A Simple Approach for Handling Out-of-Vocabulary Identifiers in Deep Learning for Source Code

Nadezhda Chirkova[*] and **Sergey Troshin**[*]
HSE University
Moscow, Russia
`{nchirkova, stroshin}@hse.ru`

Abstract

There is an emerging interest in the application of natural language processing models to source code processing tasks. One of the major problems in applying deep learning to software engineering is that source code often contains a lot of rare identifiers, resulting in huge vocabularies. We propose a simple, yet effective method, based on identifier anonymization, to handle out-of-vocabulary (OOV) identifiers. Our method can be treated as a preprocessing step and, therefore, allows for easy implementation. We show that the proposed OOV anonymization method significantly improves the performance of the Transformer in two code processing tasks: code completion and bug fixing.

1 Introduction

Natural language processing (NLP) is widely used for source code processing (SCP), e. g. for learning the meaningful vector representations of code (Feng et al., 2020; Alon et al., 2019b; Azcona et al., 2019), that can be used in various downstream tasks, e. g. code summarization (Iyer et al., 2016; Shiv and Quirk, 2019), code completion (Kim et al., 2020), or bug fixing (Hellendoorn et al., 2020).

An important question, one should answer before building an SCP model, is how to create a vocabulary? Karampatsis et al. (2020) underline that modern source code datasets may incorporate millions of unique identifiers, of which less than 1% occur in the dataset frequently, e. g. more than 5 times. The common practice is to crop the vocabulary based on top-N identifiers and replace all occurrences of out-of-vocabulary (OOV) identifiers with an `UNK` identifier to avoid huge embedding matrices and the meaningless embeddings of rare tokens. But can one process rare identifiers in a better way?

[*] The work was done while working at Samsung-HSE Laboratory, HSE University. Both authors contributed equally.

Vocabulary: {`np, sin` }
Input: `my_y = np.sin(my_x) + my_x`
Standard OOV processing procedure:
`UNK = np.sin(UNK) + UNK`
Proposed OOV anonymization procedure:
`VAR1 = np.sin(VAR2) + VAR2`

Figure 1: Illustration of the proposed OOV anonymization procedure. Out-of-vocabulary identifiers `my_y` and `my_x` are replaced with anonymized identifiers `VAR1` and `VAR2`, while in-vocabulary identifiers `np` and `sin` preserve their names.

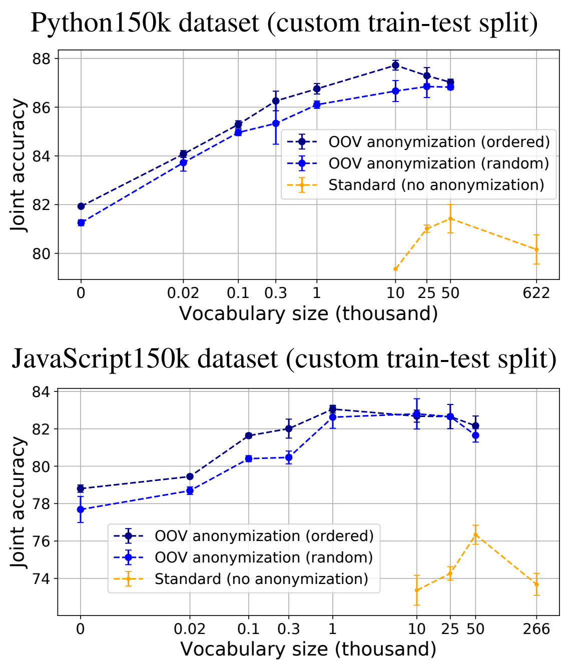


Figure 2: Results for Transformer in the variable misuse task: joint bug localization and repair accuracy, mean value ± standard deviation (over 3 runs). Models with the proposed OOV anonymization significantly outperform the standard model (all OOV identifiers are replaced with an `UNK` token). The numerical data for the plots is given in Table 2 in Appendix.

There are two main directions in the NLP literature to tackle rare tokens: open vocabulary and copy-based approaches. An open vocabulary solution implies splitting rare tokens into subtokens (Sennrich et al., 2016). The copy-based approaches are used in generation tasks and imply using the pointer mechanism (Gulcehre et al., 2016) to copy tokens from the input sequence.

We propose a new, simple, yet effective approach for processing OOV identifiers in source code, namely OOV anonymization. Anonymization implies replacing *rare* identifiers with unique placeholders, i. e. `VAR1`, `VAR2`, `VAR3` etc., while preserving the names of *frequent* identifiers. An example of OOV anonymization is shown in figure 1. The intuition behind using anonymization is that it preserves the semantics of the algorithm that the code snippet implements, i. e. renaming user-defined identifiers does not change the underlying algorithm. By contrast, replacing all rare identifiers with an `UNK` identifier changes the algorithm. We underline that we propose anonymizing only rare identifiers, because frequently used identifier names may serve as an additional source of information, and neural networks are indeed capable of capturing this information.

The proposed OOV anonymization strategy allows for easy implementation as a preprocessing step, thus no modification of model code is required. Another advantage of the OOV anonymization is that it enhances both the encoder and the decoder. The proposed approach significantly outperforms the model with all rare identifiers being replaced with an `UNK`, in code completion and bug fixing tasks, with the Transformer (Vaswani et al., 2017) architecture being used (see example comparison in Fig. 2). Our code and data split are available at `https://github.com/bayesgroup/code_transformers`.

2 Related Work

Handling OOV identifiers in source code. Code processing often borrows ideas from NLP. Source code can be represented as a sequence of identifiers. In this case, identifiers can be further split into subtokens using byte-pair encoding (BPE) (Karampatsis et al., 2020; Sennrich et al., 2016) resulting in an open vocabulary model. This approach has several drawbacks. Firstly, splitting identifiers into subtokens increases the length of the sequence several times. This substantially slows down inference, e. g. vanilla Transformer's forward pass has a complexity quadratic w. r. t. the input length. Secondly, splitting breaks one-to-one alignment between identifiers and nodes in the parsing tree, e. g. abstract syntax tree (AST), in other words, *several* subtokens correspond to *one* node in the AST, which makes it harder to apply structure-aware models such as (Hellendoorn et al., 2020) or (Alon et al., 2019a). To the best of our knowledge, all SCP works, proposing structure-aware models, either use entire tokens without subtokenization / BPE, or average the embeddings over subtokens (this strategy provides only a slight quality improvement compared to the first one), and the question of how to incorporate BPE in structure-aware models needs further investigation. Taking into account the described disadvantages of BPE, we do not consider BPE in this work and do not split tokens into subtokens.

An orthogonal direction for handling OOV identifiers in source code is the modification of the computational graph. For the task of code generation, the pointer mechanism is widely adapted (Li et al., 2018). Cvitkovic et al. (2019) also propose a graph-structured cache for inferring the representations of the rare identifiers in source code. The major drawback of the mentioned approaches is that they are quite hard to implement.

Identifier anonymization in source code. Chirkova and Troshin (2020) conduct an empirical study of Transformers for source code in a setting with *all* identifiers being anonymized and show that Transformers can make meaningful predictions in this setting. By contrast, we propose anonymizing only OOV identifiers and show that it boosts the performance of the model in the setting with frequent identifier names being present in the data. The anonymization of *all* identifiers has also been used in (Gupta et al., 2017) and (Xu et al., 2019) for training recurrent neural networks. Ahmed et al. (2018) replace variables with their types, losing information about identifier repetition.

3 Proposed method

Consider a vocabulary of all identifiers in the training data. It could be a vocabulary of all tokens if we treat input code snippets as text sequences, or a vocabulary of all user-defined variables if we parse the ASTs of code snippets. Let us now select the vocabulary V_{full} of frequent identifiers and call all others OOV identifiers.

We propose an elegant way of tackling OOV identifiers based on anonymization. Particularly, we propose replacing all OOV identifiers with placeholders `VAR1`, `VAR2`, `VAR3` etc. All occurrences of one identifier in one input sequence are replaced with the same placeholder (anonymized identifier), but different identifiers are replaced with different placeholders. One identifier may be replaced with different placeholders in different input sequences. An example of OOV anonymization is presented in figure 1.

We consider two strategies for the OOV anonymization, namely *ordered* anonymization and *randomized* anonymization. The *ordered* anonymization implies assigning an anonymized identifier `VAR1` to the first seen rare identifier, `VAR2` to the next seen rare identifier, etc. For example, the snippet from Fig. 1 is transformed into `VAR1 = np.sin(VAR2) + VAR2`. The *randomized* anonymization implies fixing the placeholder vocabulary size $|V_{an}|$ and selecting a random subset of anonymized placeholders `VAR1`, ..., $\texttt{VAR}|V_{an}|$ for each code snippet. For example, the snippet from Fig. 1 can be transformed into `VAR38 = np.sin(VAR801) + VAR801`. To ensure that we can always encode identifiers in a code snippet injectively, the size $|V_{an}|$ of the placeholder vocabulary should not be fewer than the maximum possible number of tokens per snippet. We set $|V_{an}|$ to the maximum length of code snippets.

The proposed OOV anonymization can be seen as a preprocessing step, thus no model parts change. In the encoder, the embedding matrix contains embeddings for both anonymized and in-vocabulary identifiers: $\{e_v\}_{v \in V_{full}} \cup \{e_{\texttt{VARi}}\}_{i=1}^{|V_{an}|}$. In the decoder, when generating the next identifier, the softmax is computed over all anonymized and in-vocabulary identifiers. We note that the *ordered* OOV anonymization may need a more careful implementation, e. g. of metric computation, see details in section 4.

4 Experiments

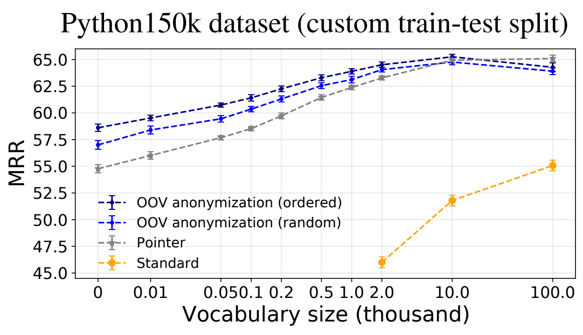
4.1 Experimental setup

We conduct experiments with Transformer (Vaswani et al., 2017) on the code completion (CC) and variable misuse (VM) tasks, on Python150k (Raychev et al., 2016a) (the redistributable version of (Kanade et al., 2020)) and JavaScript150k (Raychev et al., 2016b) datasets.

We use the problem setup, metrics and loss of Hellendoorn et al. (2020) for the VM task, and of Kim et al. (2020) for the CC task. To validate our implementation, we check that the quality we obtain with the vanilla Transformer is the same as the quality of this model reported in the corresponding works, see details in Appendix B. As a base model, we use the 6-layer Transformer equipped with the relative attention mechanism (Shaw et al., 2018) and applied over the depth-first traversal of the AST. Chirkova and Troshin (2020) show that such an approach leads to high performance and outperforms the vanilla Transformer and several techniques for capturing AST structure in Transformer. The hyperparameters are given in Appendix A. Allamanis (2019); Chirkova and Troshin (2020) emphasize the importance of the thoughtful splitting data into training and testing parts, which includes *splitting by repositories* and *removing duplicate code*. We follow the same strategy in our experiments (later referred to as custom train-test split).

Variable misuse task. For the VM task, we use the same setup as in (Hellendoorn et al., 2020), below we briefly recap this setup. In the VM task, given the code of a function, the task is to output two positions (using two pointers): in what position a wrong variable is used and which position a correct variable can be copied from (any such position is accepted). If a snippet is non-buggy, the first pointer should select a special no-bug position. We obtain two pointers by applying two position-wise fully-connected layers, and softmax over positions on top of the Transformer encoder outputs. We use the joint accuracy to assess the model quality (the portion of buggy examples for which the model correctly localizes and repairs the bug).

To obtain a dataset for the VM task, we select all top-level functions in Python150k dataset, including functions inside classes, and filter out functions longer than 250 AST nodes, and functions with less than three positions containing user-defined variables or less than three distinct user-defined variables. The resulting training / testing set consists of 417K / 231K functions (Python) and 202K / 108K functions (JavaScript). One function may occur in the dataset up to 6 times, 3 times with synthetically generated bug and 3 times without bug. The buggy examples are generated synthetically by choosing random bug and fix positions from positions containing user-defined variables. When using the ordered OOV anonymization, we firstly inject a synthetic bug and then perform anonymization, to



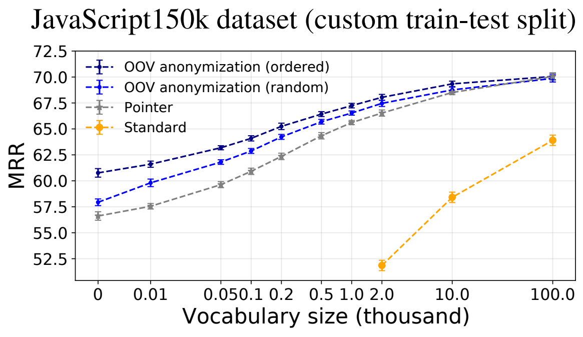


Figure 3: Results for Transformer in the code completion task (value prediction): mean reciprocal rank ± standard deviation over 3 runs. The numerical data for the plots is given in Table 3 in Appendix.

avoid data leak.

Code completion task. For the CC task, we use the setup of Kim et al. (2020), and below we briefly review it. The CC task implies predicting the type and value of the next node based on the prefix of the depth-first AST traversal. We predict the next type and value using two fully-connected heads on top of the Transformer decoder and optimize the sum of cross-entropy losses for types and values. While computing the loss, we skip the first occurrences of anonymized values and special positions, i. e. `UNK` and `PAD`. We tie the embeddings of input and output layers. In this task, we split the large AST traversals into chunks with a maximum length of 500, as described in (Kim et al., 2020). The resulting dataset includes 186K / 100K training/testing chunks for Python and 270K / 220K for JavaScript.

We use mean reciprocal rank (MRR) to measure the quality of the model: $\mathrm{MRR} = 100\% \cdot N^{-1}\sum_{i=1}^{N} 1/\mathrm{rank}_i$, where N is the total number of tokens in the dataset and rank_i is a position of the true token in the model ranking. We assign zero scores (a) if the correct token is not in the top 10 predicted tokens, (b) if the correct token is a `UNK` and (c) for the first occurrences of anonymized identifiers.

For the next value prediction task, we add the pointer mechanism to the Transformer for comparison. We re-implement the pointer mechanism following the design choice of (Deaton, 2019). Given an input sequence $[x_1, \dots, x_\ell]$ of length ℓ, Transformer outputs two distributions: the distribution over the fixed vocabulary V, $p_{\mathrm{model}}(a), a \in V$, and the probability of copying an input from position j, $p_{\mathrm{copy}}(j), j = 1, \dots, \ell$. Then both distributions are combined to obtain the final distribution over the extended vocabulary: $p(x_{\ell+1} = a) = p_{\mathrm{gen}} p_{\mathrm{model}}(a)[a \in V] + (1 - p_{\mathrm{gen}}) \sum_{j=1}^{\ell} p_{\mathrm{copy}}(j)[x_j = a]$. The switcher is computed given the current input and the output of the decoder as $p_{\mathrm{gen}}(x_\ell, h_\ell) = \sigma(w_h^T h_\ell + w_i^T x_\ell + b_{\mathrm{gen}})$. The cross entropy loss is computed over the extended vocabulary.

4.2 Results

We compare the proposed anonymization of OOV identifiers with the following baseline approaches: (1) Standard: with all OOV identifiers being replaced with an `UNK` identifier; (2) training on fully anonymized data, i. e. all identifiers are anonymized. This baseline corresponds to the zero vocabulary size in all plots. For the code completion task, we also include the baseline with the pointer mechanism.

Figure 2 presents the results for the variable misuse task, for different frequent identifier vocabulary sizes. We observe that the proposed approach, with the anonymization of OOV identifiers (dark blue and blue lines), performs substantially better than the baseline models, particularly than the standard approach with OOV identifiers being replaced with an `UNK` identifier (orange line). The leftmost point in both blue lines corresponds to the full anonymization baseline (zero vocabulary size). The ordered OOV anonymization (dark blue line) performs slightly better or similarly to the randomized OOV anonymization (blue line). We also experimented with the frequency-based OOV anonymization, i. e. sorting rare identifiers by frequencies in the code snippet and assigning `VAR1` to the most frequent one, `VAR2` to the next one etc. We found that such a strategy achieves the same quality as the ordered anonymization.

Increasing the vocabulary size for the standard model does not help much and even hurts the performance, i. e. the standard model with a vocabulary of 50K identifiers outperforms the one with the largest possible vocabulary. The reason is that the embeddings of rare identifiers are updated only several times during the training and do not change a lot after being initialized randomly. On the contrast, anonymized identifiers occur quite frequently in the data, e. g. thousands of times, so their embeddings are updated regularly. As a result, it is more beneficial to anonymize rare identifiers than to include them in vocabulary.

The intuition behind why OOV anonymization performs well is that it saves information about variable repetition and thus does not change the algorithm that the code snippet implements. For example, in the buggy snippet `with open(myfnm) as myfp: data = myfnm.read()` (should be `myfp.read()`), the model with OOV anonymization detects that OOV variables after `as` and before `read` are different and correctly predicts the bug, while the model with OOVs replaced with `UNK` does not distinguish variables `myfnm` and `myfp` and cannot detect the bug.

Figure 3 presents the results for code completion (value prediction), for different frequent identifier vocabulary sizes. In this task, the ordered OOV anonymization again slightly outperforms the randomized OOV anonymization, and they both substantially outperform the standard baseline and the baseline with full anonymization. Moreover, the proposed OOV anonymization surpasses the strong pointer baseline for almost all vocabulary sizes. The advantage of the proposed OOV anonymization approach is that it helps the Transformer to distinguish OOV identifiers in the input, while the pointer mechanism enhances only the output layer. Also, in contrast to the pointer mechanism, OOV anonymization is much easier to implement. The pointer mechanism and the OOV anonymization could be straightforwardly combined, however, in our experiments, this combination did not increase the scores compared to the maximum score of the OOV anonymization and the pointer. The results for type prediction are relatively the same as for the value prediction and can be found in Appendix C. We visualize the t-SNE representations of the learned embeddings in Appendix D.

4.3 The influence of the anonymized vocabulary size

The randomized OOV anonymization strategy comprises the hyperparameter $|V_{an}|$, i. e. the size of the anonymized vocabulary. It should not be less than the maximum sequence length, to avoid using the same placeholder for different identifiers, and we select $|V_{an}|$ as the maximum length of code snippets. We tried using the larger values of $|V_{an}|$ and observed the insignificant difference in quality in the variable misuse task, and a slight drop in quality in the code completion task, as shown in Table 1.

	PY		**JS**	
$\|V_{full}\|$:	1k	10k	1k	10k
$\|V_{an}\| = 0.5$K	63.35	64.77	66.63	68.98
$\|V_{an}\| = 1$K	63.03	64.63	66.52	68.75
$\|V_{an}\| = 3$K	62.79	64.34	66.22	68.60

Table 1: Increasing the size $|V_{an}|$ of the anonymized vocabulary for two frequent identifier vocabulary sizes V_{full}, namely 1k and 10k, in the code completion task (value prediction). Metric: MRR (%), all standard deviations are less than 0.3%.

5 Conclusion

In this work, we propose the effective anonymization-based encoding of out-of-vocabulary identifiers, with two options, namely ordered and randomized OOV anonymization. Our preprocessing technique allows for easy implementation, could be easily plugged into various Transformer models and outperforms the widely used standard approach by a significant margin. The ordered anonymization performs slightly better than the randomized anonymization but requires a more careful implementation.

Acknowledgments

We would like to thank Ivan Rubachev and the anonymous reviewers for the valuable feedback. The results on the variable misuse task were supported by the Russian Science Foundation grant №19-71-30020. The results on the code completion task were supported by Samsung Research, Samsung Electronics. The research was supported in part through the computational resources of HPC facilities at NRU HSE.

References

Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: For the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, page 78–87, New York, NY, USA. Association for Computing Machinery.

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Uri Alon, Meital Zilberstein, Omer Levy, and E. Yahav. 2019b. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3:1 – 29.

David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. User2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, LAK19, page 86–95, New York, NY, USA. Association for Computing Machinery.

Nadezhda Chirkova and Sergey Troshin. 2020. Empirical study of transformers for source code.

Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 1475–1485. PMLR.

Jon Deaton. 2019. Transformers and pointer-generator networks for abstractive summarization.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. 2016. Pointing the unknown words. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 140–149, Berlin, Germany. Association for Computational Linguistics.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1345–1351. AAAI Press.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations, ICLR 2020*.

Srini Iyer, Ioannis Konstas, A. Cheung, and L. Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, C. Sutton, and A. Janes. 2020. Big code != big vocabulary: open-vocabulary models for source code. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code prediction by feeding trees to transformers.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 4159–25. AAAI Press.

I. Loshchilov and F. Hutter. 2017. Sgdr: Stochastic gradient descent with warm restarts. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016a. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 731–747, New York, NY, USA. Association for Computing Machinery.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016b. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 761–774, New York, NY, USA. Association for Computing Machinery.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 12081–12091. Curran Associates, Inc.

L.J.P. van der Maaten and G.E. Hinton. 2008. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research 9:2579-2605*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3975–3981. International Joint Conferences on Artificial Intelligence Organization.

A Implementation details

Passing ASTs to Transformer. To pass an AST to the Transformer, we follow the strategy of Chirkova and Troshin (2020). They converted each input code snippet to the depth-first traversal of the abstract syntax tree (AST), obtaining a sequence of pairs (node type, node value). The node types denote syntactic units of the programming language, e. g. `If` or `For`, and come from a small dictionary (up to 350 types), while the node values denote user-defined identifiers, language-specific identifiers, e. g. `None` in Python, and constants. Some nodes do not store any values, we insert `<EMPTY>` values in these nodes. We store two embedding layers, one for types and one for values, and sum the embedding of type and value in each AST node. The OOV anonymization is applied to the values. To train a model on the fully anonymized data, we anonymize all values except `<EMPTY>`.

Hyperparameters. We list hyperparameters for the VM / CC tasks using slashes. Our Transformer model has 6 layers, 8 / 8 heads, d_{model} equals to 512 / 512. The number of parameters of our models (excluding embeddings) is 19M / 18M. We train all Transformers using Adam with a starting learning rate of 0.00001 / 0.001 and the batch size of 32 for 20 epochs (CC), 25 epochs (VM PY), or 40 epochs (VM JS). In the CC task, we use cosine learning rate schedule (Loshchilov and Hutter, 2017) with a warmup step of 2000 and zero minimal learning rate, and the gradient clipping of 0.2. In the VM task, we use a constant learning rate. We use residual, embedding and attention dropout with $p = 0.2 / 0.1$. We use relative attention (Shaw et al., 2018) with the maximum distance between elements of 8 / 32.

B Validating our implementation

The numbers reported in our paper are not directly comparable to the works we borrow setups from, because we use our custom (and more correct) data split rather than the commonly used split (see details in Sec. 4). We ensure the validity of our results in two ways: by relying on the code of recently published works, and by comparing our numbers achieved for the commonly used data split to the numbers in the corresponding papers. Particularly, we use the model / loss / metrics and the code of (Kim et al., 2020) for the CC task, and the model / loss / metrics of (Hellendoorn et al., 2020) for the VM task (we rewrite line-by-line their code for metrics and loss). For the vanilla Transformer in the VM task, Hellendoorn et al. (2020) report 67.7% joint accuracy and we achieved 64.4% with the similar model size. The results are close to each other. For the vanilla Transformer in the CC task (Python), for value / type prediction, (Kim et al., 2020) report 58.0 / 87.3 MRR ("TravTrans" model), and we achieve 60.0 / 89.1 MRR, again the results are close.

C Experiments with type prediction

In Fig. 4, we report the results for type prediction, code completion task. Overall, the anonymization of rare identifiers again performs better, compared to the standard Transformer with rare identifiers replaced with an `UNK`, and also improves over the pointer baseline for almost all vocabulary sizes.

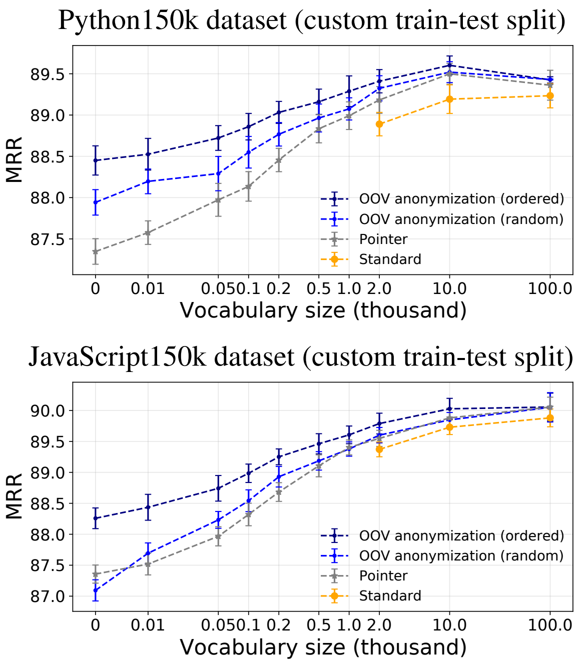


Figure 4: Type prediction for code completion task for Transformer: mean reciprocal rank ± standard deviation over 3 runs.

D Visualization of embeddings

In this section, we visualize the embeddings learned in the code completion task on Python dataset, vocabulary size 1k, OOV identifiers are anonymized randomly. We use t-SNE (van der Maaten and Hinton, 2008) with default hyperparameters and cosine metric to visualize the embeddings in a 2-dimensional space, see Figure 5. We observe that the embeddings of anonymized identifiers form a well-separated cluster in the embedding space. We also measured the inter-cluster / intra-cluster cosine similarities: the average cosine similarity between pairs of embeddings in one cluster / two different clusters. We observe the inter-cluster similarity for in-vocabulary / OOV identifiers of 0.09 / 0.05, and the intra-cluster similarity between in-vocabulary / OOV clusters of -0.06, which shows that these two clusters occupy different subspaces of the embedding space.

E Numerical data for the plots in the paper

Table 2 lists the numerical data for Figure 2 and Table 3 lists the numerical data for Figure 3.

For the code completion task, we also report accuracy scores of the best performing models for values prediction. We mark `UNK` prediction as wrong. Our random / ordered / pointer / baseline models achieve 59.31 / 59.71 / 58.88 / 50.41 accuracy (%) on the Python dataset, and 64.08 / 64.13 / 63.58 / 58.48 on the JavaScript dataset.

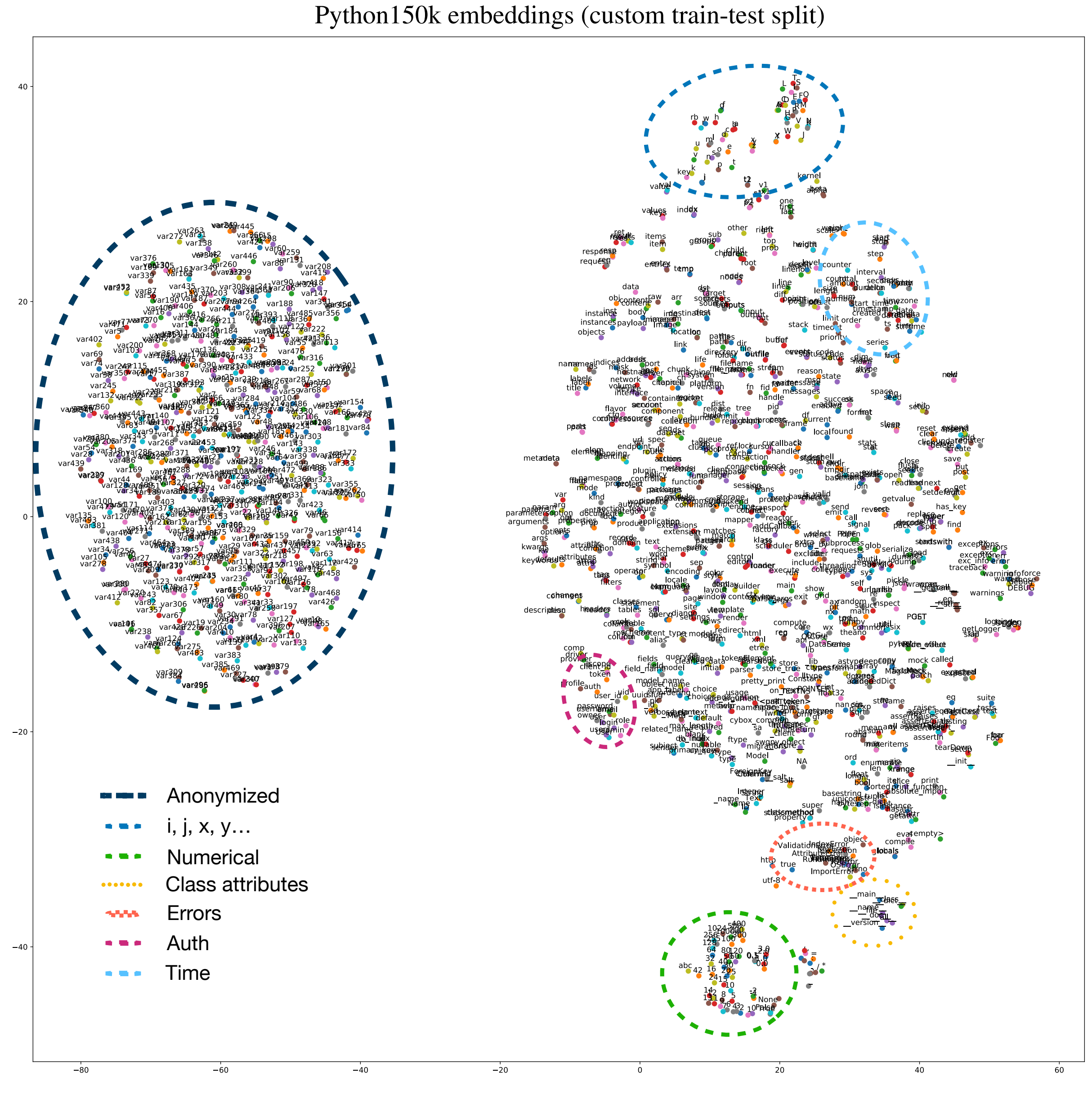


Figure 5: t-SNE visualization of Python150k embeddings. Anonymized (random) embeddings (left cluster), in-vocabulary embeddings (right cluster).

	Variable Misuse task, Joint localization and repair accuracy (%)					
	PY			JS		
$\|V_{full}\|$	Ordered	Random	Standard	Ordered	Random	Standard
1	81.93	81.26	0.00	78.80	77.69	0.00
20	84.08	83.72	40.28	79.45	78.69	6.89
100	85.29	84.96	54.71	81.63	80.40	33.66
300	86.26	85.33	64.72	82.01	80.47	50.61
1000	86.76	86.10	72.36	83.05	82.62	62.87
10000	87.72	86.66	79.36	82.68	82.80	73.36
25000	87.29	86.85	81.01	82.66	82.65	74.27
50000	87.02	86.82	81.42	82.16	82.62	76.33
Max	N/A	N/A	80.16	N/A	N/A	73.68

Table 2: Numerical data for Figure 2. Max denotes the vocabulary without any filtering: 622K for PY and 266K for JS. The standard deviations for all models are approx. 0.5%.

	Code Completion task, Values Prediction, MRR (%)							
	PY				JS			
$\|V_{full}\|$	Random	Ordered	Pointer	Standard	Random	Ordered	Pointer	Standard
1	57.00	58.61	54.77	N\A	57.93	60.77	56.61	N\A
10	58.40	59.52	56.01	N\A	59.81	61.60	57.55	N\A
50	59.45	60.74	57.68	N\A	61.81	63.19	59.63	N\A
100	60.35	61.41	58.53	N\A	62.88	64.09	60.92	N\A
200	61.31	62.26	59.72	N\A	64.23	65.25	62.36	N\A
500	62.55	63.29	61.45	N\A	65.69	66.43	64.34	N\A
1000	63.12	63.90	62.38	N\A	66.53	67.24	65.61	N\A
2000	64.05	64.51	63.28	46.0	67.46	68.04	66.53	51.85
10000	64.77	65.25	64.94	51.8	68.76	69.34	68.51	58.41
100000	63.91	64.27	65.09	55.07	69.87	70.05	70.05	63.9

Table 3: Numerical data for Figure 3. The standard deviations for all models are approx. 0.3%.