Represent Code as Action Sequence for Predicting Next Method Call

Yu Jiang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
yujiang@smail.nju.edu.cn

Liang Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
wl@nju.edu.cn

Hao Hu
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
myou@nju.edu.cn

Xianping Tao
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
txp@nju.edu.cn

ABSTRACT

As human beings take actions with a goal in mind, we could predict the following action of a person depending on their previous actions. Inspired by this, we collect and analyze more than 13,000 repositories with 441,290 python source code files from the Internet. We find the action expressed in code form lies in the developers' high-level programming language statements.

Previous code understanding and code completion research paid little attention to code editing contexts like code file name and repository name while representing the code for machine learning models. Via modeling method calls as action and modeling method names, file names, and repository names as code editing context, with the help of the modern natural language processing models and techniques, we utilize the huge open source code resources from the Internet and train a model with the understanding of the hidden meaning under the method call sequences to predict the next method call token for developers.

In the evaluation part, the experiments we conducted showed the GPT-2 model trained with our action sequence representation achieves 81.92% top-5 accuracy for next method call token prediction, compared to 61.89% of same GPT-2 model trained with same data set, the proposed approach makes an improvement of 32.36%. As for the context of the code we proposed, we find it critical for machines to understand the code. All the above contribute to code comprehension and enhance method call completion for developers via unlimited Internet resources.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Internetware 2022 (13th Asia-Pacific Symposium on Internetware), June 11-12, 2022, Hohhot, China

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

KEYWORDS

mining software repositories, software engineering with Big data, software engineering with AI

ACM Reference Format:
Yu Jiang, Liang Wang, Hao Hu, and Xianping Tao. 2022. Represent Code as Action Sequence for Predicting Next Method Call. In *Internetware 2022 (13th Asia-Pacific Symposium on Internetware), June 11-12, 2022, Hohhot, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

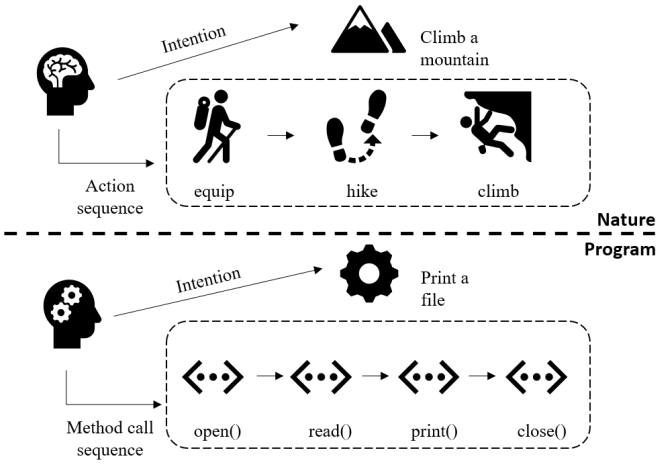

Figure 1: The similarity of action sequence in nature and method call sequence in program context for human.

If a person knows everything in a field has happened in the world, they may be the one who could see the intention behind the actions in that field. However, the memory of a person is limited. For now, only computers can remember this quantity of knowledge. As the Internet develops, code from all kinds of projects could be seen as a form of open source repositories, which makes the computer closer to being capable of knowing everything in the coding field. For the computer to be the one who could see the intention behind the

Table 1: The most frequent words in complete code text

Rank	Words
1-5	self, if, return, def, in
5-10	none, for, not, else, is
10-15	name, data, raise, false, true
15-20	i, path, len, format, np

code, the next thing it needs is thinking. Fortunately, the state of the art machine learning models and techniques could help with this. Therefore, the last problem left here is how we represent the code knowledge for computers.

Method call completion is a part of code completion. In modern software development tools, code completion is one of the most frequently used functions, directly higher the software artifact productivity. Figure 3 shows a familiar code editing environment, the code expressions in the block with gray color are in the method call prediction task.

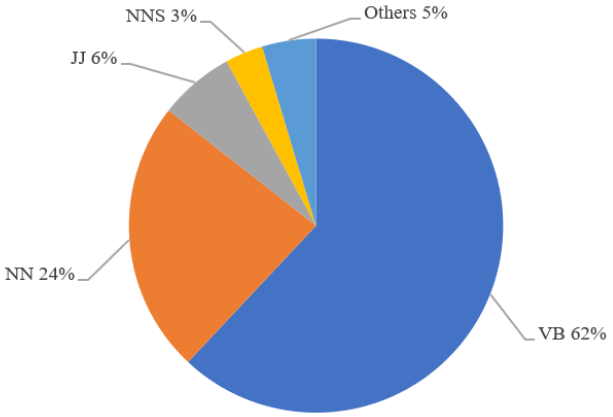

Figure 2: Word tags distribution of the first tokens in method names. We use nltk[15] to analysis tags. In this figure, *VB* refers to verb, *NN* refers to noun.

To predict the next action of a person, given their previous actions, we may guess the intention behind action sequences. For method calls in code of Code Search Net data set[10], as shown in Figure 2, the first words of method names written by developers are mostly verbs as our analysis, which are quite different from the most frequent words in complete code text as listed in Table 1.

In Wikipedia, word subroutine is described as "In computer programming, a subroutine is a sequence of program instructions that performs a specific task, packaged as a unit. ". Both from the naming of method calls and definition of a subroutine, we could somehow take method call as one kind of action expression form in code as shown in Figure 1, so that the way we find similarity between action in these different situations and model the action in code to represent for machine learning models.

To model the action in nature, who conducts the action, under which situation the action happens, what items the action uses will be key points to model the action. Similar to modeling action in codes, a method call is an action entity, which instance or object conducts the method call, under which context the method call invokes, what parameters the method call uses, will be accordingly modeling the action in code.

At present, on the Internet, a larger number of open source code repositories with nearly unlimited source code files could be utilized to teach the machine learning model with the representation mentioned above. This allows the model to remember different action permutations or combinations under a different context. With the help of the state of the art natural language processing(NLP) models like transformer-based language models, the trained model is capable of learning the relationship between who, which, and what described above, and such relations could also be learned among several continuous methods calls as actions in code. This hidden relation learned by the state of the art transformer-based language model lead to the understanding of the intention in the developer's mind, which will give a clue about the next action the developer wants the code to do, or to say the next method call the developer would like to write next. However, there is also a challenge to learn from the open source code repository because the codes on the Internet are hard to compile one by one, and even if the compilation is available, the time cost for project build of such a quantity of repositories is unacceptable. To face this challenge, the action representation form in the code context we proposed does not need to be compiled, and we only need the code to meet the syntax requirement.

After collecting more than 440000 source code files from more than 13000 repositories from GitHub, we analyze and process the original data as the above inspiration guided, then model the code text content related to actions into structural data with the help of an abstract syntax tree(AST) tools, finally conduct different arguments settings on the model which reaches 81.92% top-5 next method call token accuracy on the randomly selected test set. The comparing experiments show the effectiveness of the proposed context encoding method and action modeling method. With the combination of static code analyzing tools, the model is capable of helping build stronger code completion tools.

Our contribution includes:

1. A code representation approach for method call prediction with the advantages of generalization and efficiency.

2. A general and simple context representation for code expressions.

3. An efficient approach to utilize unlimited open source repositories on the Internet for code knowledge construction.

2 OVERVIEW

As Figure 3 shows, when a developer is writing codes in an editor or IDE, a system with our proposed model will collect informative source code content created by the developer and use the proposed model to represent the content into an action sequence, then input into the trained model to get the prediction for next action, in this case, an action is a method call the developer would like to write next.

Developer coding context When a developer is writing a source code file within a project, the project name, the directory name, the

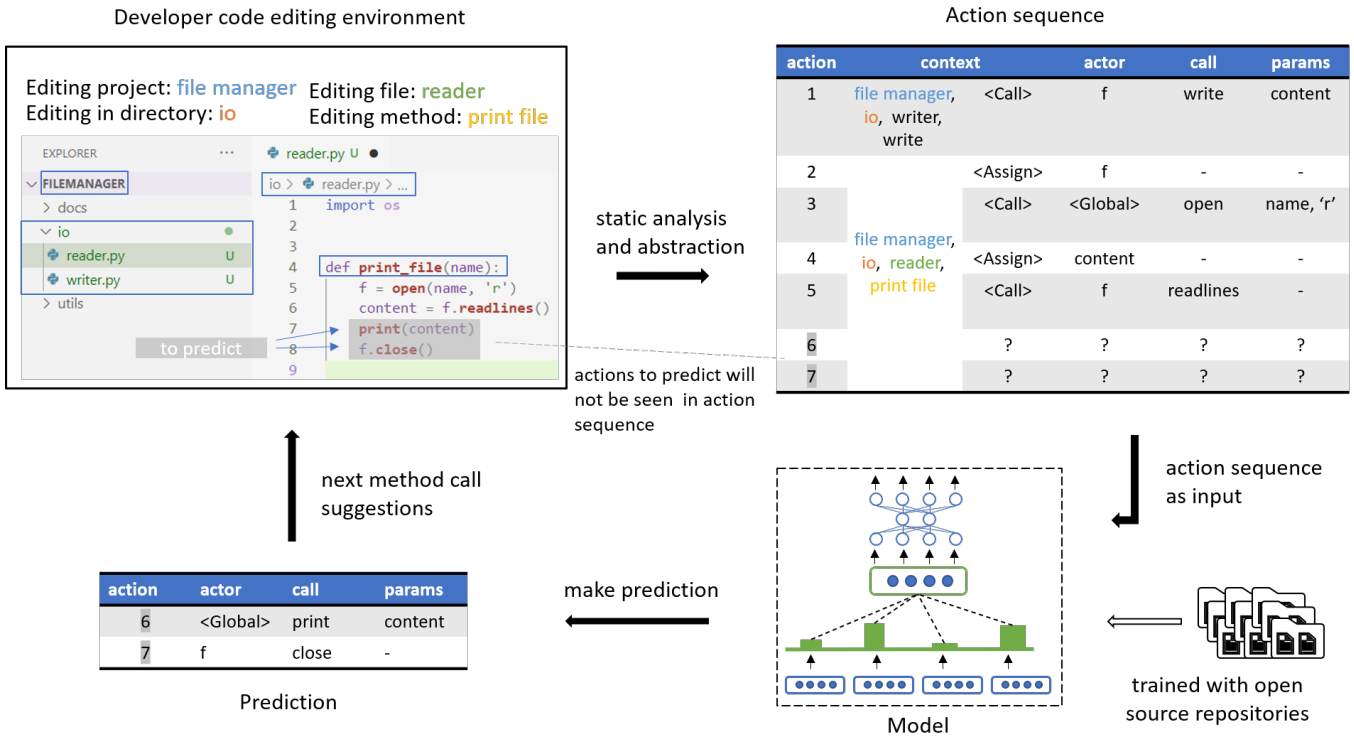

Figure 3: Approach overview. Start from code editing context, until the model make method call predictions for the block coloured in grey. The first action with actor *f* and call *write* in action sequence is constructed from a code expression previously written by the same developer in *writer.py* file, this part of code is not shown in this figure.

file name, and the method name often indicate the goal of the developer's editing content. As different words will be explained under different contexts, different developer coding contexts also guide the different comprehension of the code. To utilize this context, we encode the context into every action.

Action We've discussed the similarity between action in code and nature in the previous section. We define an action in code containing four parts of content: $\langle context, actor, call, parameters \rangle$. These four components are inspired by both describing actions in nature and the definition of a method call in program. For most situations, actions in code are built from method calls and assignment expressions.

Consider action *5* in Figure 3:

$$\langle (file, manager, io, reader, print, file), f, readlines, \emptyset \rangle$$

In this action, the context $(file, manager, io, reader, print, file)$ indicates its goal is likely to be file-reading related operation, and the latest task is probably to print the content. The actor f under this context often be regarded as a short for *file*, following call *readlines* is the specific move made by actor. Because the call doesn't need any parameters, the 4th component of the action is $\emptyset$.

Action sequence The action sequence is a group of continuous actions, the action order in a sequence should mainly depend on the order of actions in the developer's mind though this could not be directly extracted. For the case shown in the figure, the action sequence is decided by the syntax analysis result, which often reflects the execution order.

Predict next action in code Consider the action sequence in Figure 3, the first action's context indicates the developer firstly writes a method call in the same directory *io* but in a different file *writer.py* and in a different method *write*, and the subsequent actions after that are all in the same file and method. The goal of our model is to retrieve the intention behind the action sequence and then make predictions about the next action. In the shown case, the next action is a method call *print*, following that is an actor called f conducts a call named *close* with no parameters.

3 BACKGROUND

In this section, we will introduce the causal language model and transformer, the action sequence comprehension and prediction implementation is built via these models.

3.1 Causal Language Model

The language model is applied to describe a probability distribution over a sequence of words. For example, the n-gram language model made an assumption that the probability of a word is decided by its previous $n-1$ words in sequence.

A causal language model is a form of the language model, consider an assumed probability distribution of a sequence of words:

$$P(w_{m+1}...w_{m+n}) = \prod_{i=m+1}^{m+n} P(w_i|w_{i-k+1}...w_{i-1}) \quad (1)$$

where m indicates the start position of a sequence, n is the length of the sequence, and k defines the quantity of context that can be taken to calculate the probability distribution. This assumption only accepts the context before a position but not after the position, and the causal language model is based on this assumption, which means if we apply the causal language model into a neural network, the prediction position could only access the past information but not the future.

Language modeling can be seen as a task to build a language model. In deep neural natural language processing, language modeling is often a training process for models to improve sequence representation and comprehension ability. Well-known models like Google's BERT [5] and OpenAI's GPT-2 [16] apply masked language modeling and causal language modeling separately.

As for our prediction task, knowing a developer's recently done actions and making the next action prediction based on his or her past action sequence is exactly our goal. It is obvious that the program is executed instruction by instruction. Latter instructions often depend on the execution result of previous ones in a non-concurrency execution environment. So that causal language modeling will be the key process in our approach.

3.2 Transformer

Transformer[19] has been the most influential natural language processing neural architecture for a few years, and the latest popular computer vision research work named Vision Transformer(ViT)[6] is based on transformer architecture. For our approach, we need a neural architecture that could easily capture a long-term dependency in a sequence. The word *easily* refers to the inspiring training parallel efficiency than previous works.

Compared with other similar works, the attention as described in the original transformer paper is what we need for comprehension of the method call sequence or action sequence in our context. Here we will just go through the key concepts of attention. Given a query vector q, a key vector k, and a value vector v, if we would like to give out a weighted v belongs to k depending on the q, then attention mechanism together with other necessary neural network components could help to learn the proper weight for a given task. As for our experiment, we use transformer-based OpenAI's GPT-2 [16] for the training process.

4 ACTION SEQUENCE REPRESENTATION FOR CODE

In this section, we will explain our action sequence representation for code in detail. Illustration for constructing from code to action representation in the real word can be found in Figure 3.

4.1 Action

An action is a tuple with 4 components in the form:

$$action = (context, actor, call, parameters) \quad (2)$$

4.1.1 Context. The context for action in code is derived from human actions in nature shown in previous sections, which provides information about the environment and the goal of the action. Consider in a development environment a developer is working on a code project or code repository $repo$ and implementing a method named $meth$ in code file whose path is dir appended with $file$, the codes this developer has written in the method $meth$ is $code$, and then the $context$ of the code is defined as:

$$context(code) = (repo, dir, file, meth, type) \quad (3)$$

where every variable in the equation is a sequence of words in natural language, the variable type is depends on the expression type of $code$, including $Call$, $Return$ and $Assign$.

4.1.2 Actor. For human actions, an action is done by an individual or by a group of individuals. We define an actor as the source of action. For object-oriented programming, an object or a class could be the actor. For procedural programming, a static library or a source code file could be the actor. In our definition for python, if a method call has an instance or module belonging to it, then it is the actor. For assignment expression, the left value is the actor like a man or woman dressing up himself or herself, and the dressing action is done by the person who is going to wear the clothes. For those actions that do not have an instance or module belonging to them, we designate its context as its actor. In a detailed example like in Figure 3, a special variable *Global* will be designated to its actor position.

4.1.3 Call and Parameters. Call and parameters are nothing special but the original definition for method calls. For code expressions that are not method calls, the call is empty, and context will contain the expression type *Assign* at the end in it as type. For example, the action of the assignment expression code c will be defined as:

$$action(c) = (context(c), Global, \emptyset, \emptyset) \quad (4)$$

where *Global* is a predefined constant variable.

4.2 Action Sequence

4.2.1 Action sequence window size. Consider a sequence containing n continuous actions done by a developer:

$$S = (action_1, action_2, ..., action_n) \quad (5)$$

where n is the action sequence window size. Obviously, the larger the window size, the more history of current action we know. Notice that actions in a sequence of window size n may have n different contexts from actions in the sequence, the only requirement for an action sequence is the actions are in a continuous manner either ordered by the developer's input order or the possible execution order from syntax analysis, which will ensure there is a goal or hidden meaning behind the action sequence.

4.2.2 Alatten action sequence. So far, we have defined and explained an abstraction of an action sequence for code, and now we will flatten it to natural language manner. This flattening process is like giving a continuous series of scenes, actors, actions, and requirements for actions, then simply concatenating to readable sentences with the given material, which correspond to contexts, actors, calls, and parameters in our definition of action.

Flatten process return the original word sequence of every component in action sequence and insert different kinds of separator

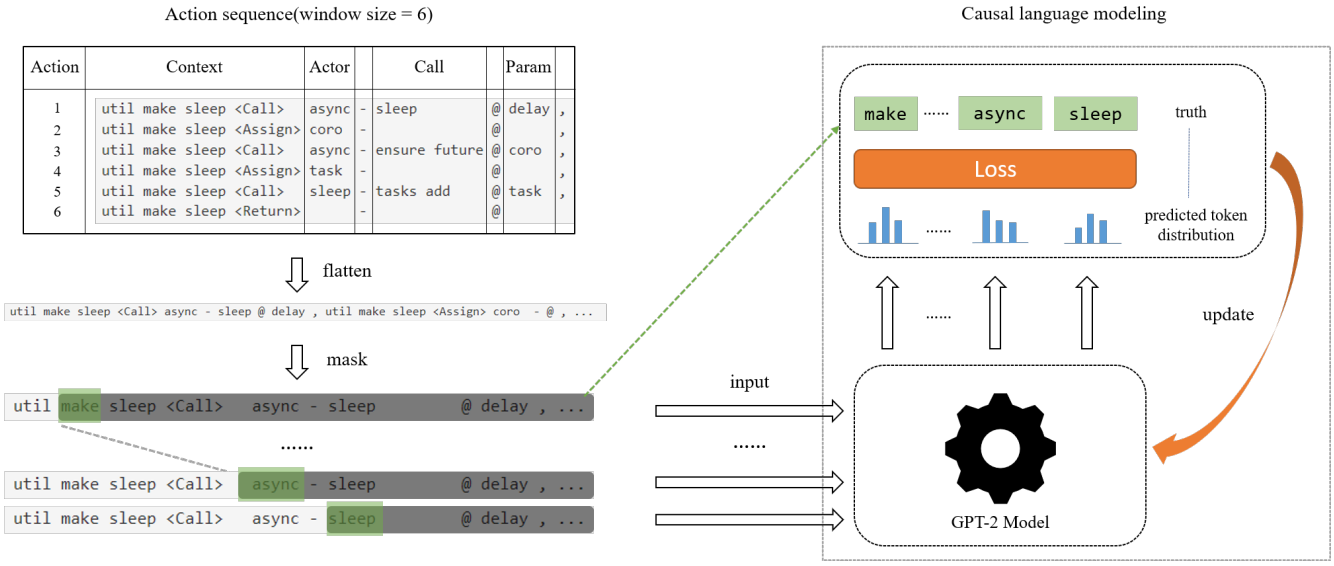

Figure 4: Training process overview. The action sequence will firstly be flattened and then mask the flattened action sequence token by token for each input sequence. Finally, the model will predict the masked token distribution and update its parameters from backpropagation.

for every action and every component in each action:

$$flat(S) = (flat(action_1), SEP_{action}, flat(action_2), SEP_{action}, ..., flat(action_n))$$

$$flat(action_i) = (flat(context_{action_i}), SEP_{context}, flat(actor_{action_i}), SEP_{actor}, flat(call_{action_i}), SEP_{call}, flat(parameters_{action_i}))$$

and finally, flatten S to a sequence of words and predefined separators.

5 DATA SET

5.1 Data Source

5.1.1 Public data set. In order to model the action concept mentioned above with the information hidden in the open source code repositories, our requirement for the dataset is to keep as much input data as possible following the order the developers wrote the code. Input data is like creating a directory with a name or creating a file with a name, then writing a method with a name, and so on. This kind of data, as we inspired from actions in nature mentioned before, should help construct the action context in code, which may reveal the goal or intention in the developer's mind while editing code. However, without specifically collecting data from a large number of developers enrolling in an extremely large scale of different code projects with a well-designed tool, the perfect dataset is almost impossible to directly collect from the Internet by us.

Though the perfect dataset for us is temporarily unavailable, we could still extract enough information from raw source code repositories to implement our model. For publicly available code datasets, we have surveyed many related code datasets for machine learning, but none of them satisfies.

On the one hand, some public dataset has got enough methods but lack the order information of methods and files, also lacks headers of the source files. This kind of information lacks because the trained models based on such datasets don't need to use the information taken by project directory structure, class or file names, and so on.

On the other hand, our assumption is based on the informative words the developers construct in their minds with other informative parts of the current editing file and the project. This requires the code repositories are not too poorly crafted by themselves, which means the developers' contribution to the repository should at least have the effort to make other developers not pay too much effort to understand the meaning of the codes. For this reason, we prefer to code with at least a small number of methods or files to have comments, which always indicate some parts of the names for variables and methods in the repository are written after thinking.

5.1.2 Our data set. Finally, to build the dataset matching the requirements, we construct a dataset ourselves based on the Code Search Net dataset constructed by Husain, Hamel et al.[10]. The based dataset is designed for code search, so the threshold number of comments in filtered code is needed originally. As for our dataset building process, we use GitHub API to collect code repositories' file structures and every source code file in each repository. The overview of our dataset is shown in Table 2.

For choosing python as our experiment programming language, firstly, python is a high-level programming language, which is designed with natural language elements, and secondly, python

Table 2: Dataset Overview

Item	Count
#Repositories	13,006
#Source code files	441,290
#Methods	3,550,572
#Lines	85,238,675
#Tokens	73,335,634
#Unique tokens	175,143

is a dynamic typing programming language, this means developers will get less accurate suggestions from traditional static analysis strengthened code editors or IDEs while editing the python code, which may lead writing more informative variable name and method names for themselves to understand and the proposed model could be more valuable for helping developers more on code completion topic.

Other repositories not satisfying the requirement are not under consideration for our action sequence representation. From the NLP aspect, developers who are willing to write sensible content will be easier to take advantage of the knowledge from the open source code repositories contributed by other developers who did the same.

5.2 Data Processing

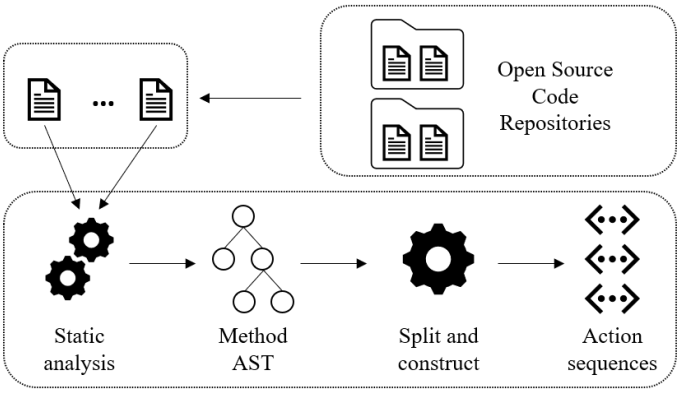

Figure 5: Data processing overview

In data processing, for parsing source code files to ASTs, we make use of python built-in ast package, and for splitting names into words, we apply Ronin[9] algorithm. The data processing overview is shown in Figure 5.

6 TRAINING

6.1 Experiment Settings

6.1.1 Hardware settings. We use Nvidia Tesla P100 GPU with 16GB graphic memory. The total training time for shown results in this paper takes 204 hours(8.5 days) of GPU time in total.

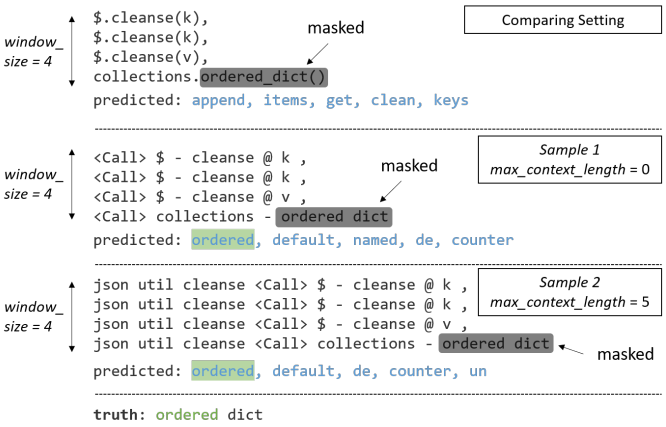

Figure 6: Input and prediction results in real word environment, different experiment settings are labelled at top right of each example, the expecting truth is demonstrated at the bottom.

Table 3: Train/Test data set split.

Statistics	Train	Test
#Methods(test window size = 4)	311,801	17,785
#Methods(test window size = 6)	311,801	13,984
#Methods(test window size = 8)	311,801	11,015

6.1.2 Data set settings. In order to get control of the static analysis processing time upper bound over source code files from more than 13,000 repositories in our data set introduced in the previous section, source code files with more than 500 lines are filtered out. Methods that could not generate at least an action sequence with three actions are also filtered out.

Training and testing data split is shown in Table 3. All the action sequences in the test data set are generated from the methods not seen in the training process. For all experiments, we select 30,000 action sequences from methods in the test data set for testing with no preference.

6.2 Training Approach

The key arguments in the training process are listed in Table 4, arguments with similar meanings like max_actor_length are not listed there. The GPT-2 model mentioned in the previous section is set with default arguments, and we use the version named DistilGPT2, which is publicly available in Hugging Face[23].

Figure 4 shows the sketched training process of an action sequence of a method. The masked tokens in the flattened action sequence could not be seen by the model, and the mask will move token by token of an input sequence. For overall training, the flattened input sequence is much longer for training efficiency. Action sequences are often concatenated with other action sequences following the corresponding methods AST traversal order in source files.

Table 4: Arguments for the training process.

Argument Name	Possible Values	Explanation
window_size	-1	The window size of input action sequence for training, -1 means no limit.
window_size_threshold	3	The threshold window size to filter action sequence for each method, method with action sequence window size less than this value will be discarded in training.
max_context_length	0,1,2,3,4,5,6,7	The max length of context in each action, context length larger than this value will be trimmed.

Since the pre-trained GPT-2 model we mentioned uses Byte Pair Encoding(BPE)[17] as its tokenizing method, the input may seem a little different for some rare words in code. The implication of the tokenizing difference could be neutralized by training on a huge data set with a causal language modeling task.

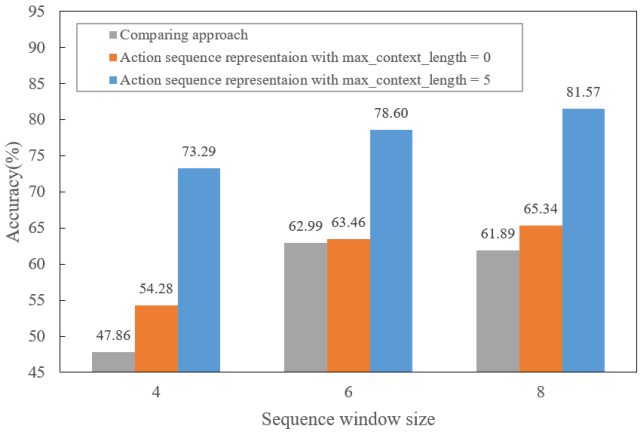

Figure 7: Next method call token prediction top-5 accuracy of comparing approach, our approach without context(*max_context_length* equals 0) and our approach with context(*max_context_length* equals 5). The results in different sequence window size settings are shown in different colors.

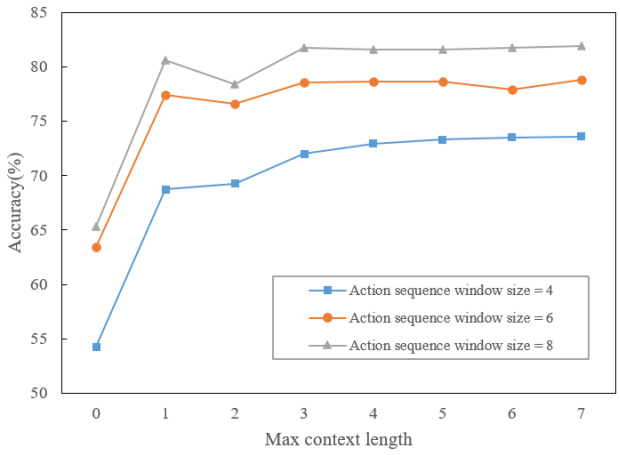

Figure 8: Next method call token prediction top-5 accuracy of models trained with our approach in different *max_context_length* and action sequence *window_size* settings.

7 EVALUATION

In this section, we will evaluate the performance of our proposed action sequence representation in the method call completion task and answer the research questions by comparing results. Overall results of our approach are shown in Table 5.

7.1 Research Questions

The evaluation experiments are mainly designed for answering the following research questions.

7.1.1 RQ1. How useful is the action sequence representation approach on method call prediction task?

7.1.2 RQ2. How useful is the context in action sequence representation on method call prediction task?

7.1.3 RQ3. How does window size influence the method call prediction result on models trained with action sequence representation?

7.1.4 RQ4. What can model trained on action sequence representation approach learn?

7.2 Comparing with Same SOTA Pre-trained Model without Applying Our Approach

To answer RQ1, we trained the same SOTA pre-trained model in the same way with formatted input sequence but not our representation from the same data set. Figure 6 shows the different input formats for the same original source code snippet. To remove the influence of the proposed context, we remove context from one of the groups, and the result shows in Figure 7 that our approach performs 13.40% better when test window size equals 4, and in setting with the context, our approach performs 53.13% better than the comparing approach.

7.3 Comparing with Different Context and Window Size Settings

For answering RQ2 and RQ3, as Figure 8 shown, with or without context in the representation significantly affects the prediction

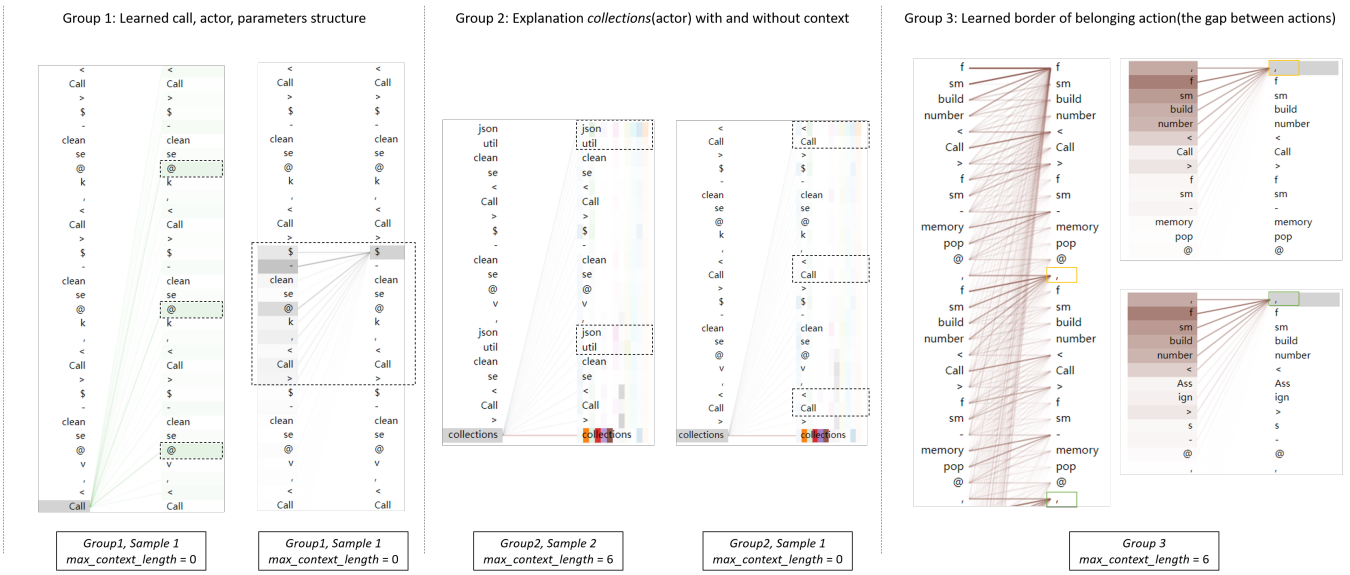

Figure 9: Learned attention of layer one for different experiment settings. The input sample is corresponding to the same sample ID in Figure 6.We use bertviz[20] for visualization of our models' attention.

accuracy. With the max allowed context length increases, the accuracy of the same window size increases much slower. Window size serves as context length. The larger the window size, the accuracy improves under the same context setting. In Figure 7, the comparing approach also benefits from larger window sizes.

7.4 Learned Attentions from Different Experiment Settings

For RQ4, Figure 9 shows the learned attention of different experiment settings. The different color refers to the different heads of learned attention, the more the attention weights, the stronger the color. From the comparing settings illustration of the group 1 in Figure 9, the model has learned a method call will set its parameters near the special sign *@* and also learned a caller *$* is related to its call and parameter content, which means the model is capable of recognizing the different components of each action.

As group 2 in Figure 9 show, the reason for the performance difference probably lies in the explanation of collections made by learned attention. The model knows context explains *collections* much with the context *json util*, which in transformer architecture will give weights on related output of *json util*.

Results that are shown in group 3 in Figure 9 reveal the model's ability to judge the border of continuous actions. This ability will guide the model to explain a token in code within an action scope, which will improve the model's comprehension for single method call or assignment expression.

8 RELATED WORK

8.1 Machine Learning for Codes

The phrase 'Big Code' often means a huge amount of code data and the potential value in the codes. Hindle et al.'s breakthrough[8] on the naturalness of software may be a sign of using NLP techniques to model code, their simple but effective application of N-gram showed the similarity between natural language and codes. M.Allamanis et al.'s paper[1] describes a model using handcrafted features and sub-token of codes to suggest method and class names. X. Gu et al.[7] build a system converting natural language queries to API sequences via encoder and decoder framework. U.Alon et al.[4] find a way not based on the text of the code but a way more program style by using the in-product of the compiler results, they extract pairs of the node to node path in AST, in which the structure information of codes is reserved, and their proposed method is capable for applying on different languages with a huge amount of data. R.-M. Karampatsis et al.'s solid work[11] directly face the challenge of open-vocabulary in big code, taking care of large vocabulary, Out of Vocabulary(OOV), and rare words problems caused by the variance of the developers' mind. They work hard to demonstrate the solution results after applying various methods on convention, string, number, etc., separately and as combinations. After the comparison, they propose Byte Pair Encoding(BPE) as their solution for this challenge and get a decent result. M. Allamanis, H. Peng, and C. Sutton[3] may be the first to take advantage of both convolutional network and attention to learn a mapping between method body and its name, which they call 'Extreme Summarization.' Their work enlightens the way of using new techniques to model the codes.

A. LeClair, S. Jiang, and C. McMillan[13] also take code as text and together with AST information as sequence input to get summary sequence output as another work mentioned before. Besides providing a way to distribute weight on code text and code structure information, they also provide a complete process description both in their dataset usage and engineering, interested researchers and practitioner could easily reconstruct the model and apply it. Wang K, Su Z. Blended[21] take method name prediction and semantic classification as the downstream task of their model. They creatively combine the static and dynamic analysis as the input

Table 5: Next method call token prediction top-5 accuracy(%) results of different settings on test data set.

max_context_length	Top-5 Accuracy(window_size=4)	Top-5 Accuracy(window_size=6)	Top-5 Accuracy(window_size=8)
0	54.28	63.46	65.34
1	68.74	77.40	80.56
2	69.27	76.63	78.36
3	71.99	78.54	81.78
4	72.96	78.60	81.61
5	73.29	78.60	81.57
6	73.50	77.90	81.75
7	**73.61**	**78.77**	**81.93**

sequence for the model, which makes the model quite precise even on a small number of source codes. However, their work depends on the compilation, which makes learning from open source code bases become too hard to apply in scale.

Code search, bug detection, auto fixing, and other similar functions have been improved by the advancement of machine learning for codes, Allamanis et al.[2] and Le, Triet H. M. et al.,[12] introduce many related techniques.

8.2 Code Completion

In this subsection, we will introduce some work out of traditional code completion methods. Wang, Yanlin, and Li, Hui's recent paper[22] proposes a quite new way to utilize information lie in AST for code completion. Instead of using the AST from top to bottom, they flatten the AST and take it as input, together with global attention and parent-child attention. A. Svyatkovskiy et al.[18] pay more attention to the memory-efficient performance of the neural code completion model. The difference is that they combine the static analysis results with the neural network, by which it will have even less noise to make predictions. F.Liu et al.[14] advance in using AST for code completion tasks. They use additional information from the hierarchical structure of AST, which the authors think will be complementary to pre-order traversal with the structure information of code snippets. At the same time, they apply a multi-task setting by training one model first to predict both the next token's type and value in order to get a more expressive embedding generator for code snippets.

The pioneers of code modeling have built up some companies focused on creating more advanced software tools for code completion and other useful functions. TabNine, aiXcoder, and Kite are part of these applications. Tools like them often support popular languages in famous IDEs and code editors. Some of these tools provide paid service for learning from the custom training dataset, which will give the users optimized models under their coding context.

9 CONCLUSION

In this paper, we proposed an action sequence representation approach for code inspired by the way developers named their methods and also inspired by the intrinsic similarity between human actions and method calls. The proposed approach has general applicability for high-level programming languages. At the same time, the efficiency of the proposed representation approach is suitable for large-scale code repository learning and software development tools to apply.

From the experiment results, the model trained with our approach performs 53.8%, 25.05%, 32.38% better than the same model trained with sequence extracted from AST of sequence window size set to 4, 6, 8 respectively, which shows the improvement in method call prediction task. The experiment results also demonstrate the context for code we proposed is a significant factor for NLP models to comprehend the codes.

We believe action sequence representation for code could be used in not only method call prediction task and not only through a few state-of-art machine learning models. With the advantages of generalization and efficiency, the knowledge of codes it can learn from the unlimited Internet resources is also unlimited.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. PMLR, 2091–2100.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[7] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 631–642.

[8] 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.

[9] Michael Hucka. 2018. Spiral: splitters for identifiers in source code files. *Journal of Open Source Software* 3, 24 (2018), 653. https://doi.org/10.21105/joss.00653

[10] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[11] 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.

[12] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38.

[13] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019*

IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 795–806.

[14] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 37–47.

[15] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. arXiv:cs/0205028 [cs.CL]

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[17] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. arXiv:1508.07909 [cs.CL]

[18] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[20] Jesse Vig. 2019. A Multiscale Visualization of Attention in the Transformer Model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics, Florence, Italy, 37–42. https://doi.org/10.18653/v1/P19-3007

[21] Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 121–134.

[22] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. *Proceedings of AAAIConference on Artificial Intellegence* (2021).

[23] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6