International Journal of Digital Crime and Forensics
Volume 15 • Issue 1

The Metric for Automatic Code Generation Based on Dynamic Abstract Syntax Tree

Wenjun Yao, Kunming University of Science and Technology, China
Ying Jiang, Kunming University of Science and Technology, China*
Yang Yang, Kunming University of Science and Technology, China

ABSTRACT

In order to improve the efficiency and quality of software development, automatic code generation technology is the current focus. The quality of the code generated by the automatic code generation technology is also an important issue. However, existing metrics for code automatic generation ignore that the programming process is a continuous dynamic changeable process. So the metric is a dynamic process. This article proposes a metric method based on dynamic abstract syntax tree (DAST). More specifically, the method first builds a DAST through the interaction in behavior information between the automatic code generation tool and programmer. Then the measurement contents are extracted on the DAST. Finally, the metric is completed with contents extracted. The experiment results show that the method can effectively realize the metrics of automatic code generation. Compared with the MAST method, the method in this article can improve the convergence speed by 80% when training the model, and can shorten the time-consuming by an average of 46% when doing the metric prediction.

KEYWORDS

Automatic Code Generation, Dynamic Abstract Syntax Tree, Extract Algorithm, Metric

INTRODUCTION

Automatic code generation technology is one of the means to improve the efficiency and quality of software development. Many researchers have studied technical means and improved automatic code generation implementation methods to enhance the quality of automatically generated code and achieve the purpose of meeting programmer's expectations. Therefore, automatic code generation technology has always been the core issue for practitioners and researchers (Hu et al., 2019).

In recent years, artificial intelligence technology has made great progress and development, which has further formed an important promotion for the research on automatic code generation technology. Automatic code generation technology has also achieved vigorous development (Yang et al., 2020). Part of the current research on automatic code generation technology has been applied to actual development. The automatic code generation tools implemented according to a certain code

DOI: 10.4018/IJDCF.325062 *Corresponding Author

generation method are usually embedded in the integrated development environment in the form of plugins. For example, integrated development environments such as IntelliJ IDEA, Eclipse, and PyCharm all support embedded code automatic generation plugins to help programmers improve development efficiency.

Among the number of automatic code generation technologies, the ability to evaluate the model includes the accuracy of predicting the next token, the MRR indicator used in the field of information retrieval, and so on. The evaluation indicators cannot be directly converted because of automated evaluation standards are different (Hu et al., 2019). Therefore, the existing research on automatic code generation technology focuses on the improvement of its model capability, ignoring the research on the quality measurement of its generated code. In addition, the existing research on software code quality measurement is based on the content of the written code, ignoring the programmer's programming behavior information. However, programming behavior information is a factor that cannot be ignored when doing the automatic code generation measurement. The programming behavior information is continuously and dynamically changing. So the automatic generation of code measurement is also a process of continuous dynamic changeable. Traditional software quality measurement methods are not suitable for code measurement generated by code automatic generation technology. Therefore, it is of great significance to study the metric method for the code generated by the automatic code generation technology.

To fully consider that the automatic code generation metric is a process of continuous dynamic changeable, this paper proposes a metric method for automatic code generation based on dynamic abstract syntax tree (DAST method). Specifically, the authors build a dynamic abstract syntax tree, and then extract metric-related content from the dynamic abstract syntax tree. Finally, the authors complete the metric for automatic code generation according to the extracted content. The experiment results show that the method can effectively realize the metrics of automatic code generation. Compared with the automatic generation metric method (Zhang, 2021) (MAST method) of which constructed by all programming codes and programming records, the method in this paper can improve the convergence speed by 80% when training the model, and can shorten the time-consuming by an average of 46% when doing the automatic code generation metric prediction.

The contributions of this paper are summarized as follows:

1. In this paper, the authors propose an algorithm to construct a Dynamic Abstract Syntax Tree (DAST) by combining programmer behavior and code generation tool behavior information. This algorithm effectively utilizes the cooperation behavior information between programmers and code generation tools in the programming process. A DAST reflects the dynamic changes of semantics, structure information and programming behavior information in a abstract syntax tree (AST), providing a basic platform for the measurement of code generated by code automatic generation tools.
2. The authors propose an algorithm to extract metric content through traversing the DAST. This algorithm effectively extracts the content which is deeply related with metric for automatic code generation.
3. The authors propose a method to build a metric model for automatic code generation based on code semantic, programmer behavior and behavior information of code automatic generation tools, and send the extracted content to this model and complete the measurement.

The rest of this article is organized as follows. First this article introduces related work on automatic code generation techniques and metric for automatic code generation. Secondly this article elaborates the construction and implementation of DAST. Thirdly this article elaborates algorithms for extracting metric content on DAST. Then this article elaborates the method of automatically generating metrics from code based on the extracted metrics content of DAST. After that the article shows the experimental setup and analyzes the results. Finally, this article concludes this article's work.

RELATED WORK

In the research work on evaluation metric for the automatic code generation methods, indicators such as accuracy, precision, recall, MRR (Mean Reciprocal Rank), and F-measure are usually used to evaluate the performance of automatic code generation methods (ZHAO et al., 2020).

Accuracy refers to the proportion of correctly predicted numbers in all sample sizes. In automatic code generation, the accuracy rate is defined as the ratio of the number of correctly generated codes to the number of all generated codes. Liu, F., et al (2022) used the accuracy to evaluate the generated code of proposed UMTLM (Unified Multi-Task learning based neural Language Model) method. When calculating the accuracy, the paper compute the top-1 accuracy on next token/AST node's value and type, i.e., the fraction of times the correct suggestion appears in the first of the predicted list. Experiments show that, in the AST-level and token-level code automatic generation tasks, the accuracy of the UMTLM model method is higher than that of PMN, LSTM, and Transformer-XL. Nguyen, A.T et al. (2022) used the accuracy to evaluate the generated code of the proposed ASTCC method. The accuracy in this paper is defined as the ratio between the number of hits over the total number of suggestions. In the task of Next-Statement Suggestion, compared with the AutoSC and PCC methods, the ASTCC method has achieved an improvement in accuracy. Lu, S et al. (2022) used the accuracy rate to evaluate the generated code of the proposed ReACC method (a Retrieval-Augmented Code Completion framework). In the token-level code generation task, the ReACC method has achieved higher accuracy than GPT-2, CodeGPT and other methods.

Precision is defined as the ratio of the number of codes correctly recommended by the automatic code generation tool to the total number of codes recommended by the automatic code generation tool. Rahman M et al. (2021) used the precision to evaluate the proposed BiLSTM method for automatic code generation. When calculating the precision in that paper, the paper uses the ratio of correct error classifications to total error classifications. Experiments show that BiLSTM method improves the precision rate of 98% by comparing with the automatic code generation methods of neural networks such as LSTM and RNN.

Recall is defined as the ratio between the number of correctly recommended codes by the code generation tool and the total number of recommended codes that programmers actually need. Bruch et al. (2009) used the recall to evaluate the BMN algorithm for automatic code generation. When calculating recall, that paper uses the ratio between the relevant (correct) recommendations made by the recommendation method for a given query and the total number of recommendations it should make. Experiments show that the recall of the BMN algorithm reaches 72%. Ding, Y., et al. (2022) used the recall to evaluate the COCOMIC framework for automatic code generation. Compared with the CodeGen method, the recall rate of the COCOMIC framework increased from 50.09% to 55.83%.

MRR is defined to mean reciprocal rank. Kim S et al. (2021) used the MRR to evaluate their proposed transformer method for the basic neural network framework to automatic generate code. When calculating MRR, that paper refers the indicator as a percentage number. The higher the correct token prediction position is, the higher the MRR value will be. By comparison, the automatic code generation method proposed by Kim S et al. improves the MRR index from 43.9% to 58% compared to the TravTrans method on the Deep3 model; on the Code2Seq model, the MRR index increases from 43.6% compared to the TravTrans method. to 58%. Yang, K., et al. (2022) use MRR to evaluate the code generated by their proposed CCGGNN method. This paper conclued that, comparing to the metric (Acc@1), MRR is closer to the realistic scenario when completion suggestions are presented to developers. This paper presentd MRR@5 as a percentage in task of various types and values of leaf token predictions. Experiments show that, the CCGGNN method achieved the highest MRR score compared to methods such as DEEP3. Liu, Y., et al. (2022) used the MRR to evaluate the code generated by approach, of which in two real-world datasets and find that sequence features are practically crucial for code completion. Compared to previous researches, the best model in this paper has a 10% improvement for the mean reciprocal rank (MRR) metric compared to previous researches.

F-measure is defined as the weighted harmonic mean of precision and recall. Allamanis M et al. (2021) evaluate the performance of a neural probabilistic language model specially designed for the method naming problem via F-Measure. When calculating the F1 score for the proposed level k > 1, the paper selects the precision, recall value of the level l ≤ k that leads to the highest F1 score. Experiments show that the evaluation of the model achieves a final F1 score of 60% in the prediction of the method name, and a F1 score of 55% in the prediction of the class name.

In the existing research on automatic code generation metrics, indicators such as validity, usability, reliability, normalization, maintainability, time efficiency, space efficiency, and assistance are usually used to measure the code generated by automatic code generation tool (Zhang, 2021).

Validity is defined as the degree that the automatic code generation tool produces the code to fit the programmer's expectation. That is, the degree to which the automatic code generation function can be realized under the predetermined environment. Usability is defined as how easily it is for programmers to obtain generated code in the process of using an automatic code generation tool. Reliability is referred to a degree of how reliable the generated code is in the context of the code. High-reliability code has a lower error rate. Normalization is referred to measure the degree of which generated code conforms to grammatical rules. Maintainability is defined as the degree to which programmers and code completion tools correct the results of automatic code generation in the process of generating code. Time efficiency is represented the efficiency of the corresponding processing time of the code automatic generation tool in the code automatic generation process. Space efficiency is represented the throughput rate of the code generation tool during the code generation process. Assistance is referred to the ease with which the automatic code generation tool assists the programmer in programming. Zhang (2021) constructed the MAST feature tree, and used the TBCNN (Tree-based Convolutional Network) (Mou et al., 2014) neural network to establish three dimensions of convolution extraction on MAST, including syntax, semantics, and automatic code generation information. That work built a metric model for codes which is generated by automatic code generation tools. The experimental results show that the metric model can effectively capture the characteristics of the impact of the interaction behavior information between programmers and automatic code generation tool. However, the model extracts the convolutional features of the entire MAST, which not only ignores the feature information of AST changes in the programming process, but also takes a long time to metric.

To sum up, in the current research on automatic code generation, there is a lack of a unified method for the metric on the code generated by automatic code generation tool. Most of the research on automatic code generation methods usually use indicators such as precision, recall, and F1-Measure to measure the performance of automatic code generation methods. But these indicators ignore the metric for the code generated by automatic code generation tool. During actual development, the programming process is a dynamic and continuous process. Programming behavior information is a factor that must be considered in the automatic code generation measurement. Differing from traditional code quality metrics analysis methods, the metric for automatic code generation is a dynamic and continuous process. Therefore, the metric for automatic code generation is an urgent problem to be solved.

This paper comprehensively considers that the metric for automatic code generation is a dynamic and continuous process. So this paper proposes a metric method for automatic code generation based on dynamic abstract syntax tree. Specifically, through the interact behavior information between the programmer and the code automatic generation tool (Zhang & Jiang, 2021), the authors build a dynamic abstract syntax tree, and then extract metric-related content from the dynamic abstract syntax tree. Finally, the authors complete the metric for automatic code generation according to the extracted content.

BUILDING A DAST

In this section, the authors elaborate the method for the construction and implementation of DAST. This method fully combines the mutual cooperation behavior information between programmers

and code automatic generation tools. This method feeds back the dynamic changes of programming behavior information to AST, thus building the basis platform for automatic code generation metric.

According to previous studies, a large number of researchers use AST to represent a program (Li et al., 2017), and this data structure can reflect the structural and semantic information of code content. When programmers are coding with the help of automatic code generation tools, the content of the code is constantly increasing, of that corresponding AST is also constantly growing. This process is a dynamic and continuous process. The authors employ an open-source Python package called ast to parse our Python source code into ASTs.

Since the traditional ASTs just include syntax information on every token, they lack the information to reflect the changeable coding process, such as time information and operation information. These information are important to construct DASTs. In order to solve this problem, the authors draw on and modifies the behavior capture plugin tool of programmers and code automatic generation tools which is published and opened source by X zhang, Y Jiang et al. (2019). The plugin tool can be obtained from https://github.com/xiaojiangzhang/Generate-the-code-acquisition-plugins. The plugin tool can not only capture the behavior information of programmers, but also capture information of automatic code generation tool.

In order to build a DAST, the information attribute of the tokens on the AST must be expanded. The authors define that the information attribute of token consists of type value, source, operation and time. As is shown in Table 1.

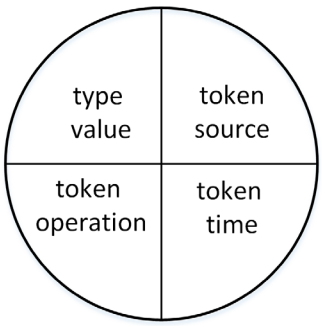
Figure 1 illustrates the attribute state of token.

As Figure 1 shows, every token on a DAST gets 4 attributes. The attribute of type value is obtained by the code contents. The attribute of token source is used to distinguish the token whether comes from programmer or automatic code generation tool. When the code is completely typed by the programmer, the token source attribute of this token is "programmer", which is abbreviated as "PG". When the code comes from the recommendation of automatic code generation tool, the token source attribute of this token is "tool". The attribute of token operation is derived from the appearance

Table 1. Information dimension of token

Name	Attribute	Description
Information dimension	type value	Type value in token
	token source	Distinguish whether the token is typed by the programmer individual or recommended by an automatic code generation tool
	token operation	Addition, deletion and modification etc. operation in token
	token time	Time in token operation

Figure 1. Token's attribute state



style of code. When the code first time appears, the attribute of token source is "addition", which is abbreviated as "ADD". When the code is modified by the programmer, the attribute of this token source is "modification", which is abbreviated as "MOD". The attribute of token time is the time that the token's operation attribute is changed.

Then A DAST can be represented by a two-tuple {V, E}, where V represents the token data set, token V_i=<$value_i$,$source_i$,$operation_i$,$time_i$>, indicating that token V_i owns attributes, such as the type value, token source, token operation and token time. E represents the set of connection relationships between tokens. E = {<m,n> | m,n ∈V and p(m,n), <m,n> represents the connection edge from token m to token n. p(m,n) defines the connection property of the connection edge <m,n>}. The assignment of p(m,n) is as follows:

$$p(m,n) = \begin{cases} Addition\ token\ connection\ edge\ EAmn,\ n\ is\ a\ newly\ added\ token \\ Modification\ token\ connection\ edge\ EMmn,\ n\ is\ a\ modified\ token \end{cases}$$

Algorithm 1 describes a method for building a DAST. Its input is a standard of abstract syntax tree. Its output is a DAST.

In the algorithm 1, two while nested loops are designed. The second while loop is used to determine that if the programmer does not press the Enter key, the new captured code will be parsed into a subtree. Then this subtree is linked to the baseTree. After that, baseTree got a new state. This new state is updated to the baseTree. When the programmer stops programming, the baseTree is updated to DAST. Finally the DAST is returned. The time complexity of the algorithm for building a DAST is O(n2). Figure 2 and Figure 3 use a simple example to illustrate how a DAST is built.

First in time t1, a programmer types the codes as shown in Figure 2(a). These codes are parsed into a benchmark tree baseTree, as shown in Figure 3(a). Then in time t2, the programmer types the new codes "k = 2", as shown in Figure2 (b). The new codes are parsed into a subtree. Since the new code is in a parallel relationship with codes "i=1", the subtree is linked as the child node of token "While". Besides, for the sake of the operation in token "While" is addition, the connecting line between token "While" and token "j" is a solid line. This process is shown in Figure 3(b). Finally in time t3, the programmer modifies the code "while" into "if", as shown in Figure2 (c). The modified code "if" is parsed into a new subtree. Since the code's operation attribute is modification, the new subtree is linked to token "While" with a dotted line. That is, the dotted edge connecting token "While" and token "If" is shown in Figure 3(c).

EXTRACTING METRIC CONTENT ON A DAST

Since the process for coding is dynamic and continuous changeable, the metric for automatic code generation is a dynamic and continuous process. Building DASTs can represents all the process of dynamic change of program semantics within a certain time range and all the behavior information of the programmer and the automatic code generation tool. But the metric for automatic code generation must be set in a certain time to metric. Applying DASTs directly could not complete the metric for automatic code generation. It is necessary for the metric to extract content on a DAST.

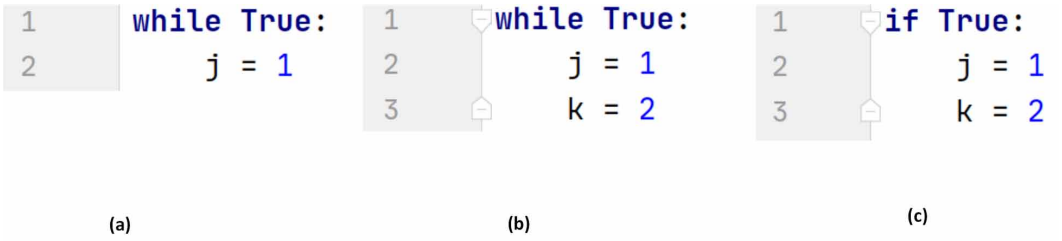
This section will introduce in detail the method of extracting content on a DAST for automatic code generation metric. This method first defines the extraction rules on a DAST, and then traverses DAST to extract content for automatic code generation metric.

In the process of automatic code generation, sometimes the code automatic generation tool generates logical code, but does not meet the programmer's expectations. So the validity in this code turns to be low when be used. Therefore, a measure of the validity of automatic code generation is related to the degree to which the generated code conforms to the programmer's expectations. The

Algorithm 1. Building a DAST

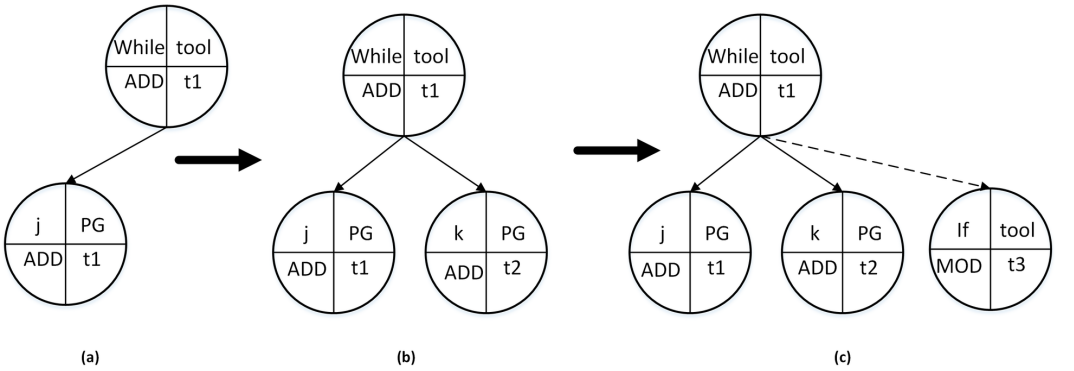
Input: A base abstract syntax tree namely baseTree
Output: A DAST
1:while Programmer does not stop programming do
2: while True do
3: if Programmer does not press enter key then
4: Capture the code typed by the programmer (codeP) and behavior information (Pinfo) = {Ptime, Psource, Poperation, Pscope}
5: Capture the code which is recommended by the automatic code generation tool (codeT) and then selected by programmer, and tool's behavior information (Tinfo)
6: Merge codeP and codeT into code = {codeP, codeT}
7: Merge Pinfo and Tinfo into info = {time,source,operation,scope}
8: else
9: Parse code into sub-AST, namely subTree
10: Find the Pnode node on the baseTree according to the scope
11: if operation == "addition" then
12: Link subTree as child node of Pnode
13: else
14: Link subTree as shadow node of Pnode
15: end if
16: Update the AST of the current moment to baseTree
17: break
18: end if
19: end while
20: end while
21: DAST = baseTree
22: return DAST

Figure 2. A sample for coding process



more codes an automatic code generation tool can generate that meets the programmer's expectations, the more valid the snippet will be. In the programming process, the valid generated variable code, function definition code, and class definition code usually meet the programmer's expectations. So these codes are extracted as the measurement reference of the validity index.

Figure 3. A sample to build a DAST



In the process of mutual cooperation between the programmer and the automatic code generation tool, the programmer enters a number of characters. And the automatic code generation tool predicts the variable name, function name, class name and code fragment in the code fragment to be generated according to the characters typed by the programmer. The automatic code generation tool generates a list of recommended codes for programmers. In this process, if the programmer can perform fewer operations, such as inputting shorter characters, the automatic code generation tool can generate the recommended code list. The programmer does not have to perform frequent complicated operations on the recommended code list, such as typing up and down keys on the keyboard to select the recommended code. The generated code is easier to be obtained for the programmers. Therefore, the generated code that is easy to be obtained by programmers can be extracted as a measurement reference for usability indicators.

The type of recommended code determines how reliable it is in the context of the code after programmers select the recommended code from the code generation tool. Generated code with high reliability exhibits a lower error rate. Codes with judgment branches, loop branches or fragments of function definitions in the programming language usually bring potential errors. Because these code snippets contain 2 or more branch codes. And the fewer branches of a code snippet, the lower the error rate. Therefore, the judgment branch, loop branch or function definition fragment in the generated code greatly affects the reliability of the generated code.

In the code context, the thing that whether the generated code conforms to the program language syntax determines the generated code is normalization or not. The higher normalized with code generated by automatic code generation tool, the higher the quality of this generated code. Therefore, the degree to which the generated code conforms to the program language syntax determines its normalization.

Sometimes programmers would make some changes to the results that the code generated by automatic code generation tool. The difficulty in maintaining the code generated process affects the quality of the code generated to a certain extent. The judgment branch, loop branch or program statement with complex syntax is often not easy to be modified, and it takes longer than coding work as usual. Therefore, the judgment branch, loop branch or program statement with complex syntax directly affects the correction degree of in generated code, which is related to the maintainability in the generated code.

Automatic code generation tool produces code recommendations in the list based on the semantic environment of the code context and the programmer's typing characters. The time it takes programmers to select recommended code reflects the rate at which code is generated in the code context. The speed of code generation directly affects the development efficiency of programmers. So the rate at which code is generated determines the measurement result of the time efficiency indicator.

Throughput refers to the space occupied by the generated code during the automatic code generation process. The larger the generated code, the higher the throughput. The space efficiency is directly related to the size of the throughput rate. Therefore, the throughput rate is selected as the space efficiency index measurement reference for the generated code.

In the process of using the automatic code generation tool, assistance indicates the improvement of the scale of generated code to the development speed of programmers. In actual development, the size of code generated according to the code context will greatly improve the development speed of programmers. Therefore, the code generated by the automatic code generation tool directly affects the development speed of programmers.

Generally, the values of the above metrics indicators are usually set the number between 0 to 1. The closer the value of metric indicator gets to number 0, the less obviously the result of the metric indicator for automatic code generation performs. The closer the value of metric indicator gets to number 1, the more obviously the result of the metric indicator for automatic code generation performs.

In summary, this paper defines the extraction rules of automatic code generation metrics on a DAST, as shown in Table 2.

Algorithm 2 describes methods for extracting metric content on a DAST. Its input is a DAST and the output is a sequence of tokens.

In the Algorithm 2, sentence 2 to sentence 9 illustrates a breadth-first manner to traverse on a DAST for finding the tokens within the evaluation time range. Sentence 11 to sentence 41determine whether the accessed tokens conform to the rules. If it is, it will be extracted, and if it does not match, it will continue to access and traverse. Finally, the extracted sequence is returned. The time complexity of the content extraction algorithm on the DAST is O(n2). Figure 4 uses a simple example to illustrate how metric content extraction is done on a DAST.

First, according to the set measurement time range, the token “For” is obtained through the breadth-first traversal method and assign it to Pnode, which its time information is within the metric time range and is first be visited. Then, starting from the Pnode, the depth-first pre-order traversal method is used to determine the sequence of visited nodes. Judging each node accessed whether meets any one of the extracting rules in Table 2. If it meets, then it is extracted. For example, as shown in Figure3(c), although the time in token “j” is within the metric time range, token “j” does not meet any one of the extraction rules in Table 2. So it would not be extracted. Finally, the extracted tokens are combined as the way with their connection on the DAST. In the Figure 4, token “k” and “return” do meet the extracting rule, and the token “if” is not within the metric time range. So the final extracted token sequence is: “for, while, return, k”.

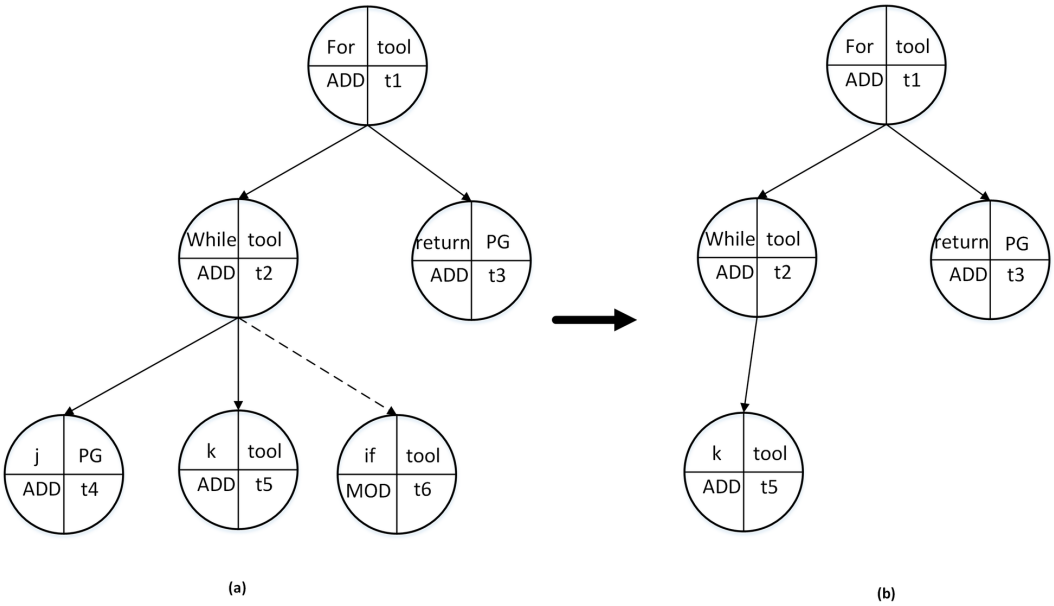
Table 2. Extraction rules on a DAST

Indicator	Extraction rule
validity	token of valid generated variable name, function definition name, and class definition name.
usability	token generated that requires less programmer operations.
reliability	token generated for judgment, loop and function definition.
normalization	token generated for function definitions and class definition that conform to program language syntax.
maintainability	token generated for judgments, loop and their child token.
time efficiency	token generated of which the generation takes less time.
space efficiency	token generated occupied much space.
assistance	tokens generated of which takes the large scale

Algorithm 2. Extract metric content on a DAST

Input: A DAST
Output: A sequence of tokens.
1: set start time, end time for metric, initialize queue=[DAST], stack, sequence
2: while queue not empty do
3: pop the top element of the queue and assign it to node
4: if the time information dimension of node is in the interval [start time, end time] then
5: assign node to Pnode,break
6: else
7: push the node's child nodes into the queue in turn
8: end if
9: end while
10:assign Pnode to sequence and push it to the stack
11: while stack not empty or Pnode != null do
12: while Pnode != null do
13: if Pnode is not leaf node then
14: update the first child node of Pnode to Pnode and push it into the stack
15: else
16: Pnode = null
17: end if
18: end while
19: while True do
20: pop the top element of the stack and assign it to Pnode
21: if Pnode meets any one of the extraction rules in Table 2 and the node time information is within the metric time range then
22: put the Pnode into the sequence
23: if Pnode has sibling node then
24: Assign the sibling node of Pnode to Pnode and push it to the stack,break
25: else
26: if stack is empty then
27: Pnode = null,break
28: end if
29: end if
30: else
31: if Pnode has sibling node then
32: Assign the sibling node of Pnode to Pnode and push it to the stack, break
33: else
34: if stack is empty then
35: Pnode = null, break
36: end if
37: end if
38: end if
39: end while
40: end while
41: return sequence

Figure 4. A sample to extracting content on a DAST



AUTOMATIC CODE GENERATION METRIC

In this paper, the automatic code generation metric model constructed by the TBCNN deep learning method is used to complete the metric for the code generated by automatic code generation tool. The metric model based on TBCNN (Zhang, 2021) is mainly divided into four parts. The first part converts the processed token tag vector into a fixed-length high-dimensional vector. The second part is information capturing. The key information of the input sequence is obtained through the TBCNN deep learning network. The third part is code representation. The extracted features are synthesized to form a vector representing the features of the generated code model. The fourth part is the output layer. Automatic code generation predictions for metric is based on characterization vector output. The authors sent the content extracted into this metric model and finally get the metric result for automatic code generation.

EXPERIMENT

This section will conduct an empirical study on the effectiveness of metric method for automatic code generation based on dynamic abstract syntax tree. The performance of the method will be considered from various sides. The data sets, performance evaluation indicators, experimental procedures and parameter settings used in the experimental research will be introduced. In the experiment, tensorflow frame is selected to build the TBCNN deep learning model, which runs under the Windows 10 operating system, i7-9850H, 2.60GHz processor, and 32G memory.

Research Question

The purpose of metric method for automatic code generation based on dynamic abstract syntax tree is to complete the code quality measurement generated by the automatic code generation technology. In order to verify the effectiveness of the metric method for automatic code generation based on dynamic abstract syntax tree, two research questions (RQs) are need to be answered as follows:

RQ1: Compared with the metric for automatic code generation built by the MAST method, do the DAST method get better performance?

RQ2: How reliable is the DAST method when it is used to measure the code generated by automatic code generation tool?

Experiment Dataset

In this paper, the plugin tool capturing programmer behavior and code automatic generation tool behavior is opened source by Zhang et al. (2019). The authors capture nearly 82,000 programing behavior information as a training set. In addition, the authors type 12 files with less than 100 lines of code (id ranges 1.1 to 1.12), 12 files with 100 to 200 lines of code (id ranges 2.1 to 2.12), and 12 files with more than 200 lines of code (id ranges 3.1 to 3.12) with the help of automatic code generation tool. In this process, the authors use plugin tool to capture programmer behavior and code automatic generation tool behavior as a test dataset.

The features in test dataset are shown in Table 3.

Although typing 12 files with different class of lines, every behavior information in typing 12 files corresponds to the same 12 kinds of metric indicator combinations. Among the 12 kinds of metric indicator combinations, 5 of them are set to meet the extraction rules in Table 2, the other 5 combinations are set to not meet the extraction rules for every indicator. The last 2 metric indicator combinations are set to no indicator meet the extraction rules and all indicators meet the extraction rules. As is shown in Table 4.

The id in Table 3 and Table 4 contains all the ids of files with 3 different levels of code lines. Specifically, the feature whose id is x (ranges 1 to 12) in Table 3 represents three files 1.x, 2.x, and 3.x with different levels of code lines. For example, the feature file with an id of 5 in Table 3 indicates an id of 1.5 in a file with less than 100 lines of code, an id of 2.5 in a file with with 100 to 200 lines of code, and an id of 3.5 in a file with more than 200 lines of code.

Table 3. The features in test dataset

ID	Proportion for Valid Variable Name	Quantity in Programmer Operation	Nested Statement	Time for Programmer Operations	Proportion for Tokens Generated by Tool in Codes
1	low	small	more	less	low
2	low	large	more	more	low
3	low	small	more	more	high
4	low	large	more	less	high
5	low	small	more	less	high
6	high	large	less	more	high
7	high	small	less	less	high
8	high	large	less	less	low
9	high	small	less	more	low
10	high	large	less	more	low
11	low	small	less	less	low
12	high	large	more	more	high

Table 4. Metric indicator combinations

ID	Validity	Usability	Reliability	Normalization	Maintainability	Time Efficiency	Space Efficiency	Assistance
1	no meet	no meet	meet	no meet	meet	no meet	no meet	meet
2	no meet	meet	meet	no meet	meet	meet	no meet	no meet
3	no meet	no meet	meet	no meet	meet	meet	meet	meet
4	no meet	meet	meet	no meet	meet	no meet	meet	meet
5	no meet	no meet	meet	no meet	meet	no meet	meet	no meet
6	meet	meet	no meet	meet	no meet	meet	meet	no meet
7	meet	no meet	no meet	meet	no meet	no meet	meet	meet
8	meet	meet	no meet	meet	no meet	no meet	no meet	no meet
9	meet	no meet	no meet	meet	no meet	meet	no meet	no meet
10	meet	meet	no meet	meet	no meet	meet	no meet	meet
11	no meet	no meet	no meet	no meet	no meet	no meet	no meet	no meet
12	meet	meet	meet	meet	meet	meet	meet	meet

Evaluation Metric for Performance

Evaluating the performance of a deep learning method can use model performance metrics and model generalization capabilities. The automatic code generation metric method based on dynamic abstract syntax tree in this paper uses the deep learning method of TBCNN. So the convergence times of the loss function of the training model and the time spent in the automatic code generation metric prediction are selected as the performance comparison of the model. The difference value of the metrics indicator is analyzed as the generalization ability of the model.

Loss function refers a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. The authors choose the square loss function as the loss function:

$$Square\ \ loss : L\left(Y, f\left(X\right)\right) = \left(Y - f\left(X\right)\right)^2$$

where X and Y are the variables defined on the input space χ and output space γ, and f is the decision function.

Experiment Result

When measuring the metric for automatic code generation, the authors choose the MAST method to compared with our DAST method.

First, nearly 82,000 programming behavior information records were used as the training set to train MAST method model and DAST method model. Then, the authors send all the test dataset to MAST model and DAST model. Finally, the authors get the predictions of results with different metric method model.

Table 5 lists the comparison of the times of iterations of the MAST method training code automatic generation metric model and the DAST method training model loss function.

Table 5. Comparison model training performance

Method	Times of iterations	MSE
MAST method	750	0.01
DAST method	150	0.0062

Table 5 shows that using the MAST method requires 4 times than using DAST method for iterations before the training model converges. Besides, the MSE value of the MAST method is greater than the DAST method.

Table 6 to Table 8 list behavior information captured by typing 12 kinds of code files at 3 different code line levels are sent to MAST method model and DAST method model to predict the results of automatic code generation metric.

Table 6 lists the measurement results of automatic code generation for coding within 100 lines of code. Each serial number in the table represents the comparative experimental group number of each group using the MAST method and the DAST method to do the automatic code generation measurement. The last column represents time cost for doing the metric predictions. Table 6 shows the values of 8 indicators predicted by MAST method and DAST method respectively when doing code automatic generation metrics. In terms of time cost, the time used by the MAST method is in the range of 0.021 seconds to 0.178 seconds, and the time used by the DAST method is in the range of 0.023 seconds to 0.103 seconds.

Table 6. Result of metric for automatic code generation by typing less than 100 lines of code

ID	Method	Validity	Usability	Reliability	Normalization	Maintainability	Time Efficiency	Space Efficiency	Assistance	Time Cost (Seconds)
1.1	MAST method	0.24	0.23	0.6	0.28	0.63	0.21	0.29	0.69	0.067
	DAST method	0.14	0.17	0.56	0.2	0.52	0.11	0.2	0.62	0.04
1.2	MAST method	0.27	0.6	0.69	0.29	0.66	0.63	0.23	0.25	0.069
	DAST method	0.18	0.51	0.63	0.2	0.55	0.62	0.13	0.21	0.037
1.3	MAST method	0.29	0.25	0.66	0.29	0.67	0.6	0.64	0.7	0.024
	DAST method	0.22	0.19	0.57	0.22	0.65	0.56	0.56	0.63	0.025
1.4	MAST method	0.3	0.62	0.69	0.27	0.61	0.28	0.66	0.61	0.084
	DAST method	0.2	0.57	0.59	0.18	0.58	0.22	0.57	0.61	0.043
1.5	MAST method	0.22	0.26	0.67	0.21	0.69	0.23	0.61	0.25	0.153
	DAST method	0.19	0.23	0.63	0.2	0.61	0.15	0.51	0.17	0.053
1.6	MAST method	0.67	0.66	0.24	0.63	0.21	0.69	0.66	0.22	0.178
	DAST method	0.65	0.59	0.17	0.6	0.15	0.61	0.65	0.19	0.103
1.7	MAST method	0.62	0.27	0.29	0.69	0.23	0.28	0.66	0.66	0.021
	DAST method	0.54	0.23	0.26	0.62	0.13	0.2	0.63	0.62	0.023
1.8	MAST method	0.63	0.68	0.22	0.69	0.26	0.21	0.21	0.28	0.063
	DAST method	0.53	0.6	0.19	0.59	0.19	0.16	0.18	0.2	0.037
1.9	MAST method	0.63	0.29	0.26	0.65	0.23	0.67	0.29	0.22	0.151
	DAST method	0.56	0.23	0.21	0.54	0.2	0.56	0.22	0.18	0.067
1.10	MAST method	0.66	0.7	0.28	0.61	0.21	0.66	0.26	0.64	0.116
	DAST method	0.63	0.65	0.18	0.51	0.16	0.64	0.22	0.61	0.069
1.11	MAST method	0.27	0.23	0.27	0.2	0.21	0.25	0.22	0.24	0.137
	DAST method	0.19	0.19	0.24	0.19	0.17	0.14	0.12	0.23	0.067
1.12	MAST method	0.69	0.62	0.63	0.66	0.63	0.65	0.7	0.65	0.068
	DAST method	0.63	0.56	0.54	0.56	0.61	0.57	0.68	0.55	0.034

Additionally, in Table 6, there is little difference between the indicator values predicted by the DAST method and the MAST method. In the same group of comparative experiments, the predicted value of a certain indicator by the DAST method and the MAST method showed a trend of being both high or low. For example, in the first group of comparative experiments with id 1.1, the difference values of the eight indicators predicted by the MAST method and the DAST method are all within 0.1, indicating that the differences are not large. On the other hand, the metric values predicted by the two methods all appear to be high or low together.

Table 7 lists the metric results of automatic code generation when typing 100 to 200 lines of code. The table shows the values of 8 indicators predicted by MAST method and DAST method respectively when doing code automatic generation metric. In terms of time cost, the time used by the MAST method is in the range of 0.22 seconds to 0.69 seconds; the time used by the DAST method is in the range of 0.064 seconds to 0.428 seconds.

Besides, in Table 7, the measurement values of indicators predicted by DAST method and MAST method are not significantly different. In the comparative experiment of the same group, the predicted values in indicators of DAST method and MAST method for a certain indicator are both high or low at the same time. For example, in the fourth group of comparative experiments with id=2.4, the difference between the eight indicator values predicted by the MAST method and the DAST method is within 0.1, indicating that the difference is small. On the other hand, the indicator values predicted by the two methods are both high or low at the same time.

Table 8 lists the metric results of automatic code generation when typing more than 200 lines of code. The table shows the values of 8 indicators predicted by MAST method and DAST method

Table 7. Result of metric for automatic code generation by typing 100 to 200 lines of code

ID	Method	Validity	Usability	Reliability	Normalization	Maintainability	Time Efficiency	Space Efficiency	Assistance	Time Cost (Seconds)
2.1	MAST method	0.3	0.24	0.62	0.21	0.69	0.27	0.2	0.61	0.349
	DAST method	0.24	0.14	0.61	0.2	0.66	0.23	0.16	0.6	0.243
2.2	MAST method	0.27	0.63	0.64	0.23	0.66	0.64	0.28	0.22	0.401
	DAST method	0.2	0.56	0.58	0.12	0.62	0.53	0.22	0.12	0.221
2.3	MAST method	0.24	0.28	0.63	0.28	0.67	0.65	0.62	0.63	0.517
	DAST method	0.22	0.26	0.52	0.17	0.63	0.61	0.57	0.54	0.297
2.4	MAST method	0.29	0.64	0.62	0.25	0.69	0.26	0.6	0.68	0.575
	DAST method	0.28	0.54	0.58	0.21	0.69	0.2	0.58	0.62	0.379
2.5	MAST method	0.24	0.22	0.67	0.24	0.64	0.25	0.7	0.28	0.69
	DAST method	0.19	0.16	0.57	0.14	0.61	0.18	0.68	0.2	0.428
2.6	MAST method	0.67	0.65	0.25	0.66	0.3	0.65	0.64	0.22	0.22
	DAST method	0.59	0.64	0.2	0.62	0.25	0.62	0.58	0.19	0.079
2.7	MAST method	0.62	0.21	0.2	0.65	0.22	0.23	0.68	0.69	0.298
	DAST method	0.61	0.16	0.11	0.6	0.11	0.23	0.68	0.59	0.125
2.8	MAST method	0.7	0.65	0.25	0.61	0.28	0.28	0.22	0.3	0.362
	DAST method	0.64	0.64	0.2	0.55	0.23	0.23	0.17	0.24	0.349
2.9	MAST method	0.67	0.2	0.23	0.68	0.23	0.62	0.28	0.21	0.339
	DAST method	0.62	0.13	0.18	0.65	0.15	0.54	0.27	0.19	0.17
2.10	MAST method	0.67	0.61	0.28	0.67	0.3	0.69	0.22	0.63	0.253
	DAST method	0.57	0.54	0.23	0.64	0.25	0.65	0.12	0.6	0.064
2.11	MAST method	0.25	0.29	0.25	0.24	0.3	0.24	0.26	0.23	0.344
	DAST method	0.23	0.2	0.2	0.22	0.24	0.23	0.18	0.13	0.168
2.12	MAST method	0.69	0.62	0.64	0.66	0.69	0.67	0.69	0.6	0.6
	DAST method	0.59	0.56	0.62	0.56	0.64	0.59	0.67	0.5	0.284

Table 8. Result of metric for automatic code generation by typing more than 200 lines of code

ID	Method	Validity	Usability	Reliability	Normalization	Maintainability	Time Efficiency	Space Efficiency	Assistance	Time Cost (Seconds)
3.1	MAST method	0.2	0.26	0.64	0.25	0.67	0.25	0.21	0.67	2.551
	DAST method	0.19	0.24	0.53	0.23	0.57	0.24	0.13	0.65	0.82
3.2	MAST method	0.28	0.68	0.7	0.28	0.65	0.63	0.29	0.24	1.348
	DAST method	0.27	0.58	0.7	0.26	0.59	0.59	0.26	0.15	0.979
3.3	MAST method	0.26	0.25	0.63	0.21	0.61	0.64	0.64	0.7	0.777
	DAST method	0.22	0.23	0.55	0.16	0.53	0.57	0.59	0.62	0.332
3.4	MAST method	0.26	0.68	0.63	0.21	0.63	0.27	0.69	0.61	1.194
	DAST method	0.23	0.65	0.54	0.15	0.59	0.25	0.6	0.52	0.648
3.5	MAST method	0.29	0.29	0.66	0.28	0.65	0.23	0.7	0.26	0.965
	DAST method	0.23	0.21	0.6	0.26	0.65	0.18	0.64	0.26	0.597
3.6	MAST method	0.64	0.68	0.24	0.64	0.26	0.63	0.69	0.23	0.865
	DAST method	0.62	0.63	0.19	0.54	0.23	0.54	0.64	0.15	0.389
3.7	MAST method	0.6	0.2	0.26	0.65	0.22	0.23	0.68	0.65	0.875
	DAST method	0.57	0.1	0.23	0.61	0.21	0.2	0.67	0.6	0.177
3.8	MAST method	0.65	0.64	0.23	0.6	0.25	0.29	0.25	0.22	1.617
	DAST method	0.61	0.55	0.23	0.55	0.19	0.25	0.21	0.11	0.765
3.9	MAST method	0.62	0.29	0.3	0.66	0.29	0.61	0.23	0.3	1.472
	DAST method	0.59	0.24	0.26	0.55	0.26	0.55	0.13	0.28	0.718
3.10	MAST method	0.63	0.69	0.27	0.67	0.21	0.67	0.25	0.69	0.743
	DAST method	0.6	0.68	0.22	0.63	0.15	0.64	0.15	0.64	0.156
3.11	MAST method	0.25	0.27	0.25	0.2	0.28	0.3	0.23	0.27	0.776
	DAST method	0.23	0.18	0.17	0.19	0.21	0.24	0.19	0.23	0.273
3.12	MAST method	0.68	0.66	0.78	0.71	0.74	0.7	0.71	0.71	1.512
	DAST method	0.65	0.63	0.67	0.61	0.65	0.67	0.64	0.69	1.076

respectively when doing code automatic generation metric. In terms of time cost, the time used by the MAST method is in the range of 0.743 seconds to 2.551 seconds; the time used by the DAST method is in the range of 0.177 seconds to 1.076 seconds.

Furthermore, in Table 8, there is little difference between the indicators predicted by the DAST method and the MAST method. In the same group of comparative experiments, both DAST method and MAST method show a tendency of both high and low predicted values of an indicator. For example, in the tenth group of comparison experiment with id 3.10, the difference between the eight measurement index values predicted by the MAST method and the DAST method is all within 0.1, indicating that the difference is not large. On the other hand, the values of measurement indexes predicted by the two methods are both high or low.

Analysis

1. Comparison between DAST method and MAST method when doing automatic code generation metric (RQ1).

For the comparison of training model performance, the authors select nearly 82,000 programming behavior information record data as the training set to train the metric model. Table 5 lists the training results of the MAST method and the DAST method to train automatic code generating metric model. In Table 5 the loss function converges in training MAST model after running 750 epochs, while the loss function converges in training DAST model after running 150 epochs. It shows that the DAST

method converges faster when training a model for automatic code generation metric. Therefore, the DAST method can improve the convergence speed by 80% compared with the MAST method in the training model. In addition, the MSE value of the DAST method training metric model is smaller, indicating that the DAST method training metric model has a higher degree of fit.

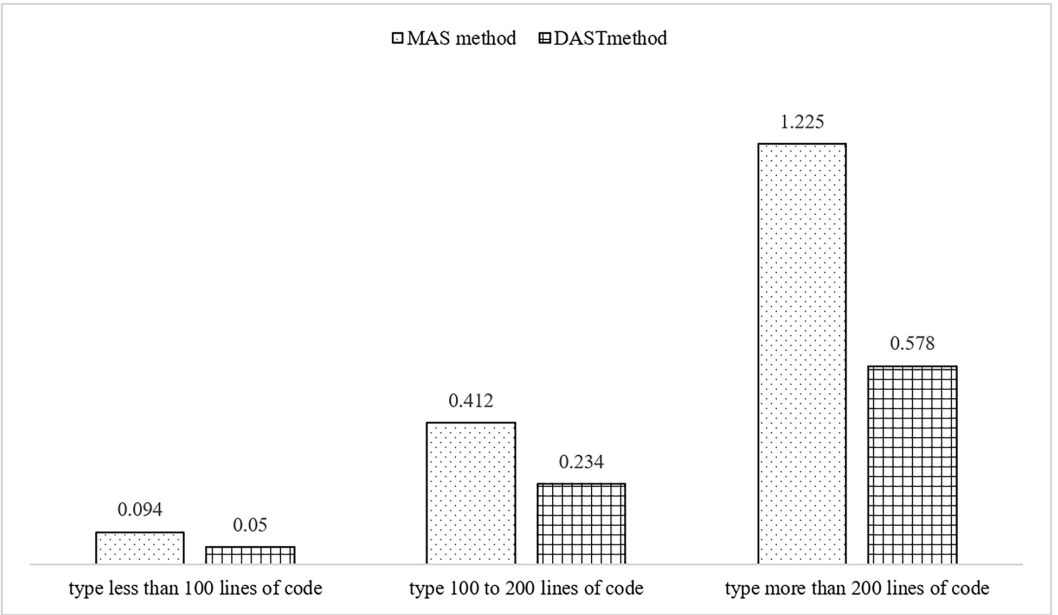
When using the trained automatic code generation metric model to predict the indicators, the authors select the test dataset described as section EXPERIMENT. The purpose of this is to prove that when a metric is automatically generated by code, at least when a certain metric exhibits a higher or lower value, there can be at least 5 other combinations of experiments to observe its value volatility. In addition, it illustrates that it is not by chance that the DAST metric is superior to the MAST metric.

According to Table 6 to Table 8, the average time cost in predict 12 groups of comparative experiments with 3 level of typing codes when using the MAST method and the DAST method for automatic code generation metrics is shown in Figure 5.

In Figure 5, the average time cost in DAST method is less 46% in MAST method when measure automatic code generation metric in the 3 levels of lines of code typed. Moreover, more lines of code typing, the average time cost reduction of using the DAST method for code automatic generation metrics is greater than that of using the MAST method.

But not in each group of comparative experiments, the time cost of DAST method is shorter than that of MAST method. In the control group that typed the content of code files with less than 100 lines as the automatic code generation metric, there were some cases where the DAST method metric model took longer than the MAST method measurement model. Taking Table 6 as an example, in the 3rd control group and the 7th control group experiment, the DAST method took longer to measure the model metric than the MAST method. The reason is that in the 3rd and 7th groups in Table 6, the code typed by the programmer and the automatic code generation tool does not exceed 20 lines. But the DAST method occupies most of the time spent in building a DAST. So the DAST method takes longer time than the MAST method when do the metric. In the control group experiment with more than 100 lines of automatic code generation, the DAST method takes an average of 46% less time than the MAST method to generate automatic code metric. It proves that the automatic code

Figure 5. The average time cost in predict indicators



generation metric model constructed by the DAST method has higher efficiency than MAST method when doing automatic code generation metric prediction.

On the other hand, as more and more lines of code are typed, the time required to do the automatic code generation metric prediction becomes higher and higher. For example, the prediction time required to automatically generate metrics for code written in 100 to 200 lines of code is longer than for code written in 100 lines or less. This may be due to the increasing width and depth of MAST and DAST as code typing increases.

To sum up, compared with the MAST method, the DAST method not only has better performance when training the code to automatic generation metric model, but also has better performance when doing the code automatic generation metric prediction.

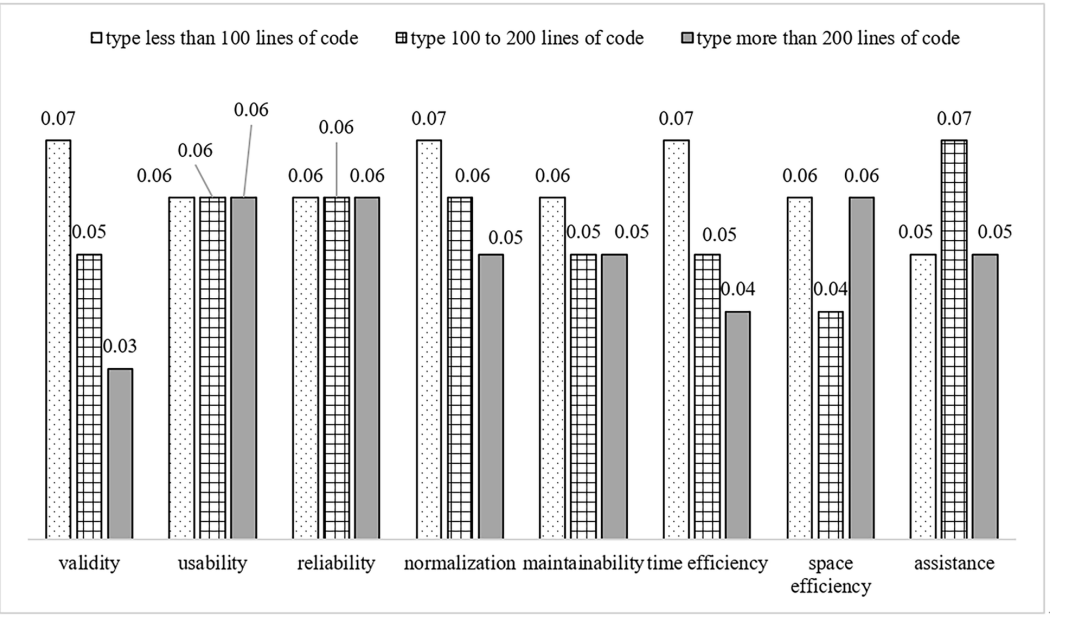
2. The reliable in DAST method when it is used to measure the code generated by automatic code generation tool (RQ2).

In order to analyze the results of the DAST method in the prediction of automatic code generation indicators are persuasive, the authors set up a comparative group experiment with 12 kinds of measurement indicators performance combinations. Each metric indicator gets at least 5 other combinations of experiments to observe whether meet the extracting rule in Table 2 or not. In addition, the authors also separately set metric indicator combinations of which no indicator meets the extraction rules and all indicators meet the extraction rules.

In Tables 6 to 8, the mean gap value in the predicted indicators when using the MAST method and the DAST method for automatic code generation metric is shown in Figure 6.

In Figure 6, the mean gap value in the automatic code generation metric between the DAST method and the MAST method using to predict 8 indicators lies 0.03 to 0.07. Furthermore, the mean gap value between the DAST method and the MAST method predicting the value of the same indicator is within 0.1. However, in some comparative experiments, such as the control experiment of the 10th group in Table 7, the gap value of the space efficiency indicator between the DAST method and the

Figure 6. The mean gap value in the predicted indicators



MAST method in the automatic code generation metric prediction reached 0.1. The reason for this phenomenon may be that some tree structures or token information were not taken into consideration when designing the DAST extraction rules (ie Table 2) for automatic code generation metrics. This led to the problem that the indicator value predicted by DAST method would loss some value.

Further analysis shows that in the experiment of the same comparative groups, the indicator value predicted by DAST method is smaller than that predicted by MAST method. The reason may be that the original tree structure of DAST is destroyed when extracting the content on DAST, resulting in the loss of measurement indicator value.

To sum up, the DAST method in Table 6 to 8 is close to or almost the same as most of the code automatic generation indicator values predicted by the MAST method, indicating that the indicator values predicted by the DAST method in code automatic generation metric have much degree of reliability.

CONCLUSION

In this paper, the authors propose a metric method for automatic code generation based on dynamic abstract syntax tree, of which comprehensively considers the behavior information of the interaction between programmers and automatic code generation tools in the process of automatic code generation. The metric method firstly builds the dynamic abstract syntax tree through the interaction in behavior information between the automatic code generation tool and programmer. Then it extracts the measurement contents on the dynamic abstract syntax tree for code automatic generation metric. Finally, code automatic generation metric is completed with measurement contents extracted. However, when extracting content on a AST, some tree structures or token information were not taken into consideration. It would miss some information related with the metric. Future work will further explore the association between tokens and extraction rules on a DAST. In addition, the work in this paper only verifies the generated code quality analysis of the Python code automatic generation tool, so the next step will apply the method to verify the generated code quality metric analysis of the Java code automatic generation tool.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China under Grant (Nos. 62162038, 61462049, 60703116 and 61063006), the National Key Research and Development Program of China (2018YFB1003904), Key Project of Yunnan Applied Basic Research (2017FA033), and Open Foundation of Yunnan Key Laboratory of Computer Technology Application (2020101).

REFERENCES

Bo, Y., Neng, Z., Li, S., & Xin, X. (2020). A Survey of Smart Code Completion Research. *Journal of Software*, *31*(5), 1435–1453.

Bruch, M., Monperrus, M., & Mezini, M. Learning from Examples to Improve Code Completion Systems. in Joint Meeting of the European Software Engineering Conference & the Acm Sigsoft Symposium on the Foundations of Software Engineering. 2009.

Ding, Y. (2022). CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. arXiv preprint arXiv:2212.10007.

Jian, L., He, P., Zhu, J., & Lyu Michael, R. (2017). Software defect prediction via convolutional neural network. *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE.

Liu, F., Li, G., Wei, B., Xia, X., Fu, Z., & Jin, Z. (2022). A unified multi-task learning model for AST-level and token-level code completion. *Empirical Software Engineering*, *27*(4), 1–38. doi:10.1007/s10664-022-10140-7

Liu, Y. (2022). Improving Code Completion by Sequence Features and Structural Features. *Proceedings of the 4th World Symposium on Software Engineering*.

Kim, S. Zhao, J., Tian, Y., & Chandra, S. (2021). Code prediction by feeding trees to transformers. in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE.

Lu, S. (2022). ReACC: A Retrieval-Augmented Code Completion Framework. arXiv preprint arXiv:2203.07722. 10.18653/v1/2022.acl-long.431

Miltiadis, A., Barr, E. T., Christian, B., & Charles, S. (2015). Suggesting accurate method and class names. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. IEEE.

Mou, L., Ge, L., Zhi, J., Lu, Z., & Tao, W. (2014). TBCNN: A tree-based convolutional neural network for programming language processing. arXiv preprint arXiv:1409.5718.

Nguyen, A. T., Yadavally, A., & Nguyen, T. N. (2022). Next Syntactic-Unit Code Completion and Applications. In 37th IEEE/ACM International Conference on Automated Software Engineering. IEEE. doi:10.1145/3551349.3559544

Rahman, M., Watanobe, Y., & Nakamura, K. (2021). A bidirectional LSTM language model for code evaluation and repair. *Symmetry*, *13*(2), 247. doi:10.3390/sym13020247

Xing, H., Ge, L., Fang, L., & Zhi, J. (2019). Research progress of program generation and completion technology based on deep learning. *Journal of Software*, *30*(5), 1206–1223.

Yang, K., Yu, H., Fan, G., Yang, X., & Huang, Z. (2022). A graph sequence neural architecture for code completion with semantic structure features. *Journal of Software (Malden, MA)*, *34*(1), e2414. doi:10.1002/smr.2414

Zhang, X., Jiang, Y., & Wang, Z. (2019). Analysis of Automatic Code Generation Tools based on Machine Learning. In *2019 IEEE International Conference on Computer Science and Educational Informatization (CSEI).* IEEE. doi:10.1109/CSEI47661.2019.8938902

Zhang, X. (2021). *Research and Implementation of Quality and Efficiency Evaluation Method for Automatic Code Generation Based on TBCNN*. Kunming university of science and technology.

Zhang, X. J., & Jiang, Y. (2021). A semi-supervised learning method for automatic code generation performance evaluation. *Journal of Chinese Computer Systems*, *42*(3), 8.

Zhao, H., Min, L. I., Qing-kui, C. H. E. N., & Jian, C. A. O. (2020). Code Review in Open Source Software Development. *Journal of Chinese Computer Systems*, *41*(4), 861–867.