DeMinify: Neural Variable Name Recovery and Type Inference

Yi Li
New Jersey Inst. of Technology
New Jersey, USA
yl622@njit.edu

Aashish Yadavally
University of Texas at Dallas
Texas, USA
aashish.yadavally@utdallas.edu

Jiaxing Zhang
New Jersey Inst. of Technology
New Jersey, USA
jz48@njit.edu

Shaohua Wang*
New Jersey Inst. of Technology
New Jersey, USA
davidshwang@ieee.org

Tien N. Nguyen
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

To avoid the exposure of original source code, the variable names deployed in the wild are often replaced by short, meaningless names, thus making the code difficult to understand and be analyzed. We introduce DeMinify, a Deep-Learning (DL)-based approach that formulates such recovery problem as the prediction of missing features in a Graph Convolutional Network–Missing Features. The graph represents both the relations among the variables and the relations among their types, in which the names or types of some nodes are missing. Moreover, DeMinify leverages dual-task learning to propagate the mutual impact between the learning of the variable names and that of their types. We conducted experiments to evaluate DeMinify in both name recovery and type prediction on a Python dataset with 180k methods and a JavaScript (JS) dataset with 322k files. For variable name prediction, in 76.7% and 81.6% of the cases in Python and JS code respectively, DeMinify can predict correctly the variables' names with a single suggested name. DeMinify relatively improves 15.3%–40.7% and 7.7%–49.7% in top-1 accuracy over the state-of-the-art variable name recovery approaches for Python and JS code, respectively. It also relatively improves 14.5%–51.9% in top-1 accuracy over the existing type prediction approaches. Our experimental results showed that learning of data types helps improve variable name recovery and vice versa.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Language types**.

KEYWORDS

Type Inference; Name Recovery; Deep Learning; Minified Code

ACM Reference Format:
Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. DeMinify: Neural Variable Name Recovery and Type Inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616368

*Corresponding Author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616368

1 INTRODUCTION

Code minification is the process in which source code is minified such that the variable names are replaced with short, opaque, and meaningless names. It is useful in software development as it could improve rendering time due to payload size. E.g., in Web technology, AWS Cloudformation templates may have lambda function source code in Python embedded in them, but only if the function is less than 4KB. In other cases, it is a code protection scheme that slows down those who have bad intentions since program is a valuable asset to companies. Variable name minification hides the business logics from the readers while maintaining the essence of the code.

For better code readability and understandability [2], especially when the original source code is unavailable, there is a natural need to automatically recover the minified code with meaningful variable names. With the recovered names and types, the source code will be accessible for code review, analysis, enhancement, reuse, etc.

Several automated approaches have been proposed to automatically recover the names of the variables in the minified source code. The approaches can be broadly classified into three directions: *information retrieval*, *statistical learning*, and *machine learning*. JSNeat [35] follows an information retrieval (IR) approach to recover names by searching for them in a large corpus of open-source JS code. JSNeat integrates three types of contexts to match a variable in given minified code against the corpus including 1) the properties and roles of the variable, 2) its relations with other variables under recovery, and 3) the task of the function to which the variable contributes. Despite its successes, due to the inherent limitation of the information retrieval direction, JSNeat *cannot generate a new variable name* that was not encountered in the corpus.

JSNice [32], following a statistical direction, is an automatic variable name recovery approach that represents the program properties and relations among program entities in JavaScript (JS) code as dependence graphs. JSNice [32] uses a probabilistic model with the dependency network including variables and surrounding program entities. It formulates the problem of variable name recovery as the structured prediction via conditional random fields (CRFs) [32]. Unfortunately, it still has low accuracy. In contrast, JSNaughty [38] formulates that problem as a *statistical machine translation* (SMT) from minified code to the recovered code. However, its phrase-based

translation approach cannot capture well the relations among the variables to be recovered, leading to low effectiveness.

In this work, we present DeMinify, a deep learning (DL)-based variable name recovery and type inference approach. We address both tasks as parts of the dual-task learning between a variable name prediction model and a type prediction model. *Correct learning of one model can benefit for the learning of the other and vice versa* due to the naturalness of names in source code [15]. Except for a few un-important variables (e.g., running variables in a loop or temporary variables), the majority of the variables carry some contextual meaning toward achieving the task intended in the current function. *The names chosen for such a variable in the original code should be natural (unsurprising) with respect to its type.* For example, for easy comprehension, a variable of the type `Offset` might have a name relevant to the notion of *offset* or its abbreviations, e.g., `startOffset`, `endOffset`, etc. Similarly, if a model learns the name of a variable, its type should be in accordance with the name. For example, if a model recovers the name of a variable as *count* or *index*, its type might likely be of `int` or `Integer`.

Exploring this duality can provide useful constraints to predict both the variable names and their types. To build the variable name prediction model and the type prediction model, we leverage Graph Convolution Network - Missing Features (GCNmf) [34] to model different kinds of *relations/dependencies among the variables and among the types*. We formulate *the name recovery problem as the predicting the missing features* in GCNmf. With the philosophy that *"Tell Me Your Friends, I'll Tell You Who You Are"*, DeMinify decides a variable's name by learning at once the names of the variables connecting to one another. Our DL-based model is expected to have better predictive power than CRF in JSNice and SMT in JSNaughty, especially in predicting the missing features when considering both relations among variables and types. To propagate the mutual impact of type learning and name learning, we apply a dual-task learning mechanism between the two models.

We have conducted experiments to evaluate DeMinify in both name recovery and type prediction on a Python dataset with +180k methods and a JavaScript (JS) dataset with 322k files. For variable name prediction, in 76.7% and 81.6% of the cases in Python and JS code respectively. DeMinify can predict correctly the variables' names with a single suggested name. In 82% and 83.3% of the cases in Python and JS code, the correct names of local variables are in the top-5 candidate suggested lists. For Python code, DeMinify relatively improves 40.7%, 28.7%, and 15.3% top-1 accuracy over the state-of-the-art variable name recovery approaches JSNice [32], JSNaughty [38], and JSNeat [35], respectively. For JS code, the relative improvements over those baselines are 49.7%, 36.9%, and 7.7%, respectively. For variable type prediction in Python, in 79% of the cases, DeMinify can predict correctly the types with a single predicted type. Top-5 accuracy for type prediction is 88%. It relatively improves 14.5%–51.9% in top-1 accuracy and 22.2%–46.6% in top-5 accuracy over the state-of-the-art type prediction approaches HiTyper [25], Type4Py [21], Typilus [4], and TypeWriter [28].

In brief, the contributions of this paper includes:

1. **DeMinify**: a Deep Learning (DL)-based approach to recover variable names with type inference for minified code in dual-task learning. DeMinify's type inference can be used for regular code.
2. An extensive evaluation and analysis on DeMinify's accuracy.

```
1  def exportSelection(self, root, doc):
2    if not root:
3       return null
4    selection = doc.getSelection()
5    if selection.rangeCount > 0:
6       range_ = selection.getRangeAt(0)
7       preSelectionRange = range_.cloneRange()
8       preSelectionRange.selectNodeContents(root)
9       preSelectionRange.setEnd(range_.startContainer, range_.startOffset)
10      start = len(str(preSelectionRange))
11      selectionState = {
12          "start": start,
13          "end": start + len(str(range_))
14      }
15      if self.doesRangeStartWithImages(range_, doc):
16          selectionState.startsWithImage = true
17      trailingImageCount = self.getTrailingImageCount(root,
              selectionState, range_.endContainer, range_.endOffset)
18      if trailingImageCount:
19        selectionState.trailingImageCount = trailingImageCount
20      if start != 0:
21        emptyBlocksIndex = self.getIndexRelativeToAdjacentEmptyBlocks(doc,
              root, range_.startContainer, range_.startOffset)
22        if emptyBlocksIndex != -1:
23          selectionState.emptyBlocksIndex = emptyBlocksIndex
24        ...
```

Figure 1: An Original Code from a Project in GitHub

```
1  def exportSelection(self, w, b):
2    if not w:
3      return null
4    q = b.getSelection()
5    if q.rangeCount > 0:
6      r = q.getRangeAt(0)
7      d = r.cloneRange()
8      d.selectNodeContents(w)
9      d.setEnd(r.startContainer, r.startOffset)
10     m = len(str(d))
11     p = {
12          "start": m,
13          "end": m + len(str(r))
14        };
15     if self.doesRangeStartWithImages(r, b):
16       p.startsWithImage = true
17     a = self.getTrailingImageCount(w, p, r.endContainer, r.endOffset)
18     if a:
19       p.trailingImageCount = a
20     if m !== 0:
21       y = self.getIndexRelativeToAdjacentEmptyBlocks(b, w,
             r.startContainer, r.startOffset)
22       if y != -1:
23         p.emptyBlocksIndex = y
24       ...
```

Figure 2: The Minified Code for the Code in Figure 1

3. A novel formulation of variable name recovery and type inference in minified code as a missing-feature, graph-based neural network in a dual-task learning framework to benefit both tasks.

2 MOTIVATING EXAMPLE

Let us start with a real-world example to motivate our approach. Figures 1 and 2 show the original and minified versions of the function `exportSelection` in Python. The function is aimed to export/retrieve the selection from a document. In the minified code, all local variables were randomly renamed by a minification tool with short and meaningless names, *e.g.*,`root` becomes `w`, `doc` becomes `b`, etc. This makes the code difficult to comprehend.

We aim to recover the names of the variables in the minified code. Such process is not trivial and affected by multiple factors. Let us explain the following observations then to motivate our solution:

Observation 1. *The mutual impact between variable name learning and variable type learning.* If a model learns correctly the type of a variable, it would help learn better the name of the variable and vice versa. Let us consider the name `emptyBlocksIndex` for the variable `y` at line 22 in Figure 2: `if y != -1:`. From that comparison, a model could learn that `y` is of the type `int`. Knowing that it is an integer, a model could combine that with the knowledge learned from line 21 (`y = self.getIndexRelativeToAdjacentEmptyBlocks(...)`), and predict the name for `y` could be *"Index"* or similar. On the other hand, correct learning of a variable name can also benefit for learning of its type. At line 17 of Figures 1–2, if the name `trailingImageCount` is recovered for the variable `a`, its type is likely to be `int` if the model could make sense of the sub-token `count` in that name `trailingImageCount`.

Key Idea 1. [**Dual-task Learning between Name Prediction and Type Prediction**] *While aiming to recover variable names in minified code, we leverage the duality between the learning to predict the variable names and the learning to predict the variable types.* In the original code, the name of a variable should be natural (unsurprising) with respect to the type of that variable. We build a name prediction model and a type prediction one, and we apply a dual-task learning mechanism connecting the two models.

Observation 2. *The name and type of a variable are affected by the names and types of the surrounding variables in a function.* Intuitively, because multiple variables are used together to achieve the task in the function, their names are often consistent with one another. For example, at line 6, the choice of the name `range_` in the recovery process could be derived by the choice of the variable `selection` and the call to `getRangeAt` as it is made on that variable as in the statement `range_ = selection.getRangeAt(0)`. The choice of the name `preSelectionRange` could be affected by the choice of the variable `range_` due to the statement `preSelectionRange = range_.cloneRange()`. Moreover, the type system in a programming language always requires the concordance between the types of variables.

Key Idea 2. [***"Tell Me Your Friends, I'll Tell You Who You Are"***] *A variable name or type are influenced by the names or the properties of the other variables having the relations with that variable* in the surrounding context. We *treat the problem of variable name generation as predicting the missing features* in a graph neural network by leveraging Graph Convolutional Network - Missing Features (GCNmf) [34]. We also use Edge-Enhanced Graph Convolutional Network (EE-GCN) [9] to model different kinds of relations among the variables and types in the function/method.

Observation 3. *The actual variable name must also be in accordance with the names of the accessed fields and called methods.* For example, in the original code in Figure 1, the variable name `range_` makes sense in `range_.startOffset` and `range_.endOffset` because a range could have a starting offset and an ending offset. The rationale is that in the original code, for easy comprehension, developers tend to follow naming conventions and use meaningful names with respect to the surrounding variable names in the code. That is, the predicted name of a variable and the names of its properties (fields and methods) are in accordance. As an example, `preSelectionRange` and `setEnd` are in accordance with each other in `preSelectionRange.setEnd` (*"Setting the end of the selected range"*). In fact, Pradel *et al.* [29] explore the concordance between the method's name and the names of its arguments to detect name-based bugs in a program.

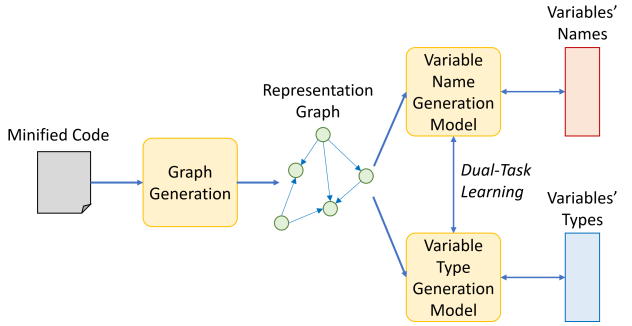

Figure 3: Neural Variable Name Recovery and Type Inference

Observation 4. *The fields and methods of a variable are kept intact after minification.* If the names of the fields and methods were minified, the corresponding field accesses and method calls would not be valid anymore. For example, `cloneRange()` in `range_.cloneRange()` at line 7 and `startContainer` in `range_.startContainer` at line 9 in Figure 1 are unchanged in Figure 2. Thus, a model can rely on the names of those properties of a variable to predict the variable's name.

Key Idea 3. [**Properties of a Variable**] *The name of a variable is in accordance with its own properties* including the names of its fields in field accesses and the names of its methods in the method calls on that variable. Moreover, the names of fields and methods are kept intact after minification, thus, a model can rely on those names to predict the variables' names. For example, a model can learn from the variables that have the field accesses to `startContainer`, `endContainer`, `startOffset`, and `endOffset`, and have the method calls to `cloneRange()`. It can use that knowledge to predict the name `range_`.

In brief, to recover the name and type of a variable, a model could examine the *properties of the variable, its relations* with other variables and their properties, and the relations among *their types.*

3 APPROACH OVERVIEW

We propose DeMinify that accepts minified code and at the same time, recovers the variable names and derives the types for variables/expressions. It also can take regular code and derive the types. Figure 3 illustrates the overall process. The input is the minified code with all the original variables' names and types during training and without them during the prediction. The process contains the following key steps. First, the minified code is parsed and two feature graphs are extracted: 1) the Type Dependency Graph [25] representing the relations among the types of the variables in a function/method according to type inference rules, and 2) the Relation Graph [35] represent the relations among the variables including the ones via field accesses and method calls (Section 4). The two graphs can always be extracted for minified code in both training or prediction. They will be merged into a representation graph.

We have two models dedicated to the two tasks: Variable Name Generation (VNG) and Variable Type Generation (VTG). DeMinify first extracts the features in a representation graph, and converts them into the input vectors for the VNG and VTG models.

The VNG model processes them as follows. For the nodes that represent the variables with the minified names, we mask the node features and regard them as the missing features, and feed the

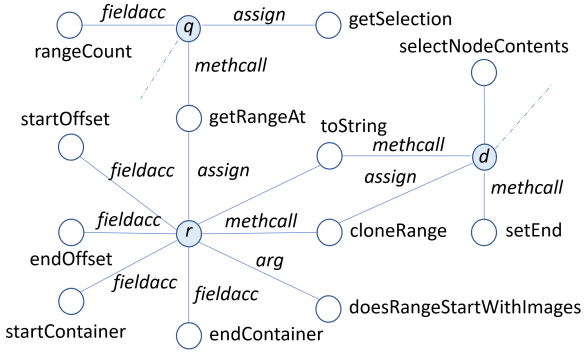

Figure 4: Relation Graph for the Code in Figure 2

graph into a Graph Convolutional Network–Missing Features (GCN-mf) [34]. The actual variables' names in the input minified code are used as the labels during training. For name prediction, the same process is used except that the variables' names are predicted by the trained VNG model. To generate names, we also use program rules to ensure scoping, and valid and consistent names.

The VTG model leverages Edge-Enhanced Graph Convolutional Network (EE-GCN) [9] with the support of an embedding model [26] as well as a Gate Recurrent Unit (GRU). The actual variables' types in the input minified code are used as the labels during training. For prediction, the same process is used except that the types are predicted using the trained model. To generate the types, we use type inference rules to eliminate the impossible candidates.

To propagate the impact between VTG and VNG, we apply a dual-task learning scheme between them. We use the uncertainty weighted multi-task loss as the loss function and use the maximum of the top-1 accuracy scores from two tasks as the training target.

For non-minified code, to infer the types, the representation graph is extracted as explained, and then fed to DeMinify whose VTG will produce the types for variables/expressions. The full variable names will help VNG improve VTG's type inference task.

4 IMPORTANT CONCEPTS

In this section, we present the important concepts used in DeMinify. To identify the name of a variable, first, DeMinify examines its own **attributes and behaviors** via field accesses and method calls, then the **relations** of the variables to learn the concordance among the variables' names. At the same time, it examines **the relations among the types of the variables**.

Definition 1. **[Attributes and Behaviors]** *The fields and methods of the object represented by a variable are referred to as the attributes and behaviors, respectively. The names for those fields and methods of a variable are intact after code minification.*

In Figure 2, at lines 7–9, we explore the field accesses and method calls of the variable `r` in the method calls and field accesses made to `r`, e.g., `r.cloneRange()`, `r.startOffset`, and `r.startContainer`.

We denote an instance of field access and method call as a triple (v, p, t), where v is the variable, p is the name of the field or method, and t is either `fieldAccess` or `methodCall`. The examples are $(r, cloneRange, methodCall)$ and $(r, startOffset, fieldAccess)$.

$$
\begin{array}{rcl}
\theta \in Type(\Theta) & ::= & \gamma \mid \alpha[\theta, ..., \theta] \mid u \mid \textbf{None} \mid \textbf{type} \\
\gamma \in ElementaryType(\Gamma) & ::= & \textbf{int} \mid \textbf{float} \mid \textbf{str} \mid \textbf{bool} \mid \textbf{bytes} \\
\alpha \in GenericType(A) & ::= & \textbf{List} \mid \textbf{Tuple} \mid \textbf{Dict} \mid \textbf{Set} \mid \\
 & & \textbf{Callable} \mid \textbf{Generator} \mid \textbf{Union} \\
b \in BuiltinType(B) & ::= & \gamma \mid \alpha[\theta] \\
u \in UserDefinedType(U) & ::= & \textit{all classes and named tuples} \\
o \in OverloadingUserDefType(O) & ::= & \textit{all classes with operator} \\
 & & \textit{overloading in code}
\end{array}
$$

Figure 5: Types in Python

Definition 2. **[Argument Relation]** *A variable v is said to have an argument relation with a method m if it is used as an argument of a call to that method as in $o.m(..., v, ...)$.*

Definition 3. **[Assignment Relation]** *A variable v is said to have an assignment relation with a method m or a field f if it is used as a left-hand side in an assignment from a method call or a field access as in $v = o.m(...)$, or $v = o.f$.*

The idea is that the name of the minified variable v in the original code is often in accordance with the names of the method or the field in such an assignment or an argument. For example, `selectNodeContents` and `root` are in accordance with each other in `preSelectionRange.selectNodeContents(root)`; or `range_` and `getRangeAt` are in accordance with each other in `range_ = selection.getRangeAt(0)`. We will use the triple notations $(v, m, argument)$, $(v, m, assignment)$, and $(v, f, assignment)$ to denote those three cases, where v is a variable, m is a method name, and f is a field name.

Definition 4. **[Relation Graph]** *[35] A relation graph (RG) is a directed graph in which each node of the RG represents a variable. The connected nodes represent the methods/fields in method calls or field accesses, respectively, and are labeled with their names. Edges represent relations among nodes and are labeled with relation types.*

Figure 4 shows the relation graph for the variables in the code in Figure 2. For example, there are an *assign* edge from the variable `r` to the node `getRangeAt`, and a *methodcall* edge from `q` to `getRangeAt` because we have `r = q.getRangeAt(0)` at line 6.

Regarding type inference, Figure 5 shows the type system in Python [25]. To represent the dependencies among the types, we adopt Type Dependency Graph (TDG) [25], which aims to capture the type inference rules for variables/expressions.

Definition 5. **[Type Dependency Graph]** *[25] A Type Dependency Graph is a graph $G = (N, E)$ in which N is the set of nodes representing all the variables and expresions, and E is the set of edges from $n_i \rightarrow n_j$ indicating that the type of n_j can be derived from the type of n_i by the type inference rules in the type system.*

In Figure 2, let us consider line 4: `q = b.getSelection()`. The TDG will contain a node for the expression `b.getSelection()` connecting to a node for the variable `q` because its type can be derived from the return type of the method call `getSelection`. We also have a node for the variable `b` connecting to a node of the method `getSelection` since the type of `getSelection` can be derived from that of variable `b`.

`b.getSelection` → `q`

`b` → `getSelection`

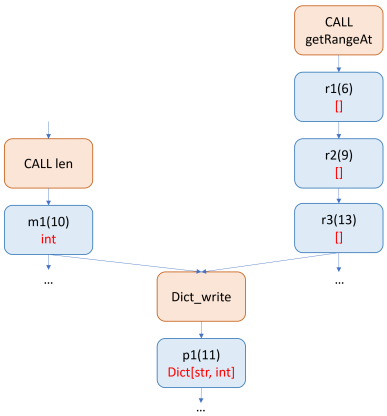

Figure 6: Type Dependency Graph for the Code in Figure 2

Connecting all the dependencies among the types of variables and expressions, we have the type dependency graph for a function/method. Note that both RG and TDG can be built for either the original or minified code. The merging of RG and TDG is straightforward. The nodes of the same variables, method calls, field accesses in two graphs are unified. The other nodes in two graphs are kept. All edges are combined including the type-dependency edges.

Figure 6 displays the TDG for the running example. The type of the variable *m* at line 10 (Figure 2) can be derived by the function call `len`. The subscript is used to denote the same variable at different locations. At line 11 (Figure 2), the type of the variable `p` is derived from the type of the left-hand-side of line 11 (`Dict`). Thus, we have an edge from `Dict_write` to `p`. Similarly, the type of the variable `r` at line 6 can be derived from the call to `getRangeAt`. The types at line 13 and line 9 can be derived from the type at line 6.

5 VARIABLE NAME GENERATION MODEL

This section presents the Variable Name Generation Model (VNG). During training, the input is the minified code with all the original variables' names and types, and during predicting, the input does not have names and types. First, we build TDG and RG for the given code. The two graphs are combined into a representation graph G. For each node in G, we tokenize the names in the corresponding code sequence of the node. We consider each of them as a sentence and use an embedding model (e.g., GloVe [26]) to build the representation vector for each node in G.

Next, for training, we process the graph G with the node vectors as follows. For the node n that represents a variable with minified name, we perform masking the node feature with a special mask token for the variable name and consider it as a missing feature. In the VNG model, we leverage an advanced neural network called Graph Convolutional Network–Missing Features (GCNmf) [34]. The actual names are used as the ground truth labels to train the GCNmf model. The key characteristic of GCNmf is its ability to deal with incomplete and missing features in a GCN. It represents the missing data by Gaussian Mixture Model (GMM) and calculates the expected activation of neurons in the first hidden layer of GCN, while keeping the other GCN layers unchanged. The GMM parameters and GCNmf weight parameters are learned within the same architecture, enabling the learning of missing features. The GCNmf model is trained with the masks for the minified variable names. For prediction, applying on the minified code (without names), the GCNmf model outputs the vectors for the nodes in the graph G and the vectors for the missing features.

The vectors representing the missing features, i.e., the missing variables' names in the input graph G are used next. We leverage an Gate Recurrent Unit (GRU) as a decoder. The decoder accepts those vectors for missing names as input and generates the names for the variable nodes (During training, the name labels are known and used). Finally, we apply the semantic checkers to make sure that the variables' names are valid in the scope and the same variable is assigned with a consistent name.

Let us use Figure 4 to illustrate the benefit of *modeling the variable name generation/recovery problem as predicting the missing features* using GCNmf [34]. A variable with a minified name is modeled as a nodel in our graph, e.g., r, q, and d. The prior works based on machine translation (e.g., JSNaughty [38]) aiming to translate the minified code to the orginal code, face an issue of different naming schemes used by different minification tools. For example, the variable `range` might become r_1, instead of r, by a different minification tool or by alpha-renaming. In DeMinify, the minified names themselves do not play a crucial role in deciding the original names as in prior work. They help mainly in recognizing the occurrences of the same variable. In our graph G, DeMinify considers a node for a minified variable as a placeholder with a missing feature that it aims to fill in. In prediction, GCNmf will create the vectors v_r, v_q, and v_d for the nodes. During the convolution process, those vectors are automatically updated based on the neighboring node features and the relations. After convolution, the final vectors will be fed into the GRU decoder for variable name prediction.

Next, let us present the variable type generation model, and then the dual-task learning scheme between VNG and VTG models.

6 VARIABLE TYPE GENERATION MODEL

This section presents the Variable Type Generation Model (VTG). The input is the minified code with all the original variables' names and types during training, and without the types during prediction. Similar to VNG model, the combined graph G is processed in which the names in the code sequence of each node n is tokenized and an embedding model is applied to build the vector for n considering each sequence of sub-tokens for n as a sentence.

Next, we feed the graph G with those vectors to Edge-Enhanced Graph Convolutional Network (EE-GCN) [9]. EE-GCN could accept both node and edge features. We use the above vectors as node features, and the edge types in the graph G (built from TDG and RG) as the edge features. The rationale for choosing EE-GCN is its capability producing the embeddings emphasizing on the edges in G, which represent the relations/dependencies among the data types. A key characteristic of EE-GCN is that it has an edge-aware node update module and a node-aware edge update module, and two modules work in a mutual way by updating each other iteratively. Specifically, "for each layer, the edge-aware node update module is first performed for aggregating information from neighbors of each node through specific edges. Then, a node-aware edge update module is used to dynamically refine the edge representation with

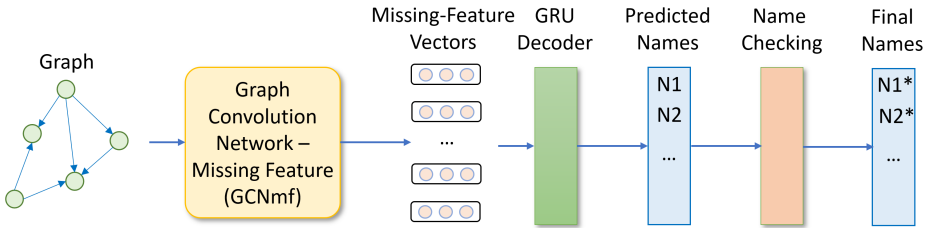

Figure 7: Variables Name Generation Model (VNG)

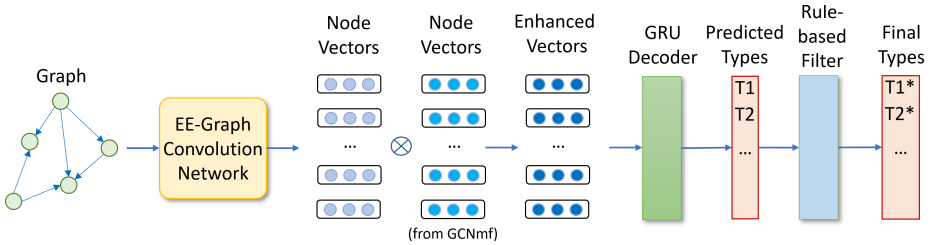

Figure 8: Variable Type Generation Model (VTG)

its connected node representations, making the edge representation more informative" [9]. The output of the EE-GCN model includes the list of the representation vectors V_n for all the nodes in G.

To further propagate the impact from variable name learning to type learning, we combine the above vectors V_n with the vectors obtained from the GCNmf in the Variable Name Generation model. Specifically, we use the cross-product between the two vectors to produce the final vectors V_f for the type prediction for all the nodes. Next, we leverage a Gate Recurrent Unit (GRU) as a decoder, which accepts the vectors V_fs as input and generates the type for the node. (During training, the type labels are used). DeMinify handles the primitive and non-primitive types in the same way as the decoder generates them as texts for the type names. Finally, we also apply the rule-based filter, which performs type-checking to eliminate the candidates that violate the type inference rules. In the current implementation, we used a small subset of the static type inference rules for Python [18]. For example, we check the types of the left-hand-side and the right-hand-side of an assignment, the type of the condition in an `if` statement, the type of a comparison operation, the simple sub-typing rules for primitive types and for lists, tuples, and dictionaries, etc. The final result contains the types for all the nodes including the the nodes for the variables.

7 DUAL-TASK LEARNING FOR VNG AND VTG

In DeMinify, to propagate the mutual learning between variable name learning and variable type learning, we leverage a dual-task learning framework to train both VNG and VTG models together. As for these two tasks, DeMinify regards them as the regression problem and multi-task loss [8] to learn both of them at the same time. Specifically, for each regression, DeMinify uses a smooth L1 loss function to estimate the accuracy of output as follows:

$$L = \{l_1, l_2, ..., l_n\}^T \tag{1}$$

$$l_n = \begin{cases} 0.5(f(x)_n - y_n)^2 & |f(x)_n - y_n| < 1 \\ |f(x)_n - y_n| - 0.5 & otherwise \end{cases} \tag{2}$$

Where $f(x)_n$ is the output for a variable n in a regression task f; y_n is the ground truth. To get the joint loss function for the dual-task learning with uncertainty weighting, following Kendall *et al.*'s [8], we have:

$$L_i(W) = \{l_1^W, l_2^W, ..., l_n^W\}^T \tag{3}$$

$$l_n^W = \begin{cases} 0.5(f(x)_n^W - y_n)^2 & |f(x)_n^W - y_n| < 1 \\ |f(x)_n^W - y_n| - 0.5 & otherwise \end{cases} \tag{4}$$

$$L(W, \sigma_1, \sigma_2) = \sum_i \frac{1}{2\sigma_i^2} L_i(W) + log\sigma_i^2 \tag{5}$$

Where W is the weight adding to the input, σ_i is the i^{th} noise scalar, and W and σ_i are both trainable parameters in the model.

By combining all the loss functions into one as in Formula 5, DeMinify trains the variable name model and the type prediction model together. We choose the smallest loss results to get the most suitable model parameters for DeMinify.

8 EMPIRICAL EVALUATION

8.1 Research Questions and Datasets

To evaluate DeMinify, we seek to answer the following questions:
RQ1. Comparative Study on Variable Name Prediction. How well does DeMinify perform in comparison with the state-of-the-art variable name prediction approaches on the Python dataset?
RQ2. Comparative Study on Variable Name Prediction. How well does DeMinify perform in comparison with the state-of-the-art variable name prediction approaches on the JavaScript dataset?
RQ3. Comparative Study on Variable Type Prediction. How well does DeMinify perform in comparison with the state-of-the-art variable type prediction approaches on the Python dataset?
RQ4. Ablation Study. How do the key features in DeMinify affect its overall performance?

Dataset. We have conducted our experiments to evaluate DeMinify on the well-established Python dataset, ManyTypes4Py provided in the work by Mir *et al.* [20]. The dataset includes +180k methods

from 4,000 Python projects with 37,408 different variable types. We applied Pyminifier [30], a minification tool for Python to minify the variable names in the source files. We also used a dataset in JavaScript (JS) provided in a prior work, JSNeat [35]. The JS dataset includes +320k methods from 20,000 projects with 176k unique variable names. We minified them with the minifying tool UglifyJS [37].

8.2 Experimental Methodology

8.2.1 Comparison on Variable Name Prediction in Python (RQ1). *Baselines.* We compared DeMinify against the state-of-the-art variable name recovery approaches for minified code including JSNeat [35], JSNice [32], and JSNaughty [38].

Procedure. We took all the methods in the Python dataset and used Pyminifier to produce the minified code with variable name mininification. We randomly split all the methods into 80%, 10%, 10% in which 80% of the methods as the training dataset, 10% of the methods as the tuning dataset, and 10% of the methods as the testing dataset for all the baselines and DeMinify.

Parameter Tuning. We tuned DeMinify with autoML [1] for the following key hyper-parameters to have the best performance: (1) Epoch size (100, 150, 200); (2) Batch size (64, 128, 256); (3) Learning rate (0.001, 0.005, 0.010); (4) Vector length of feature embeddings and its output (32, 64, 128); (5) Number of GCN layers (4, 8, 16).

8.2.2 Comparison on Variable Name Prediction in JS (RQ2). We build the JS representation graph and use the same model.

Baselines. We compared DeMinify against the state-of-the-art approaches JSNeat [35], JSNice [32], and JSNaughty [38].

Procedure and Tuning. We used UglifyJS for code minification in the JS dataset. We used the same splitting and tuning as in RQ1.

8.2.3 Comparison on Variable Type Prediction (RQ3). Baselines: we compared DeMinify against the state-of-the-art variable type prediction approaches: Ivanov *et al.* [16], Typilus [4], TypeWriter[28], Type4Py [21], and HiTyper [25].

We did not compare our approach with Xu *et al.* [40], DeepTyper [14], NL2Type [19], LAMBDANET [39], OptTyper [24], and TypeBERT [17], because in its paper, Type4Py has been shown to perform better than those tools. We did not compare on JS code because JS variables must not always have type declarations.

Procedure. With all variable type information summarized in the ManyTypes4Py dataset, we directly used the types for the variables as the ground truth for all the approaches. We used the same data splitting for training, tuning, and testing as in RQ1. We ran the baselines on the original source code with the parameters in their documentation. We fed to DeMinify the minified code but with the original names and no types (i.e., the original code), and obtained the resulting types from the variable type generation (VTG) model for all the variables/expressions in the code.

Parameter Tuning. Tuning was done via autoML [1] as in RQ1.

8.2.4 Ablation Study (RQ4). We built several variants of DeMinify to evaluate different factors in its components by measuring their accuracies and making comparisons. First, we evaluated our hypothesis that mutual impact exists between name learning and type learning via dual-task learning. We built two variants VNG and VTG separately without connecting them via dual-task learning. Second, the graph representation for source code is important in the name and type prediction. Thus, we kept the same architecture for DeMinify and fed to it different graph representations for source code. This allows us to evaluate the impact of our chosen graph representations for this problem. Third, the key technical solution in DeMinify is the formulation in which predicting names and types for variables is considered as predicting the missing features of connected program elements. We currently used the EE-GCN model to capture the relations and dependencies among program elements. We replaced it with the Label-GCN [6] and made a comparison.

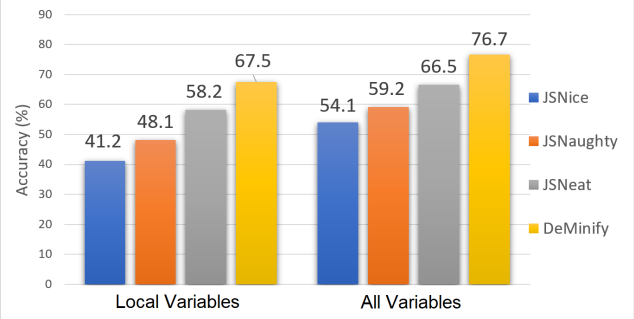

Figure 9: Top-1 Accuracy on Name Prediction (Python) (RQ1)

Evaluation Metrics. For the variable name prediction, we follow a prior work [35] to calculate the prediction accuracy on local variables and all variables. We compared the resulting names from a tool against the original names. A tool is considered to correctly recover the name of a variable v if the recovered name is matched exactly with its original name. For v, if matching, we count it as a hit, otherwise, it is a miss. Accuracy is measured by the ratio between the total number of hits over the total number of cases. Top-k accuracy is measured similarly, however, a hit is achieved when the correct name is in the top-k candidate list from a tool.

For the variable type prediction, we use two metrics, `Exact-Match` and `Parametric-Match` with the top-k accuracy as in HiTyper [25]. `Exact-Match` occurs when the resulting type matches exactly with the human-annotations. `Parametric-Match` occurs if the result matches with the correct type but those of the parameters might not. For example, `Dict[int,str]` is parametric-matched with `Dict[int,int]`.

9 EXPERIMENTAL RESULTS

9.1 Comparison on Name Prediction (RQ1)

As seen in Figure 9, for all variables in minified Python code, DeMinify achieves high top-1 accuracy of **76.7%**: i.e., in 76.7% of the cases, it can recover the correct variable names with a single prediction. The relative improvements in top-1 accuracy for all variables over JSNice, JSNaughty, and JSNeat are **40.7%, 28.7%**, and **15.3%**, respectively. The absolute improvements in top-1 accuracy over those state-of-the-art approaches are from 10.2%–22.2%.

Considering only local variables, in **67.5%** of them, DeMinify correctly predicts their original names with a single result. The relative improvements in top-1 accuracy in recovering local variables' names over JSNice, JSNaughty, and JSNeat are **62.2%, 39.8%**, and **15.9%**, respectively. The absolute improvements in top-1 accuracy over those state-of-the-art approaches are from 9.3%–25.9%.

Table 1: Comparison on Variable Name Prediction (RQ1)

	Top-1		Top-3		Top-5	
	Local	All	Local	All	Local	All
JSNice [32]	41.2	54.1	52.2	63.0	59.5	67.8
JSNaughty [38]	48.1	59.2	59.8	69.7	66.3	75.0
JSNeat [35]	58.2	66.5	65.3	75.4	71.6	80.1
DeMinify	67.5	76.7	75.4	84.3	82.1	90.2

As seen in Table 1, the result is also consistent for top-3 and top-5 accuracies for DeMinify. The relative improvements are with same trends in comparison with the baselines.

We examined the predicted names from all the baselines. Compared to JSNeat, an information retrieval approach, we found that it often failed in the following cases. (1) It has not seen the names before in the database. DeMinify can generate a new name with its decoder. Due to its explicitly setting of similarity thresholds, JSNeat faces two other issues. (2) The correct name was not returned since the relations and contexts are not similar enough with pre-defined thresholds. (3) If two variables in the same function are assigned with the same name via the similarity measure, JSNeat cannot decide one, turning to a random selection. Avoiding feature matching and explicit similarity threshold, DeMinify with its neural network can implicitly do so without any pre-defined threshold.

Compared to JSNaughty [38] with statistical machine translation from minified code to original code, we found that it relies much on the *minified names* in minified code. We examine the *phrase mapping table*, a byproduct of their machine translation model, which contains the knowledge it learned from training. We reported that JSNaughty learns *inconsistent mappings* between the minified names and original ones. That is, there are several mappings for the same minified names with different weights depending on their occurrences in the corpus. For example, b ↔ doc, b ↔ book, etc. In contrast, DeMinify does not rely on the minified names themselves, by modeling them as the nodes with missing features (i.e., missing names and types) in the graph-based GCNmf model [34].

DeMinify is similar in spirit to JSNice [32] in which both formulates the problem as predicting the features/attributes of the nodes in a graph. The Conditional Random Field in JSNice uses probabilistic name prediction graph. In comparison, DeMinify leverages more advanced neural network in GCNmf [34] as well as a more specialized graph with the help of the type prediction model.

9.2 Comparison on Name Prediction in JS (RQ2)

As seen in Figure 10, for all variables in minified JS code, DeMinify achieves high top-1 accuracy of **81.6%**: i.e., in 4 out 5 cases, it can recover the correct variable names with a single prediction. The relative improvements in top-1 accuracy for all variables over JSNice, JSNaughty, and JSNeat are **49.7%, 36.9%,** and **7.7%**, respectively. The absolute improvements in top-1 accuracy over those state-of-the-art approaches are from 5.8%–27.1%.

Considering only local variables, in **76.8%** of them, DeMinify correctly predicts their original names with a single result. The relative improvements in top-1 accuracy in recovering local variables' names over JSNice, JSNaughty, and JSNeat are **84.6%, 59.0%**, and **11.1%**, respectively. As seen in Table 2, the comparison result is also consistent for top-1, top-3, and top-5 accuracies.

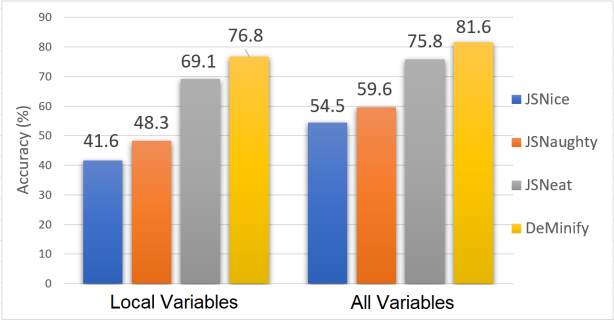

Figure 10: Top-1 Accuracy on Name Prediction in JS (RQ2)

Table 2: Comparison on Name Prediction in JS (RQ2)

	Top-1		Top-3		Top-5	
	Local	All	Local	All	Local	All
JSNice [32]	41.6	54.5	56.4	68.2	64.2	72.4
JSNaughty [38]	48.3	59.6	64.1	74.5	71.8	79.6
JSNeat [35]	69.1	75.8	69.6	79.5	76.9	86.0
DeMinify	76.8	81.6	76.2	85.6	83.3	91.8

As seen in Figures 9 and 10, DeMinify improves over the baselines in both languages. However, its relative improvement over the top baseline, JSNeat [35], for Python is higher than that for JS. The reason is that JS is a weakly typed language. In JS code, the types of variables do not need to be specified, and can be changed (that is, type information is not always available). DeMinify is less effective in those cases, while JSNeat [35], an IR approach, is still effective since the variable names might be seen in the JS dataset.

9.3 Comparison on Type Prediction (RQ3)

While we focus on variable name prediction, the result of variable type prediction is also useful since the types in the minified code are exactly the same for the original code. In this study, we evaluate DeMinify's accuracy and compare it with the state-of-the-art approaches in variable type prediction. As seen in Figure 11, for all variables, DeMinify achieves high top-1 accuracy of **79%** for exact-matches of the types and **89%** for the parametric matches. That is, in 79% of the cases, it can recover the correct variable types with a single prediction. The relative improvements in top-1 accuracy in exact-matching over Ivanov *et al.* [16], TypeWriter[28], Typilus [4], Type4Py [21], and HiTyper [25] are **51.9%, 43.6%**, **33.8%**, **27.4%**, and **14.5%**, respectively. The absolute improvements in top-1 accuracy over those type prediction approaches are from 10%–27%.

Regarding the parametric matches (disregarding the types of the parameters), DeMinify achieves higher top-1 accuracy. In 89% of the cases, it can recover the correct variable types (regardless of parameters' types) with a single prediction. The relative improvements in top-1 accuracy in parametric-matching over Ivanov *et al.* [16], TypeWriter [28], Typilus [4], Type4Py [21], and HiTyper [25] are **53.4%, 45.9%**, **34.8%**, **34.8%**, and **15.5%**, respectively. The absolute improvements in top-1 accuracy with parametric matching over those state-of-the-art approaches are from 12%–31%.

Moreover, DeMinify also achieves high top-k (k=3,5) accuracies: as seen in Table 3, in 88% of the variables, the correct types

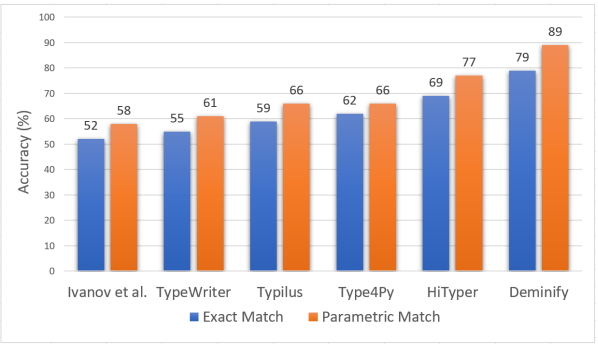

Figure 11: Top-1 Accuracy on Type Prediction (RQ3)

Table 3: Comparison on Type Prediction (RQ3)

	Top-1		Top-3		Top-5	
	EM	PM	EM	PM	EM	PM
Ivanov *et al.* [16]	52	58	55	63	60	67
TypeWriter [28]	55	61	59	66	62	70
Typilus [4]	59	66	63	71	64	73
Type4Py [21]	62	66	66	72	67	73
HiTyper [25]	69	77	72	81	72	82
DeMinify	79	89	87	90	88	94

EM: Exact Match, **PM**: Parametric Match

of the variables are in the list of five candidate types. The relative improvements in top-5 accuracy in parametric-matching over Ivanov *et al.* [16], TypeWriter[28], Typilus [4], Type4Py [21], and HiTyper [25] are **46.6%, 42%**, **37.5%**, **31.3%**, and **22.2%**, respectively.

We examined the cases that DeMinify predicted correctly and the baselines missed. Compared with HiTyper [25], HiTyper did not perform well for the isolated groups of a couple variables. That is, the groups are isolated, i.e., not type-dependent to other variables and HiTyper did not correctly detect any of those variable types. In those cases, DeMinify could rely on the concordance between the names and types of a variable. For example, the type of a variable named `index` or `count` will likely be `int`.

Type4Py [21] uses a hierarchical neural network model to learn to distinguish between similar/dissimilar types in a vector space. The candidate types are predicted via nearest neighbor search. In contrast, DeMinify also encodes the information on variable names into the embeddings in the high-dimensional space via two mechanisms: dual-task learning and cross-product of vectors. Thus, DeMinify could perform better in the cases in which the embeddings for types are not sufficiently distinguishable from one another, which poses challenges for Type4Py. Regarding the used neural networks, it uses Recurrent Neural Network (RNN) operating on the code token embeddings built from the AST. In contrast, we use GCNmf that captures better the dependencies with different types of edges.

Compared to Typilus [4], DeMinify relatively improves 31.6% in top-1 accuracy. Similar to Type4Py, Typilus builds a vector space for type embeddings. It uses a graph neural network to learn to map variables, parameters, and function returns to a type embedding space using deep similarity learning. For type inference, using the type map, it accepts unannotated code, computes type embeddings with the trained GNN and finds the concrete k nearest neighboring

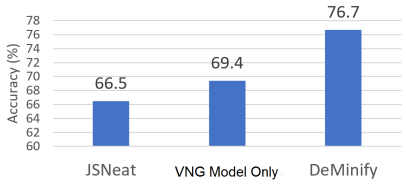

Figure 12: Dual-Task Learning on Name Prediction (RQ4)

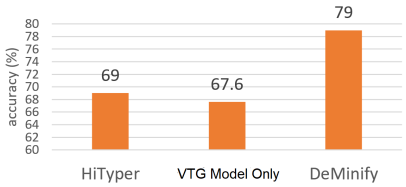

Figure 13: Dual-Task Learning on Type Prediction (RQ4)

types as the candidates. There are two key limitations in Typilus that DeMinify overcomes. First, their graph representation does not directly encode the type dependencies. It encodes among the code tokens the next-lexical-token relation, structural/syntactical relations, and data dependencies. Second, it does not have the assistance of the learning of variable names via dual-task learning as explained in the comparative analysis with HiTyper.

TypeWriter [28] builds token embeddings and uses two RNNs for identifiers and for code to merge them to form type vectors. It then uses feedback-directed search on the results from a static type checker to search for consistent types. The key limitation is that the type dependencies are not encoded during the learning. Instead, TypeWriter leverages an external static type checker and relies on the feedback-directed searching for the right types. If the RNN models do not produce the right types at the first place, searching via a static type checker will not result in any better output.

For Ivanov *et al.* [16], DeMinify has the same advantages as the comparison to the above approaches with graph embeddings.

9.4 Ablation Study (RQ4)

9.4.1 Impact of Dual-Task Learning. Figure 12 shows the Top-1 accuracy in variable name prediction when we removed the dual-task learning scheme and measured only the accuracy of the variable name generation (VNG) model. As seen, without the impact from variable type generation (VTG) via dual-task learning, VNG still performs better than the best baseline, JSNeat (69.4% versus 66.5%). The drop in Top-1 accuracy from DeMinify is 10.5% (from 76.7% down to 69.4%). Similarly, as seen in Figure 13, without the impact from VNG due to the removal of the dual-task learning scheme, VTG performs slightly worse than the best baseline HiTyper. The drop in Top-1 accuracy from DeMinify is 14.4%. These results indicate the positive contribution to DeMinify from the dual-task learning for the mutual impact of VNG and VTG (Key Idea 1).

9.4.2 Impact of Different Types of Program Graphs. As in any approach, code representation is important and affects the performance. In this study, we kept the same neural network architecture, however, we changed different input graphs extracted from source code. In addition to the graphs used in DeMinify (Relation Graph

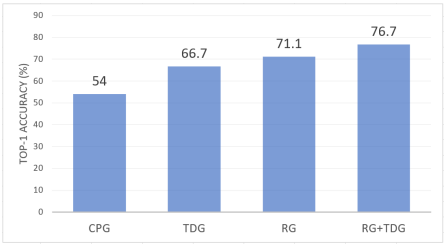

Figure 14: Impact of Input Graphs on Name Prediction (RQ4)
CPG: Code Property Graph, **TDG**: Type Dependency Graph, **RG**:Relation Graph.

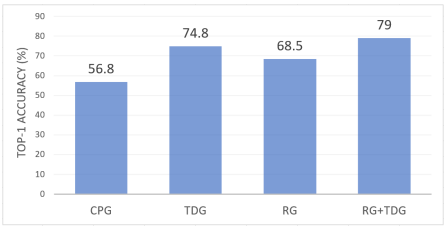

Figure 15: Impact of Input Graphs on Type Prediction (RQ4)

Table 4: Impact of EE-GCN on Top-1 Accuracy (RQ4)

Accuracy (%)	Name Prediction	Type Prediction
Label-GCN+GCNmf	68.5	71.1
EE-GCN+GCNmf (DeMinify)	76.7	79

Label-GCN: Label Graph Convolutional Network,
EE-GCN: Edge-Enhanced Graph Convolutional Network,
GCNmf: Graph Convolutional Network - Missing Features

(RG), Type Dependency Graph (TDG)), we experimented with code property graph (CPG) [41] since it has been used in several machine learning approaches for code [41]. We did not experiment with program dependence graph (PDG) because it works at the statement level, which does not help with variable name prediction.

As seen in Figure 14 and Figure 15, for variable name prediction, RG helps the model more than TDG and for variable type prediction, TDG helps the model more than RG. This is expected because each of them is designed toward capturing the key features for its problem. For the dual tasks, the combined graph (RG+TDG) in DeMinify yields the highest accuracies in both name prediction and type prediction. In contrast, CPG capturing the dependencies among program elements are not specifically designed for handling variable names and types, thus, did not yield high accuracy as TDG, RG, and CG. An interesting observation is that CPG, which contains lexical, syntax, and dependency information, did not help the model as much as others. It seems that CPG contains too much irrelevant information for variable name and type prediction.

9.4.3 Impact of Graph Models. Table 4 shows the accuracy of the variant model as we replaced the graph neural network EE-GCN with the Label Graph Convolutional Network (Label-GCN) [6]. We did not replace GCNmf because it is the core component in our solution, which formulates the name/type recovery as the prediction of missing features. As seen, Edge-Enhanced GCN helps improve accuracy more than Label-GCN. EE-GCN enables the modeling of different types of relations because it handles different edge types in different channels, while Label-GCN just integrates the edge information as simple labels. This is crucial for our problem in which the type dependencies and the variable relations are well captured.

9.5 Limitations, Examples, Threats to Validity

9.5.1 Limitations. First, DeMinify does not predict names well for the original code with the short and meaningless names, e.g., `i`, `j`, etc. In this case, the variable type could not help because the name and type are not in accordance. The relations among the variables in the context do not help either because there does not exist the natural-language semantic connections among their names. The following example shows such a case: the variable `r` at line 2 was minified into `e` at line 6. DeMinify failed this case and JSNeat predicted correctly as it has seen the code with the same names.

```
1 if (word.indexOf('G') === 0 || word.indexOf('M') === 0) {
2   r = _find(SMOOTHIE_MODAL_GROUPS, (group) => {
3     return _includes(group.modes, word); ...
4 // Minified code
5 if(s.indexOf("G")===0||s.indexOf("M")===0){
6   e =_find(SMOOTHIE_MODAL_GROUPS,t=>{
7     return _includes(t.modes,s); ...
```

Second, DeMinify does not produce well long variable names with several sub-tokens. In the next example, the variable `resetDocumentPropertyIsSet` (line 2) was minified into `t` (line 6). DeMinify predicted an incorrect name `resetDocumentOwnProperty` with its current decoder. A solution could be to replace the GRU decoder with a better model that can predict the length of the name and the name itself.

```
1 function shouldResetDoc(config) {
2   var resetDocumentPropertyIsSet = config.hasOwnProperty("...");
3   return config.resetDocument || !resetDocumentPropertyIsSet; ...
4 // Minified code
5 function shouldResetDoc(e) {
6   var t = e.hasOwnProperty("...");
7   return e.resetDocument || !t; ...
```

Third, for the type prediction, DeMinify works not so well for the user-defined types with long names. Fourth, in some cases, generic types (e.g., String) did not help learn variable names.

Finally, we currently implemented DeMinify for two languages: Python and JS. However, the approach is general for any language with a weak/strong type system in which it works better for a strong one. To expand for a new such language, one just needs to have a parser and a representation graph building module for that language. The graphs will be fed to the same architecture model.

9.5.2 Examples. Figure 16 shows an example of correct predictions from DeMinify. At line 16 (minified code), our model can derive the type `bool` for `p` due to the `if` statement. At line 15, `p` has a relation via an assignment with `check_request_limit()`, then our model can produce the name `is_limited` (instead of `limit`) for `p`. At line 14, the variable `q` has a relation with `extract_ip_from`, DeMinify can use the name relevant to *ip* for `q`, e.g., `client_ip`.

At line 18 with a minus operation, DeMinify can derive the type of the variable `m` as `int`. It also has a relation with `BirdNameDatabase.objects.count()`, thus, our model can derive the name `count` for `m`.

At line 20, `n` is used as the index of an array type. Moreover, at line 19, our model can derive its type of `int` due to the relation

```
1  def get(self, request):
2    client_ip = self.extract_ip_from(request)
3    is_limited = self.check_request_limit(client_ip)
4    if is_limited:
5      return Response({}, status=rest_framework.status.HTTP_403_FORBIDDEN)
6    count = BirdNameDatabase.objects.count() - 1
7    index = randint(0, count)
8    bn = BirdNameDatabase.objects.all()[index]
9    serialized = BirdNameSerializer(bn, many=False)
10   self.save_general_statistics(client_ip, bn)
11   return Response(serialized.data)
12 // Minified code
13 def get(self, r):
14   q = self.extract_ip_from(r)
15   p = self.check_request_limit(q)
16   if p:
17     return Response({}, status=rest_framework.status.HTTP_403_FORBIDDEN)
18   m = BirdNameDatabase.objects.count() - 1
19   n = randint(0, m)
20   t = BirdNameDatabase.objects.all()[n]
21   s = BirdNameSerializer(t, many=False)
22   self.save_general_statistics(q, t)
23   return Response(s.data)
```

Figure 16: A Correct Prediction Example by DeMinify

```
1  def post(self, event: Subscribable) -> None:
2    event_class = type(event)
3    if event_class not in self._listeners:
4      return
5    for listener in self._listeners[event_class]:
6      listener(event)
7  // Minified code
8  def post(self, B:Subscribable)->None:
9    C=type(B)
10   if C not in self._listeners:
11     return
12   for D in self._listeners[C]:
13     D(B)
```

Figure 17: Another Correct Prediction Example by DeMinify

with `randInt()`. Thus, our model can assign the name `index` for `n`. Moreover, due to the not-so-much-meaningful name `bn` at line 8, our model did not produce the name for `t` at line 20.

Figure 17 shows another example in the project `EventBus` in our dataset. At line 12, our model can recognize the variable `D` used in a `for` loop with an array of `listeners`, thus, can assign for `D` the name `listener`. From line 8, our model can learn that `B` is of the type `Subcribable` and at line 13 and it knows that `B` is related to `D` (`listener`). Thus, from *Subcribable* and *listener*, it could derive the name for `B` as `event`. At line 9, if our model can learn that `type` is to return the class for `B`, it can assign `event_class` for `C` due to the assignment.

9.5.3 Threats to Validity. We evaluated in one dataset, which might not be representative. However, the dataset has been verified and used in prior work. DeMinify currently works for Python and JS. Other programming languages could be supported as explained above. Because some of the baselines for variable name recovery were designed for JS, we re-implemented for Python based on their source code and documentation. We kept all the parameters in their tools as in their documentation or original source code.

10 RELATED WORK

Variable Name Recovery approaches. The approaches follow the three categories: *information retrieval* (IR), *statistical learning*, and *machine learning* (ML). JSNeat [35] follows an IR approach to search for the names in large code corpus, but is not effective for un-seen names. JSNice [32] infers the variable names via structured prediction with conditional random fields (CRFs) [32]. We showed that deriving missing features via a neural network yields higher accuracy than predicting program properties with statistical learning. DeMinify is computationally heavier than those approaches.

JSNaughty [38] uses a statistical machine translation from the minified code to recovered code. First, it faces the issue of relying on minified names as explained. Second, it uses a phrase-based translation model, which enforces a strict order between the recovered variable names in a function. The other methods for code deobfuscation mainly leverage static/dynamic analyses [7, 22, 36].

ML-based Type Inference approaches. HiTyper [25] is a hybrid approach between static inference and deep learning. It uses TDG to encode type inference rules to conduct type rejection to inspect the output predictions. It iteratively conducts static inference and DL-based prediction until the TDG is fully inferred. As shown, DeMinify also improves over HiTyper due to the propagation of VNG to VTG. Type4Py [21] is a deep similarity learning-based hierarchical neural network model. It learns to discriminate between similar and dissimilar types in a high-dimensional space. DeMinify avoids finding similar types via embeddings. Typilus [4] proposes TypeSpace containing the embeddings with type properties of a symbol. TypeWriter [28] learns to infer the return and argument types for functions from partially annotated code bases by combining the natural language properties of code with programming languages. DeMinify does not use natural-language properties. Ivanov *et al.* [16] show that graph-based embeddings could improve type prediction. We use GCNmf to predict the missing features.

Statistical NLP approaches have been used for name and code style suggestions [2, 3]. Other applications include code suggestion [15, 23], code convention [2], name suggestion [3], API suggestions [33], code mining [5], type resolution [27], pattern mining [11], code generation *e.g.,* SWIM [31], DeepAPI [12], Anycode [13].

11 CONCLUSION

We introduce DeMinify, a Deep-Learning (DL)-based approach that formulates name recovery problem as the predicting the missing features in Graph Convolution Network-Missing Features. The learning of types and names are mutual to support both tasks of name and type recovery. The graph represents both the relations among the variables and those among their types. DeMinify also leverages dual-task learning to propagate the mutual impact between the learning of the variable names and that of their types. Our empirical evaluation on real-world data shows that DeMinify relatively improves from 15.3–40.7% in top-1 accuracy over the existing variable name recovery approaches. It relatively improves 14.5%–51.9% in top-1 accuracy over the existing type prediction approaches. We plan to explore the GCN-Missing Features in other problems in traditional languages, e.g., Java in which the attributes of a node could possess the domain logic, or other properties, e.g., values.

12 DATA AVAILABILITY

Data and code is available in a website [10].

ACKNOWLEDGMENTS

This work was supported in part by the US NSF grant CNS-2120386 and the NSA grant NCAE-C-002-2021.

REFERENCES

[1] 2021. *The NNI autoML tool.* https://github.com/microsoft/nni

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[4] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

[5] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, 207–216.

[6] Claudio Bellei, Hussain Alattas, and Nesrine Kaaniche. 2021. Label-GCN: An Effective Method for Adding Label Propagation to Graph Convolutional Networks. *CoRR* abs/2104.02153 (2021). arXiv:2104.02153 https://arxiv.org/abs/2104.02153

[7] Mihai Christodorescu and Somesh Jha. 2003. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, 12–12. http://dl.acm.org/citation.cfm?id=1251353.1251365

[8] R. Cipolla, Y. Gal, and A. Kendall. 2018. Multi-task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 7482–7491. https://doi.org/10.1109/CVPR.2018.00781

[9] Shiyao Cui, Bowen Yu, Tingwen Liu, Zhenyu Zhang, Xuebin Wang, and Jinqiao Shi. 2020. Edge-Enhanced Graph Convolution Networks for Event Detection with Syntactic Relation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 2329–2339. https://doi.org/10.18653/v1/2020.findings-emnlp.211

[10] DeMinify [n. d.]. DeMinify. https://github.com/variable-name-type-prediction/variable-name-type-prediction/.

[11] Jaroslav M. Fowkes and Charles A. Sutton. 2015. Parameter-Free Probabilistic API Mining at GitHub Scale. *CoRR* abs/1512.05558 (2015). http://arxiv.org/abs/1512.05558

[12] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[13] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-Form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 416–432. https://doi.org/10.1145/2814270.2814295

[14] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[15] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

[16] Vladimir Ivanov, Vitaly Romanov, and Giancarlo Succi. 2021. Predicting Type Annotations for Python using Embeddings from Graph Neural Networks. In *Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,*. INSTICC, SciTePress, 548–556. https://doi.org/10.5220/0010500305480556

[17] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning Type Annotation: Is Big Data Enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. https://doi.org/10.1145/3468264.3473135

[18] Eva Maia, Nelma Moreira, and Rogerio Reis. 2011. A Static Type Inference for Python. In *DYLA 2011*.

[19] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 304–315. https://doi.org/10.1109/ICSE.2019.00045

[20] Amir M. Mir, E. Latoskinas, and G. Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-Based Type Inference. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 585–589. https://doi.org/10.1109/MSR52588.2021.00079

[21] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124

[22] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, 231–245.

[23] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. *CoRR* abs/1409.5718 (2014). http://arxiv.org/abs/1409.5718

[24] Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2020. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. *CoRR* abs/2004.00348 (2020). arXiv:2004.00348 https://arxiv.org/abs/2004.00348

[25] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038

[26] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[27] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 632–642. https://doi.org/10.1145/3180155.3180230

[28] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715

[29] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (oct 2018), 25 pages. https://doi.org/10.1145/3276517

[30] pyminifier [n. d.]. pyminifier. https://github.com/liftoff/pyminifier.

[31] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean - Code Search and Idiomatic Snippet Synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 357–367. https://doi.org/10.1145/2884781.2884808

[32] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[33] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

[34] Hibiki Taguchi, Xin Liu, and Tsuyoshi Murata. 2021. Graph convolutional networks for graphs containing missing features. *Future Generation Computer Systems* 117 (04 2021), 155–168. https://doi.org/10.1016/j.future.2020.11.016

[35] Hieu Tran, Ngoc Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Recovering Variable Names for Minified Code with Usage Contexts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1165–1175. https://doi.org/10.1109/ICSE.2019.00119

[36] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. 2005. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE Computer Society, 45–54.

[37] Uglify [n. d.]. Uglify. https://github.com/mishoo/UglifyJS.

[38] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JS Names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 683–693. https://doi.org/10.1145/3106237.3106289

[39] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference*

on Learning Representations. https://openreview.net/forum?id=Hkx6hANtwH

[40] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 607–618. https://doi.org/10.1145/2950290.2950343

[41] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

Received 2023-02-02; accepted 2023-07-27