5-2023

A study of variable-role-based feature enrichment in neural models of code

Aftab. HUSSAIN

Md. Rafiqul Islam. RABIN

Bowen. XU

David LO
Singapore Management University, davidlo@smu.edu.sg

Mohammad Amin. ALIPOUR

The 1st International Workshop on Interpretability and Robustness in Neural Software Engineering (InteNSE'23),
Co-located with ICSE'23, Melbourne, Australia

A Study of Variable-Role-based Feature Enrichment in Neural Models of Code

Aftab Hussain
ahussain27@uh.edu
University of Houston
Houston, TX, USA

Md Rafiqul Islam Rabin
mrabin@uh.edu
University of Houston
Houston, TX, USA

Bowen Xu
bowenxu@smu.edu.sg
Singapore Management University
Singapore

David Lo
davidlo@smu.edu.sg
Singapore Management University
Singapore

Mohammad Amin Alipour
maalipou@central.uh.edu
University of Houston
Houston, TX, USA

***Abstract*—Although deep neural models substantially reduce the overhead of feature engineering, the features readily available in the inputs might significantly impact training cost and the performance of the models. In this paper, we explore the impact of an unsuperivsed feature enrichment approach based on variable roles on the performance of neural models of code. The notion of variable roles (as introduced in the works of Sajaniemi et al. [1, 2]) has been found to help students' abilities in programming. In this paper, we investigate if this notion would improve the performance of neural models of code. To the best of our knowledge, this is the first work to investigate how Sajaniemi et al.'s concept of variable roles can affect neural models of code. In particular, we enrich a source code dataset by adding the role of individual variables in the dataset programs, and thereby conduct a study on the impact of variable role enrichment in training the Code2Seq model. In addition, we shed light on some challenges and opportunities in feature enrichment for neural code intelligence models.**

***Index Terms*—feature enrichment, variable roles, neural models of code**

I. Introduction

In recent years, there has been significant growth in the research and development of AI models for software engineering tasks (referred to as *code intelligence models* or *neural models of code*) in large software companies like Facebook, Microsoft, and Google; for example, automatic code completion [3, 4], method name prediction [5, 6], bug prediction [7], and even code generation from natural language (e.g. Github Copilot) [8]. A key, costly, and often necessary factor that allows deep neural models in performing these tasks more accurately is scaling up: training massive models on huge datasets [9] for extensive time periods – putting significant stress on computation resources. For instance, it has been estimated that training Lachaux et al.'s [4] final published model with 32 GPUs costs around $30,000, while training the published GPT-3 [10] language model would cost more than $2 million [9]. These tough realities underscore the need for further investigation in approaches that can improve the performance of deep neural models, under more sustainable settings.

One approach that aims to help neural networks (NNs) learn faster and more effectively is *feature enrichment*, which adds extra information in individual inputs, e.g., by explicating the data dependence relations between variables. On the other hand, such techniques are likely to benefit only up to the extent to which the added "explicit" knowledge brings in "new" knowledge to the dataset. For instance, if the added knowledge captures relations in the data that are already being learned by the neural model, the enrichment process is unlikely to bring substantial gains in the NN's performance.

In this work, we evaluate the potential of an unsupervised source code feature enrichment approach for addressing the scalability bottleneck of deep neural models: enriching a given source code dataset with explicit variable role information, where a *role* reflects the manner in which a variable interacts with other variables in a program. The notion of variable roles was introduced in the works of Sajaniemi et al. [1, 2], who classified variables in programs into role categories based on how they are used (e.g., *fixed value*, *temporary variable*, *stepper*, etc.). They applied the role concepts in programming learning tasks and found that roles facilitated the abilities of students to mentally process program information and apply programming steps in writing programs. Inspired by this work, we are interested in how data enrichment with variable role information can impact neural models of code. In particular, we seek to answer the following question: *can explicitly injecting the information about variable roles to the input data help neural models of code achieve better performance and robustness with lesser effort spent on training?*

To pursue this investigation, we (1) build a static analyzer to identify two common kinds of variable roles (*steppers* and *walkers*), and (2) augment the dataset with explicit roles information. We then train a popular code intelligence model, Code2Seq [5], using the role-augmented dataset. Next, we compare our trained model against a pretrained version of Code2Seq, released by the creators of the model, who had pretrained it with the original (unaugmented) dataset. In addition, we evaluate the models' robustness by testing them on noise-induced data (data in which variables in a program are randomly transformed). For both the role augmentation and noise induction steps, we use the semantic-preserving transformation technique [11]. Since we aimed to do an exploratory study, we chose Code2Seq for its ease-of-use.

Contributions. This paper provides the following contributions:

- We present a novel exploratory study to investigate the impact of augmenting variable roles information in a large code dataset on the performance and robustness of Code2Seq.
- We present and implement a technique to automatically detect certain roles of variables in programs, and consequently embed the roles information into variables in the input programs.
- We provide insights on the usefulness of adding signals like roles to code datasets in using deep neural models to perform software engineering tasks.

Paper Organization. The rest of this paper is organized as follows: In Section II, we discuss the methodology of our work, where we provide definitions of the variable roles we used, and present our variable role detection and augmentation approach. In Sections III and IV, we present our experimental design and results, respectively. In Section V, we provide a discussion on the insights we gained from the results, giving some future directions. In Section VI, elaborate on the threats to validity of our work. In Section VII, we present some related literature. We conclude the paper in Section VIII.

II. Methodology

In this section, we discuss our approach. We first, present the definitions of the two specific variable roles, which we studied in this work. Then we discuss how we added roles information to variables in a program in Subsection II-B.

A. Roles Definitions

Towards augmenting variable role information into programs, we focused on two common kinds of variable roles: *steppers* and *walkers*. We define these roles based on the work of Hermans [12], who reintroduced Sajaniemi's cite variable role categories [1, 2]. In contrast to these previous works, we provide more refined definitions that are syntax-oriented and geared towards object-oriented programming languages like Java:

- *Stepper variable*. A stepper variable (or *stepper*) is a for-loop variable of numeric type, which iterates through a list of values via arithmetic-operation based updates, such as, increment operations (e.g., `i++`), or more complex operations (e.g., `size = size/2`). In the following piece of code, `i` is a stepper,

```
for (int i=0 ; i<5; i++){...}
```

- *Walker variable*. A walker variable (or *walker*) can be of two kinds: (1) an iterator object, that enables traversal through containers via APIs, e.g., `iter` in the following piece of code,

```
while (iter.hasNext()){...}
```

or (2) an enhanced for-loop variable, e.g., `elem` in the following,

```
for (String elem: Elements) {...}
```

Unlike steppers, walkers can only be used for sequential access and can be of any type.

B. Role Detection and Role Augmentation

Figure 1 shows the overall workflow of the role detection and augmentation process. We first built a static analyzer, Role Detector, which is based on a popular Java parser library `javalang`[1] written in Python. Using Role Detector we detect steppers and walkers based on the definitions presented in Section II-A from an input program. Then, for implementing role augmentation, we built Role Augmenter, which is based on `JavaMethodTransformer`[2], a semantic-preserving program transformation tool written in Java [11]. Role Augmenter augments roles information obtained from Role Detector into variables in the input program; specifically, all instances of the variable name in the actual program are prefixed with the role type. Here is an example:

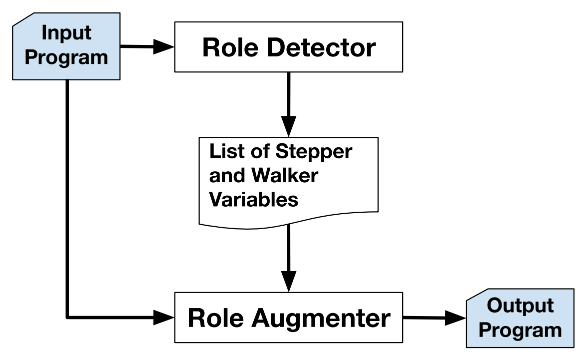

Fig. 1: Role detection and augmentation workflow.

```
for (int count=0 ; count<10; count++){...}
```

Role Detector identifies `count` as a stepper in the above code. Role Augmenter then adds the prefix "stepper_" to all instances of `count`, generating the following,

```
for (int stepper_count=0 ;
stepper_count<10; stepper_count++){...}
```

An alternative approach can be to implement both role detection and augmentation in a single pass using `JavaMethodTransformer`. However, our approach using `javalang` for role detection is easier to implement as it provides an easy-to-read parse tree representation that can be checked for syntactic properties of steppers and walkers. In addition, our two-pass approach allows us to independently test the detection and augmentation phase more cleanly.

Our variable roles augmentation gives potential hints to the model about both the structure and semantics of the code,

[1]https://github.com/c2nes/javalang
[2]https://github.com/mdrafiqulrabin/tnpa-generalizability/

TABLE I: Size of JAVA-LARGE (JL) and JAVA-LARGE-ROLES (JLR) datasets, and number of variables augmented in these datasets. In parentheses, we provide the number of JAVA-LARGE methods augmented with roles information to create the JAVA-LARGE-ROLES dataset.

Dataset	No. of methods in each split			No. of vars. augmented		
	Test	Train	Val	*Steppers*	*Walkers*	Total
JL	370,930	13,376,784	274,604	N/A	N/A	N/A
JLR	370,930 (32,358)	13,376,784 (1,063,868)	274,604 (20,933)	624,701	813,334	1,438,035

where specific tokens are added for variables in certain kinds of program structures (in our case, loops). Code models can be very dependent on variable names (e.g., CodeBERT) [13], while others may have more reliance on the structure of the program [14]. Code2Seq encodes paths in the abstract syntax tree, in addition to encoding tokens into subtokens, and has been found to perform well even for unseen inputs. Our investigation thus throws light on whether Code2Seq can leverage any benefit, if there is useful information in the input tokens.

III. EXPERIMENTAL DESIGN

In this section, we discuss the datasets, models, and the evaluation approach we used in our experiments.

A. Datasets

We deployed our role detection and augmentation techniques on the JAVA-LARGE methods dataset [5] to generate the role-augmented dataset, JAVA-LARGE-ROLES. Each sample in these datasets corresponds to a Java method. Table I shows the number of samples in each split of the datasets, the number of augmented samples in each split of JAVA-LARGE-ROLES (shown in parentheses), and the number of variables augmented as steppers and walkers. (No steppers and walkers were detected in the un-augmented samples.)

B. Models

We study the Code2Seq model trained from scratch using the JAVA-LARGE-ROLES dataset. We compare our trained version with the pretrained version released by the model's creators [5]. We refer to these two versions as Code2Seq-R (Code2Seq-Roles) and Code2Seq-O (Code2Seq-Original). In total, 50 epochs were spent to generate Code2Seq-R (i.e., the best performing model found in terms of F-1 score after 50 epochs of training). The pretrained released version, Code2Seq-O, was obtained after 52 epochs of training.

C. Prediction Task

The prediction task is to predict the method name of a function body, when only provided the function body. This task is widely considered in many prior works, e.g. [6, 15, 16].

D. Model Architecture

Code2Seq [5] uses an encoder-decoder architecture. The encoder encodes paths in the abstract syntax tree (AST) of programs, where each path corresponds to a sequence of nodes in the AST. Code2Seq treats the input function code as a sequence of tokens, which are further split into sub-tokens. The decoder uses attention mechanism [17] to extract features from relevant paths and predicts sub-tokens of a target sequence in order to generate the output, which in our study is the method name.

E. Evaluation Test Sets

We evaluated the models using the eight test sets obtained in the way shown in Figure 2.

The "Original" and "Original, Roles-added" sets directly correspond to the test sets in JAVA-LARGE and JAVA-LARGE-ROLES, respectively. The "Original, Roles-added" test set consists of both augmented and unaugmented methods. The "Original" set consists of all methods in the "Original, Roles-added" set, in un-augmented form. These test sets help us evaluate whether training a model with role augmented data affects its predictions for diverse inputs, with and without the variable roles we considered.

To more accurately gauge the effects of role augmentation on the models, we also tested the models with filtered versions of each of the above test sets, where we only kept augmented methods. Thus, "Original, Roles-added, Filtered" set only consists of augmented methods, and "Original, Filtered" set consists of all methods in "Roles-added, Filtered" set, in un-augmented form.

Finally, we evaluated the models on the face of semantically-transformed test data to examine their robustness; we applied variable name transformations randomly on the JAVA-LARGE and JAVA-LARGE-ROLES test sets using the approach in [11] to obtain transformed test sets. In the transformation, all occurrences of a randomly picked variable in a program was changed to a generic variable name, "varN", where "N" is an integer. This transformation thus strips away any semantic meaning in a variable, and thus is a form of noise induction.
We summarize all the test sets in Table II.

F. Metrics

In order to evaluate the model's predictions, we use the same metrics of precision, recall, and F1-score used at the sub-token level, as in [11]. These metrics are typically used for method name prediction tasks [5, 6].

IV. RESULTS

In this section, we present the results of our experiments, where we seek to answer the following research questions:

RQ.1 How does role augmentation impact Code2Seq's **effectiveness** in making predictions? (Subsection IV-A)

RQ.2 How does role augmentation impact Code2Seq's **robustness** in making predictions? (Subsection IV-B)

TABLE II: Sizes of both unfiltered and filtered JAVA-LARGE (JL), JAVA-LARGE-ROLES (JLR), JAVA-LARGE-TRANSFORMED (JLT), JAVA-LARGE-TRANSFORMED-ROLES (JLTR) test sets. (F) corresponds to the filtered versions.

Type	**Test Set**	**No. of methods**
Untransformed	JL Test	370,930
	JLR Test	370,930
	JL Test (F)	32,358
	JLR Test (F)	32,358
Transformed	JLT Test	916,611
	JLTR Test	916,611
	JLT Test (F)	261,832
	JLTR Test (F)	261,832

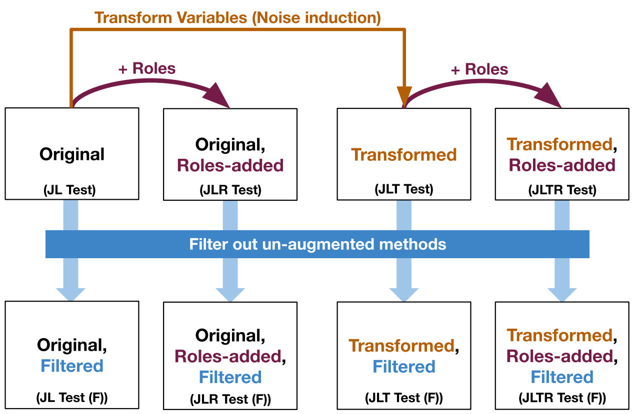

Fig. 2: The approach for obtaining the eight test sets that were used for evaluating the models. The "Original" and "Original, Roles-added" test sets correspond to the test sets in JAVA-LARGE and JAVA-LARGE-ROLES (JL Test and JLR Test), respectively.

A. RQ.1: Impact of Role Augmentation on the Effectiveness of Code2Seq

To evaluate the overall effectiveness of Code2Seq-R and Code2Seq-O, we tested the models with the untransformed test sets, the results of which are shown in Table III (Test type (a)). We observed that Code2Seq-R attained slightly better precision values with all four test sets. On the contrary, the recall values were slightly better for Code2Seq-O compared to Code2Seq-R for three of the untransformed test sets. Overall, however, the F_1-scores were very similar between the two models, for all the test sets.

To gain further insight into the impact of role-based feature enrichment, we wanted to observe the behaviour of Code2Seq when it is trained with a dataset containing a greater share of role-augmented samples. We thus curated the dataset JAVA-VSMALL-ROLES by randomly extracting a subset of only role-augmented samples in JAVA-LARGE-ROLES (In total, 157,046 train set examples and 19,931 validation set examples). Thus 100% of the samples in JAVA-VSMALL-ROLES are role-augmented. In addition, we constructed an equivalent un-augmented version, JAVA-VSMALL, which consists of all samples in JAVA-VSMALL-ROLES in un-augmented form.

The smaller versions of the datasets made it feasible to train with both the augmented and un-augmented datasets for longer epochs. We thus separately trained Code2Seq from scratch on JAVA-VSMALL and JAVA-VSMALL-ROLES for 65 epochs, and saved the best models in each train session based on F_1-score; the saved models are referred to as Code2Seq-O_s and Code2Seq-R_s, respectively.

TABLE III: Precision, recall, and F_1 scores for predictions by the two trained Code2Seq models on the eight test sets obtained from JAVA-LARGE and JAVA-LARGE-ROLES as shown in Figure 2. O and R indicate the values for Code2Seq-O and Code2Seq-R, respectively. The numbers in type (a) and (b) correspond to the results for untransformed and transformed test sets respectively.

Type	**Test Set**	**Precision**		**Recall**		**F_1-Score**	
		O	R	O	R	O	R
(a)	JL	0.636	0.643	0.541	0.536	0.585	0.584
	JL (F)	0.458	0.462	0.375	0.365	0.412	0.408
	JLR	0.636	0.643	0.54	0.536	0.584	0.584
	JLR (F)	0.447	0.462	0.36	0.367	0.399	0.409
(b)	JL-Transformed	0.493	0.497	0.401	0.398	0.442	0.442
	JL-Transformed (F)	0.418	0.42	0.346	0.34	0.379	0.376
	JLR-Transformed	0.493	0.497	0.398	0.398	0.44	0.442
	JLR-Transformed (F)	0.415	0.42	0.336	0.341	0.372	0.377

We evaluated the overall effectiveness of Code2Seq-O_s and Code2Seq-R_s using the same test sets used for evaluating Code2Seq-O and Code2Seq-R. The results are shown in Table IV (Test type (a)). We observed very similar marginal differences for Code2Seq-O_s and Code2Seq-R_s, as we did for Code2Seq-O and Code2Seq-R. In precision, Code2Seq-R_s was slightly better with all four test sets. However, overall, the F_1-scores were very similar between the two models, for all the test sets.

Observation. *Overall, the experiment results demonstrate that the impact of role augmentation is* ***indistinguishable*** *with respect to Code2Seq's performance in making method name predictions. In other words, role augmentation neither boosted nor harmed the performance significantly.*

B. RQ.2: Impact of Role Augmentation on the Robustness of Code2Seq

To investigate the impact of role augmentation on the robustness of Code2Seq, we tested Code2Seq-O and Code2Seq-R on transformed data, the results of which are summarized in Table III (Test type (b)). We see the F_1-scores values of both Code2Seq-O and Code2Seq-R dropped for the transformed test sets, with the difference between them being marginal, just as was seen with the untransformed tests. Overall, Code2Seq-O and Code2Seq-R exhibited almost the same degrees of

TABLE IV: Precision, recall, and F_1 scores for predictions by the two trained Code2Seq models on the eight test sets obtained from JAVA-VSMALL and JAVA-VSMALL-ROLES as shown in Figure 2. O_S and R_S indicate the values for Code2Seq-O_S and Code2Seq-R_S, respectively. The numbers in type (a) and (b) correspond to the results for untransformed and transformed test sets respectively.

Type	Test Set	Precision		Recall		F_1-Score	
		O_S	R_S	O_S	R_S	O_S	R_S
(a)	JL	0.465	0.467	0.307	0.306	0.37	0.37
	JL (F)	0.356	0.358	0.256	0.255	0.298	0.298
	JLR	0.465	0.466	0.306	0.306	0.369	0.37
	JLR (F)	0.355	0.356	0.256	0.258	0.297	0.299
(b)	JL-Transformed	0.362	0.361	0.239	0.241	0.288	0.289
	JL-Transformed (F)	0.337	0.335	0.24	0.236	0.281	0.277
	JLR-Transformed	0.362	0.36	0.239	0.241	0.288	0.289
	JLR-Transformed (F)	0.336	0.333	0.239	0.238	0.28	0.278

robustness (produced almost the same performance) with the transformed test sets.

Similar to the approach for evaluating the effectiveness of Code2Seq, we also tested the Code2Seq-O_s and Code2Seq-R_s models (refer Subsection IV-A), this time with the transformed test sets. A similar trend of marginal differences between the scores for Code2Seq-O_s and Code2Seq-R_s were also observed for these noise-induced test sets.

Observation. *Overall, the experimental findings show that the impact of role augmentation is* ***indistinguishable*** *with respect to Code2Seq's robustness in making method name predictions. In other words, role augmentation neither boosted nor harmed the robustness of Code2Seq significantly.*

V. DISCUSSION

The results of our experiments are negative. Our feature-enrichment scheme did not significantly impact Code2Seq. One reason for this observation could be that since variable roles are determined from the way variables are syntactically used, Code2Seq may already be capable of capturing the surrounding structural context of a certain variable role, and this may explain the largely similar performances seen by Code2Seq-O and Code2Seq-R. However, more investigations may be necessary to determine how much benefit does explicitly inserting the role in a variable name add to the predictions of the models, in cases where the role is less apparent from just the structure of the code (e.g., *fixed-value* variables, i.e., constants, may be used in significantly different ways in code). *Thus, towards investigating the impact of variable roles in neural program models that capture code structure, there is a need to distinguish between variables with roles that are more strictly reliant on code structure, e.g., steppers, and those that can be more flexibly used, despite playing the same role, e.g., fixed-value (constant) variables.*

Another direction that needs further investigation is towards understanding how capable are the models in exploiting enriched features, as it is "possible" that the models may have an intrinsic incapability in exploiting enriched features in datasets. Separate investigations can be carried out in this direction for models that are more reliant on code semantics (e.g., large language-model-based deep neural networks like CodeBERT [18] and CodeGPT [19]).

Finally, one more reason for the similarity in performances of Code2Seq-O and Code2Seq-R could be the fact that only two variable roles were augmented, an aspect of our experiment we discuss in more detail in the next section.

VI. THREATS TO VALIDITY

Although we considered two common types of variable roles as our first attempt, still, they only covered 8% of the samples of the whole dataset (1,117,159 of the approximately 14 million methods). In the future, we plan to extend this work by considering more variable roles to further mitigate this threat.

Furthermore, the stepper variable names are not as diverse. Most steppers were seen to be the variables `i` and `j`: of 624,701 steppers detected in the entire JAVA-LARGE dataset, 475,383 (76.1%) were `i`, and 47,221 (7.56%) were `j`. Hence, adding the "stepper" keyword may not have had a significant effect in terms of adding additional information towards learning in the experiments. For future explorations, more of Sajaniemi's roles [1] could be added, which may require more complex static analysis techniques (e.g., data-flow analysis for detecting *most-wanted-holder* and *gatherer* variables [2].).

VII. RELATED WORKS

In this section, we discuss two branches of works that have also focused on improving the performance of neural models of code:

Code Modeling. There are numerous works that have tried to use different code representations in developing better performing neural program models. Some early works used natural language processing models to capture textual patterns in code, without capturing structural information [20, 21]. Better performances were achieved by approaches that leveraged tree- and graph-forms of source code to better grasp code structure [5, 6, 22, 23, 24]. E.g., GraphCodeBERT has shown that representing variables with a static data flow graph and augmenting the graph with original source code can improve the transformer-based model's comprehension of code. [24].

In [25], the author developed dynamic embeddings, a recurrent mechanism that adjusts the learned semantics of the variable when it obtains more information about the variable's role in the program. They show that using the proposed dynamic embeddings significantly improves the performance of the recurrent neural network, in code completion and bug fixing tasks. In contrast to these techniques, our approach directly adds variable semantic information, as derived from Sajaniemi's pedagogical notion of variable roles [1, 2], into the raw training source code dataset without adding a new representation of the source code (we only renamed the variables).

Feature Enrichment. Allamanis et al. [26] showed that adding features that capture global context can increase the performance of a model. Rabin et al. [27] found that code complexity features can improve the classification performance of some labels up to about 7%. While this work focused on extracting a set of handcrafted features for better transparency, we study how feature enrichment affects in model's training behavior. Recent studies have shown that state-of-the-art models heavily rely on variables [13, 28], specific tokens [29], and even structures [30]. Chen et al. [31] focus on semantic representations of program variables, and study how well models can learn similarity between variables that have similar meaning (e.g., `minimum` and `minimal`). Ding et al. [32] explore the problem of learning functional similarities (and dissimilarities) between codes, towards which they rename variables to inject variable-misuse bugs in order to generate buggy programs that are structurally similar to benign ones. Neither of these works investigated or deployed variable-role based augmentation, as was done in this work.

VIII. Conclusion

In this paper, we investigated the impact of explicitly adding variable role information in code datasets on the performance of Code2Seq. To the best of our knowledge, this is the first work to evaluate the impact of Sajaniemi et al.'s notion of variable roles, a concept that was found to help students learn programming, to neural models of code. The work presents guidelines and challenges on enriching source code datasets for using code intelligence models more productively, encouraging the development of a systematic framework to investigate how to provide such models meaningful information to enable them to learn faster and perform better.

Acknowledgements

We would like to thank our paper's reviewers from the InteNSE 2023 Program Committee for their valuable feedback towards the revision of this paper.

References

[1] P. Byckling and J. Sajaniemi, "Roles of variables and programming skills improvement," in *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '06, 2006, p. 413–417.

[2] J. Sajaniemi, "The roles of variables home page," http://saja.kapsi.fi/var_roles/, 2008, [Online; accessed 11-May-2022].

[3] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019.

[4] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[5] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

[7] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[8] GitHub and OpenAI, "GitHub Copilot: Your AI pair programmer," https://copilot.github.com/, 2021, [Online; accessed 10-May-2022].

[9] V. J. Hellendoorn and A. A. Sawant, "The growing cost of deep learning for source code," *Commun. ACM*, vol. 65, no. 1, p. 31–33, 2021.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[11] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, p. 106552, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000379

[12] F. Hermans, *The Programmer's Brain - What Every Programmer Needs to Know about Cognition*. Manning, 2021.

[13] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1443–1455. [Online]. Available: https://doi.org/10.1145/3510003.3510049

[14] S. Gao, C. Gao, C. Wang, J. Sun, D. Lo, and Y. Yu, "Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension," 2022.

[15] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020.

[16] N. D. Q. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting

subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1186–1197.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[19] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[20] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

[21] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 2016.

[22] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, 2014, pp. 649–657.

[23] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 1287–1293.

[24] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[25] N. Chirkova, "On the embeddings of variables in recurrent neural networks for source code," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2679–2689. [Online]. Available: https://aclanthology.org/2021.naacl-main.213

[26] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[27] M. R. I. Rabin, A. Mukherjee, O. Gnawali, and M. A. Alipour, "Towards demystifying dimensions of source code embeddings," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*, ser. RL+SE&PL 2020, 2020, p. 29–38. [Online]. Available: https://dl.acm.org/doi/10.1145/3416506.3423580

[28] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *17th International Conference on Mining Software Repositories (MSR) 2020, Seoul, Republic of Korea.*, 2020.

[29] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "Understanding neural code intelligence through program simplification," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 441–452. [Online]. Available: https://doi.org/10.1145/3468264.3468539

[30] M. R. I. Rabin and M. A. Alipour, "Code2snapshot: Using code snapshots for learning representations of source code," 2021.

[31] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2327–2339. [Online]. Available: https://doi.org/10.1145/3510003.3510162

[32] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Towards learning (dis)-similarity of source code from program contrasts," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 6300–6312. [Online]. Available: https://aclanthology.org/2022.acl-long.436