A Methodology for Refined Evaluation of ML-based Code Completion Approaches

Kim Tuyen Le
Heidelberg University
Heidelberg, Germany
tuyen.le@informatik.uni-heidelberg.de

Gabriel Rashidi
Heidelberg University
Heidelberg, Germany
g.rashidi@stud.uni-heidelberg.de

Artur Andrzejak
Heidelberg University
Heidelberg, Germany
artur.andrzejak@informatik.uni-heidelberg.de

Abstract

Code completion has become an indispensable feature of modern Integrated Development Environments. In recent years, many ML-based approaches have been proposed to tackle this task, with deep learning models achieving the best results. However, almost all of these works report the accuracy of the code completion models as aggregated metrics averaged over all types of code tokens. Such evaluations make it difficult to assess the potential improvement for particularly relevant types of tokens (such as method or variable names), and blur the differences between the performance of the methods. In this paper, we propose a methodology called *Code Token Type Taxonomy* (*CT3*) to address this problem. We identify multiple dimensions relevant for code prediction (e.g. syntax type, context, length), partition the tokens into meaningful types along each dimension, and compute individual accuracies by type. We illustrate the utility of this methodology by comparing the code completion accuracy of a Transformer-based model in two variants: with closed, and with open vocabulary. The results show that the refined evaluation provides a more detailed view of the differences, and indicates where further work is needed. Furthermore, the open vocabulary model is significantly more accurate for relevant code token types such as variables and literals.

CCS Concepts: • **Computing methodologies** → **Machine learning approaches**; • **Software and its engineering** → **Development frameworks and environments**.

Keywords: code completion, accuracy evaluation, code token types, open/closed vocabulary, Transformers, Python

KDD Workshop '21, August 14–18, 2021, Virtual

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Kim Tuyen Le, Gabriel Rashidi, and Artur Andrzejak. 2021. A Methodology for Refined Evaluation of ML-based Code Completion Approaches. In *Proceedings of KDD Workshop on Programming Language Processing (PLP) (KDD Workshop '21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Code completion is a widely used feature of modern IDEs, where the most likely next token is offered based on the code already present up to the cursor position [7]. This feature not only helps developers to save typing effort, but also assists them in learning new libraries, as it offers information about available functions or attributes. Machine-learning (ML) approaches for code completion are leading the field, and in particular the Transformer models excel here by outperforming the Recurrent Neural Networks (RNNs). Multiple state-of-the-art solutions are using Transformers with variations of the code representation and/or the attention mechanisms [3, 7, 10].

However, most of the proposed modifications for Transformers use aggregated metrics (i.e. averaged over all types of code tokens) to evaluate their accuracy. This eliminates valuable information about the improvements for relevant code token types, and as a consequence make it more difficult to compare approaches or identify weaknesses of a method. For instance, code completion for identifiers is considered highly relevant for developers [4], yet aggregated metrics cannot compare the accuracy of two approaches in this regard. To our knowledge, only few previous works [7] consider token categories and evaluate the mean reciprocal rank (MRR) per category. However, the token subdivision (into attribute access, numeric constant, variable/module name, and parameter name) is rather crude, and it is difficult to apply this taxonomy to other works. We aim to provide a more refined types of code tokens which can be used for evaluating existing and future code completion approaches with minimum effort. Our contributions are as follows:

- We propose a methodology called *Code Token Type Taxonomy* (*CT3*) for a refined evaluation of code completion accuracy by proposing multiple dimensions for identifying code token types. For each dimension we obtain the types

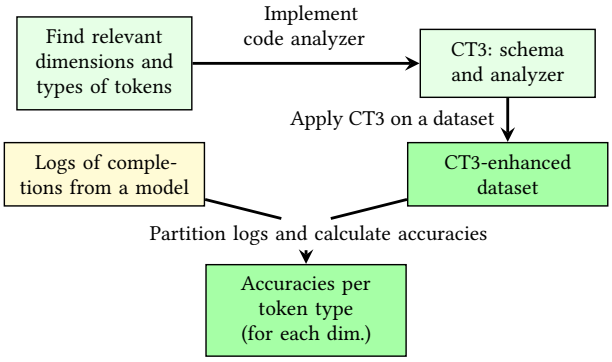


Figure 1. Implementation and usage of Code Token Type Taxonomy (CT3).

by analyzing the Abstract Syntax Tree (AST), and the relationships between tokens in the AST. CT3 can be used for a comprehensive comparison between approaches, to gain a detailed view of the impact of each component in a prediction model, and to identify model challenges.

- We demonstrate the utility of this methodology by conducting an empirical study on the Python150k[1] dataset of a Transformer-based code completion approach. We compare the impact of using closed vocabulary vs. open vocabulary [6], and find significantly better accuracy of the latter for relevant token types.
- To facilitate reproducibility and reuse of our methodology, we published the Python150k dataset with pre-computed token types according to CT3[2] and CT3 source code[3].

The rest of the paper is organized as follows. Section 2 describes the approach, while Section 3 discusses the experimental evaluation. We outline related work in Section 4, and conclude in Section 5.

2 Approach

2.1 Methodology for a refined evaluation

Figure 1 illustrates the implementation and usage of the proposed methodology. Given a programming language, we identify the code token properties relevant for effective code completions. This gives rise to multiple dimensions (i.e. criteria for partitioning), and the token types within each dimension (i.e. a complete subdivision of all tokens into types). Table 1 shows such a schema for Python. In the next step, a static code analyzer must be implemented. Given a dataset, it assigns each code token a type for each dimension. Our prototypical implementation for Python analyses the Abstract Syntax Tree (AST) and the relationships between its nodes. Note that we consider only terminal (leaf) nodes in the AST.

After applying the analyzer, we combine the results with logs of code completions provided by the investigated model (see middle of Figure 1). The result is a log file which contains for each code token its CT3-data and information on the correctness of the completion. In the final step, the log is partitioned according to the types per dimension, and the desired evaluation metric (e.g. MRR, accuracy) is computed for each type (per dimension).

In the following we discuss the CT3 schema used for Python, see Table 1. Details of meaning and identifying types in each dimension are presented in Appendix A.1.

Table 1. CT3 schema proposed for Python

Syntax Type	Context	Origin	Length	Frequency
attribute	in_arithmetic_op	from_builtin	long	high_frequent
base_class	in_assign	from_extlib	medium	low_frequent
class	in_bool_op	from_infile	short	medium_frequent
class_def	in_class_def	from_stdlib		
const	in_comparison			
exception	in_else			
func	in_except			
func_def	in_for			
func_keyword	in_func_def			
import_ID	in_if			
literal	in_parameter			
method_call	in_raise			
method_def	in_return			
module	in_try			
pre_attribute	in_while			
python_keyword	in_with			
sub_import				
var				
unknown				

Syntax Type refers to the syntactic category of a token in source code. Values of syntax type (first column of Table 1) can mostly be generalized for various programming languages and offer information regarding the code token's purpose. Most of types describe identifiers since predicting them is the most relevant completion in practice [4]. For instance *func* indicates a function call, while *func_def* denotes a function definition; analogously for *method_call* and *method_def*. A class instantiation is expressed by *class* value. The *var* represents usage of any kind of variable in general, e.g. parameters in function calls, free variables, or global variables. Syntax Type dimension also contains *keyword*, *literal* and *constant*. The *unknown* value is used for any syntax type other than the first 18.

Context describes surrounding code structures (e.g. loop body, condition expression) in which the token is found. The context types (second column of Table 1) aim to reflect the local context, which plays a large role in code completions [4]. For each code token, we record in how many contexts of a given type it is included. Listing 1 illustrates the token *var_example* which is in the context of *in_class_def*, *in_func_def*, *in_for* (twice) and *in_assign*.

```
class ClassDef():
    def func_def(self, ...):
        for i in ...:
            for j in ...:
                var_example = ...
```

Listing 1. Context Example

[1] https://www.sri.inf.ethz.ch/py150
[2] https://doi.org/10.5281/zenodo.5148586
[3] https://gitlab.com/pvs-hd/published-code/code-token-type-taxonomy

Origin indicates the location where an identifier or a keyword is defined. The *from_builtin* value represents built-in code tokens which do not require an explicit import such as keywords (e.g. `True` and `False`). Tokens categorized as *from_extlib* originate from an external (non-standard) library or a package. Identifiers defined in the same file have *from_infile* as their origin value. Ultimately, *from_stdlib* refers to identifiers defined in standard libraries.

Length of a code token is the number of characters in the token. This dimension is motivated by the fact that long code tokens are benefit more from code completions [4]. The length also correlates with the importance of a code token. Short tokens usually hold temporary values (e.g. "i" as a loop counter), which are less significant. A code token is labeled as *short* if it has up to 3 characters, label *medium* is used for 4 to 10 characters, and label *long* indicates longer tokens.

Frequency refers to code token's frequency relative to the frequency distribution of all code tokens within an AST. We use three values here: *low*, *medium*, and *high*, based on intervals explained in Appendix A.1. Long and frequent code tokens are likely to be significant. On the other hand, while short code tokens can be frequent, in general they carry in-significant (e.g. temporary) values.

2.2 Open vocabulary for Transformers

To evaluate whether CT3 is beneficial for improving code completion models, we implemented a Transformer-based code completion approach in two variants: with a *closed vocabulary model* (i.e. Transformer learns on a fixed set of strings), and with an *open vocabulary model* using Byte-Pair Encoding (BPE) [6]. In the latter version, each token can be encoded by several sub tokens (potentially even letters). The motivation for focusing on the open vocabulary model is the notorious out of vocabulary (OOV) issue encountered in source code, caused mainly by the arbitrariness of identifiers. We use HuggingFace Tokenizers[4] as the implementation.

The original code token sequence (i.e. with closed vocabulary) is created by traversing ASTs in depth-first search order. The sub tokens sequence can be much longer than the original one. Due to the limited memory capacity of our GPUs, a *window* is used to slide through the sub tokens sequence to divide it into smaller pieces. Each *window* is defined with *window_size* (i.e. number of sub tokens within a window) and *step_size* of the sliding window. A padding symbol is used to ensure all windows have the same size. After performing several experiments, we selected (1,000, 500) and (2,000, 1,000) for (*window_size*, *step_size*) of closed and open vocabulary cases, respectively. Due to the space limits, these experiments are not presented here.

The open vocabulary model uses greedy search for finding the next possible sub token. We assume that a prediction is correct in this model if all sub tokens are suggested correctly.

[4] https://huggingface.co/docs/tokenizers/python/latest/index.html

3 Experimental Evaluation

3.1 Research Questions

We address two following research questions:

RQ1. Does the refined evaluation reveal useful information for comparing and characterizing code completion approaches? To answer this question, we conduct an experiment of code completions with a Transformer-based model. We compare two variants of the model: closed vocabulary vs. open vocabulary, and investigate whether the refined evaluation reveals more information about each variant and so facilitates their comparison.

RQ2. Does the open vocabulary model improve the prediction accuracy compared to the closed vocabulary model? The utility of open vocabulary models is assessed by comparing the accuracy of completions provided by each of the both models.

3.2 Evaluation Results

Experiment setup is described in Appendix A.3.

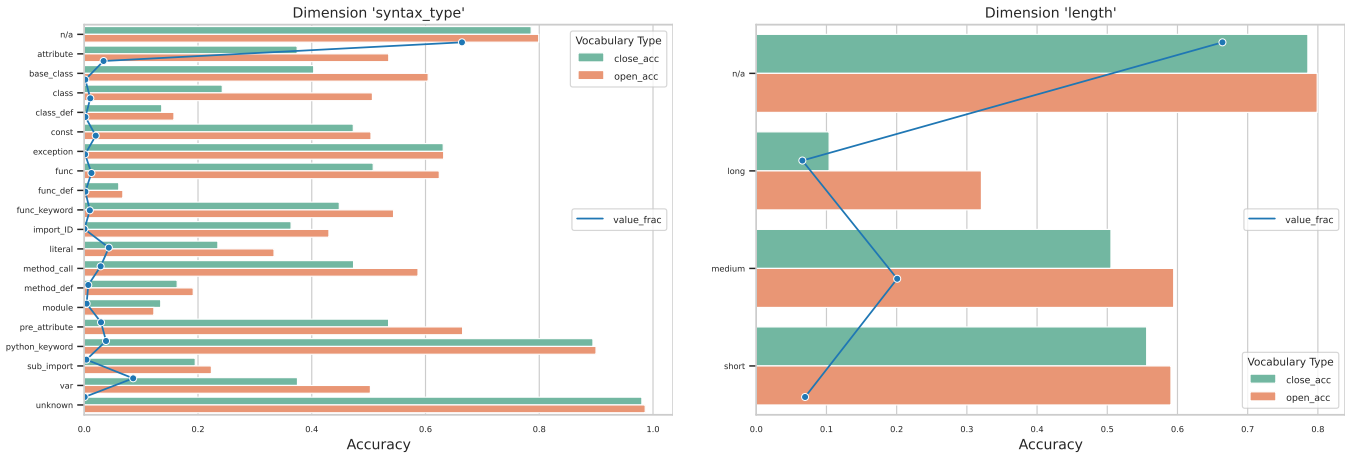
RQ1: The refined evaluation gives a more detailed information about the completion models. The experimental results show that evaluating accuracy for individual token types provides a better understanding on the prediction approaches than using the aggregated metrics. The first line of Table 2 compares the aggregated accuracy results of the closed and open vocabulary models on the Python50k dataset. Figure 2 shows the refined evaluation for dimensions *Syntax Type* and *Length*. Additionally, in Appendix A.2 we discuss the refined evaluation for the dimensions *Origin* and *Frequency* (see Figure A.1 there). The analysis of dimension *Context* is more complex and not presented in this paper. The special value *n/a* is mostly used for code tokens that are non-terminal nodes or their types could not be identified.

Table 2. Aggregated accuracy of closed and open vocabulary models

Dataset	c_acc.	o_acc.	c_oov_o_true *	oov_c **
PY150k	0.6687	0.7121	162,750	626,087
JS150k	0.6964	0.7485	164,924	609,391

We evaluate one tenth of the evaluation dataset.
c/o denotes closed vs. open vocabulary model, acc. is accuracy.
* Number of OOV tokens in closed vocabulary model that can be predicted by the open vocabulary model.
** Total number of OOV tokens in closed vocabulary model.

While the aggregated metric indicates that the open vocabulary model increases the prediction accuracy by only 6.49%, the refined evaluation clearly reveals that the open vocabulary model outperforms the closed model in every dimension of CT3, with improvements ranging from 6.2% to 2.09 times. One of the reasons that aggregate metrics shows



(a) (b)

Figure 2. Comparing the accuracy of the closed and open vocabulary models for dimensions *Syntax Type* and *Length*. The bars represent accuracies and the line shows the relative fraction a token type over all tokens.

only a moderate improvement is caused by the token type *n/a* (mostly internal AST nodes), which makes up more than a half of test instances, but does not benefit much from the open vocabulary model.

Figure 2(a) shows that for the dimension *Syntax Type* the open vocabulary model achieves a higher accuracy for all token types except the *module* type. There is substantial difference for the *class* type (1.07 times). Increase of the accuracy for token types *var* and *literal* (two most relevant completions in practice) is 34.1% and 41.7%, respectively. Both closed and open vocabulary models perform quite well for tokens categorized as *python_keyword*.

Open vocabulary model also outperforms the closed model in all values of dimension *Length* (Figure 2(b)). Although the overall accuracy for *long* tokens is not high (ca. 32.1%), the improvement of 2.09 times is still remarkable.

RQ2: Open vocabulary model outperforms the closed vocabulary model on predicting variables and literals. The results in Table 2 and Figure 2 show that the open vocabulary model enhances the prediction accuracy of the Transformer model, especially in completing *variables* and *literals*, and *long* tokens. The last two columns in Table 2 are computed to clarify the utility of open vocabulary model in addressing the out of vocabulary (OOV) issue. Around 25.9% of OOV tokens encountered when using the closed vocabulary model can be recommended correctly by the open vocabulary model (Python dataset). An additional experiment conducted on JavaScript[5] dataset (second line of Table 2) confirms the advantage of the open vocabulary approach.

4 Related Work

State-of-the-art approaches for code completions or general code predictions use ML-based techniques [8]. Methods include *n*-gram language models [5], Probabilistic Higher Order Grammars [2], Recurrent Neural Networks (RNNs) [1, 9], or hybrid approaches [11]. Recent works [3, 7, 10, 12] use Transformer models [13] which outperform RNNs.

An important aspect of the prediction approaches is code representation. While some works use as input a sequence of AST nodes linearized by a tree traversal [9, 11, 12], more recent approaches attempt to capture the high-level structural representation [1, 7]. Authors of [3] indicate only syntactic information is needed to make meaningful predictions.

Another factor of code representation is to capturing the code identifiers as a closed vocabulary or as an open vocabulary, e.g. via Byte-Pair Encoding (BPE) [6]. While only few works use the open vocabulary model, e.g. [12], the results of this work show that this variant can significantly improve the accuracy of relevant token types.

Almost all of the prior works use aggregated metrics to evaluate the accuracy by averaging over all code token types ([7] provides a rough analysis, see Section 1). However, the authors of [4] show that there are large differences of relevance of completions from the point of view of developers. We propose a more detailed way of evaluating the accuracy of code completions which might facilitate comparison and improvement of prediction models for the relevant cases.

5 Conclusion

We proposed a methodology called CT3 for a refined evaluation and comparison of code completion approaches. Our empirical study shows that CT3 is helpful in characterizing and comparing the accuracy of approaches. As a side-effect, we demonstrated that the open vocabulary model significantly enhances the accuracy of Transformers on code completion for relevant tokens like variables and literals. We also published the CT3 information for the Python150k dataset, and will publish the CT3 code analyzer for Python. Further work

[5] https://www.sri.inf.ethz.ch/js150

will include extending the method to other programming languages and datasets, and implementation of specialized code predictors according to the proposed CT3 schema.

References

[1] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International conference on machine learning*. 245–256. tex.organization: PMLR.

[2] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. PMLR, 2933–2942.

[3] Nadezhda Chirkova and Sergey Troshin. 2020. Empirical study of transformers for source code. *arXiv preprint arXiv:2010.07987* (2020).

[4] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 960–970.

[5] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[6] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.

[7] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[8] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation. *Comput. Surveys* 53, 3 (Jul 2020), 1–38. https://doi.org/10.1145/3383458

[9] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 4159–4165. https://doi.org/10.24963/ijcai.2018/578

[10] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 473–485.

[11] Max Eric Henry Schumacher, Kim Tuyen Le, and Artur Andrzejak. 2020. Improving Code Recommendations by Combining Neural and Classical Machine Learning Approaches *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 7 pages. https://doi.org/10.1145/3387940.3391489

[12] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. 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, 1433–1443. https://doi.org/10.1145/3368089.3417058

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

A Appendix

A.1 Details on Identifying Token Types in CT3

We provide here additional details of identifying the token types for three more involved dimensions of the CT3 schema for Python introduced in Section 2.1 and shown in Table 1. The identification of the token types for the dimensions *Context* and *Length* is straightforward and omitted here.

Syntax Type values (i.e. first column in Table 1) are derived based on the syntactic information of the source code and the patterns in ASTs. The complexity of identifying these token types ranges from simple to very complex. Simpler types mostly depend on conditions of identifying AST node types. Complex types are aggregations of conditions which identify feature-specific AST patterns.

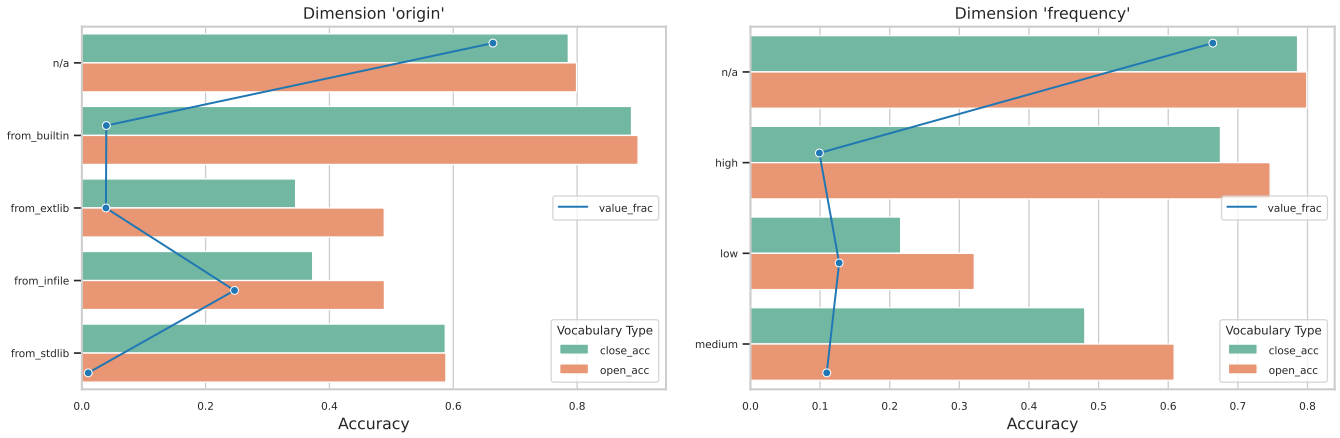
An example of a simple type is *literal*, which indicates that the code token is a string, or the AST node type is "Str". The *var*, however, is a case of high complexity. Several AST patterns related to various sorts of variables are inspected to identify token type *var*. These are: (1) parameters in function calls, (2) function or method definitions, (3) free variables, (4) subscripted variables, (5) global variables, and optional function arguments indicated by "vararg" and "kwarg".

Parameters of a function call (1) are located in children's leaves of an AST "Call" node, except the first child, which refers to the function itself. The node type of ancestors of those leaf nodes must not be "attr". Otherwise, this would indicate a method call or class initiation. Parameters of function or method definition (2) can easily be identified by checking for the "NameParam" node type. Free variables (3) are found based on the exclusion of other variable types. This exclusion incorporates any nodes which are children of "Call", "Subscript", "bases" and "Attribute" node types to ensure that the variable is not involved in any call, or it does not have a subscript, it is not a base class inherited to a child class, or is not an attribute of a class, respectively. Variables with a subscript (4) can be classified by checking for the "Subscript" node type. Ultimately, global variables, "vararg" and "kwarg" variables (5) are detected by examining the ancestors of leaf nodes for "Global", "vararg" or "kwarg" types.

Identifying the remaining syntax types is less complicated. For instance, the *pre_attribute* and *attribute* indicate a pre-attribute (if any) and attributes of an object (e.g. class instance). A base class parameter in a class definition is labeled as *base_class*. An imported library is identified by *module* and its alias (if any) is classified as *import_ID*.

Origin labels (i.e. third column in Table 1) are obtained by analyzing the `import` commands in each AST to determine the origin of the code tokens. Code tokens that appear as attributes of a particular library are then categorized accordingly. Built-in code tokens are those within a predefined *python_keyword* set. Tokens from within the file are determined by exclusion. Those tokens are neither from the standard library nor external libraries nor built-in.

Frequency of a code token is computed for each individual AST. Three intervals specifying the frequency of occurrence as *low*, *medium*, and *high* are adjusted by the min, mean, and max value of the frequency distribution of all tokens within the AST. Equations (1a) to (1e) explain calculation for these intervals. An illustration of this process is presented in Figure A.2.



(a) (b)

Figure A.1. Comparing the accuracy of the closed and open vocabulary models for dimensions *Origin* and *Frequency*. The bars represent accuracies and the line shows the relative fraction a token type over all tokens.

$$boundary_1 = (mean\_freq - min\_freq)/2 \quad (1a)$$

$$boundary_2 = (max\_freq - mean\_freq)/2 \quad (1b)$$

$$low\_interval = [min\_freq, boundary_1] \quad (1c)$$

$$medium\_interval = [boundary_1, boundary_2] \quad (1d)$$

$$high\_interval = [boundary_2, max\_freq] \quad (1e)$$

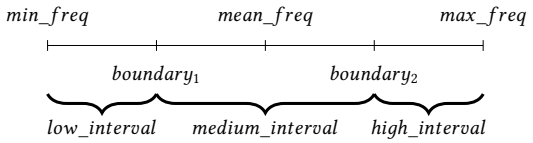


Figure A.2. Intervals for determining frequency labels of code tokens

A.2 Additional Evidence Showing Advantage of the Open Vocabulary Model

The refined evaluation for the dimensions *Origin* and *Frequency* shown in Figure A.1 indicates that the open vocabulary model outperforms closed vocabulary model for all token types in these dimensions. The notable points are the results for the *in-file* and *low-frequent* token types, which not only constitute a relatively significant fraction of the dataset, but are also quite difficult to predict when using the closed vocabulary. The increased accuracy for these cases (31.1% and 48.6% for in-file and low-frequent tokens), together with the results of the analysis in Section 3.2 emphasize the advantage of using the open vocabulary model instead of the traditional closed vocabulary model.

A.3 Experimental Setup

We conduct the experiments using the datasets Python150k and JavaScript150k. The model is fitted on the original train datasets (i.e. Python100k and JavaScript-100k), but due to performance reasons evaluated on 1/10th of the evaluation datasets (i.e. Python50k and JavaScript50k). We use Python 3.7.9 and TensorFlow 2.3.0 for our implementation.

Data preprocessing. The code tokens and subtokens sequences are created by traversing ASTs in a depth-first search order. Due to the considerable noise amount in the dataset, we eliminate all white spaces, tabs and new lines before collecting tokens for building encoders and creating input files for our Transformer model. Besides, we performed additional experiments on the effect of token length on the built vocabulary and prediction accuracy. The experimental results show that there should be a threshold for token length when building encoders (e.g. 50) or creating *tfrecord* files (e.g. 30). Due to the space limits, these additional experiments are not presented in this paper.

Experiment configuration. Table A.1 presents the settings used for conducted experiments. The model is trained and evaluated on one GPU (GeForce RTX 2080 Ti or TITAN Xp). The whole computation process (i.e. preprocessing, training and evaluating) takes more than two weeks.

Table A.1. Experiment Configuration

Parameter	Value
vocabulary_size	10,000
(window_size, step_size) for closed vocabulary	(1,000, 500)
(window_size, step_size) for open vocabulary	(2,000, 1,000)
batch_size	8
epochs	10
max_len_encoder	50
max_len_data	30
optimizer	Adam