Method Name Prediction for Automatically Generated Unit Tests

Maxim Petukhov
Huawei Technologies Co., Ltd
St. Petersburg, Russia

Evelina Gudauskayte
Huawei Technologies Co., Ltd
St. Petersburg, Russia

Arman Kaliyev
kaliyev.arman@yandex.kz
Huawei Technologies Co., Ltd
St. Petersburg, Russia

Mikhail Oskin
Huawei Technologies Co., Ltd
St. Petersburg, Russia

Dmitry Ivanov
dmitry.ivanov@huawei.com
korifey@gmail.com
Huawei Technologies Co., Ltd
St. Petersburg, Russia

Qianxiang Wang
wangqianxiang@huawei.com
Huawei Technologies Co., Ltd
Beijing, China

Abstract

Writing intuitively understandable method names is an important aspect of good programming practice. The method names have to summarize the codes' behavior such that software engineers would easily understand their purpose. Modern automatic testing tools are able to generate potentially unlimited number of unit tests for a project under test. However, these tests suffers from unintelligible unit test names as it is quite difficult to understand what each test triggers and checks. This inspired us to adapt the state-of-the-art method name prediction approaches for automatically generated unit tests. We have developed a graph extraction pipeline with prediction models based on Graph Neural Networks (GNNs). Extracted graphs contain information about the structure of unit tests and their called functions. The experiment results have shown that the proposed work outperforms other models with precision = 0.48, recall = 0.42 and F1 = 0.45 results. The dataset and source codes are released for wide public access.

Keywords: method name, unit test, abstract syntax tree, graph, deep learning

1 Introduction

Automatic unit test generation [3, 5, 13] attracts significant amount of attention from researchers as it is proving its effectivity for regression suite generation and exposing unexpected scenarios that were defective [25]. The main goal of such tools is to generate unit test suite which will execute as many as possible computation paths of method under test (MUT). Auto-generated tests are often used in the areas where it is mandatory to cover all the lines of code with tests. And whenever they find bugs it is not problematic for programmers to read and to fix code despite that auto-generated takes a longer time to inspect than manually written ones [9, 26].

One of the main challenges of these tools is the incomprehensible unit test names. Potentially the tools can generate a large number of unit tests for one function where each test will trigger a particular behavior of the MUT. Consequently, the name of each unit test has to reflect the behavior it triggers, so that a developer could easily grasp it without even looking at generated code.

Many of the research works on function name prediction heavily use syntactic information of the code [1, 14, 23]. Function name prediction is directly reflected from variable names and other syntactic information. The main intuition behind such approach is that a high proportion of tokens in the method name can be found in the method's body, parameters type and class' name. This approach is unreliable for automatically generated unit tests as they usually contain obscure variable names and syntactically randomly ordered code statements. Moreover, a syntactic change of the automatically generated code may reflect in the change of predicted function name while the unit test would execute the same computation path.

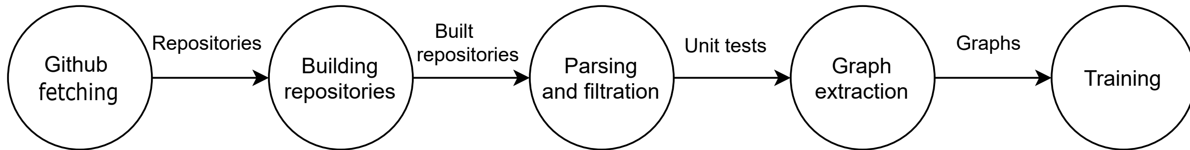
Another approach is to represent structural information of the code as an abstract syntax tree (AST) in Graph Neural Networks (GNNs). GNNs are connectionist models that capture the dependence of graphs via message passing between the nodes of graphs. In encoder-decoder architecture, the GNNs are usually used as encoding part. However, finding invariants about data structures in GNNs is an open problem[19]. The

2022 International Conference on Code Quality (ICCQ) | 978-1-6654-1203-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCQ53703.2022.9763112

978-1-6654-1203-2/21/$31.00 © 2022 IEEE

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

Figure 1. Diagram of data preparation pipeline.



most popular and well-known solution options to this problem[11]:

- Sum or calculate the mean of all graph nodes.
- Introduce a new "virtual node" to represent nodes of the sub graphs in the graph, and start conventional training of GNNs.

Both options are vulnerable to the vanishing gradient problem, as code graphs often consist of thousands of nodes. One way to deal with this weakness is to incorporate attention mechanism, which will point to only important or relevant nodes in the graph.

In this research work, we have developed a neural networks model that predicts Java unit test name based on structural information of the code as control-flow-graph, abstract syntax tree, etc. To train our model, we have downloaded the most popular GitHub projects with unit tests. Next, the structural representations of each unit test and MUTs are extracted as it done in [6]. Except, we also incorporated call graph information to link each unit test with called functions. On a fly, we erase variable names inside of each unit test in order to minimize their influence on prediction and better fit our main goal. So that, during application on generated unit tests, the generated variable names would not effect on prediction result.

We extend [6] approach with the modified self attention mechanism to node types in the aggregation layer and the simplified version of decoder. Our experiments show that proposed modifications help neural networks to better generalize and faster converge. For more detailed review the dataset and source code is available on link https://github.com/kk-arman/graph_names/.

Our main contribution and novelties in this paper are the followings:

- The first attempt to apply the state-of-the-art models of method name prediction for automatically generated Java unit tests.
- The dataset of Java unit test graphs with MUTs.
- A simplified version of decoder with modified aggregation layer for GNN encoder.

The rest of this paper is structured as follows. Section 2 introduces a dataset. Section III presents prior works, and Section IV proposes our approach of network setting. Section V presents an experiment and obtained results. Section VI discusses the work. Section VII gives a conclusion.

2 Dataset

Naturally, software engineers name unit tests based on the functionality they are testing. The body of unit test carries only setup and assertion information while code behavior is triggered on called function. A developer can create a number of unit tests for one function, each time triggering different code behavior. Each such unit test would have different name that reflect the behavior it is testing or capturing. Therefore, well written open-source projects with unit test sets are well suited for training neural network models to predict names of generated unit tests.

Data were collected from open-source projects on the GitHub[15] platform. We used GitHub API to filter out Java projects based on GitHub star counts. GitHub star counts is ranking metric based on software developers' votes on their interested or liked projects.

We chose to fetch the projects with the highest number of appraisals, as they most likely to follow best practices of code naming. Figure 1 illustrates our data collection and graph extraction pipeline.

This way, we acquired around 1000 repositories with the highest star counts. Although, many of them contained only a small number of tests or no tests at all, we ended up with 171151 unit tests built with Maven and Gradle.

After analysis of test methods and their names, we developed a list of criteria to filter out test functions. In order our model would learn only comprehensible function names with a corresponding representative body. Criteria for filtration:

- Method name must contain no more than five words, excluding the word "*test*". Many of the tests in the dataset had long names which were

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

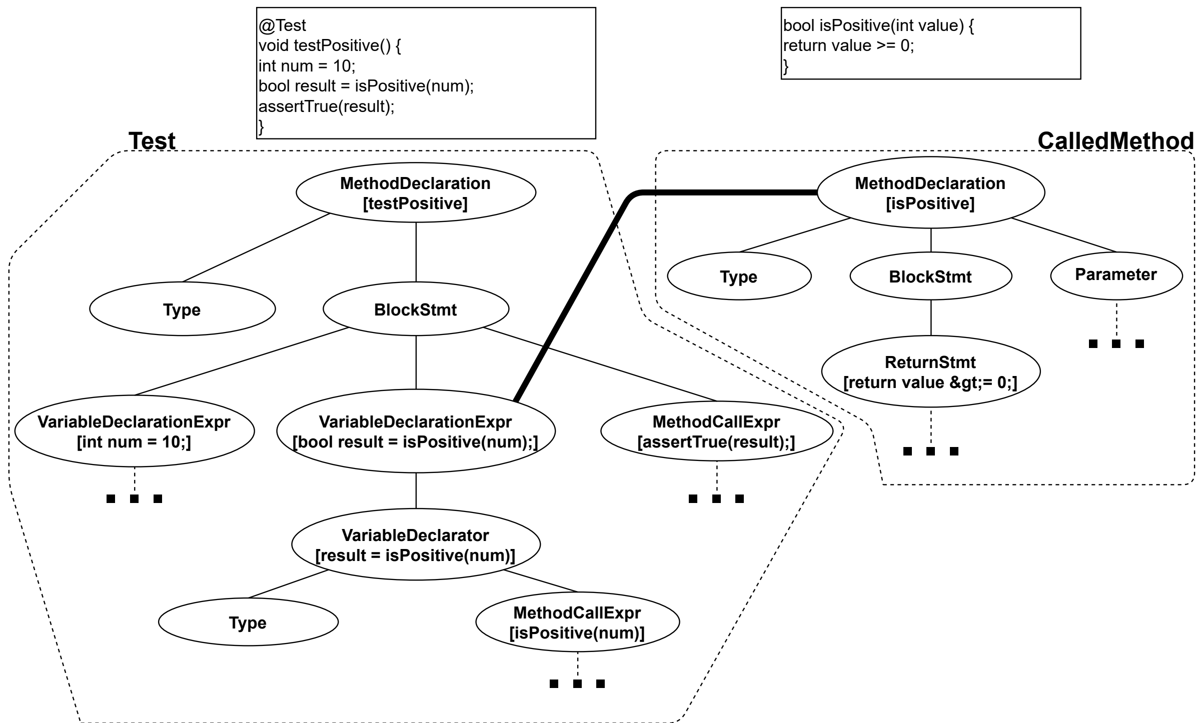


Figure 2. An example of unit test, MUT and their AST representations. During preprocessing ASTs of unit test and MUT are connected to a single graph.

difficult to read and understand. To avoid such long name generation, this filter is applied.

- Method name must include at least one verb. Some of the test names are consisted of only nouns and pronouns. As a result, the names lacked description of actions that are triggered.
- The body of method must contain at least 4 lines of code. 2 or 3 lines of code structurally have a little difference. The real difference, they have, would be in the specifics of the execution paths which we intend to integrate in the future work.

Application of this filtration resulted in 43931 unit test methods. An average number of tokens in test names is 4.33 words.

Next, AST representations of unit tests and their MUTs are built as their graphs. Details of this stage are discussed in the preprocessing step of the Experiment section.

2.1 Abstract syntax tree

Abstract syntax tree is a structural representation of a program code. Each node in an AST represents a source code unit: operators, variables, literals, function calls etc. ASTs translate the structure of source code into parent-child or sibling relationships allowing the efficient traversing [16], where every node has at least a type specifying what it is representing. It is widely used by researchers and software developers for code completion [29], plagiarism detection [30], bug localization [20] and etc.

Figure 2 presents a code of unit test with MUT and their ASTs. At the top a simple example of unit test and called function are illustrated, at the bottom their ASTs representation are shown.

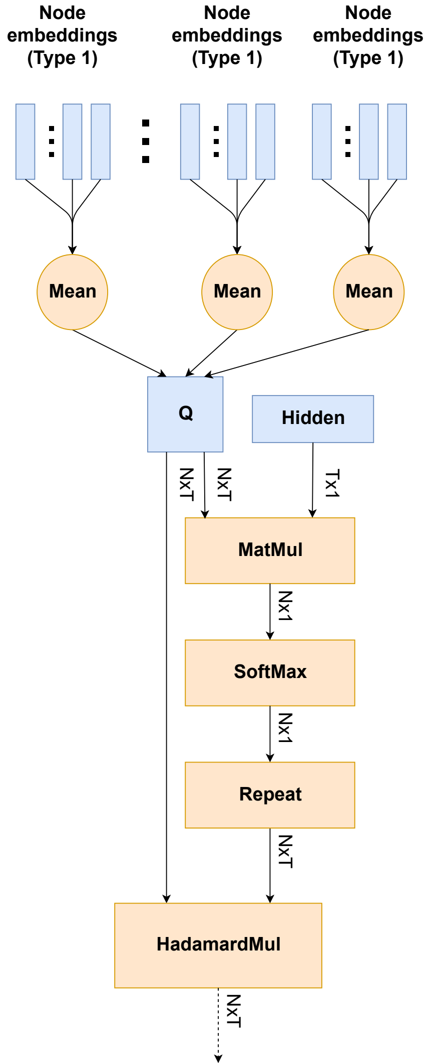
3 Prior Works

Early works on source code summarization are based on key words retrieval out of source code, and then those words are considered as annotation or summary. Most of these summarization techniques used TF-IDF, LSI, and LDA methods to create summary [18, 22, 27].

A more recent approach of structural representation of the source code is to parse into AST. Once it in the graph format, an additional edges can be added which can represent other semantic information as data flow or control flow graph. Such data can be feed to neural networks which in turn can exploit its structural formatting. Allamanis et al. [2] proposed to apply GNNs on

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

Figure 3. Diagram of self-attention to node types mechanism inside encoder.



AST graphs to learn representations of code (though for variable names prediction and variable misuse detection). They have showed that GNNs can be effective in extracting information in a program graph structure, and using that information for downstream tasks. Following this approach, researchers proposed several modifications, Cvitkovic et al. [6] introduced Graph structured Cache (GSC) format to handle open vocabulary issue which arose from wide range words use by programmers for method and variable naming. GSC added an additional 'caching' nodes of vocabulary words to these code graphs. Each variable that used vocabulary words had connection to respective 'caching' nodes. Fernandes et al. [7] proposed to use recurrent neural network (RNN) to learn the sequence representation of each token in a program, then apply GNN to compute the state for every node in the AST. For decoder, they use another RNN, which generates the method name as a sequence of words. Wang et al. [28] presented GINN that derives an abstracted representation from a hierarchy of subgraphs in the control flow graph for graph encoding.

U. Alon et al. [4] in 2018 presented an alternative approach for encoding source code that leverages the syntactic structure of programming languages. According to their observation, common methods have similar syntactic paths, and the difference can often be a single node in AST. This has encouraged them to develop Code2Seq model, which would randomly select a set of paths for encoding to a standard encoder-decoder model. A random set of paths encoding provided a good generalization of unseen examples.

We follow [6] approach by extending GNN with aggregation layer based on node types and simplified decoder.

4 Our Approach

We adopt encoder-decoder architecture for test name generation based on the graph representation of source code. Encoding mechanism consist of Gated Graph Convolutional neural network and node type attention layer Figure 3.

Intention behind using attention mechanism on node types:

- Large number of nodes have empty identifiers.
- Some of the node types are rich with different values.
- Only a few of the node types carry syntactic and semantic information of code, many other nodes are repetitive on each level in AST representation.

4.1 Gated Convolutional Graph

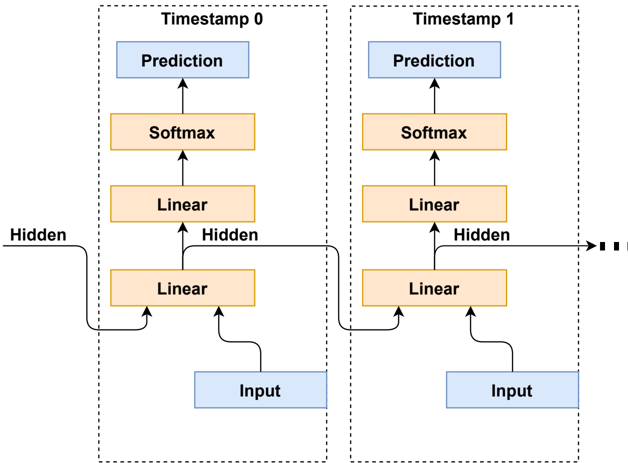
Following notation described at [2, 19], a graph G consist of a sets of nodes V, X, and E such that $G = (V, E, X)$ where

- V - set of nodes
- X - node features
- $E = (E_1, ..., E_k)$ - a set of directed edge sets, k is the number of edge types

Each node $v \in V$ is labeled with a feature vector $x^v \in R^D$. For each node v we also initialize a state vector h_t^v from node label x^v and node type $t^v \in T$. A "message" is sent from each node v to its neighboring nodes where "message" of type k is calculated as $m_k^v = f_k(h_t^v)$. f_k is a linear

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

Figure 4. Simple decoder



layer function. The state in the graph are updated by aggregation of all messages from each neighboring node as $\tilde{m}^v = g(\{m_k^u | there\,exist\,an\,edge\,of\,type\,k\,from\,u\,to\,v\})$, where g is an aggregation function. New state $\hat{h}$ for each node v is computed at the same time by equation $\hat{h}_t^v = GRU(\tilde{m}^v, h_t^v)$, where h is node state in a previous time step and GRU is the recurrent cell function of gated recurrent unit (GRU). For a given n number of time steps, all the states in the graph are updated n times by propagation of all "messages".

4.2 Aggregation Layer

We implemented a new aggregation layer with attention mechanism on node types. After message propagation steps inside graph, node annotation vectors h_t^v are grouped based on node type t into $H = \{H^t | t \in T\}$ where $H^t = \{h_t^v | v \in V\}$ and T is set of node types. For each such group mean $\bar{h}_t = mean(H^t)$ are calculated, on which an attention mechanism is placed. After application of attention mechanism, vectors send to encoder layer to squeeze the output vector size.

$$output = enc(\sum_{t \in T} atten(\bar{h}_t))$$

4.3 Decoder

We implemented **simple decoder** which consist of hidden layer and output layer. Sampling strategy is applied to our decoder. The output of hidden layer is used for prediction of the current token and as the feedback to neural network for prediction of the next token Figure 4.

In our training and testing procedures, test names are presented as sequences of tokens.

While data collection and preprocessing pipeline were written on Java and Python languages, the neural networks were implemented on Python using PyTorch framework [24] and PyTorch Geometric Library [8]. Training and testing were run on Intel Xeon Gold 6140 2.3 GHz with GeForce RTX 2080Ti graphic card.

5 Experiment

5.1 Preprocessing

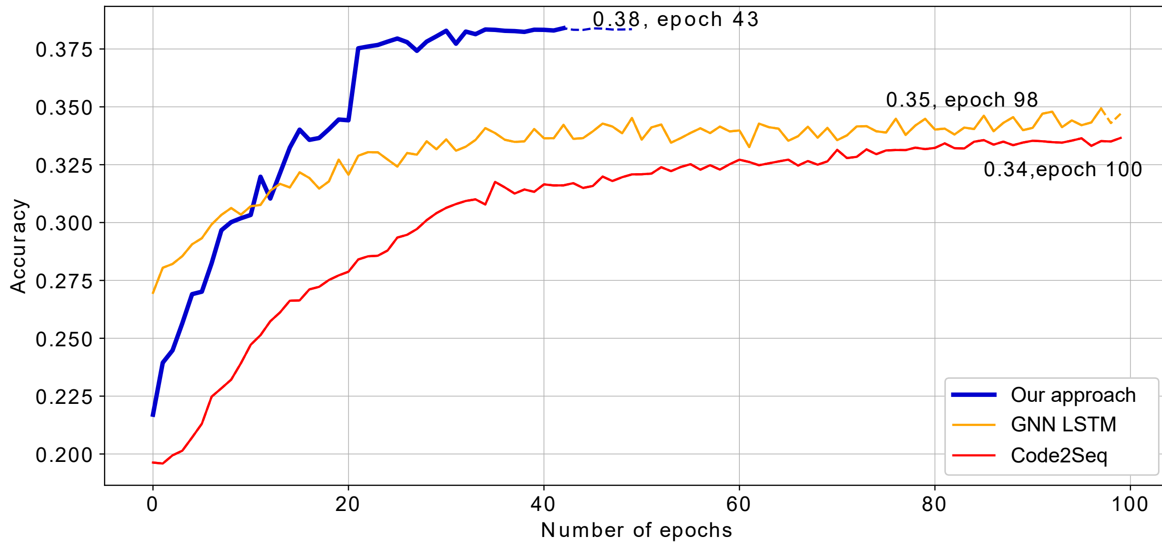
At the preprocessing stage, for each unit test we generated a separate graph from the source file. This graph captures the structure of the test body and its MUTs. To extract the graph from the source file, we perform the following steps:

1. We extract the AST from the source code file that contains a unit test.
2. Semantic edges of data-flow and control-flow between variables are added to the extracted AST.
3. A subgraph related to the unit test body is cut from the AST.
4. New edges of control-flow are added to the cut AST, which describe the overall progress of the code execution, described in more detail below.
5. From unit test body called methods are identified using call graph.
6. We repeat steps 1 through 4 for each called method from the unit test body. Next, we connect the graph of the unit test body with the graphs of the called methods using call graph.

Steps 1, 2 are the same as in M. Cvitkovic et al. [6]. In step 3, we keep information only about the structure of the unit test body. We discard other information, such as the structure of the unit test class itself, as we find them little informative for our task and unnecessary burdensome for the final graph. Steps 4 and 5 are performed using the Soot framework[17]. Soot is a framework for static analysis of Java programs. It contains a number of intermediate representations of Java source code, including Jimple. We use Jimple together with Soot to build call graph and control flow graph, which are then added to the final graph. Call graph allows us to determine which specific methods were called from the unit test body, and control flow graph gives us an additional information about the structure of the source code. However, we built the AST at the source code level, while the control flow graph and the call graph from Soot are built at the Jimple representation level. Jimple is an intermediate representation of a Java program based on

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

Figure 5. Comparison of training progress of our approach, GNN LSTM and Code2Seq based on validation accuracy.



three-address code. Here, the control flow graph of Jimple can be mapped to the lines of source code. In this regard, we made the following assumption:

a. The Jimple statements are mapped to the lines of source code as each Jimple statement contains information about the line of code, it is extracted from.
b. Javaparser[12] converts each line of code into a hierarchical representation of the AST nodes. We select the node at the highest level of the hierarchy for the given line during comparison of the Jimple instructions with the AST nodes.

In step 6, we bound our analysis with the methods called from the unit test body. Methods that are inside the MUTs are not analyzed.

To sum up, we end up with the structural representation of unit tests and their called functions. We attach called methods representations as the actual code behavior is triggered on the MUTs while unit tests themselves contain information of concrete input and output. Here, it is assumed that the deep neural networks will able to find a relation between such representation of unit tests with attached MUTs and unit test names. Due to the complexity of our data collection approach, we were able to pipeline around 18k such graph examples.

5.2 Experimental Setup

As the reference for the performance comparison we took Code2Seq [4] and GNN LSTM [2][1]. Both models are state-of-the-art models for method name generation.

[1] https://github.com/bdqnghi/ggnn.method_name_prediction

Code2Seq predicts function name based on AST representation of the source code. Model follows the standard encoder-decoder architecture; it decomposes AST into a set of paths, where each path is encoded to a vector using LSTM. Then, the decoder uses averaged path vector of AST, attention mechanism and another LSTM to predict sequences of tokens.

Code2Seq's setup: parameters are initialized with Glorotand Bengio heuristic [10] while cross-entropy and Nesterov momentum are used as loss and optimization functions .

GNN LSTM uses the average of all nodes in the graph as the initial state of the decoder, as well Luong Attention [21] to link the hidden state of the decoder to the "important" nodes in the graph. However, the main disadvantage of this approach is the potential appearance of extremely small gradients with an increase of the number of nodes in the graph. The attention vector is calculated over all nodes during decoding, while there can be from several hundred to several thousand of them in the graph.

The dataset were randomnly split into training (80%), testing (10%) and validation (10%) parts. During training GNN LSTM and our model, we erased variable names inside of unit tests to simulate nature of generated code. All models are trained on 14.5k data set and evaluation are performed on 1.8k dataset.

All models except Code2Seq are trained on the graphs described above. To train on our dataset Code2Seq requires major changes in its mechanism. Therefore, it is

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

trained on the graph that did not contain MUT instead it had token identifiers.

We would want to point out that a number of researchers apply different approaches of metrics calculation for the task of method name prediction. Our investigation of public repositories related to this task revealed that some researchers train their networks to predict the sequence of tokens as the method name but metrics calculation is done by symbol-to-symbol scheme. Similar evaluation setups are detected on other repositories with different approaches of neural networks training. We believe that such way of presenting information misleads the readers about the true performance of the models, as many others, for example, would train their models to predict sequence of tokens and calculate metrics based on the predicted tokens' ids.

To ensure the fairness of the comparison of different models we implemented an identical metrics calculation algorithm for all models. Accuracy, F1 and other metrics are calculated based on token-to-token matching scheme. Figure 5 shows training progress for all models based on validation accuracy. Table 1 presents the final results on the test set.

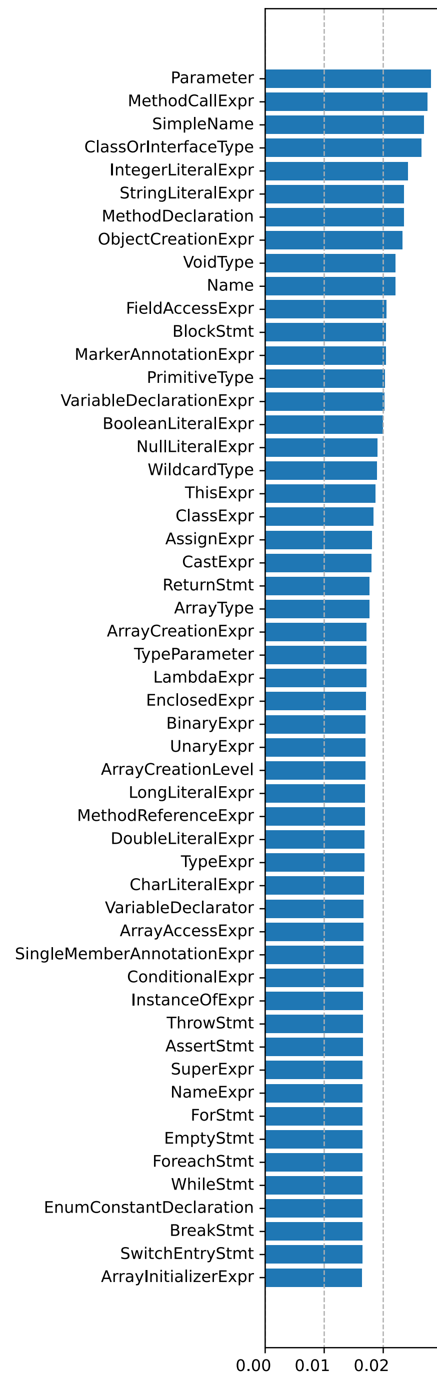
5.3 Results

Table 1. Final results on test set run.

Model	Prec	Rec	F1
GNN LSTM	0.28	0.33	0.30
Code2Seq	0.44	0.38	0.41
Our approach	0.48	0.42	0.45

Figure 5 shows the progress of training neural networks based on validation accuracy metrics. As it can be seen, our approach with simple decoder outperforms other models with better accuracy and faster convergence. Simple decoder reaches 37.5 percent accuracy already on 21th epoch, far above others. Moreover, according to the Table 1 our approach is superior to other models in precision, recall and F1 scores. A sample of predicted examples can be found on the Table 2. Quite unexpectedly, GNN LSTM and Code2Seq showed slightly controversial results. During training GNN LSTM surpassed Code2Seq in accuracy metric, converging at 35 percent accuracy on validation set at 98th epoch, while Code2Seq stopped improving at 100th epoch with 34 percent accuracy. On the final run on the test set, Code2Seq and GNN LSTM demonstrated 0.41 and 0.30 F1 scores respectively.

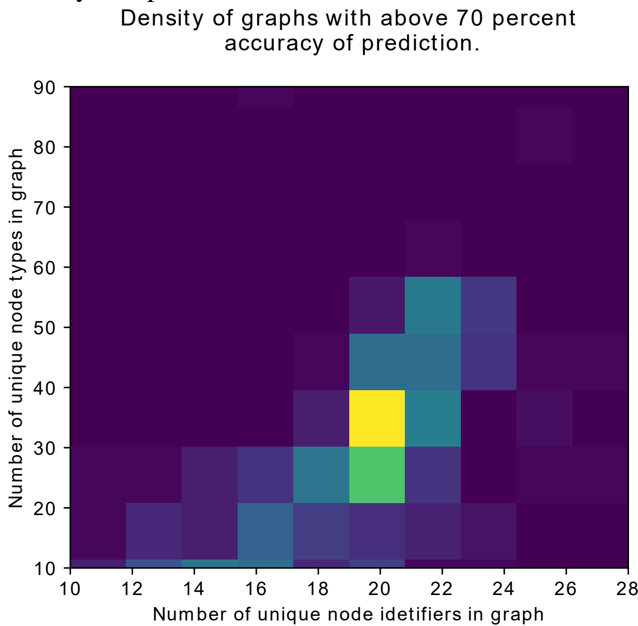
Figure 6. Ranking node types based on attention scores.

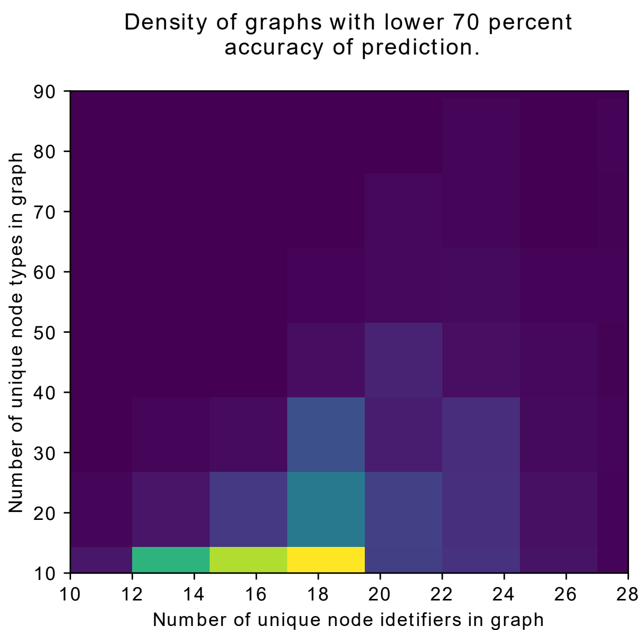


The attention scores for different types of nodes are ranked on Figure 6. These scores were calculated on validation set with our approach. According to this ranking,

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

Figure 7. Difference between graphs based on their density and prediction results.





the most important types of nodes for test name prediction are MethodCallExpr, MethodDeclaration, SimpleName, Name etc. It seems that the neural network puts high importance on the nodes, which:

- have links between unit test and MUTs.
- keeps some token or parameter identifiers, for example variable names inside called function.

Figure 7 shows the difference between good and bad predicted examples on the validation set with our approach. The criteria for good and bad prediction is a barrier of 70 percent accuracy of token sequences matching. Our model shows high accuracy prediction on graph representations, which consist of at least of 25 unique node types and 18 unique parameter identifiers. On the other hand, the neural network has troubles in prediction with graphs that are less dense with these type of parameters. In our dataset, almost all the graphs that pass through this “density” threshold show good results.

Table 2. Some random examples of predicted unit test names by simple decoder on the test set.

Prediction	Reference
should place service	should place blank sub process
should deserialize partial object	should deserialize json object
test server	test server delete path
test invoke config	configures module
classify role provides module name	classify role uses type name

6 Discussion and Future Work

We find method name prediction based on a graph as a hard problem. Some of the unit tests are written after occurrence of random bugs in the code. There are also examples of unit tests that were created to increase test coverage. As result, these and some other tests can have obscure or unintelligible names. While our approach relies on factors:

- Each unit test triggers one particular code behavior.
- Developers write meaningful unit test names.
- If there is unintelligible test name then our pipeline will filter out them using criteria filtration.

Despite of this, our approach shows promising results of name prediction for Java unit tests. It outperformed other models in this task with attention to node types and almost perfect prediction for densely packed ASTs. As future work in this area, we intend to incorporate information related to trace execution with depiction of nodes order execution. We believe that program traces could bolster our results on test name prediction. Unfortunately, trace collection from program execution is another hard problem that requires a significant amount of engineers’ effort and their time. Our team have already

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

developed a such plugin for trace collection but because of the complexity of Java virtual machine debugging and other issues, our plugin currently can automatically traverse about 6 thousand unit tests and their called functions. These unit tests were initially filtered by our pipeline, then traversed by the plugin. It is dozens of automatically debugged projects. As it is still insufficient for training and testing, we made public all collected dataset as well as trace collection plugin itself for other researchers and engineers.

We are planning to continue working in this area by enhancing the plugin and incorporating execution traces to the dataset. Although, it seems that execution path would have higher relation to the names, traces alone may be not sufficient. As they present information from the code coverage point of view. While real "code behavior" summarization may require additional background information of the unit tests and project.

7 Conclusion

In this paper, we applied the state-of the art approaches of method name prediction for Java unit test name generation. We have developed:

- A pipeline for data preparation and preprocessing of Java unit tests and their MUTs;
- Own implementation of neural networks for the task of Java unit name prediction;

Our proposed approach showed a competitive result in comparison to the-state-of-art models. However, we believe that a real world application requires a further research in this area and additional feature engineering. Therefore, we plan to continue to work in this direction and invite other researchers to take interest in this task. Our future direction of research will be enhancing trace collection plugin for Java virtual machine and expanding the dataset.

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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2018). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[3] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[4] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. *CoRR* abs/1808.01400 (2018). arXiv:1808.01400 http://arxiv.org/abs/1808.01400

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[6] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. 2018. Open Vocabulary Learning on Source Code with a Graph-Structured Cache. *CoRR* abs/1810.08305 (2018). arXiv:1810.08305 http://arxiv.org/abs/1810.08305

[7] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured Neural Summarization. *CoRR* abs/1811.01824 (2018). arXiv:1811.01824 http://arxiv.org/abs/1811.01824

[8] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[9] Gordon Fraser and Andrea Arcuri. 2015. 1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite. *Empirical Softw. Engg.* 20, 3 (June 2015), 611–639. https://doi.org/10.1007/s10664-013-9288-2

[10] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks.. In *AISTATS (JMLR Proceedings, Vol. 9)*, Yee Whye Teh and D. Mike Titterington (Eds.). JMLR.org, 249–256. http://dblp.uni-trier.de/db/journals/jmlr/jmlrp9.html#GlorotB10

[11] William L. Hamilton, Rex Ying, Jure Leskovec, and Rok Sosic. 2018. Representation Learning on Networks. http://snap.stanford.edu/proj/embeddings-www/

[12] R Hosseini and P Brusilovsky. 2013. JavaParser: A fine-grain concept indexing tool for java problems. *CEUR Workshop Proceedings* 1009, 60 – 63. http://d-scholarship.pitt.edu/26270/

[13] Dmitry Ivanov, Nikolay Bukharev, Alexey Menshutin, Arsen Nagdalian, Gleb Stromov, and Artem Ustinov. 2021. UtBot at the SBST2021 Tool Competition. In *Proceedings of the ACM/IEEE 43th International Conference on Software Engineering: Companion Proceedings (ICSE '21)*. Association for Computing Machinery, New York, NY, USA.

[14] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the*

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.

11th Working Conference on Mining Software Repositories (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

[16] Richard E. Korf. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27, 1 (1985), 97–109. https://doi.org/10.1016/0004-3702(85)90084-0

[17] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. 2011. The Soot framework for Java program analysis: a retrospective.

[18] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. arXiv:2004.02843 [cs.SE]

[19] Y. Li, Daniel Tarlow, Marc Brockschmidt, and R. Zemel. 2016. Gated Graph Sequence Neural Networks. *CoRR* abs/1511.05493 (2016).

[20] H. Liang, L. Sun, M. Wang, and Y. Yang. 2019. Deep Learning With Customized Abstract Syntax Tree for Bug Localization. *IEEE Access* 7 (2019), 116309–116320. https://doi.org/10.1109/ACCESS.2019.2936948

[21] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. *CoRR* abs/1508.04025 (2015). arXiv:1508.04025 http://arxiv.org/abs/1508.04025

[22] Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing Software Artifacts: A Literature Review. *J. Comput. Sci. Technol.* 31, 5 (2016), 883–909. https://doi.org/10.1007/s11390-016-1671-1

[23] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1372–1384. https://doi.org/10.1145/3377811.3380926

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[25] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 121–125. https://doi.org/10.1109/MSR.2019.00028

[26] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 201–211. https://doi.org/10.1109/ASE.2015.86

[27] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. 2019. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *CoRR* abs/1907.10863 (2019). arXiv:1907.10863 http://arxiv.org/abs/1907.10863

[28] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. arXiv:2005.09997 [cs.SE]

[29] Yanlin Wang and Hui Li. 2021. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. arXiv:2103.09499 [cs.SE]

[30] Mengya Zheng, Xingyu Pan, and David Lillis. 2018. CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Tree. In *Proceedings for the 26th AIAI Irish Conference on Artificial Intelligence and Cognitive Science Trinity College Dublin, Dublin, Ireland, December 6-7th, 2018 (CEUR Workshop Proceedings, Vol. 2259)*, Rob Brennan, Jöran Beel, Ruth Byrne, Jeremy Debattista, and Ademar Crotti Junior (Eds.). CEUR-WS.org, 362–373. http://ceur-ws.org/Vol-2259/aics_33.pdf

Authorized licensed use limited to: Huawei Technologies Co Ltd. Downloaded on September 23,2022 at 15:49:48 UTC from IEEE Xplore. Restrictions apply.