

CodeGRU: Context-aware Deep Learning with Gated Recurrent Unit for Source Code Modeling

Yasir Hussain^{a,*}, Zhiqiu Huang^{a,b,c,*}, Yu Zhou^a, Senzhang Wang^a

^a*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NCAA), Nanjing 211106, China*

^b*Key Laboratory of Safety-Critical Software, NCAA, Ministry of Industry and Information Technology, Nanjing 211106, China*

^c*Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210093, China*

Abstract

Context: Recently deep learning based Natural Language Processing (NLP) models have shown great potential in the modeling of source code. However, a major limitation of these approaches is that they take source code as simple tokens of text and ignore its contextual, syntactical and structural dependencies.

Objective: In this work, we present CodeGRU, a gated recurrent unit based source code language model that is capable of capturing source codes contextual, syntactical and structural dependencies.

Method: We introduce a novel approach which can capture the source code context by leveraging the source code token types. Further, we adopt a novel approach which can learn variable size context by taking into account source codes syntax, and structural information.

Results: We evaluate CodeGRU with real-world data set and it shows that CodeGRU outperforms the state-of-the-art language models and help reduce the vocabulary size up to 24.93%. Unlike previous works, we tested CodeGRU with an independent test set which suggests that our methodology does not require the source code comes from the same domain as training data while providing suggestions. We further evaluate CodeGRU with two software engineering applications: source code suggestion, and source code completion.

Conclusion: Our experiment confirms that the source codes contextual information can be vital and can help improve the software language models. The extensive evaluation of CodeGRU shows that it outperforms the state-of-the-art models. The results further suggest that the proposed approach can help reduce the vocabulary size and is of practical use for software developers.

Keywords: Deep Neural Networks, Source Code, Code Suggestion, Code Generation

*Corresponding author

Email addresses: yaxirhussain@nuaa.edu.cn (Yasir Hussain), zqhuang@nuaa.edu.cn (Zhiqiu Huang), zhouyu@nuaa.edu.cn (Yu Zhou), szwang@nuaa.edu.cn (Senzhang Wang)

1. Introduction

Source code suggestions, code completion, bug fixing, etc. are vital features of a modern integrated development environment (IDE). These features help software developers to build and debug software rapidly. In the last few years, there have been a massive amount of increase in code related databases over the internet. Many open source websites (i.e. w3school, GitHub, Stack Overflow, etc.) provides API libraries, code usage examples, bug fixing, and much more. Software developers exceedingly rely on such resources for above-mentioned purposes.

Natural language processing (NLP) [9, 27, 36] explores, understands and manipulates natural language text or speech to do serviceable things. NLP techniques have shown its effectiveness in many fields such as speech recognition [6], information retrieval [7], text mining [35], machine translation [48] and source code modeling [20, 37, 22, 44]. One of the most common NLP technique for source code modeling is statistical language models (SLM), which calculates the probability distribution over sequences in a corpus. Given a sequence S of length N it assigns the probability to the whole sequence $Pb(t_1, \dots, t_n)$ and then calculates the likelihood of all sub-sequences to find the most likely next sequence.

The advancement in the neural network based NLP models [47, 37, 52] have recently shown that they can effectively overcome the context issue that cannot be effectively addressed by SLM [20, 44] based models. Many deep learning based approaches have been applied for different tasks for source code modeling such as code summarization [21, 3], code readability classification [28], code generation [41], error fixing [18, 49], and code recommendation [47, 37, 16, 11]. A major limitation of these approaches [20, 47, 37] is that they take source code as simple tokens of text sequence and ignore its contextual, syntactical and structural dependencies. Another limitation is that they learn source code as a sequence to sequence problem with fixed size context where the right context may not be captured in the fixed size window, which leads to the inaccurate prediction of the next code token.

Compared with natural language text, source code tends to have richer contextual, syntactical and structural dependencies. Treating source code as a simple text cannot effectively capture these dependencies. Software developers usually choose to have different names for methods, classes, and variables, which makes it difficult to capture the right context. For example, one software developer may choose an identifier name `num` for an `INT` data type, while another one may choose `size` for the same purpose. Consider another example where a common method `i.toString()` converts a variable to `String` data type. A similar method `person.toString()` refers to an object of a `person` class that returns a persons information. In addition, the source code must follow the rules defined by its grammar. For example, a `try` block must be

followed by a `catch` block. Another example is that when a developer uses `do` block, the next block should be `while()` and the possible next token suggestion should be `;` according to the syntax of java language grammar. Previous works [20, 47, 37] lack to consider such information which can be valuable for source code modeling tasks.

In this work, we present CodeGRU which considers the source code’s context, syntax, and structure while suggesting the next source code token. This work does not simply consider source code as simple tokens of text. The CodeGRU introduces a novel approach which can correctly capture the source code context by leveraging the token type information. The CodeGRU can effectively capture the right context even it is separated far apart in the code. CodeGRU further proposes a novel approach which can learn variable size context while modeling source code. Unlike previous works [20, 47, 37, 32], we do not treat the source code as a single sequence of text tokens instead we use a novel approach which builds the sequences based on source codes structural and syntactical information. We evaluated CodeGRU with real-world data set with an independent test set with two software engineering applications: source code suggestion, and source code completion.

This work makes the following unique contributions:

- A novel approach for source code modeling is proposed, which can capture the source code context by leveraging the token type information.
- A novel method which learns the variable size context of the source code is proposed. Unlike previous works, we do not treat the source code as a single sequence of text tokens instead we use a novel approach which builds the sequences based on source codes structural and syntactical information.
- An extensive evaluation of CodeGRU on the real-world data set shows that CodeGRU outperforms the state-of-the-art language models. We further evaluated CodeGRU with two software engineering applications: (1) source code suggestion, which can suggest multiple predictions for the next code token, and (2) code completion, which can complete the whole next code sequence.

2. Preliminaries

In this section, we will discuss the preliminaries and technical overview of this work.

Fig. 1 shows the architecture of the RNN for source code modeling, where τ is input layer, c is context layer also known as hidden layer and y is the output layer. The hidden state activation at a time step i is computed as a function on the previous h_{i-1} along with current code token τ_i .

$$h_i = f(\tau_i, h_{i-1}) \quad (1)$$

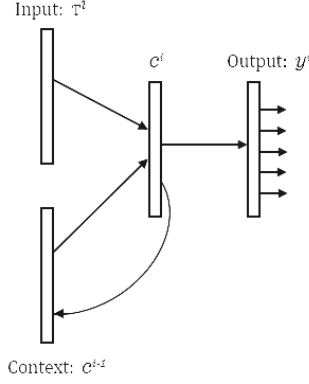


Figure 1: An architecture of a RNN neuron where input is a code token vector at index i , and the outputs are different next code tokens y^i based on the context and probabilities

Usually f is composed of an element-wise nonlinear and affine transformation of τ_i and h_{i-1} .

$$h_i = \phi(W\tau_i, Uh_{i-1}) \quad (2)$$

Here W is the weight matrix for the input to hidden layer and U is the weight matrix for the state to state matrix, and ϕ is an activation function. The RNN models [29, 14] tends to look back further than $n - 1$. But vanilla RNN suffers from vanishing gradient problem which can be overcome by using Gated Recurrent unit (GRU) model.

The GRU exposes [52] the full hidden content without any control which is ideal for source code modeling. It is composed of two gates, the reset gate r_i and the update gate z_i . Further, it entirely exposes its memory context at each time step i . Exposing the entire context on each time step helps to learn contextual dependencies better than vanilla RNN. It can be expressed as

$$h_i = (1 - z_i)h_{i-1} + z_i\tilde{h}_i \quad (3)$$

Where h and \tilde{h} is prior context and fresh context respectively.

$$z_i = \phi(W_z\tau_i + U_zh_{i-1}) \quad (4)$$

$$\tilde{h}_i = \tanh(W\tau_i + r_i \otimes Uh_{i-1}) \quad (5)$$

$$r_i = \phi(W_r\tau_i + U_rh_{i-1}) \quad (6)$$

A major difference from Eq. 2 is that the \tilde{h} is modulated by the reset gates r_i . Here \otimes is element-wise multiplication and ϕ is the activation function.

3. The CodeGRU Model

In this section, we introduce CodeGRU in detail. The overall workflow of CodeGRU is illustrated in Fig. 2. The first step is data collection, which we will discuss in section 4.1. Next step is *Code Analyzer*, which pre-processes the source code files and captures the token type information. The *Code Analyzer* parses the source code and encode the type information to capture the source code context. Next, the *Variable Size Context Learning* approach is used to build the sequences based on the source code syntax and structure. Next, we tokenize the sequences and build vocabulary. Finally, we train deep learning classifier for source code suggestion and completion task. Each step is discussed in detail in the following subsections.

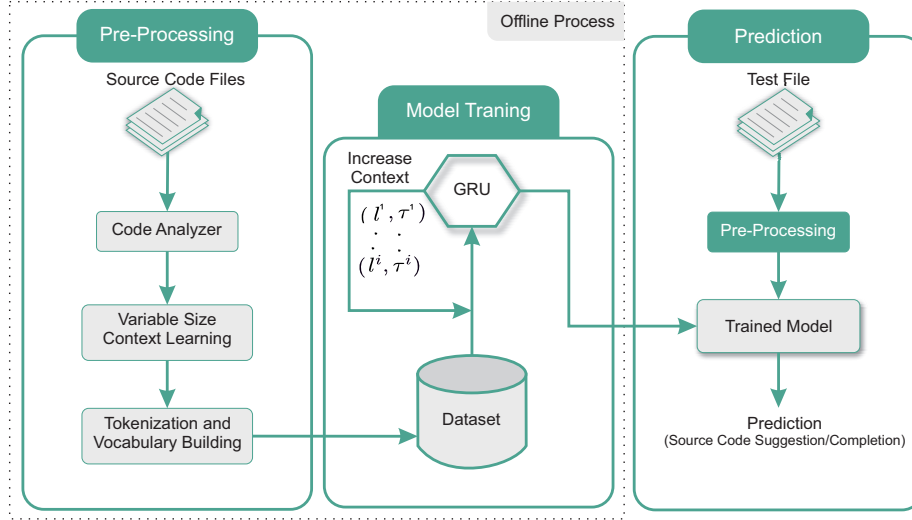


Figure 2: The framework of CodeGRU, which is a context aware deep learning model for source code modeling.

3.1. Code Analyzer

Code Analyzer first takes a source code file and pre-process it to capture the source code context. First, we normalize the files by removing all blank lines, inline and block level comments as they have no impact on source code suggestion or completion task. Source code consists of different kinds of tokens such as classes, functions, variables, literals, language-specific keywords, data types, stop words, etc. Among all these, language dependent keywords, stop words, library functions, and data types form a shared vocabulary which can be considered as context. A key insight is that to capture the context of source code tokens we capture their token types. Here we care about the token type rather than the token identifier. As discussed earlier code token identifiers can differ from developer to developer, so we use the token types to help capture the

context information. The exact values of literals (*Int*, *float*, *long*, *double*, *byte*, *String*, etc) are unnecessary in source code modeling task. So, we transform all literal values to their abstract data types according to the Java language grammar¹. For example given `System.out.println("Hello World")`, the string value "Hello World" is of type `String Literal` according to java language grammar. We transform it with its token type `StringLiteral`. Similarly, the value of identifier *a* in `a = 1.1` is not important, so we transform `1.1` with its token type `FloatLiteral`.

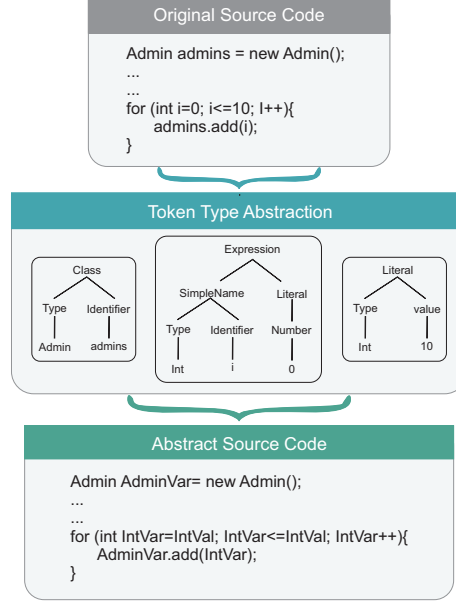


Figure 3: An example of Java code with our *Code Analyzer* approach.

A challenging issue in source code modeling is to capture the variable and class object identifies declaration types. Static type languages (Java, C++, C#, etc.) are strongly type defined languages, which means types of such declarations need to be defined before use. In Fig. 3 one can see a transformation example of variable and class object identifiers. We capture all identifiers types into their declared data types. In Fig. 3 one can see, we transform all instances of identifier *i* with its declared data type `Int` combined with a special token `Var`. Similarly, the class object identifier *admins* is replaced with `AdminVar`. We leave special code tokens (*true*, *false*, *null*) unchanged. Unlike literals, variables, and class objects, such tokens reflect constant behavior, which does not need transformation. Table 1 shows some common code tokens and their

¹<https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html>

resolved types captured by our novel approach.

Table 1: Common source code examples by our *Code Analyzer* approach

Code Token	Token Type	Special Token
<code>i</code>	Int	IntVar
<code>"HelloWorld"</code>	String Literal	StringLiteral
<code>null</code>	Literal	NullLiteral
<code>outFile</code>	File	FileVar
<code>ex</code>	Exception	ExceptionVar
<code>arr</code>	ArrayList<String>	ArrayListStringVar
<code>lstID</code>	List<Int>	ListIntVar
<code>'c'</code>	char	CharLiteral
<code>true</code>	Boolean	true
<code>Int * x</code>	Int Pointer	Int * IntPtrVar
<code>Int ** x</code>	Int Pointer	Int ** IntPtrToPointerVar
<code>true</code>	Boolean	true
<code>inputfile.open()</code>	File	FileVar.open()

3.2. Variable Size Context Learning

Programming languages strictly follow the rules defined by their grammar. Each line in a programming language starts with a language reserved identifier, variable or class object deceleration, assignment statements, etc. Whereas an assignment statement can only have a variable name, object instance or array index on the left-hand side. Further, source code follow block rules such as `try-catch-final`, `do-while` where one must follow the other. We use such information while building the sequences. Unlike previous works [47, 37, 32], we do not consider the source code as a single sequence of text tokens and divide them into fixed-size context window. Instead, we leverage from the syntactical and structural information to learn variable size context of the source code. The CodeGRU takes a source code program and split it based on a code statement or a block statement. In source code a single statement ends with `;` token where a block statement starts with the `{` token and ends with `}` token according to the Java language grammar². With this approach, we split each file into multiple sequences of code tokens X . Here the goal is to produce the next token y by satisfying the context of X . We can express a source code file S at line L . Then a source code program can be represented as (l^i, τ^i) where l^i is the line number and τ^i is tokenization of S at l^i . It breaks each l^i into several τ^i by iteratively increasing the context on each iteration. The CodeGRU learns the source code context at (l^i, τ^i) and keeps increase the τ^{i+1} until it reaches the upper bound limit of l^i . When CodeGRU reaches the upper bound limit of l^i , it increases l^{i+1} and keeps learning the source code context. Fig. 4 shows an example of proposed variable size context learning approach.

²<https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html>

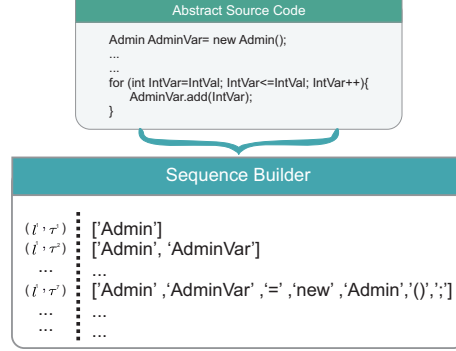


Figure 4: An example of variable size context learning approach.

Further, here we expect the model to assign the high probability to the correct next source code suggestion by having a low *Cross-entropy*. The *Cross-entropy* is a cost function to observe how best the model works. A low value of *Cross-entropy* indicates a good model. It can be expressed as

$$H(C) \approx -\frac{1}{N} \sum_{i=1}^N \log_2 Pb_C(\tau^i | \tau_{i-n+1}^{i-1}) \quad (7)$$

3.3. Tokenization and Vocabulary Building

To convert the source code files into a form that is suitable for training we perform a series of transformations. First, we tokenize the source code files as shown in Fig. 5. Each unique source code token corresponds to an entry in the vocabulary. Each source code token is then assigned a unique positive integer value. In Table 2 one can see the vocabulary statistics, where V_{Norm} shows the vocabulary statistics without *Code Analyzer* and $V_{CodeGRU}$ shows the vocabulary statistics with *Code Analyzer* approach. The vocabulary statistics without *Code Analyzer* is reported by calculating the unique code tokens found in source code files after removing blank lines, inline and block level comments. Further, we replace all literal values to their abstract data types and leaving the rest of the file to its original state.

Tokenization	['Admin', 'admins', '=', 'new', 'Admin', '()', ',']
Abstract Tokenization	['Admin', 'AdminVar', '=', 'new', 'Admin', '()', ',']
Vectorization	[5,0,4,1,5,2,3]

Figure 5: Process of building vocabulary for source code language models.

3.4. Training and Prediction

We parse each source code file in the training set and use *Code Analyzer* to capture the token type information as discussed earlier in Section 3.1. After

Table 2: Vocabulary statistics between projects.

	Min	Max	Mean	Median	S.D.	Total
V_{Norm}	6813	47756	20742.7	19699.5	11579.53	177,342
$V_{CodeGRU}$	5566	33867	15358.7	13733.5	8503.74	133,135

normalizing and type encoding the source code files we use the variable size context approach to generate the sequences based on source code’s syntax and structure as discussed in Section 3.2. To convert the sequences into a form that is suitable for training and to build vocabulary, the sequences are converted into vector form as discussed in Section 3.3. We use these sequences as input to CodeGRU for model training. The trained model is then used for the test purpose. For the prediction of next source code token y in a file S , we capture the token type information using *Code Analyzer* prior to the prediction position y . Then, we use the captured type information as context and input it to the model for the next source code suggestion. Then, each code token in the *Vocabulary* is computed and ranked for the possible next source code suggestion.

4. Empirical Evaluation

In this section, we provide an empirical evaluation of CodeGRU. We train and evaluate our models on Intel(R) Xeon(R) CPU E5-2620 v4 with 16 cores running at *2.10GHz* with *64GB* of ram equipped with four NVIDIA Tesla K20m GPUs running CentOS v7 operating system. It took 8 days to fully train and test all models. One important thing to mention here is that the training and testing are offline and thus have no impact on prediction time. It takes less than 30 milliseconds for source code suggestion and source code completion tasks.

To evaluate the performance of the proposed approach, we aim at answering the following research questions:

- RQ1: Does the proposed approach outperform the state-of-the-art approaches? if yes, to what extent?
- RQ2: How well does the proposed approach perform in source code suggestion and source code completion tasks?
- RQ3: To what extent the proposed approach helps reduce the vocabulary size?

To answer the research question (RQ1), we compare the performance of the proposed approach with the state-of-the-art approaches [47, 20, 32]. To answer the research question (RQ2), we evaluate and compare CodeGRU for source code suggestion and completion tasks with state-of-art approaches [47, 20, 32]. To answer the research question (RQ3), We provide the statistical results of vocabulary with and without our proposed approach. We further evaluate

CodeGRU with two software engineering applications: source code suggestion, and source code completion tasks which show CodeGRU is of practical use.

4.1. Dataset

To build our code database³, we collect open-source java projects from GitHub. We download an archive containing the latest snapshot of the projects default branch. For comparison, we choose the projects used in previous studies [20, 32, 47] summarized in Table 3. The Table 3 shows the version of the projects, total number of code lines, total code tokens and unique code tokens found in each project. We randomly choose one project(Cassandra) as an independent test set and use the rest of the projects for the model training purpose. Further, to evaluate the effectiveness of the proposed approach we adopt a random testing approach. In random testing we train the model on one project and test it with a different project (beside Cassandra and itself). To empirically evaluate our work, we repeat our experiment on each project separately. Each project is subdivided into ten equal lines of code folds from which one fold is used for validation and rest are used for training purpose. One important thing to mention here is that the test sets was never used while training or validating the models and was only used for evaluation purpose.

Table 3: List of java projects used for evaluation. The table shows name of the project, version of the project, line of code (LOC), total code tokens and unique code tokens found in each project.

Projects	Version	LOC	Code Tokens	
			Total	Unique
ant	1.10.5	149,960	920,978	17,132
cassandra	3.11.3	318,704	2734218	33,424
db40	7.2	241,766	1,435,382	20,286
jgit	5.1.3	199,505	1,538,905	20,970
poi	4.0.0	387,203	2,876,253	47,756
maven	3.6.0	69,840	494,379	8,066
batik	1.10.0	195,652	1,246,157	21,964
jts	1.16.0	91,387	611,392	11,903
itext	5.5.13	161,185	1,164,362	19,113
antlr	4.7.1	56,085	407,248	6,813

4.2. Training

We train several baseline models for the evaluation of this work. In this section, we briefly describe the baselines for comparison in detail. We train the N-gram model used in Hindle et al. [20]. We train the RNN [37] model used in White et al. [47]. We train the DNN model used in Nguyen et al. [32]. We implement our own version of DNN since it is not publicly available. In our

³<https://github.com/yaxirhuxxain/Source-Code-Suggestion>

reproduction, we follow the process described in Nguyen et al. [32]. Further, we train four different models N-gram+, RNN+, GRU, and CodeGRU by using our proposed approach. The N-gram+ and RNN+ are trained similar to the previous studies by adopting our proposed *Code Analyzer* approach. The GRU based model is trained by using our proposed approach, but without using the variable size context learning approach. The CodeGRU model is trained by using our proposed approach with variable size context as described in Section 3. For the training purpose, each source code file is tokenized as discussed earlier in Section 3.3. Then, we map the vocabulary to a continuous feature vector of dense size 300 similar to Word2Vec [39]. This approach helps us build a dense vector representation for each vocabulary index without compromising over the semantic meaning of the source code tokens.

The Table. 4 shows the architecture of deep learning based models. We choose 300 hidden units with context size 20. We use *Adam* [24] optimizer with the learn rate set to its default 0.001. To control over fitting we use *Dropout* [15] at the rate of 0.25. We employ *early stopping* [5, 40] to help stop model training when it achieves the best performance.

Table 4: Deep learning models architecture summary.

	Type	Size	Activations
Input	Code embedding	300	
Estimator	RNN,GRU	300	tanh
Over Fitting	Dropout	0.25	
Output	Dense	V	softmax
Loss	Categorical cross entropy		
Optimizer	Adam	0.001	

4.3. Prediction

For the prediction of next source code token y in a source code file S , the model takes the context information (l^i, τ^i) prior to the prediction position y . We use the *CodeAnalyzer* to capture the token’s context information as discussed in section 3. We then use the trained model to predict the most likely top-k suggestions for the given context. CodeGRU predicts the token type for an identifier and the actual token for rest of the code tokens. We use the top-k accuracy and Mean Reciprocal Rank (MRR) metrics for the evaluation of this work.

5. Results

5.1. Accuracy Comparison

For comparison, we evaluate the models with *top-k* accuracy score as done in the previous works [20, 47, 37, 32]. We calculate top-k accuracy, where

$k=1,3,5,10$. The accuracy scores for independent test (Cassandra) of all models are shown in Table 5. One can see that CodeGRU model outperforms other baseline [20, 47, 37, 32] models. We can see in Table 5 that our proposed CodeGRU achieves the max accuracy score of $46.74 @ k=1$ and $74.32 @ k=10$, whereas previous models gains much lower score $39.26 @ k=1$ and $69.12 @ k=10$. Further, we can observe that our proposed approach help improve the performance of previous studies. The simple N-gram model achieves the max score of $21.62 @ k=1$ and $31.62 @ k=10$ where as with our proposed approach the N-gram+ achieves the max score of $25.57 @ k=1$ and $37.84 @ k=10$. Similarly, by using our proposed approach the RNN+ achieves the max score of $38.90 @ k=1$ and $70.09 @ k=10$ where as the RNN model gains the max score of $37.78 @ k=1$ and $67.63 @ k=10$.

Further, to see the effectiveness of the proposed approach, we evaluate its performance with the random test set. In random testing the model is trained on one project and then tested with different a project. The random test results are presented in Table 6 where P_{train} shows the project name used for training and p_{test} shows the project name used for testing. On average it improves the accuracy ($K@1$) by 5.66% in random test and by 6.62% in independent (Cassandra) testing from the best baseline (DNN). From the results, we conclude that the proposed approach outperforms other baseline approaches in both cases, independent (Cassandra) and random testing.

5.2. Source Code Suggestion and Completion

To quantify the accuracy of our proposed approach for the source code suggestion and completion task we calculate the *Mean Reciprocal Rank (MRR)*. The MRR is a rank based evaluation metric in which suggestions that occur earlier in the list are weighted higher than those that occur later in the list. The MRR produces a value between 0-1, where the value 1 indicates perfect source code suggestion model. The MRR can be expressed as

$$MRR(C) = \frac{1}{|C|} \sum_{i=1}^{|C|} \frac{1}{y^i} \quad (8)$$

where C is code sequence and y^i refers to the index of the first relevant prediction. $MRR(C)$ is the average of all sequences C in the test data set.

Table 7 shows the MRR score of independent test set (Cassandra). The MRR score lies between 0-1, and a higher value indicates a better source code suggestion and completion model. CodeGRU achieves the lowest MRR score of 0.447 and the highest MRR score is 0.559 , while previous models lowest MRR score is 0.433 and the highest MRR score is 0.492 . On average proposed approach gains the MRR score of 0.500 , while previous studies only gain 0.465 .

The random test results are presented in Table 8. From Table 8 and Fig. 6, we make the following observations:

- The average MRR result of proposed approach is 0.524 and 0.543 , while the best baseline (DNN) average MRR score is 0.469 and 0.497 for independent (Cassandra) and random test respectively.

Table 5: Accuracy comparison of proposed approach with previous works with independent test set (Cassandra).

	K	Previous Works			Our Work			
		N-gram	RNN	DNN	N-gram+	RNN+	GRU	CodeGRU
antlr	1	17.49	36.58	37.09	21.51	38.06	41.75	43.38
	3	23.53	56.50	57.31	29.11	58.28	61.17	61.82
	5	25.51	61.93	64.01	31.63	64.98	66.49	67.50
	10	26.60	66.31	68.28	32.93	69.25	70.83	72.22
ant	1	20.42	37.73	38.38	23.78	39.35	41.32	45.01
	3	26.17	57.85	57.90	31.45	58.87	61.29	63.50
	5	27.47	63.40	63.87	33.35	64.84	66.80	68.33
	10	28.83	67.58	68.58	34.91	69.55	71.13	73.13
batik	1	16.90	34.22	34.00	20.04	34.97	35.91	38.82
	3	21.91	51.58	52.32	26.53	53.29	55.71	56.27
	5	23.43	57.61	59.41	28.27	60.38	61.76	62.66
	10	24.57	62.65	64.69	29.79	65.66	66.33	68.62
db4o	1	18.06	34.44	34.29	21.42	35.26	35.91	41.41
	3	23.25	54.41	54.89	28.30	55.86	57.03	59.44
	5	24.85	60.96	61.33	30.25	62.30	63.07	66.44
	10	26.16	65.20	66.71	31.84	67.68	68.02	71.72
itext	1	19.24	35.34	34.77	23.00	35.74	38.90	41.66
	3	24.80	56.25	56.10	30.06	57.07	59.59	62.12
	5	27.05	61.82	62.35	32.71	63.32	65.45	67.78
	10	28.38	66.16	67.53	34.35	68.50	69.79	72.22
jgit	1	21.62	35.92	36.35	25.57	37.32	41.31	45.98
	3	28.01	56.96	58.23	33.67	59.20	61.71	65.33
	5	30.27	63.73	64.12	36.32	65.09	67.46	70.12
	10	31.62	67.60	69.12	37.84	70.09	71.52	74.32
jts	1	16.20	35.60	35.01	19.78	35.98	38.18	39.31
	3	21.11	54.50	54.41	26.08	55.38	57.42	56.53
	5	23.01	59.40	60.29	28.34	61.26	62.50	63.29
	10	23.94	63.47	64.62	29.55	65.59	66.88	68.11
maven	1	18.82	37.78	39.26	22.00	40.23	43.11	44.35
	3	23.58	57.20	58.23	28.32	59.20	61.41	61.71
	5	25.42	61.79	63.59	30.17	64.56	65.89	67.15
	10	26.39	65.38	67.44	31.63	68.41	69.95	71.68
poi	1	21.37	37.78	37.93	25.09	38.90	43.38	46.74
	3	28.08	57.66	57.66	33.48	58.63	62.89	65.11
	5	29.67	63.62	64.55	35.49	65.52	68.15	69.29
	10	30.93	67.63	68.98	37.04	69.95	72.08	73.74

Table 6: Accuracy comparison of proposed approach with previous works with random test set where P_{train} shows the project name used for training and p_{test} shows the project name used for testing.

P_{train}	P_{test}	K	Previous Works			Our Work			
			N-gram	RNN	DNN	N-gram+	RNN+	GRU	CodeGRU
antlr	batik	1	25.20	35.72	39.29	32.41	40.25	43.27	44.06
		3	33.48	55.04	58.05	45.43	59.01	63.14	62.27
		5	36.21	60.16	64.29	48.75	65.25	68.00	68.06
		10	37.80	65.03	69.15	51.18	70.12	72.23	72.16
ant	db4o	1	24.50	36.63	36.85	30.48	37.82	39.45	42.01
		3	31.97	56.22	56.14	39.58	57.11	59.24	59.89
		5	34.50	61.75	62.29	42.59	63.26	65.52	64.40
		10	36.40	66.62	67.78	45.35	68.74	70.30	71.18
batik	ant	1	33.75	40.29	43.81	42.05	44.78	44.61	48.12
		3	42.83	57.10	61.47	54.92	62.43	63.39	64.76
		5	45.56	62.37	66.78	58.09	67.74	68.85	70.50
		10	47.64	67.09	71.92	61.50	72.89	73.56	75.89
db4o	jgit	1	24.21	35.52	35.93	30.56	36.90	37.94	41.97
		3	34.59	55.61	56.61	42.56	57.57	58.86	60.19
		5	36.70	61.89	62.30	59.36	63.27	64.47	67.01
		10	38.74	65.88	67.39	49.36	68.35	69.28	72.07
itext	jts	1	24.88	36.84	39.14	33.29	40.11	41.66	44.14
		3	34.03	55.81	58.61	47.58	59.57	62.29	63.77
		5	37.14	61.72	64.24	51.00	65.21	67.57	68.74
		10	38.80	66.28	68.88	53.19	69.84	71.99	73.14
jgit	itext	1	27.49	35.39	38.13	33.81	39.10	41.99	47.79
		3	35.12	56.00	58.26	46.75	59.22	62.52	65.33
		5	37.65	61.84	64.15	49.95	65.12	67.67	70.77
		10	39.56	66.92	69.24	52.71	70.65	72.31	76.32
jts	poi	1	30.09	40.21	40.56	33.29	41.53	43.61	44.65
		3	40.01	57.88	58.35	48.40	59.31	62.07	60.89
		5	43.50	62.72	63.47	53.08	64.44	66.67	66.94
		10	44.84	66.63	67.75	54.86	68.72	70.64	71.69
maven	antlr	1	25.99	39.77	41.67	35.53	42.63	46.43	47.68
		3	32.58	58.26	60.90	45.12	61.87	65.40	61.25
		5	34.95	63.66	66.13	48.22	67.87	69.75	67.47
		10	37.04	67.18	70.37	50.99	71.33	73.35	73.41
poi	maven	1	33.12	41.42	41.26	36.11	42.22	48.17	49.14
		3	41.85	60.76	60.90	52.21	61.87	65.22	66.16
		5	44.34	65.48	66.38	55.89	67.34	69.65	69.66
		10	46.37	69.15	70.30	58.25	71.27	72.90	73.58

Table 7: Mean Reciprocal Rank comparison of our proposed approach with previous works with independent test set (Cassandra).

	Previous Works			Our Work			
	N-Gram	RNN	DNN	N-gram+	RNN+	GRU	CodeGRU
antlr	0.209	0.467	0.479	0.258	0.486	0.517	0.528
ant	0.235	0.478	0.487	0.280	0.494	0.515	0.544
batik	0.197	0.433	0.440	0.236	0.447	0.461	0.481
db4o	0.209	0.447	0.451	0.252	0.458	0.467	0.511
itext	0.225	0.457	0.458	0.270	0.465	0.494	0.520
jgit	0.252	0.467	0.477	0.301	0.484	0.516	0.557
jts	0.190	0.450	0.452	0.233	0.459	0.480	0.485
maven	0.215	0.478	0.492	0.255	0.499	0.523	0.534
poi	0.250	0.475	0.485	0.296	0.492	0.533	0.559
Average	0.220	0.461	0.469	0.265	0.476	0.501	0.524

- From the results, we conclude that the proposed approach outperforms other baseline approaches.

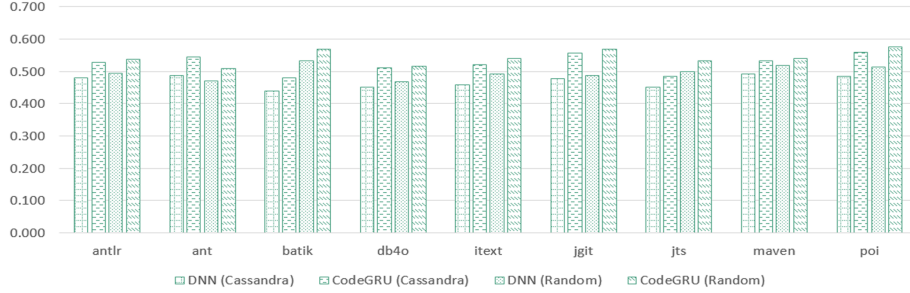


Figure 6: MRR Score Comparison.

We conduct ANOVA to measure the statistical significant difference between the proposed approach and the best baseline (DNN). ANOVA is conducted on MRR, where the unit of analysis is a project. We conduct ANOVA on Microsoft Excel version 2016 with its default setting, and no adjustments are involved. Table 9 shows $F > F_{crit}$ and $p\text{-value} < (\alpha = 0.05)$ are true for MRR in both cases (Cassandra and Random test); therefore, we reject the null hypothesis, suggesting that a statistically significant difference exist.

Table 8: Mean Reciprocal Rank comparison of our proposed approach with previous works with random test set where P_{train} shows the project name used for training and P_{test} shows the project name used for testing.

P_{train}	P_{test}	Previous Works			Our Work			
		N-Gram	RNN	DNN	N-gram+	RNN+	GRU	CodeGRU
antlr	batik	0.298	0.455	0.495	0.395	0.501	0.533	0.537
ant	db4o	0.286	0.465	0.471	0.356	0.478	0.496	0.508
batik	ant	0.388	0.490	0.532	0.491	0.539	0.542	0.570
db4o	jgit	0.297	0.458	0.467	0.371	0.474	0.485	0.515
itext	jts	0.299	0.493	0.493	0.410	0.499	0.521	0.540
jgit	itext	0.319	0.458	0.487	0.410	0.494	0.526	0.568
jts	poi	0.357	0.490	0.499	0.416	0.505	0.530	0.532
maven	antlr	0.298	0.492	0.518	0.410	0.525	0.560	0.540
poi	maven	0.380	0.510	0.515	0.444	0.521	0.568	0.576
Average		0.326	0.476	0.497	0.411	0.504	0.529	0.543

Table 9: ANOVA Analysis on MRR Scores

<i>Source</i>	<i>SS</i>	<i>df</i>	<i>MS</i>	<i>F</i>	<i>P-value</i>	<i>F crit</i>
<i>MRR (Cassandra)</i>						
Between Groups	0.013661	1	0.013661	23.71224562	0.00017043	4.493998
Within Groups	0.009218	16	0.000576			
Total	0.022879	17				
<i>MRR (Random)</i>						
Between Groups	0.009253	1	0.009253	17.99466376	0.00062147	4.493998
Within Groups	0.008227	16	0.000514			
Total	0.01748	17				

Where, SS = sum of squares, df = degree of freedom, MS = mean square.

5.3. Impact of CodeGRU on Vocabulary

NLP based deep learning models suffer from the vocabulary size. Usually, they are trained on a fixed size vocabulary and replace the out of vocabulary tokens with some special token while testing. The same approach has been employed by previous studies [20, 47, 20]. In this work, we employ a novel approach of capturing the token types which helps reduce the vocabulary size up to 24.93% which suggest that our approach can help minimize the out of vocabulary issue. Table 10 shows the vocabulary statistic with and without our proposed approach. By capturing the token type information the vocabulary size reduces significantly, in some cases over 30%. Vocabulary size reduction helps overcome two limitations; It helps minimize the out of vocabulary issue and reduce the time, and computation cost for model training. It took approximately 4-6 hours to train the *jts* model by using the proposed approach whereas without our proposed approach it took 8-20 hours for the same project.

Table 10: Vocabulary comparison with and without our proposed approach.

Projects	Code Tokens	V_{Norm}	$V_{CodeGRU}$	% Decrease
ant	920,978	17,132	12,417	27.52%
cassandra	2,734,218	33,424	26,960	19.34%
db40	1,435,382	20,286	16,397	19.17%
jgit	1,538,905	20,970	16,433	21.64%
poi	2,876,253	47,756	33,867	29.08%
maven	494,379	8,066	6,770	16.07%
batik	1,246,157	21,964	14,643	33.33%
jts	611,392	11,903	7,710	35.23%
itext	1,164,362	19,113	12,824	32.90%
antlr	407,248	6,813	5,566	18.30%
Total	13,429,274	177,342	133,135	24.93%

5.4. Improving the Performance of CodeGRU

We conduct an experiment to study the impact of different hyper parameters on CodeGRU performance. We tested various model settings varying the hidden units (*100,200,300*) with the context size of (*10,15,20*) and optimizer (*Adam, RMSprop*). In our experiments, we found that CodeGRU performs well when the hidden units are *300* and context size is *20*. The high values of both parameters (context size and hidden units) are not surprising and most commonly used in various source code modeling tasks [47, 40]. The value of context size (20)

reflects that the model is capable of effectively learning the long-term context dependencies of the source code by utilizing 300 hidden units(neurons) whereas smaller context size or hidden units cause the model to under-fit. Further, to alleviate the issue of overfitting, we have adopted dropout regularization which helps prevent the model from overfitting. We choose the code embedding size of 300 to match the hidden units size. Further, in our experiments we found that *Adam* optimizer performed well as compared to *RMSprop* optimizer with the learn rate set to its default 0.001 in both cases. We conclude from the experiments that the architecture matching Table 4 performed well as compared to other settings. Furthermore, we empirically selected GRU over RNN and LSTM because of its good performance. GRU is an advanced version of LSTM which works similar to it but performs better [42] by exposing whole hidden context. We train RNN, LSTM and GRU based models by adopting the proposed approach. Fig. 7 shows the average accuracy (K@1) score of each model with independent test set. The performance of RNN and GRU is fairly similar. However, RNN suffers from the vanishing gradient issue which can be overcome by using GRU. Further, GRU exposes the whole hidden context on each time step which is ideal for source code modeling tasks where context dependencies are separated far apart. From the results, we can perceive that the GRU based approach performs well as compared to others.

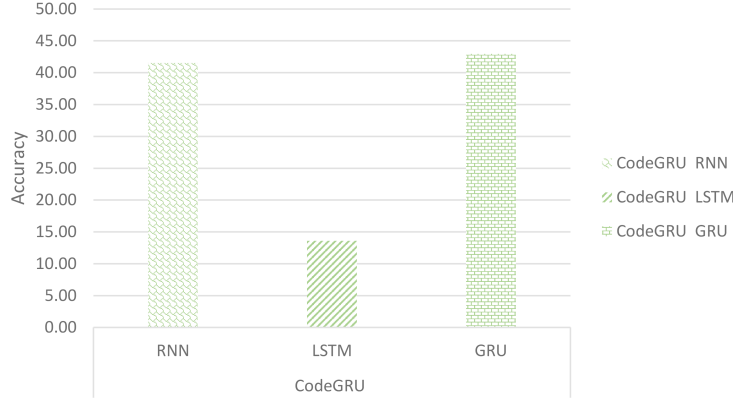


Figure 7: Performance of CodeGRU with different recurrent neural architectures.

5.5. Time and memory Cost

While conducting our experiments we retain time and space cost of using *Code Analyzer*. Table 11 shows the time and space cost of each project. For other processes, including tokenization, building sequences, models training, and testing are all common procedures, so we do not analyze their costs. Among all the projects, the time and space cost varies from 4.59 seconds with memory cost of 2.29MB for *antlr* to 32.0 seconds with memory cost of 16.3MB for *poi*.

Table 11: Time and space cost of Code Analyzer.

Project	Time(sec)	Space(MB)
antlr	4.59	2.29
ant	10.55	4.93
batik	14.66	7.08
cassandra	30.4	15.7
db4o	17.37	7.58
itext	13.81	6.25
jgit	16.7	8.37
jts	6.94	3.44
maven	5.98	3.04
poi	32.3	16.3

6. Applications of CodeGRU

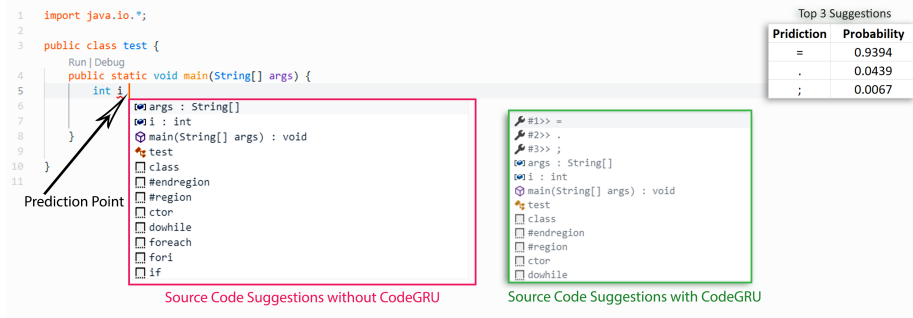
In this section, we discuss CodeGRU with two software engineering applications: code suggestion, which aims to suggest multiple predictions for the next code token, and code completion, which aims to complete the whole next source code sequence. We use Visual Studio Code (VSC) IDE version *1.31.1* for the demonstration purpose. The demonstration is conducted on Intel(R) Core(TM) i5-6500 3.20GHz with 4 cores and 8GB of ram running windows 10 operating system. The CodeGRU takes less than 260MB memory space for the prediction purpose with the largest subject project (POI). We use CodeGRU to help predict the top-k suggestions given a source code context and then use the type information from the IDE to help suggest the identifier names as discussed earlier in section 3.

6.1. Code Suggestion

CodeGRU is capable of ranking the next code token suggestions by calculating the likelihood based on a given source code context. In Fig. 8a one can see an example for source code suggestion task where a software developer is defining a variable at line five and the most probable next source code token can be `=` but in visual studio code it does not provide any relevant suggestion whereas CodeGRU suggests the correct next code token at first position of its suggestion list. Consider a more complex example Fig. 8b where a software developer is about to print a `ListIterator` item at line 31. We can see from the given context the CodeGRU suggest the possible next source code token at its first index whereas visual studio code rank it on its 4th index. In Fig. 8 one can see CodeGRU suggest the next code token in its top three suggestion list.

6.2. Code Completion

The CodeGRU is capable of completing the whole sequence of code with correct syntax and context. Lets take the example discussed in Fig. 9 where a



(a)



(b)

Figure 8: Source Code suggestion example in visual studio code IDE.

developer is writing source code `if (i <=` at line 6 in visual studio code. As Fig. 9 shows CodeGRU provides the top-3 possible code completions. We can observe that CodeGRU capture the data types of the identifiers `i` and `arg` and assign the highest probabilities to the sequence `if (i <= IntLiteral) {` and `if (i <= arg . Length) {`. We can also observe that it give low probability to `if (i <= i) {` which is the most unlikely sequence given the context.

6.3. Utility Applications

The application of the proposed approach is not limited to the preceding ones. Usually in the modeling of source code identifiers are removed or replaced by some generic code token (e.g. *idf*, *unk*, etc.) which may present vital information to help improve the model's performance. The proposed approach can help improve the performance of such approaches [28, 40, 46, 30, 49] by considering the token type information. One possible application of our work can be syntax error correction similar to Santos et al [40]. They present NLP based recurrent neural model to detect and correct syntax errors. Their work considers the single token syntax error (113 syntax token) and replacing the

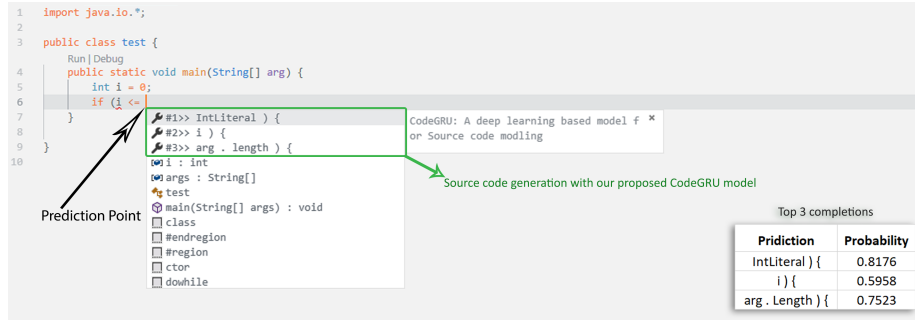


Figure 9: Source code completion example.

identifies with a generic token. Another possible application of our approach can be bug prediction similar to Wang et al. [46]. Our proposed approach may help improve these works [28, 40, 46, 30, 49] by capturing the token type information.

7. Limitations and Threats to Validity

7.1. Limitations:

One of the limitations of NLP based models is vocabulary size. Vocabulary size reduction helps in many ways, first, it helps reduce the train time of the model. Second, it helps reduce the computational complexity. Larger the vocabulary size of a model, more time and computational power it will require to train the model. Although our approach reduces the vocabulary size by 24.93% by capturing the token type information, still there may be some identifier types which may not be captured while training the model. This limitation can be overcome by training the model with even larger data set, but it will increase the vocabulary size which will increase the computational complexity. A solution to this problem is to use char level encoding. Another limitation of this work is the gap concerning the recommendation on the identifier type and the possible identifier names that developers may pick for the source code suggestion task. In this work, we capture the source code context by capturing the token type information and fill the identifier names based on the captured token types. This approach may possibly overload the developer when there are a large number of possible initializations for a token type. In our future work, we consider to overcome these limitations and to provide a more robust approach.

7.2. Construct Validity:

All models are developed using *keras* version 2.2.4 with *tensorflow* version 1.13.1 backend. Although our experiments are detailed and results have shown the effectiveness of our approach but still neural networks are in its infancy. Change in neural network settings, training or evaluating with a different test set may produce different results. Another threat to construct validity is the

suitability of the evaluation metrics. The Top-k [47, 20, 32] metric is commonly used for the evaluation of deep learning based source code models. However, the proposed approach is further evaluated with MRR [32, 40] and ANOVA metrics to show its effectiveness and to alleviate this threat.

7.3. Internal Validity:

Our tool suggests the most relevant suggestions provided by the language model given a source code context but the choice of identifier names is still up to the developer. Our tool will suggest the token type for an identifier and actual code token for the rest of the code tokens. Further, We have adopted a beam search approach for the source code completion task. The completion provided by CodeGRU model may not be accurate in some cases as it selects the first prediction for each next token where the correct source code token may not be on the first index. Another threat to internal validity is the implementation of the baselines. We follow the same process as described in the original manuscript of baselines. To alleviate this threat we double checked the implementation and the results; However, there could be unnoticed imperfections.

7.4. External Validity:

A threat to external validity is that the generalization of our results. The data set used in this study was collected from *GitHub*, a well-known source code repositories provider. It is not necessary that the projects used in this study represent Java language or other languages code entirely. Another threat to external validity is the choice of hyper-parameters. There is no universal approach to learn the best model parameters, thus the parameter tuning is mainly empirical.

8. Related Work

Most of the modern IDEs provide code completion and code suggestion features. In recent years, deep neural techniques have been successfully applied to various tasks in natural language processing, and also have shown its effectiveness to problems such as code completion, code suggestion, code generation, API mining, code migration, and code categorization. In this section, we discuss prior works which are relevant to this research.

Hindle et al. [20] have shown how natural language processing techniques can help in source code modeling. They provide a *n-gram* based model which helps predict the next code token in *Eclipse IDE*. Raychev et al. [37] used statistical language model for synthesizing code completions. They applied *n-gram* and *RNN* language model for the task of synthesizing code. Tu et al. [44], proposed a cache based language model that consists of an *n-gram* and a *cache*. Hellendoorn et al. [19] further improved the cache based model by introducing nested locality. White et al. [47] applied deep learning for source code modeling purpose. Another approach for source code modeling is to use probabilistic context-free grammars(PCFGs) [8]. Allamanis et al. [1] used a

PCFG based model to mine idioms from source code. Maddison et al. [26] used a structured generative model for source code. They evaluated their approach with *n-gram* and *PCFG* based language models and showed how they can help in source code generation tasks. Raychev et al. [38] applied decision trees for predicting API elements. Chan et al. [10] used a graph-based search approach to search and recommend API usages.

Recently there has been an increase in API usage [45, 23, 12] mining and suggestion. Thung et al. [43] introduced a recommendation system for API methods recommendation by using feature requests. Nguyen et al. [33] proposed a methodology to learn API usages from byte code. Allamanis et al. [1] introduced a model which automatically mines source code idioms. A neural probabilistic language model introduced in [2] that can suggest names for the methods and classes. Franks et al. [13] created a tool for Eclipse named *CACHECA* for source code suggestion using a *n-gram* model. Nguyen et al. [31] introduced an *Eclipse plugin* which provide context-sensitive code completion based on API usage patterns mining techniques. Chen et al. [10] created a web-based tool to find analogical libraries for different languages. Wang et al. [46] proposed a Deep Belief Network (DBN) based model for inter and cross-project bug prediction. They use source code’s AST to learn the semantic features of source code. Mou et al. [30] proposed a Convolutional Neural Networks over tree structures to capture source code structural information. They show the effectiveness of their approach for the task of categorizing source code programs based on their functionality. Both works [46, 30] considers certain parts of AST and remove the rest which present vital information for other tasks such as Source code suggestion, code completion etc.

Yin et al. [51] have proposed a source code generation approach that serially apply actions from a grammar model to generate an abstract syntax tree. A similar work conducted by Rabinovich et al. [34], which introduced an abstract syntax networks modeling framework for tasks like code generation and semantic parsing. Sethi et al. [41] introduced a model which automatically generate source code from deep Learning based research papers. Allamanis et al. [4] proposed a bimodal to help suggest source code snippets with a natural language query. It is also capable of retrieving natural language descriptions with a source code query. Recently deep learning based approaches have widely been applied for source code modeling. Such as code summarization [21, 3, 17], code mining [50], clone detection [25], API learning [16] etc. Different from the Santos et al. [40] and Gupta et al. [18], our work focuses on source code suggestion tasks, whereas their works focus on fixing syntax errors. Their work requires the compilation of code after making a fix whereas our work does not require compilation of source code and can provide suggestions instantly.

Our work is relevant to the [20, 47, 37, 32] works, however, it varies from them in several important ways. Hindle et al. [20] have shown that source codes are natural and a simple SLM (n-gram) based model can capture the regularities in them. White et al. [47] and Raychev et al. [37] have shown that the neural network based models can capture the regularities much more effectively than n-gram [47] based models. They applied *RNN* based model to

show how deep learning can help improve source code modeling. However, these works [20, 47, 37] consider source code as simple tokens of text whereas our work considers the source code contextual, syntactical and structural information. Further, they treat source code as a single sequence of text tokens with fixed size context window, while we employed a novel variable size context learning approach which shows improvement in the modeling of source code. The most similar work to ours is DNN [32] however, it varies in several important ways. First, They apply deep neural networks for source code modeling with a fixed size of context. Their work considers the context size of $n=4$, where larger size may cause scalability problem as mentioned in their work [32], while our work employed a novel approach of variable size context learning with the upper bound limit of 20 tokens. Further, their approach is limited to Java language, while our work adopts a general approach which can work with different static type languages. Finally, they train and test the model on the same project by splitting each project into ten folds from which one fold is used for test purpose and rest are used for training purpose. Our model is trained on one project and tested on a separate independent project which shows the proposed approach is capable of predicting cross-project source code tokens.

9. Conclusion

This paper presented CodeGRU, a novel approach for source code modeling which captures source code’s contextual, structural and syntactical information. The work proposed a novel approach which can capture the source code context by leveraging the token type information. The CodeGRU can effectively capture the right context even it is separated far apart in the code. CodeGRU further proposed a novel approach which learns variable size context while modeling the source code. The evaluation has shown that CodeGRU outperforms the state-of-the-art language models and help reduce the vocabulary size up to 24.93% which suggest it can help minimize the out of vocabulary issue. We further evaluated CodeGRU with two software engineering applications: (1) source code suggestion, which can suggest multiple predictions for the next code token, and (2) code completion, which can complete whole next code sequence which shows that it is of practical use.

In the future, we would like to evaluate our approach for the dynamic typed languages. In dynamic type languages, a source code token can have different types which makes it difficult to capture the right context. We also aim at providing an end to end solution with a large data set which can help software developers directly utilize these models for both static and dynamic typed languages. Another limitation of deep learning based approaches is computation power, where training a new model require additional resources. A common software developer cannot afford to have a server or GPU based system to train these models. There is a need for centralizing these language models which can directly benefit software developers with minimum effort.

Acknowledgments

This work was supported by the National Key R&D (grant no. 2018YFB1003902), Natural Science Foundation of Jiangsu Province (No. BK20170809), National Natural Science Foundation of China (No. 61972197) and Qing Lan Project.

References

- [1] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM, 2014.
- [2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.
- [3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.
- [4] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *International conference on machine learning*, pages 2123–2132, 2015.
- [5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.
- [6] Jerome R Bellegarda. Large vocabulary speech recognition with multispans statistical language models. *IEEE Transactions on Speech and Audio Processing*, 8(1):76–84, 2000.
- [7] Adam Berger and John Lafferty. Information retrieval as statistical translation. In *ACM SIGIR Forum*, volume 51, pages 219–226. ACM, 2017.
- [8] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.
- [9] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.
- [10] Chunyang Chen and Zhenchang Xing. Similartech: automatically recommend analogical libraries across different programming languages. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 834–839. IEEE, 2016.

- [11] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *arXiv preprint arXiv:1608.02715*, 2016.
- [12] Andrea Renika DSouza, Di Yang, and Cristina V Lopes. Collective intelligence for smarter api recommendations in python. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 51–60. IEEE, 2016.
- [13] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 705–708. IEEE Press, 2015.
- [14] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.
- [15] Amit Gajbhiye, Sardar Jaf, Noura Al Moubayed, A Stephen McGough, and Steven Bradley. An exploration of dropout with rnns for natural language inference. In *International Conference on Artificial Neural Networks*, pages 157–167. Springer, 2018.
- [16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.
- [17] Latifa Guerrouj, David Bourque, and Peter C Rigby. Leveraging informal documentation to summarize classes and methods in context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 639–642. IEEE, 2015.
- [18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language error by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- [19] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.
- [20] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.
- [21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 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)*, pages 2073–2083, 2016.

- [22] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184. ACM, 2014.
- [23] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675. ACM, 2014.
- [24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [25] Balwinder Kumar and Satwinder Singh. Code clone detection and analysis using software metrics and neural network-a literature review. *Complexity*, 1(2):3, 2015.
- [26] Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657, 2014.
- [27] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.
- [28] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. Improving code readability classification using convolutional neural networks. *Information and Software Technology*, 104:60–71, 2018.
- [29] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.
- [30] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- [31] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 69–79. IEEE, 2012.
- [32] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N Nguyen. A deep neural network language model with contexts for source code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 323–334. IEEE, 2018.

- [33] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Learning api usages from bytecode: a statistical approach. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 416–427. IEEE, 2016.
- [34] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.
- [35] Martin Rajman and Romaric Besançon. Text mining: natural language techniques and text mining applications. In *Data mining and reverse engineering*, pages 50–64. Springer, 1998.
- [36] Nihar Ranjan, Kaushal Mundada, Kunal Phaltane, and Saim Ahmad. A survey on techniques in nlp. *International Journal of Computer Applications*, 134(8):6–9, 2016.
- [37] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.
- [38] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *ACM SIGPLAN Notices*, volume 51, pages 761–774. ACM, 2016.
- [39] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.
- [40] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE, 2018.
- [41] Akshay Sethi, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. Dlpaper2code: Auto-generation of code from deep learning research papers. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [42] Yi Tay, Anh Tuan Luu, and Siu Cheung Hui. Recurrently controlled recurrent networks. In *Advances in Neural Information Processing Systems*, pages 4731–4743, 2018.
- [43] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. Automatic recommendation of api methods from feature requests. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 290–300. IEEE Press, 2013.
- [44] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM, 2014.

- [45] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.
- [46] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.
- [47] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE Press, 2015.
- [48] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology*, 99:58–61, 2018.
- [49] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 105:17–29, 2019.
- [50] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.
- [51] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.
- [52] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *IEEE Computational intelligence magazine*, 13(3):55–75, 2018.