_t \subset V$ refer to the set of all type-correct variables in scope at the location of $t$, i.e., those variables that can be used at $t$ without raising a compiler error. We call the location $t$ where we want to predict the correct variable usage a slot. We define a separate task for each slot $t_\lambda$. Given $t_0 \ldots t_{\lambda-1}$ and $t_{\lambda+1}, \ldots, t_N$, correctly select $t_\lambda$ from $\forall_{t_\lambda}$. For training and evaluation purposes, a correct solution is one that simply matches the ground truth, but note that in practice, several possible assignments could be considered correct (i.e., when several variables refer to the same value in memory).

4 MODEL: PROGRAMS AS GRAPHS

In this section, we discuss how to transform program source code into program graphs and learn representations over them. These program graphs not only encode the program text but also the semantic information that can be obtained using standard compiler tools.

Gated Graph Neural Networks Our work builds on Gated Graph Neural Networks (Li et al., 2015) (GGNN) and we summarize them here. A graph $G = (V,E,X)$ is composed of a set of nodes $V$, node features $X$, and a list of directed edge sets $E = (E_1, \ldots, E_K)$ where $K$ is the number of edge types. We annotate each $v$ with a real-valued vector $x^{(v)} \in \mathbb{R}^D$ representing the features of the node (e.g., the embedding of a string label of that node).

We associate every node $v$ with a state vector $h^{(v)}$, initialized from the node label $x^{(v)}$. The sizes of the state vector and feature vector are typically the same, but we can use larger state vectors through padding of node features. To propagate information throughout the graph, “messages” of type $k$ are sent from each $v$ to its neighbors, where each message is computed from its current state vector as $m_k^{(v)} = f_k(h^{(v)})$. Here, $f_k$ can be an arbitrary function; we choose a linear layer in our case. By computing messages for all graph edges at the same time, all states can be updated at the same time. In particular, a new state for a node $v$ is computed by aggregating all incoming messages as $\tilde{m}^{(v)} = g(\{m_k^{(u)} \mid \text{there is an edge of type } k \text{ from } u \text{ to } v\})$. $g$ is an aggregation function, which we implement as elementwise summation. Given the aggregated message $\tilde{m}^{(v)}$ and the current state vector $h^{(v)}$ of node $v$, the state of the next time step $h_{\lambda}^{(v)}$ is computed as $h_{\lambda}^{(v)} = \text{GRU}(\tilde{m}^{(v)}, h^{(v)})$, where GRU is the recurrent cell function of gated recurrent unit (GRU) (Cho et al., 2014). The dynamics defined by the above equations are repeated for a fixed number of time steps. Then, we use the state vectors from the last time step as the node representations.

Program Graphs We represent program source code as graphs and use different edges to model syntactic and semantic relationships between different tokens. The backbone of a program graph is the program’s abstract syntax tree (AST), consisting of syntax nodes (corresponding to non-terminals in the programming language’s grammar) and syntax tokens (corresponding to terminals). We label syntax nodes with the name of the nonterminal from the program’s grammar, whereas syntax tokens
are labeled with the string that they represent. We use Child edges to connect nodes according to the AST. As this does not induce an order on children of a syntax node, we additionally add NextToken edges connecting each syntax token to its successor. An example of this is shown in Fig. 2a.

To capture the flow of control and data through a program, we add additional edges connecting different uses and updates of syntax tokens corresponding to variables. For such a token \( v \), let \( D^R(v) \) be the set of syntax tokens at which the variable could have been used last. This set may contain several nodes (for example, when using a variable after a conditional in which it was used in both branches), and even syntax tokens that follow in the program code (in the case of loops). Similarly, let \( D^W(v) \) be the set of syntax tokens at which the variable was last written to. Using these, we add LastRead (resp. LastWrite) edges connecting \( v \) to all elements of \( D^R(v) \) (resp. \( D^W(v) \)). Additionally, whenever we observe an assignment \( v = expr \), we connect \( v \) to all variable tokens occurring in \( expr \) using ComputedFrom edges. An example of such semantic edges is shown in Fig. 2b.

We extend the graph to chain all uses of the same variable using LastLexicalUse edges (independent of data flow, i.e., in \( \text{if ( ... )} \{ \ldots \ v \ldots \} \) \text{else} \( \{ \ldots \ v \ldots \} \)), we link the two occurrences of \( v \). We also connect return tokens to the method declaration using ReturnsTo edges (this creates a “shortcut” to its name and type). Finally, inspired by Rice et al. (2017), we connect arguments in method calls to the formal parameters that they are matched to with FormalArgName edges, i.e., if we observe a call \( \text{Foo(bar)} \) and a method declaration \( \text{Foo(InputStreamName)} \), we connect the bar token to the stream token.

Finally, for all types of edges we introduce their respective backwards edges (transposing the adjacency matrix), doubling the number of edges and edge types. Backwards edges help with propagating information faster across the GGNN and make the model more expressive.

**Leveraging Variable Type Information** We assume a statically typed language and that the source code can be compiled, and thus each variable has a (known) type \( \tau(v) \). To use it, we define a learnable embedding function \( r(\tau) \) for known types and additionally define an “UNKTYPE” for all unknown/unrepresented types. We also leverage the rich type hierarchy that is available in many object-oriented languages. For this, we map a variable’s type \( \tau(v) \) to the set of its supertypes, i.e. 
\[ \tau^\ast(v) = \{ \tau : \tau(v) \text{ implements type } \tau \} \cup \{ \tau(v) \} \]. We then compute the type representation \( r^\ast(v) \) of a variable \( v \) as the element-wise maximum of \( \{ r(\tau) : \tau \in \tau^\ast(v) \} \). We chose the maximum here, as it is a natural pooling operation for representing partial ordering relations (such as type lattices). Using all types in \( \tau^\ast(v) \) allows us to generalize to unseen types that implement common supertypes or interfaces. For example, \( \text{List<K>} \) has multiple concrete types (e.g. \( \text{List<int>} \), \( \text{List<string>} \)). Nevertheless, these types implement a common interface (\( \text{IList} \)) and share common characteristics. During training, we randomly select a non-empty subset of \( \tau^\ast(v) \) which ensures training of all known types in the lattice. This acts both like a dropout mechanism and allows us to learn a good representation for all types in the type lattice.

**Initial Node Representation** To compute the initial node state, we combine information from the textual representation of the token and its type. Concretely, we split the name of a node into

---

**Figure 2:** Examples of graph edges used in program representation.
subtokens (e.g. classTypes will be split into two subtokens class and types) on camelCase and pascal_case. We then average the embeddings of all subtokens to retrieve an embedding for the node name. Finally, we concatenate the learned type representation \( \mathbf{r}^*(v) \), computed as discussed earlier, with the node name representation. This representation is then padded with zeros to match the target hidden layer size of the GGNN.

**Programs Graphs for **\textsc{VarNaming}\** Given a program and an existing variable** \( v \), we build a program graph as discussed above and then replace the variable name in all corresponding variable tokens by a special <SLOT> token. To predict a name, we use the initial node labels computed as the concatenation of learnable token embeddings and type embeddings as discussed above, run GGNN propagation for 8 time step\(^1\) and then compute a variable usage representation by averaging the representations for all <SLOT> tokens. This representation is then used as the initial state of a one-layer GRU, which predicts the target name as a sequence of subtokens (e.g., the name inputStreamBuffer is treated as the sequence [input, stream, buffer]). We train this graph2seq architecture using a maximum likelihood objective. In [section 5](#section5) we report the accuracy for predicting the exact name and the F1 score for predicting its subtokens.

**Program Graphs for **\textsc{VarMisuse}\** To model **\textsc{VarMisuse} with program graphs we need to modify the graph. First, to compute a context representation \( c(t) \) for a slot \( t \) where we want to predict the used variable, we insert a new node \( v_{\text{SLOT}} \) at the position of \( t \), corresponding to a “hole” at this point, and connect it to the remaining graph using all applicable edges that do not depend on the chosen variable at the slot (i.e., everything but LastUse, LastWrite, and LastLexicalUse edges). Then, to compute the usage representation \( u(t, v) \) of each candidate variable \( v \) at the target slot, we insert a “candidate” node \( v_{t,v} \) for all \( v \) in \( V_t \), and connect it to the graph by inserting the LastUse, LastWrite and LastLexicalUse edges that would be used if the variable were to be used at this slot. Each of these candidate nodes represents the speculative placement of the variable within the scope.

Using the initial node representations, concatenated with an extra bit that is set to one for the candidate nodes \( v_{t,v} \), we run GGNN propagation for 8 time steps\(^1\) The context and usage representation are then the final node states of the nodes, i.e., \( c(t) = h^{v_{\text{SLOT}}} \) and \( u(t, v) = h^{v_{t,v}} \). Finally, the correct variable usage at the location is computed as \( \text{arg max}_v c(t)^T u(t, v) \). We train using maximum likelihood.

### 4.1 Implementation

Using GGNNs for sets of large, diverse graphs requires some engineering effort, as efficient batching is hard in the presence of diverse shapes. An important observation is that large graphs are normally very sparse, and thus a representation of edges as an adjacency list would usually be advantageous to reduce memory consumption. In our case, this can be easily implemented using a sparse tensor representation, allowing large batch sizes that exploit the parallelism of modern GPUs efficiently. A second key insight is to represent a batch of graphs as one large graph with many disconnected components. This just requires appropriate pre-processing to make node identities unique. As this makes batch construction somewhat CPU-intensive, we found it useful to prepare minibatches on a separate thread. Our TensorFlow (Abadi et al., 2016) implementation scales to 55 graphs per second during training and 219 graphs per second during test-time using a single NVidia GeForce GTX Titan X with graphs having on average 2,228 (median 936) nodes and 8,350 (median 3,274) edges and 8 GGNN unrolling iterations, all 16 edge types (forward and backward edges for 8 original edge types) and the size of the hidden layer set to 64. The number of types of edges in the GGNN contribute proportionally to the running time. For example, a GGNN run for our ablation study using only the two most common edge types (NextToken, Child) achieves 105 graphs/second during training and 419 graphs/second at test time with the same hyperparameters. We intend to release our (generic) sparse GGNN implementation soon.

### 5 Evaluation

**Dataset** We collected a dataset for the **\textsc{VarMisuse}** task from open source C\(^6\) projects on GitHub. To select projects, we picked the top-starred (non-fork) projects on GitHub. We then filtered out

---

\(^1\)We found fewer steps to be insufficient for good results and more propagation steps to not help substantially.
which essentially replaces the log-bilinear context model by a bidirectional RNN.

This baseline allows us to evaluate how important the initial node labels in the GGNN are. As L

Table 1 shows the evaluation results of the models for both tasks. As L

Table 1: Evaluation of models. UNSEENPROJTEST refers to projects that have no files in the training data, SEENPROJTEST refers to the test set containing projects that have files in the training set.

<table>
<thead>
<tr>
<th>Variable</th>
<th>SEENPROJTEST</th>
<th>UNSEENPROJTEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>AVGLBL</td>
<td>AVGBI RNN</td>
</tr>
<tr>
<td>VAR MISUSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>15.8</td>
<td>73.5</td>
</tr>
<tr>
<td>PR AUC</td>
<td>0.363</td>
<td>0.931</td>
</tr>
<tr>
<td>FR (%)</td>
<td>22.0</td>
<td>25.5</td>
</tr>
<tr>
<td>FI (%)</td>
<td>36.1</td>
<td>42.9</td>
</tr>
</tbody>
</table>

For VARNAMING, we collect data from all projects by selecting all variable usage locations, filtering out variable declarations, where at least one other type-compatible replacement variable is in scope. The task is then to infer the correct variable that originally existed in that location. Thus, by construction there is at least one type-correct replacement variable, i.e., picking it would not not raise an error during type checking. In our test datasets, at each slot there are on average 3.8 type-correct alternative variables (median 3, σ = 2.6).

From our dataset, we selected two projects as our development set. From the rest of the projects, we selected three projects for UNSEENPROJTEST to allow testing on projects with completely unknown structure and types. We split the remaining 23 projects into train/validation/test sets in the proportion 60-10-30, splitting along files (i.e., all examples from one source file are in the same set). We call the test set obtained like this SEENPROJTEST.

**Baselines** For VAR MISUSE, we consider two bidirectional RNN-based baselines. The local model (LOC) is a simple two-layer bidirectional GRU run over the tokens before and after the target location. For this baseline, \( c(t) \) is set to the slot representation computed by the RNN, and the usage context of each variable \( u(t, v) \) is the embedding of the name and type of the variable, computed in the same way as the initial node labels in the GGNN. This baseline allows us to evaluate how important the usage context information is for this task. The flat dataflow model (AVGBI RNN) is 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 \( v \). The local context, \( c(t) \), is identical to LOC. AVGBI RNN is a significantly stronger baseline that already takes some structural information into account, as the averaging over all variables usages helps with long-range dependencies. As in GGNN, both models pick the variable that maximizes \( c(t)^T u(t, v) \).

For VARNAMING, we replace LOC by AVGLBL, which uses a log-bilinear model for 4 left and 4 right context tokens of each variable usage, and then averages over these context representations (this corresponds to the model in [Allamanis et al. (2015)]) We also test AVGBI RNN on VARNAMING, which essentially replaces the log-bilinear context model by a bidirectional RNN.

### 5.1 Quantitative Evaluation

Table 1 shows the evaluation results of the models for both tasks. As LOC captures very little information, it performs relatively badly. AVGLBL and AVGBI RNN, which capture information from many variable usage sites, but do not explicitly encode the rich structure of the problem, still lag behind the GGNN by a wide margin. The performance difference is larger for VAR MISUSE, since the structure and the semantics of code are far more important within this setting.

**Generalization to new projects** Generalizing across a diverse set of source code projects with different domains is an important challenge in machine learning. We repeat the evaluation using the

---

<http://roslyn.io>
Table 2: Ablation study for the GGNN model on SEENPROJTEST for the two tasks.

<table>
<thead>
<tr>
<th>Ablation Description</th>
<th>VARNISUSE Accuracy (%)</th>
<th>VARNAMING Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Model (reported in Table 1)</td>
<td>82.1</td>
<td>30.7</td>
</tr>
<tr>
<td>Only NextToken, Child, LastUse, LastWrite edges</td>
<td>79.0</td>
<td>15.4</td>
</tr>
<tr>
<td>Only semantic edges (all but NextToken, Child)</td>
<td>74.3</td>
<td>29.7</td>
</tr>
<tr>
<td>Only syntax edges (NextToken, Child)</td>
<td>49.6</td>
<td>20.5</td>
</tr>
<tr>
<td>Node Labels: Tokens instead of subtokens</td>
<td>82.1</td>
<td>16.8</td>
</tr>
<tr>
<td>Node Labels: Disabled</td>
<td>80.0</td>
<td>14.7</td>
</tr>
</tbody>
</table>

UNSEENPROJTEST set stemming from projects that have no files in the training set. The right side of Table 1 shows that our models still achieve good performance, although it is slightly lower compared to SEENPROJTEST. This is expected since the type lattice is mostly unknown in UNSEENPROJTEST.

We believe that some of the most important issues when transferring to new domains is the fact that projects have significantly different type hierarchies and that the vocabulary used (e.g. within tokens) is very different from the training projects.

### Ablation Study

To study the effect of some of the design choices for our models, we have run some additional experiments and show their results in Table 2. First, we varied the edges used in the program graph. We find that restricting the model to syntactic information has a large impact on performance on both tasks, whereas restricting it to semantic edges seems to mostly impact performance on VARNISUSE. Similarly, the ComputedFrom, FormalArgName and ReturnsTo edges give a small boost on VARNISUSE, but greatly improve performance on VARNAMING. As evidenced by the experiments with the node label representation, syntax node and token names seem to matter little for VARNISUSE, but naturally have a great impact on VARNAMING.

#### 5.2 Qualitative Evaluation

Figure 3 illustrates a prediction that GGNN makes on a sample test snippet. The snippet recursively searches for the global directives file by gradually descending into the root folder. Reasoning about the correct variable usages is hard, even for humans, but the GGNN correctly predicts the variable usages at all locations except two (slot 1 and 8). As a software engineer is writing the code it is imaginable that she may make a mistake misusing one variable in the place of another. Since all variables are string variables, no type errors will be raised. As the probabilities in Fig. 3 suggest most potential variable misuses can be flagged by the model yielding valuable warnings to software engineers. Additional samples with comments can be found in Appendix A.

Furthermore, Appendix B shows samples of pairs of code snippets that share similar representations as computed by the cosine similarity of the usage representation $u(t, v)$ of GGNN. The reader can notice that the network learns to group variable usages that share semantic similarities together. For example, checking for null before the use of a variable yields similar distributed representations across code segments (Sample 1 in Appendix B).

#### 5.3 Discovered Variable Misuse Bugs

We looked at the top 500 locations of a project’s test files where our model was most confident about a variable other than the ground-truth. Within this set of locations we noticed some bugs. Specific information about the project is excluded to avoid breaking the double-blind review. The bugs we found appear in the released code of a well-tested popular GitHub project with over 1.5k stars, attesting to the practical applicability of our model. The bug shown in Figure 1, possibly caused by copy-pasting, is caught since the pattern of usage is highly irregular. Such a bug cannot be easily caught by traditional methods. A compiler will not warn about unused variables (since first is used) and virtually nobody would write a test testing another test. The bugs shown in Figure 4 and Fig. 2 are two other examples of issues that were caught by the VARNISUSE model. Figure 4 shows an issue that, although not critical, can help reduce memory consumption. Figure 5 shows another issue arising from a non-informative error message. Finding these issues in the released code suggests that our model can be quite useful during the software development process and can be complementary to classic program analysis tools.
bool TryFindGlobalDirectivesFile(string baseDirectory, string fullPath, out string path){
    baseDirectory = baseDirectory.TrimEnd(Path.DirectorySeparatorChar);
    var directivesDirectory = Path.GetDirectoryName(fullPath).TrimEnd(Path.DirectorySeparatorChar);
    while(directivesDirectory != null && directivesDirectory.Length >= baseDirectory.Length){
        path = Path.Combine(directivesDirectory, GlobalDirectivesFileName);
        if (File.Exists(path)) return true;
        directivesDirectory = Path.GetDirectoryName(directivesDirectory).TrimEnd(Path.DirectorySeparatorChar);
    }
    path = null;
    return false;
}

Figure 3: VARMISUSE predictions on slots within a snippet of the SEENPROJTEST set for the ServiceStack project. Additional visualizations are available in Appendix A. The underlined tokens are the correct tokens. The model has to select among a number of string variables at each slot, where all of them represent some kind of path. The GGNN accurately predicts the correct variable usage in 11 out of the 13 slots reasoning about the complex ways the variables interact among them.

public ArraySegment<byte> ReadBytes(int length){
    int size = Math.Min(length, _len - _pos);
    var buffer = EnsureTempBuffer(length);
    var used = Read(buffer, 0, size);
    return new ArraySegment<byte>(buffer, 0, used);
}

Figure 4: A bug found (yellow) highlighted slot in an open-source project. The code unnecessarily ensures that the buffer is of size length rather than size (which our model predicts as the correct usage for the slot).

6 DISCUSSION & CONCLUSIONS

Although source code is well understood and studied within other disciplines such as programming language research, it is a relatively new domain for deep learning. It presents novel opportunities

protected void ValidateRestorePreconditions(string backupFilename) {
    if (IsValidBackup(backupFilename) == false) {
        output("Error:"+ backupLocation +" doesn’t look like a valid backup");
        output("Error: Restore Canceled");
        throw new InvalidOperationException(
            backupLocation + " doesn’t look like a valid backup");
    }
}

Figure 5: A bug found (yellow) highlighted slot in an open-source project (code modified to fit in page). Although backupFilename is found to be invalid by IsValidBackup, the user is notified that backupLocation is invalid instead.
compared to textual or perceptual data, as its (local) semantics are well-defined and rich additional information can be extracted using well-known, efficient program analyses. On the other hand, integrating this wealth of structured information poses an interesting challenge. Our VARMISUSE task exposes these opportunities, going beyond simpler tasks such as code completion. We consider it as a first proxy for the core challenge of learning the meaning of source code, as it requires to probabilistically refine standard information included in type systems.

REFERENCES


Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from Big Code. In *Principles of Programming Languages (POPL)*, 2015.


A VarMisuse Prediction Samples

Below we list a set of samples from our SEENPROJTEST projects with comments about the model performance. Code comments and formatting may have been altered for typesetting reasons. The ground truth choice is underlined.

Sample 1

```
for (var port = #1 ; #2 < #3 ; #4 ++) {
  if (!activePorts.Contains( #5 ))
    return #6 ;
}
```

#1 startingFrom: 97%, endingAt: 3%
#2 port: 100%, startingFrom: 0%, endingAt: 0%
#3 endingAt: 100%, startingFrom: 0%, port: 0%
#4 port: 100%, startingFrom: 0%, endingAt: 0%
#5 port: 100%, startingFrom: 0%, endingAt: 0%
#6 port: 100%, startingFrom: 0%, endingAt: 0%

▷ The model correctly predicts all variables in the loop.

Sample 2

```
var path = CreateFileName( #1 );
bitmap.Save( #2 , ImageFormat.Png);
return #3 ;
```

#1 name: 86%, DIR_PATH: 14%
#2 path: 90%, name: 8%, DIR_PATH: 2%
#3 path: 76%, name: 16%, DIR_PATH: 8%

▷ String variables are not confused their semantic role is inferred correctly.

Sample 3

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute]
public void MergeFrom(pb::CodedInputStream input) {
  uint tag;
  while ((tag = input.ReadTag()) != 0) {
    switch(tag) {
      default:
        input.SkipLastField();
        break;
      case 10: {
        #1 .AddEntriesFrom(input, _repeated_payload_codec);
        break;
      }
    }
  }
}
```

#1 Payload: 66%, payload_: 44%

▷ The model is commonly confused by aliases, i.e. variables that point to the same location in memory. In this sample, either choice would have yielded identical behavior.
Sample 4

```csharp
public override bool IsDisposed
{
    get
    {
        lock (_gate) // #1
        {
            return _isDisposed; // #2
        }
    }
}
```

#1 _gate: 99%, _observers: 1%
#2 _isDisposed: 90%, _isStopped: 8%, HasObservers: 2%

➢ The ReturnsTo edge can help predict variables that otherwise would have been impossible.
/// <summary>
/// Notifies all subscribed observers about the exception.
/// </summary>
/// <param name="error">The exception to send to all observers.</param>
public override void OnError(Exception error) {
    if (#1 == null)
        throw new ArgumentNullException(nameof(#2));

    var os = default(IObserver<T>[]);
    lock (#3)
    {
        CheckDisposed();
        if (!#4)
        {
            os = _observers.Data;
            _observers = ImmutableList<IObserver<T>>.Empty;
            #5 = true;
            #6 = #7;
        }
    }

    if (os != null)
    {
        foreach (var o in os)
        {
            o.OnError(#8);
        }
    }
}

◆ The model predicts the correct variables from all slots apart from the last. Reasoning about the last one, requires interprocedural understanding of the code across the class file.
Sample 6

```csharp
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.

Sample 7

```csharp
var response = ResultsFilter(typeof(TResponse), #1 , #2 , request);
```

- #1 httpMethod: 99%, absoluteUrl: 1%, UserName: 0%, UserAgent: 0%
- #2 absoluteUrl: 99%, httpMethod: 1%, UserName: 0%, UserAgent: 0%

▷ The model knows about selecting the correct string parameters because it matches them to the formal parameter names.

Sample 8

```csharp
if ( #1 >= #2 )
    throw new InvalidOperationException(Strings_Core.FAILED_CLOCK_MONITORING);
```

- #1 n: 100%, MAXERROR: 0%, SYNC_MAXRETRIES: 0%
- #2 MAXERROR: 62%, SYNC_MAXRETRIES: 22%, n: 16%

▷ It is hard for the model to reason about conditionals, especially with rare constants as in slot #2.

B  Nearest Neighbor of GGNN Usage Representations

Here we show pairs of nearest neighbors based on the cosine similarity of the learned representations \( u(t,v) \). Each slot \( t \) is marked in dark blue and all usages of \( v \) are marked in yellow (i.e. \( \text{variableName} \)). This is a set of hand-picked examples showing good and bad examples. A brief description follows after each pair.
Sample 1

```java
public void MoveShapeUp(BaseShape shape) {
    if (shape != null) {
        for(int i=0; i < Shapes.Count -1; i++){
            if (Shapes[i] == shape){
                Shapes.Move(i, ++i);
                return;
            }
        }
    }
}
```

```java
lock(lockObject) {
    if (unobservableExceptionHanler != null)
        return false;
    unobservableExceptionHanler = handler;
}
```

▷ Slots that are checked for null-ness have similar representations.

Sample 2

```java
public IActorRef ResolveActorRef(ActorPath actorPath) {
    if (HasAddress(actorPath.Address))
        return _local.ResolveActorRef(RootGuardian, actorPath.ElementsWithUid);
    ...
```

```java
ActorPath actorPath;
if (TryParsePath(path, out actorPath)) {
    if (HasAddress(actorPath.Address)){
        if (actorPath.ToStringWithoutAddress().Equals("/"))
            return RootGuarding;
    }
    ...
}
```

▷ Slots that follow similar API protocols have similar representations. Note that the function HasAddress is a local function, seen only in the testset.
Sample 3

foreach(var filter in configuration.Filters){
    GlobalJobFilter.Filters.Add(filter);
}

public void Count_ReturnsNumberOfElements(){
    _collection.Add(_filterInstance);
    Assert.Equal(1, _collection.Count);
}

Adding elements to a collection-like object yields similar representations.

C  DATASET

The collected dataset and its characteristics are listed in Table 3. The full dataset as a set of projects and its parsed JSON will become available online.

Table 3: Projects in our dataset. Ordered alphabetically. kLOC measures the number of non-empty lines of C# code. Projects marked with Dev were used as a development set. Projects marked with † were in the test-only dataset. The rest of the projects were split into train-validation-test. The dataset contains in total about 2.9MLOC.

<table>
<thead>
<tr>
<th>Name</th>
<th>Git SHA</th>
<th>kLOCs</th>
<th>Slots</th>
<th>Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Akka.NET</td>
<td>719335a1</td>
<td>240</td>
<td>51.3k</td>
<td>51.2k</td>
<td>Actor-based Concurrent &amp; Distributed Framework</td>
</tr>
<tr>
<td>AutoMapper</td>
<td>2ca7c2b5</td>
<td>46</td>
<td>3.7k</td>
<td>10.7k</td>
<td>Object-to-Object Mapping Library</td>
</tr>
<tr>
<td>BenchmarkDotNet</td>
<td>1670ca34</td>
<td>28</td>
<td>5.1k</td>
<td>6.1k</td>
<td>Benchmarking Library</td>
</tr>
<tr>
<td>BotBuilder</td>
<td>190117c3</td>
<td>44</td>
<td>6.4k</td>
<td>8.7k</td>
<td>SDK for Building Bots</td>
</tr>
<tr>
<td>choco</td>
<td>93895688</td>
<td>36</td>
<td>3.8k</td>
<td>5.2k</td>
<td>Windows Package Manager</td>
</tr>
<tr>
<td>commandline†</td>
<td>09677b16</td>
<td>11</td>
<td>1.1k</td>
<td>2.3k</td>
<td>Command Line Parser</td>
</tr>
<tr>
<td>CommonMark.NETDev</td>
<td>f3d54530</td>
<td>14</td>
<td>2.6k</td>
<td>1.4k</td>
<td>Markdown Parser</td>
</tr>
<tr>
<td>Dapper</td>
<td>931c700d</td>
<td>18</td>
<td>3.3k</td>
<td>4.7k</td>
<td>Object Mapper Library</td>
</tr>
<tr>
<td>EntityFramework</td>
<td>fa0b7ec8</td>
<td>263</td>
<td>33.4k</td>
<td>39.3k</td>
<td>Object-Relational Mapper</td>
</tr>
<tr>
<td>Hangfire</td>
<td>ff4912f</td>
<td>33</td>
<td>3.6k</td>
<td>6.1k</td>
<td>Background Job Processing Library</td>
</tr>
<tr>
<td>Humanizer†</td>
<td>cc11a77e</td>
<td>27</td>
<td>2.4k</td>
<td>4.4k</td>
<td>String Manipulation and Formatting</td>
</tr>
<tr>
<td>Lean†</td>
<td>f574bfd7</td>
<td>190</td>
<td>26.4k</td>
<td>28.3k</td>
<td>Algorithmic Trading Engine</td>
</tr>
<tr>
<td>Nancy</td>
<td>72e1f614</td>
<td>70</td>
<td>7.5k</td>
<td>15.7</td>
<td>HTTP Service Framework</td>
</tr>
<tr>
<td>Newtonsoft.Json</td>
<td>6057d9b8</td>
<td>123</td>
<td>14.9k</td>
<td>16.1k</td>
<td>JSON Library</td>
</tr>
<tr>
<td>Ninject</td>
<td>7006297f</td>
<td>13</td>
<td>0.7k</td>
<td>2.1k</td>
<td>Code Injection Library</td>
</tr>
<tr>
<td>NLog</td>
<td>643e326a</td>
<td>75</td>
<td>8.3k</td>
<td>11.0k</td>
<td>Logging Library</td>
</tr>
<tr>
<td>OpsServer</td>
<td>51b032e7</td>
<td>24</td>
<td>3.7k</td>
<td>4.5k</td>
<td>Monitoring System</td>
</tr>
<tr>
<td>OptiKey</td>
<td>7d35c718</td>
<td>34</td>
<td>6.1k</td>
<td>3.9k</td>
<td>Assistive On-Screen Keyboard</td>
</tr>
<tr>
<td>Orleans</td>
<td>e0d6a150</td>
<td>300</td>
<td>30.7k</td>
<td>35.6k</td>
<td>Distributed Virtual Actor Model</td>
</tr>
<tr>
<td>Polly</td>
<td>0afdbc32</td>
<td>32</td>
<td>3.8k</td>
<td>9.1k</td>
<td>Resilience &amp; Transient Fault Handling Library</td>
</tr>
<tr>
<td>quartznet</td>
<td>b33e6f86</td>
<td>49</td>
<td>9.6k</td>
<td>9.8k</td>
<td>Scheduler</td>
</tr>
<tr>
<td>ravenDBDev</td>
<td>55230922</td>
<td>647</td>
<td>78.0k</td>
<td>82.7k</td>
<td>Document Database</td>
</tr>
<tr>
<td>RestSharp</td>
<td>70de357b</td>
<td>20</td>
<td>4.0k</td>
<td>4.5k</td>
<td>REST and HTTP API Client Library</td>
</tr>
<tr>
<td>Rx.NET</td>
<td>2d146fe5</td>
<td>180</td>
<td>14.0k</td>
<td>21.9k</td>
<td>Reactive Language Extensions</td>
</tr>
<tr>
<td>scripts</td>
<td>f3cc8bcb</td>
<td>18</td>
<td>2.7k</td>
<td>4.3k</td>
<td>C# Text Editor</td>
</tr>
<tr>
<td>ServiceStack</td>
<td>6d59da75</td>
<td>231</td>
<td>38.0k</td>
<td>46.2k</td>
<td>Web Framework</td>
</tr>
<tr>
<td>ShareX</td>
<td>718dd711</td>
<td>125</td>
<td>22.3k</td>
<td>18.1k</td>
<td>Sharing Application</td>
</tr>
<tr>
<td>SignalR</td>
<td>fa88089e</td>
<td>53</td>
<td>6.5k</td>
<td>10.5k</td>
<td>Push Notification Framework</td>
</tr>
<tr>
<td>Wox</td>
<td>cda6f272</td>
<td>13</td>
<td>2.0k</td>
<td>2.1k</td>
<td>Application Launcher</td>
</tr>
</tbody>
</table>