SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2020, Vol. 63 190101:1–190101:22
https://doi.org/10.1007/s11432-019-2830-8

Special Focus on Big Data-Driven Intelligent Software Development for Onsite Programming

LSTM-based argument recommendation for non-API methods

Guangjie LI[1], Hui LIU[1*], Ge LI[2], Sijie SHEN[2] & Hanlin TANG[1]

[1]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing* 100081*, China;*
[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China*

Received 19 September 2019/Revised 27 December 2019/Accepted 20 January 2020/Published online 14 August 2020

Abstract Automatic code completion is one of the most useful features provided by advanced IDEs. Argument recommendation, as a special kind of code completion, is widely used as well. While existing approaches focus on argument recommendation for popular APIs, a large number of non-API invocations are requesting for accurate argument recommendation as well. To this end, we propose an LSTM-based approach to recommending non-API arguments instantly when method calls are typed in. With data collected from a large corpus of open-source applications, we train an LSTM neural network to recommend actual arguments based on identifiers of the invoked method, the corresponding formal parameter, and a list of syntactically correct candidate arguments. To feed these identifiers into the LSTM neural network, we convert them into fixed-length vectors by Paragraph Vector, an unsupervised neural network based learning algorithm. With the resulting LSTM neural network trained on sample applications, for a given call site we can predict which of the candidate arguments is more likely to be the correct one. We evaluate the proposed approach with ten-fold validation on 85 open-source C applications. Results suggest that the proposed approach outperforms the state-of-the-art approaches in recommending non-API arguments. It improves the precision significantly from 71.46% to 83.37%.

Keywords argument recommendation, LSTM, deep learning, non-API

Citation Li G J, Liu H, Li G, et al. LSTM-based argument recommendation for non-API methods. Sci China Inf Sci, 2020, 63(9): 190101, https://doi.org/10.1007/s11432-019-2830-8

1 Introduction

Code completion is a feature in which an IDE predicts the rest (or a part of the rest) of a statement that a developer is typing. It may speed up coding if it frequently predicts correctly the statement that a user intends to enter after only a few characters have been typed [1]. Code completion has been found to be one of the top ten commands in Eclipse IDE used by developers [2], which suggests that code completion is widely used.

Recommendation of arguments for method calls is a special kind of code completion [3]. While a developer is typing a method call, e.g., updateTitle(), IDEs will check possible arguments and recommend the most likely one or present a list of ranked candidates. Most of the mainstream IDEs recommend actual arguments according to the types of candidate arguments and the types of corresponding formal parameters, i.e., candidate arguments should be type compatible with the corresponding formal parameter. However, it is quite often that there are a large number of type compatible candidates. In this case, type based argument recommendation results in a long list of candidates, and it takes quite a long time to select.

* Corresponding author (email: liuhui08@bit.edu.cn)

© Science China Press and Springer-Verlag GmbH Germany, part of Springer Nature 2020

To improve the performance of argument recommendation, a few approaches have been proposed. Zhang et al. [4] proposed Precise. It employs an k-nearest neighbor algorithm to recommend arguments for API usage based on the nearest k usages of the invoked API method. Asaduzzaman et al. [5] tried to improve the Precise by exploiting the localness of API argument recommendation, i.e., the lines of source code before the call site. Raychev et al. [6] proposed a statistical language model based approach, called SLANG, to recommend API methods as well as API arguments based on the sequences of API method calls and their associated arguments. Liu et al. [3] proposed a similarity-based approach to select correct argument from candidate arguments. Hellendoorn et al. [7] proposed SLP-Core to model and complete source code based on cached n-gram, which can be used to recommend arguments as well. Although such approaches [3–7] work well in recommending arguments for popular APIs, they may not be extended easily to recommend arguments for non-API methods (methods defined within the project where they are invoked). For example, Precise depends on k usages of the invoked method. Consequently, it may not work for non-API methods because it is likely that no usage of the invoked method is available at the point of recommendation.

By analyzing 85 open-source C applications (see details in Section 5), we observe that 47% (= 972660/2097709) of the method invocations are non-API invocations. Such a large number of non-API invocations suggest that there is a great need for approaches that can recommend arguments for non-API method invocations. To this end, in this study we propose a novel approach to recommending arguments for non-API methods which have no rich invocation histories or even no invocation available at the point of recommendation. We observe (as explained in Section 2) that in most cases developers can select the correct argument from a list of syntactically correct candidates just according to the formal signature of the invoked method and the list of candidates without looking into the invocation history of the invoked method. Consequently, with a large number of real data from open-source applications, deep learning techniques may learn generic quantitative rules as well to choose the correct arguments from a list of candidate arguments without looking into the invocation history of the invoked method. To this end, we propose an LSTM-based approach to recommending arguments for non-API methods. For each of the arguments in the applications, the proposed approach generates a tuple containing the actual argument, the invoked method, the corresponding formal parameter, and a list of syntactically correct alternative arguments. Such alternatives, together with the actual argument, compose the solution space for the call site. The solution space, generated by static source code analysis, turns the problem of generating a correct argument into the problem of selecting the correct argument from a number of candidates. As a result, our approach may succeed even if the correct arguments have never appeared in the training data (it is common for non-API invocations).

To evaluate the proposed approach, we carry out a ten-fold validation on 972660 non-API arguments (and their context) extracted from 85 open-source C applications. Evaluation results suggest that the proposed approach outperforms the state-of-the-art approaches in recommending non-API arguments. The precision is improved from 71.46% to 83.37%.

In summary, this study makes the following contributions:

- An LSTM based approach to recommending arguments for non-API method calls. To the best of our knowledge, we are the first to adapt LSTM technique to argument recommendation. We are also the first one to train neural networks to select the correct argument from a small list of syntactically correct candidates.

- Ten-fold validation of the proposed approach on open-source applications whose results suggest that the proposed approach outperforms the state-of-the-art approaches.

The rest of the paper is structured as follows. Section 2 presents a case study on how developers select arguments from a list of candidates. Section 3 presents the problem statement. Section 4 proposes the approach to recommending arguments. Section 5 presents an evaluation of the proposed approach on open-source applications. Section 6 discusses related issues. Section 7 presents a short review of related research. Section 8 provides conclusion and future work.

2 Case study on argument selection

2.1 Questions

To investigate whether developers can choose the correct argument from a list of candidates without looking into the invocation histories, we carry out a case study which is interested in investigating the following questions.

- **Q1:** Whether developers can choose the correct argument when only given current invocation context whereas without invocation histories? If yes, how many percentage of correct arguments can be chosen?
- **Q2:** If developers can often choose the correct argument based on current invocation context, what are the quantitative rules that the selections are based on?

Question Q1 concerns how often developers can choose the correct argument when only the current invocation context are available. Answering this question may reveal whether we can recommend arguments based on context information. If the developers can choose the correct argument in most cases, it may reveal that there are potential semantic information embedded in the context that we may learn from, which is the assumption of this study.

Question Q2 concerns how developers choose the correct arguments. If developers can abstract quantitative rules of argument selections, we can conduct further research based on such rules. If developers cannot figure out any quantitative rules, we may have the reason to resort to deep learning techniques to learn the potential relationship between given contexts and correct arguments, because deep learning techniques are specialized in mapping relationship from big data.

2.2 Subject applications

We search for well-known and popular open-source C applications from GitHub[1)] that have at least 5 releases and can be imported into Eclipse. From such applications, we select the top 85 resulting applications with most stars. Such applications would be used in the evaluation in Section 5 as well. For each actual argument in the non-API method invocation from the resulting 85 applications, we extract the argument name, method name, formal parameter name, and top five lexically similar and syntactically correct candidate arguments. In total, we get 972660 non-API arguments. From such non-API arguments, we randomly select 30 arguments that satisfy the following conditions as the subject arguments of our case study.

- **C1:** For each argument, there are at least four syntactically correct alternative candidates available.
- **C2:** The name of the invoked method is composed of no less than two words.
- **C3:** The corresponding parameter name is composed of no less than four characters.
- **C4:** The potential relationship between the semantic of argument and that of their corresponding context is confirmed by the authors.

The first condition makes sure that we will not choose the extremely simple cases where there is only a very small number of alternatives (or even no alternatives at all). The second and third conditions make sure that identifiers of software entities are named properly. The last condition makes sure that the underlining semantics are noticeable for experienced developers. Among the 972660 arguments, there are 747437 arguments matching the criteria. The average syntactically correct alternative candidates for each non-API argument is 5.6.

2.3 Respondents

We invite 25 respondents to attend this case study. All respondents are software developers with at least one-year software development experience in real projects. It should be noted that all developers involved in the case study have no access to source code of the subject applications.

We choose such developers as the respondents for the following reasons. First, it is difficult to invite so many experienced developers from industries to attend such case study. Second, we constrain

1) GitHub. 2019. https://github.com/.

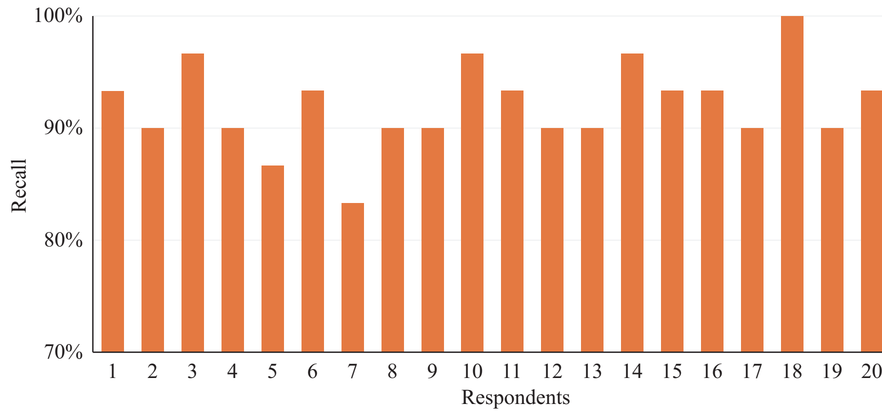

Figure 1 (Color online) Results of manual argument selection.

that all respondents have at least one-year software development experience because we assume that only experienced developers can mine the semantics between actual arguments and context of method invocations.

2.4 Set up

We set up the experiment according to the following process. First, we design an questionnaire which includes 30 argument selection questions. Each question includes two parts: the first part is the question part, which gives the method name, the formal parameter name, and asks developers to select correct arguments from the following choices. The second part is the answer choices, which include alphabetical list of 4–5 syntactically correct candidates (including the current argument) marked as choices A–E. Second, we publish the questionnaire through the online questionnaire platform Wenjuanxing[2)], send the URL of the questionnaire to developers, and ask each developer to choose the correct argument from the alphabetical list of candidates. Third, once developers complete the argument selection, we ask them to write out the their quantitative rules for argument selection or how they make selections. If they do not have such rules, we suggest them to say "do not know". After that, they submit the responses. Finally, we recall the responses from the back-end of Wenjuanxing and conduct statistical analysis on the data.

In total we send such questionnaires to 25 developers and receive 20 complete responses (who finished all of the 30 selection questions), 2 developers do not return back the questionnaires, and 3 developers return uncompleted responses.

2.5 Results

2.5.1 *Q*1*: Correct argument selections*

Results are presented in Figure 1. The horizontal axis represents different respondents, and vertical axis presents recall, i.e., how many percentage of correct arguments have been correctly chosen by the respondents. Assuming that the current arguments in the original source code are correct, the chosen arguments are correct only if they are identical to the current arguments.

From Figure 1, we observe that developers can select the correct arguments in most cases (92% = 552/600). The recall varies between 83.3% and 100%, with an average of 92%. The minimal recall is up to 83.3%, suggesting that all of the respondents can frequently make correct selection without looking into the related invocation histories.

2) https://www.wjx.cn/.

2.5.2 *Q2: Quantitative rules*

We also note that none of the respondents specify quantitative rules for argument selection. Most (16 out of 20) of them declare that they select arguments based on the semantics of identifiers. They guess the functionality of the invoked method according to the method name, the semantics of parameters, and candidate arguments. However, the semantics and functionality are difficult to quantify, and thus they fail to specify the quantitative selection rules.

We conclude from the case study that: in most cases developers can select the correct argument from a list of syntactically correct candidates just according to their identifiers as well as the formal signature of the invoked method, although most of them cannot specify quantitative rules for their selection.

2.6 Validity threats

The first threat to the external validity of the case study is that the data involved in the case study are extracted only from a small set of applications. As a result, the conclusion drawn from such data may not hold for other applications. To reduce this threat, we select the 85 most popular open-source applications from GitHub, where the source code is more likely to be representative.

The second threat to the external validity of the case study is that we only randomly sample 30 arguments and their corresponding context. Results may vary between different samples. Consequently the results of the case study may not hold on other samples. To reduce such threat, we constraint the 30 cases based on 3 conditions, which makes sure that all identifiers involved in the cases are properly named.

The third threat to the external validity of the case study is that all of the analyzed applications are implemented in C languages. However, the conclusion drawn on C applications may not hold for programming languages (e.g., Java) because there are essential differences between C and Java language: C is a kind of procedure-oriented programming language, whereas Java is an object-oriented programming language. It would be interesting to validate the result on more programming languages in future research.

The fourth threat to the external validity of the case study is that all respondents involved in the case study are not developers from the industries but less experienced developers from college school. Because the respondents are not the original developers of the source code, they may miss the abstraction of the quantitative selection rules embedded in their selection. Meanwhile, the personal abilities of such respondents may affect the results of the case study. To reduce such threat, all respondents chosen are software developers with at least one year of project experiences. They are familiar with the software development process.

The threat to the construct validity of the case study is that the arguments extracted from the applications may be incorrect. For example, the open-source code may include bugs, i.e., the developers provide wrong arguments for the method invocations. To reduce this threat, we select the most popular open source applications, where the bug is more likely to be fixed if it ever occurred.

The threat to the internal validity of the case study is that there is no direct rules for mapping the semantic of arguments to their corresponding contexts. To reduce such threat, we manually inspect each selected argument and confirm that there are potential semantic relationships.

3 Problem statement

Definition 1 (Non-API method invocation). A non-API method invocation is an invocation of method that is defined within the same project as the invocation appears.

The proposed approach is to recommend arguments for non-API method invocations. The input of the proposed approach is the incomplete program that is available at the point of the recommendation, including the signature of the invoked method, and the context of the invocation. The output of the proposed approach is the recommended argument for the given invocation.

IDEs can decide instantly when the proposed approach should be employed. Once developers type in a method invocation, IDEs decide whether the invoked method is an API method or a non-API method. If

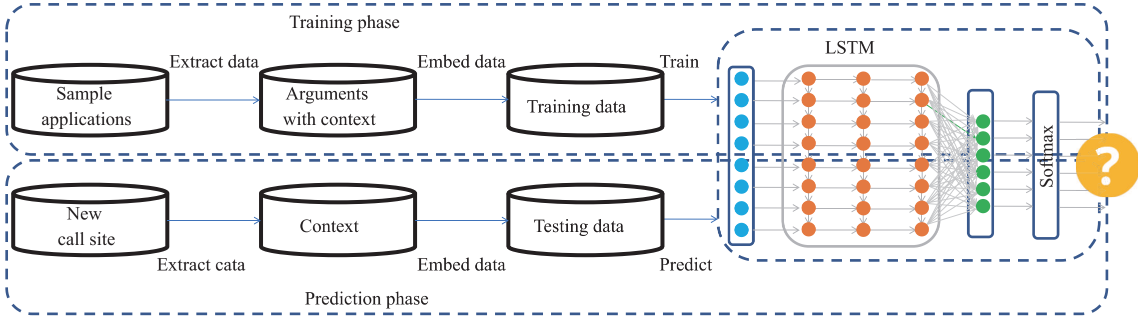

Figure 2 (Color online) Overview of the approach.

the method is defined within the working project, it is a non-API method, and thus IDEs should employ the proposed approach to recommend arguments. Otherwise, traditional API specific approaches should be employed to recommend arguments.

4 Approach

4.1 Overview

The key idea of the approach is that correct argument may be recommended based on the invoked method and its context (especially a list of syntactically correct candidate arguments), although we do not exactly know the quantified relationship between arguments and their context. Consequently, we train an LSTM (long short-term memory) neural network model with training data, i.e., actual arguments and their context from sample applications (corpus), and then predict correct arguments for new call sites with the resulting neural network. An overview of the proposed approach is presented in Figure 2.

To train the neural network, we convert the actual arguments and their context into Paragraph Vector [8]. Such vectors are then used as the training data for the LSTM. During prediction phrase, the input of the neural network is the vector representing the context of the call site whereas the output of the network is the recommended argument. Details of the proposed approach are presented in Sections 4.2–4.6.

4.2 Context data

We represent the context of each call site with names of the formal parameter and the invoked method, as well as candidate arguments as follows:

$$\text{context} = \langle \text{parName}, \text{mName}, \text{cd}_1, \ldots, \text{cd}_5, \text{idt} \rangle, \tag{1}$$

where parName and mName are the names of the formal parameter and the invoked method, respectively. $\langle \text{cd}_1, \ldots, \text{cd}_5 \rangle$ is a list of candidate arguments. idt indicates whether there is a candidate argument whose name is identical to that of the corresponding formal parameter. It is taken as an element of the context because the empirical study [3] suggests that it is quite often (in around 30% of the cases) that the chosen argument and its corresponding formal parameter share the same name. A candidate argument is an expression, e.g., local variable, global variable, or method invocation, that could be used there as an argument without introducing any syntactic errors.

We collect candidate arguments in the same way as Liu et al. [3] did. For a given call site, we consider the following expressions as candidate arguments if passing them as the actual argument will not introduce syntactical errors. (1) Formal parameters of the enclosing method where the call site appears. (2) Local variables of the enclosing method where the call site appears. (3) Global variables and method declarations. (4) For a variable (or formal parameter) v, we consider not only v itself but also its members, e.g., $v \rightarrow \text{id}$. (5) For a pointer p, we consider not only p itself but also the value it points to, i.e., $*p$.

```
char adj;
int num=0;
int a[100]={10000};
void siftDown(int pos)
{
    ... ...
}
```

```
void sort()
{
    int v;
    int k=num/2;
    int j;
    for(j=k;j>=1;j--)
        siftDown(j);
}
```

Figure 3 (Color online) Example.

It should be noted that literals (e.g., numbers and text) and complex expressions (e.g., $2 \times (\text{length} + \text{heigh})$) are not included. The reason for the exclusion is that considering all candidate literals or complex expressions may dramatically increase the number of candidate arguments if it is not infinite. As a result, it becomes challenging to select the correct one from such a large number of candidates. For the same reason, type conversion (e.g., (Person)obj) is not considered, either. For convenience, we call such arguments unsupported arguments. By analyzing 972660 non-API arguments from 85 open-source applications, we find that unsupported arguments account for 14.86% of the total.

The number of candidate arguments may vary dramatically from call site to call site. The variation of the input size causes difficulty in the design of neural networks for machine learning. To this end, we empirically limit the size of candidates to a fixed number (5), i.e., we consider only the top 5 candidate arguments that are most similar with the corresponding formal parameter. We measure the lexical similarity between the actual argument name and the formal parameter name via Euclidean distance [9] because identifiers are represented with vectors. The selection of top 5 similar candidate arguments is based on the following two findings. First, as reported recently, correct arguments are often more lexically similar with their corresponding parameters than other alternative candidates [3]. Second, only a small number of correct arguments would be excluded because of the length limitation. For example, by analyzing 972660 non-API arguments from 85 open-source applications, we find that only 8597 (0.9%) of the correct arguments are excluded because of the length limitation.

Take the method invocation to siftDown(j) in Figure 3 as example, the method name is siftDown, the formal parameter is pos, the syntactically correct candidate arguments are num, a, v, k, and j, and there is no candidate argument identical to the formal parameter. Consequently, the context we extract from the method invocation is represented as

$$\text{context} = \langle \text{pos}, \text{siftDown}, \text{num}, a, v, k, j, 0 \rangle.$$

4.3 Representation of identifiers

To feed the context into the LSTM, we convert each identifier into a fixed-length vector by Paragraph Vector [8], a well-known neural network based encoder. We first decompose identifiers into a sequence of terms (a paragraph) according to underscores and capital letters, assuming that the names follow the popular camel case or snake case naming convention. As a result, an identifier is composed of a sequence of terms: $\text{Name} = \langle t_1, t_2, \ldots, t_k \rangle$, where t_i is the ith term composing the identifier. Each identifier that is represented as a sequence of terms could be taken as a paragraph because paragraphs are essentially sequences of terms. We feed such identifier (represented as a sequence of terms) into the Paragraph Vector, and empirically set the length of vector to 30. Each name n is represented as x_n, which is a learned 30-dimensional vector.

For efficiency, we train the vector model by exploiting the most frequently used 10000 identifiers from training data set as the vocabulary. The vocabulary covers 90% of identifiers involved in the data set. Identifiers beyond the vocabulary are replaced with placeholder "unknown". Such coverage has been proved effective in several applications [10–12].

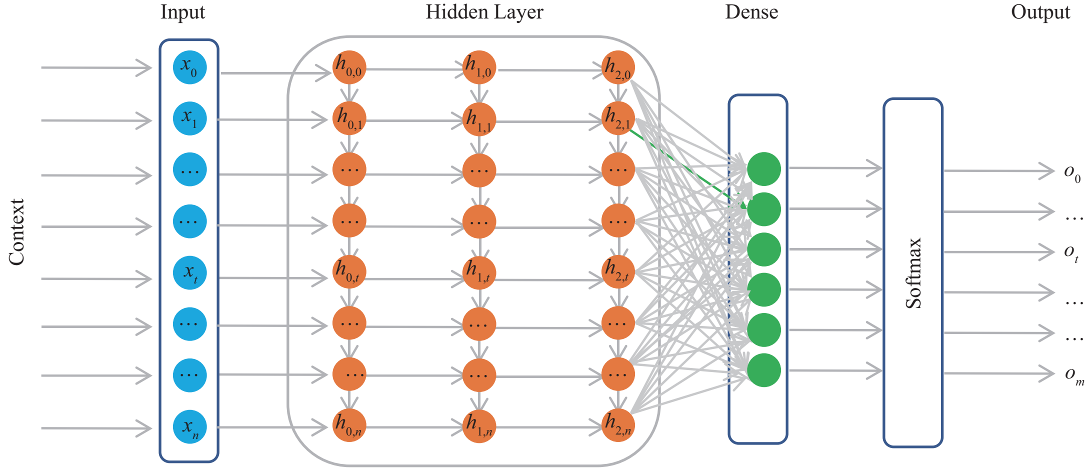

Figure 4 (Color online) Architecture of our network.

4.4 Network architecture

The architecture of the neural network for argument recommendation is presented in Figure 4. It is composed of an input layer, an output layer and four hidden layers (including three LSTM-layers [13,14] and a dense layer). The dense layer with Softmax activation guarantees that only one candidate argument could be recommended.

4.4.1 *Input and output*

For a given argument, the input of the network is a sequence of 30-dimensional vectors that are generated by Paragraph Vector according to the context introduced in Subsection 4.3:

$$\text{Input} = \langle x_1, x_2, \ldots, x_n \rangle,$$

where $n = 8$ is the length of context for each call site.

The output of the network is

$$\text{Output} = \langle o_1, o_2, \ldots, o_m \rangle,$$

where $m = 6$, output is a sequence of one-hot vectors and only one of them can be 1. If the ith candidate argument is the correct one, $o_i = 1$ and $o_k = 0$ where $k \neq i$. If none of the candidate arguments is correct, $o_m = 1$ and $o_k = 0$ where $k < m$. The last dimension (o_m) is indispensable because there are a few arguments (e.g., literals) that do not appear in the list of candidate arguments. In such cases, the neural network can never make correct recommendation by selecting from the input (a list of candidate arguments), and thus it should refuse to make any recommendation by setting $o_m = 1$.

4.4.2 *LSTM*

As shown in Figure 4, our approach employs 3-layer LSTM as the hidden layer, a linear dense layer to map the n dimensional output of the LSTM layer into m dimensional output, a Softmax regression function as the activation layer to normalize the output of the dense layer and generate an one-hot vector matrix. We feed each vector x_t in the input sequences into a chunk of neural unit $h_{0,t}$ in the first layer, and the LSTM neural network computes the output $h_{0,t}$ according to the following procedure.

First, it applies a sigmoid function f_t to decide whether information in cell state C_{t-1} should be thrown away by

$$f_t = \sigma(W_f \cdot [h_{0,t-1}, x_t]) + b_f), \tag{2}$$

where x_t is the current input information, $h_{0,t-1}$ is the output of the previous time step.

Second, it computes what information should be added in the cell state by

$$i_t = \sigma(W_i \cdot [h_{0,t-1}, x_t]) + b_i), \tag{3}$$

$$\tilde{C}_t = \tanh(W_c \cdot [h_{0,t-1}, x_t]) + b_C). \tag{4}$$

Third, it updates the cell state C_t by

$$C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t. \tag{5}$$

Finally, it gets the output $h_{0,t}$ by running a sigmoid function

$$o_t = \sigma(W_o \cdot [h_{0,t-1}, x_t]) + b_o) \tag{6}$$

to decide the output parts of the cell state, and running a tanh function

$$h_{0,t} = o_t \times \tanh(C_t) \tag{7}$$

to assure that the output value is between -1 and 1.

The output $h_{0,t}$ will be fed as input of the next time step to compute $h_{0,t+1}$ in the same layer and that of $h_{1,t}$ in the second layer. For the chunk of neural unit $h_{i,t}$ in the second and the third layer, it works the same way as $h_{0,t}$ in the first layer. We finally use the Softmax regression function to normalize the output (i.e., $\langle h_{2,0}, h_{2,1}, \ldots, h_{2,t}, \ldots, h_{2,n}\rangle$) of hidden layer into the final output $\langle o_1, o_2, \ldots, o_m\rangle$, where the index with max value indicates the prediction.

We employ LSTM in the hidden layer for the following reasons. First, neural networks like LSTM and deep learning are good at building the complex mapping between input and output, which is desirable because we know the input can determine the output in most cases but the accurate mapping is unknown. Second, while deciding whether the ith candidates argument should be recommended, code complete tools should take into account the previous input (i.e., the method name, parameter name, and the preceding candidates). LSTM is especially suitable for such kind of learning tasks where current input is somehow related to previous input. Finally, our empirical evaluation on different neural network models (i.e., fully-connected neural network (FNN) and convolutional neural network (CNN)) suggests that LSTM-based approach has the best performance in argument recommendation.

4.5 Training

Training data are generated from sample applications. For each named argument from the applications, we collect the argument itself as well as its context: $\text{data}_i = \langle \text{argName}_i, \text{context}_i\rangle$. To train the neural network, we feed context_i to the neural network as input, and compare the output against the correct argument. If the correct argument argName_i does not equal to any candidates within context_i, we expect the last dimension of the output equals to one ($o_m = 1$) whereas other dimensions are zero. The neural network adjusts its parameters using stochastic gradient descent training algorithm with the training data.

4.6 Prediction

To recommend an actual argument for a given call site, we collect its context first, which includes the invoked method, the corresponding formal parameter, and a list of syntactically correct candidate arguments. The context is then converted into a sequence of fixed-length vectors that are fed into the trained neural network in turn. The output of the neural network indicates the recommendation of the approach. If $o_m = 1$, the approach refuses to make recommendation, suggesting that the correct argument is not on the list of candidates. If $o_i = 1$ and $i < m$, the ith candidate argument is recommended.

It should be noted that the training of the neural network could be done on special machines in advance. The training of deep neural networks, even with advanced deep learning algorithms, is usually time consuming and requires special devices, e.g., powerful GPU. However, in the prediction phase it often takes neural networks only a few milliseconds to generate output for a given input. Consequently, with neural networks trained on special devices in advance, the proposed approach can response instantly, which makes it practical to integrate the proposed approach into IDEs.

5 Evaluation

5.1 Research questions

The evaluation investigates the following research questions.

- **RQ1:** Does the proposed approach outperform the state-of-the-art approaches in recommending arguments for non-API method invocation? If yes, to what extent?
- **RQ2:** How does the machine learning technique influence the performance of the proposed approach?
- **RQ3**: Is the performance of the proposed approach influenced by the types of formal parameters? If yes, to what extent?
- **RQ4:** Does inner-type recommendation outperform inter-type recommendation? When should inner-type and inter-type recommendation be employed, respectively?
- **RQ5:** How do different vectorization approaches influence the performance of the proposed approach? Can we improve the performance of the proposed approach by replacing the current vectorization approach, i.e., paragarph2vector, with alternative vectorization approaches, e.g., one hot vectorization?
- **RQ6:** How do different evaluation models influence the performance of the proposed approach?
- **RQ7:** How many percentages of arguments are not supported by the proposed approach, and to what extent do they influence the performance of the approach?
- **RQ8:** How long does it take to train the neural network and to generate recommendation for a given call site, respectively? Will IDEs become unresponsive if the proposed approach is integrated into such IDEs?

Research question RQ1 concerns the performance (e.g., precision and recall) of the proposed approach. We compare the proposed approach against the similarity-based approach [3]. The major contribution of Liu et al.'s [3] study is the empirical analysis of a large number of arguments and parameters. As an application of their findings, they manually propose an argument selection rule: Select the candidate argument that is most lexically similar to the corresponding parameter. Because their approach does not rely on the invocation history of the invoked method, it is similar and comparable to our approach. Comparing the proposed approach against this one may reveal whether the LSTM can learn better quantitative rules than human experts. We also compare the proposed approach against a recently proposed code completion approach SLP-Core [7] because it is the state-of-the-art approach that could be employed to recommend arguments. SLP-Core builds a cached n-gram language model for source code by assigning higher probability to tokens most recently used. It should be noted that the comparison between the proposed approach and SLP-Core is not to validate that the former is better than the latter. The only purpose of the comparison is to validate whether the proposed approach can outperform generic code completion algorithms in recommending arguments for non-API methods based on the context of invocations. We implement SLP-Core with default configuration.

Research question RQ2 investigates whether replacing the LSTM model in the proposed approach with other deep learning models, e.g., CNN, could result in further improvement in performance of the proposed approach. To this end, we replace the LSTM model with FNN and CNN respectively, and repeat the evaluation. Evaluation results are presented in Subsection 5.6.2.

Investigation into research question RQ3 may help to reveal the quantitative relationship between the performance (e.g., precision and recall) of the proposed approach and the types of parameters. It is likely that the proposed approach works better for some special types of parameters. Consequently, focusing on such types of parameters may improve the performance of the proposed approach.

Research question RQ4 concerns the comparison between inner-type and inter-type recommendation. Inter-type recommendation is to train a single LSTM neural network with all of the training data, and make recommendation with the same resulting neural network disregarding the types of parameters. In contrast, inner-type recommendation employs different neural networks for recommendation according to the types of parameters. For example, the inner-type recommendation trains a neural network with only those sample arguments (and their context) whose corresponding parameters are of type char. The resulting neural network is employed to make recommendation only for such call sites where the type of

the parameters is char. Different type of arguments may have different characters, and thus it is likely that in some cases inner-type recommendation may outperform inter-type recommendation. To this end, research question RQ4 would investigate whether and when the inner-type recommendation outperforms inter-type recommendation.

Research question RQ5 concerns the influence of vector representations. In Subsection 4.3, we convert identifiers into vectors by the well known Paragraph Vector. To answer research question RQ5, we replace the Paragraph Vector with one-hot vectorization[3)], and repeat the evaluation.

Research question RQ6 concerns the influence of different evaluation models. The proposed approach evaluates the performance based on cross-project model, i.e., training data and testing data come from different projects. To investigate the influence of different evaluation models, we evaluate the performance based on within-project model in Subsection 5.6.6, i.e., training data and testing data come from same project.

Research question RQ7 concerns a limitation of the approach: it cannot recommend literals, complex expressions, or type conversions as arguments. Answering research questions RQ7 helps to reveal how severe the limitation is.

Research question RQ8 concerns the efficiency of the approach. Answering this question helps to validate whether the proposed approach can be applied to recommend arguments in practice.

5.2 Prototype implementation

For evaluation, we implement the proposed approach as a prototype, called Lar (LSTM based argument recommendation). The neural network for argument recommendation is implemented based on an open-source generic implementation of neural networks, called Keras[4)]. It is a high-level library that may run on top of either TensorFlow[5)] or Theano [15][6)]. We use Keras because it is fast and simple with rich documents. It is also one of the most popular open-source neural network libraries.

We evaluate the proposed approach on 972660 non-API arguments (and their context) extracted from 85 open-source C applications. We focus on C for following reasons. First, C is popular. According to the well-known TIOBE Index[7)], C is the second most popular programming language. Second, object-oriented programming languages (e.g., Java) may dramatically increase the number of syntactically correct candidate arguments. There may exist a large number of candidate arguments in the form of a.b or a.b.c. The increase in candidates makes argument selection more challenging, and thus reduces the precision and the recall in argument recommendation. To this end, as an initial try we implement the proposed approach to handle C applications only. In future, it would be interesting to extend it to handle applications in other languages.

The conversion from identifiers to vectors is accomplished with an open-source implementation[8)] of Paragraph Vector. We use this implementation because it is one of the most popular third-part implementation of Paragraph Vector whereas the original implementation introduced in [8] is not publicly available.

5.3 Subject applications

We search for open-source C applications with most stars from GitHub that have at least 5 releases and can be imported into Eclipse. We select the top 85 resulting applications[9)] satisfying such constraints. The size of these applications varies from 322 LOC to 2202580 LOC, with an average of 181764 LOC. These applications are composed of 15450016 lines of source code in total, and contain a large number

3) one-hot. 2018. https://en.wikipedia.org/wiki/one-hot.
4) Keras. 2019. https://keras.io/.
5) Google. 2019. http://www.tensorflow.org/.
6) Theano Development Team. 2019. https://github.com/theano/theano/.
7) March 2018, https://www.tiobe.com/tiobe-index/.
8) Paragraph Vector. 2018. https://github.com/klb3713/sentence2vec.
9) March 2, 2019. https://www.github.com.

(972660) of non-API arguments, of which 88617 unique arguments included. The resulting dataset is available online[10].

We select such applications because of the following reasons. First, the source code of such applications is publicly available, which helps other researchers to repeat the evaluation. Second, such applications are popular, and thus it is likely that most of the incorrect arguments, if there is any, have been identified and fixed. It is important because the evaluation is based on the assumption that arguments in the source code are correct.

5.4 Measurements

To measure the performance of the approaches, we define precision and recall as follows:

$$\text{precision} = \frac{\text{number of accepted recommendation}}{\text{number of generated recommendation}}, \tag{8}$$

$$\text{recall} = \frac{\text{number of accepted recommendation}}{\text{number of arguments}}, \tag{9}$$

where the "number of accepted recommendation" is the number of true positives (correct recommendation), the "number of argument" is the total non-API arguments extracted from 85 open-source C applications, and the "number of generated recommendation" is the cases where the proposed approach can make recommendations. The proposed approach makes no recommendation when the arguments are unsupported arguments (e.g., literals, complex expressions, and type conversions), the identifiers are beyond vocabulary (i.e., "unknown"), or the actual argument is not in the list of candidate arguments.

The evaluation takes the assumption that arguments in the subject applications are correct. Consequently, a recommendation is accepted if and only if the recommended argument is identical to the current one appearing in the source code.

5.5 Process

To validate the proposed approach, we extract all of the arguments from the selected subject applications. With such arguments, we carry out cross-project based ten-fold validation. The subject applications are randomly partitioned into ten roughly equal sized groups notated as G_i ($i = 1, \ldots, 10$). For the ith cross-validation, we consider all arguments except for those in G_i as the corpus of training data, and we consider the arguments in G_i as the testing data.

We convert each actual argument name and its context (i.e., the name of the invoked method, the name of the corresponding formal parameter, and the names of alternative arguments available at the call site) into vectors. The conversion is done by Paragraph Vector as introduced in Subsection 4.3.

To compare the proposed approach against the state-of-the-art approaches, we evaluate the baseline approaches (i.e., the similarity-based approach, SLP-Core approach, and one-hot approach) with the same data set.

5.6 Results

5.6.1 *RQ*1*: Comparison against existing approaches*

Results of the ten-fold cross-project validations on the proposed approach and baseline approaches are presented in Table 1. From this table, we observe that the proposed approach significantly outperforms existing approaches in recommending non-API arguments. First, the precision of the proposed approach is significantly greater than that of existing approaches. Its average precision is up to 83.37%. Compared to the similarity-based approach and the SLP-Core approach, the proposed approach improves the precision by 16.7% (= (83.37% − 71.46%)/71.46%) and 135% (= (83.37% − 35.47%)/35.47%), respectively.

10) Dataset. 2019. https://github.com/d12126977/dataset.

Table 1 Comparison against existing approaches

Ten-fold	Proposed approach			Similarity-based approach			SLP-Core approach		
	Precision (%)	Recall (%)	F1	Precision (%)	Recall (%)	F1	Precision (%)	Recall (%)	F1
1#	87.77	52.63	0.66	74.65	47.37	0.58	38.14	38.14	0.38
2#	79.26	65.44	0.72	73.09	58.18	0.65	37.94	37.94	0.38
3#	92.92	55.38	0.69	69.55	50.00	0.58	27.86	27.86	0.28
4#	93.97	68.74	0.79	85.27	64.30	0.73	52.96	52.96	0.53
5#	65.30	56.90	0.61	55.78	52.58	0.54	20.81	20.81	0.21
6#	84.23	70.22	0.77	66.79	59.35	0.63	28.03	28.03	0.28
7#	84.42	60.53	0.71	77.01	56.97	0.65	53.73	53.73	0.54
8#	84.42	57.96	0.69	66.37	51.10	0.58	28.46	28.46	0.28
9#	89.89	78.07	0.84	89.62	83.33	0.86	40.34	40.34	0.40
10#	71.54	58.99	0.65	56.43	42.91	0.49	26.48	26.48	0.26
Avg	**83.37**	**62.49**	**0.71**	**71.46**	**56.61**	**0.63**	**35.47**	**35.47**	**0.35**

Second, the recall of the proposed approach is greater than that of the similarity-based approach and the SLP-Core approach as well. The average recall of the proposed approach is 62.49%, whereas the recall of the similarity-based approach and the SLP-Core is 56.61% and 35.47%.

Third, the F1 of the proposed approach is greater than that of the existing approaches. Its average F1 is 0.71. Compared to the similarity-based approach and SLP-Core approach, the proposed approach improves F1 by 12.7% ($= (0.71 - 0.63)/0.63$), and 102.9% ($= (0.71 - 0.35)/0.35$) respectively. It should be noted that during the computation of recall (Eq. (9)) the denominator is the total number of involved arguments, including unsupported arguments like literals, complex expressions, and type conversions, although the proposed approach would never succeed in recommending such arguments.

We conclude from the results that the proposed approach outperforms the state-of-the-art in recommending non-API arguments. Compared to the manually drawn rule (lexical similarity based rule), LSTM can learn better rules for argument selection.

5.6.2 *RQ2: Influence of learning models*

We empirically compare the LSTM-based approach with FNN and CNN based approaches. Results of ten-fold cross-project validations on them are presented in Table 2.

From this table, we make the following observations.

- First, LSTM-based approach significantly outperforms FNN-based approach. It improves precision significantly from 16.80% to 83.37%, and recall from 11.50% to 62.49%.
- Second, LSTM-based approach significantly outperforms CNN-based approach as well. It improves precision from 65.92% to 83.37%, and recall from 45.26% to 62.49%. As a result of the improvements in both precision and recall, F1 is improved from 0.54 to 0.71.

We conclude based on the preceding analysis that the LSTM-based approach significantly outperforms FNN-based approach and CNN-based approach.

5.6.3 *RQ3: Influence of parameter types*

We investigate the influence of parameter types, and results are presented in Table 3 and Figure 5. The first column of Table 3 presents all of the formal parameter types encountered in the subject applications. Numeral includes all numeral types provided by C, e.g., int, long, short, and float. Typedef in Table 3 refers to all types declared with the keyword “typedef”. The second and third columns (inter-type recommendation) present the precision and recall of the proposed approach on recommending specific arguments, respectively. The last two columns present the performance of inner-type recommendation.

From Table 3 and Figure 5 we observe that the proposed approach works best for struct where the greatest precision and recall are achieved. It works well for union as well. From the table, we also observe that the proposed approach is less accurate in recommending void, char, numeral and bool arguments where the lowest precision and recall are achieved. One of the reasons for the poor performance on

Table 2 Influence of learning models

Ten-fold	Proposed approach			FNN-based approach			CNN-based approach		
	Precision (%)	Recall (%)	F1	Precision (%)	Recall (%)	F1	Precision (%)	Recall (%)	F1
1#	87.77	52.63	0.66	15.63	10.72	0.13	67.64	46.42	0.55
2#	79.26	65.44	0.72	19.50	13.11	0.16	59.94	40.30	0.48
3#	92.92	55.38	0.69	15.76	10.75	0.13	67.65	46.22	0.55
4#	93.97	68.74	0.79	10.75	7.61	0.09	73.86	52.26	0.61
5#	65.30	56.90	0.61	13.45	9.61	0.11	59.29	42.35	0.49
6#	84.23	70.22	0.77	20.88	14.90	0.18	60.52	41.19	0.49
7#	84.42	60.53	0.71	15.0	10.73	0.13	64.24	44.48	0.53
8#	84.42	57.96	0.69	16.72	11.39	0.14	67.62	46.09	0.55
9#	89.89	78.07	0.84	19.32	13.29	0.16	69.54	47.85	0.57
10#	71.54	58.99	0.65	19.46	12.86	0.15	68.83	45.48	0.55
Avg	**83.37**	**62.49**	**0.71**	**16.80**	**11.50**	**0.14**	**65.92**	**45.26**	**0.54**

Table 3 Influence of parameter types

Type of parameters	Number of parameters	Inter-type recommendation		Inner-type recommendation	
		Precision (%)	Recall (%)	Precision (%)	Recall (%)
Struct	395262	94.91	81.51	92.39	84.51
Union	1572	84.02	56.46	73.01	67.70
Enum	12953	88.60	23.77	80.54	82.52
Void	14608	72.22	22.99	43.67	17.56
Char	14106	75.70	26.47	51.84	40.25
Typedef	10333	81.01	30.21	82.08	18.76
Numeral	117500	76.83	22.07	68.97	13.66
Const	65914	80.03	21.75	62.13	21.69
Bool	17251	81.71	18.04	44.56	13.59

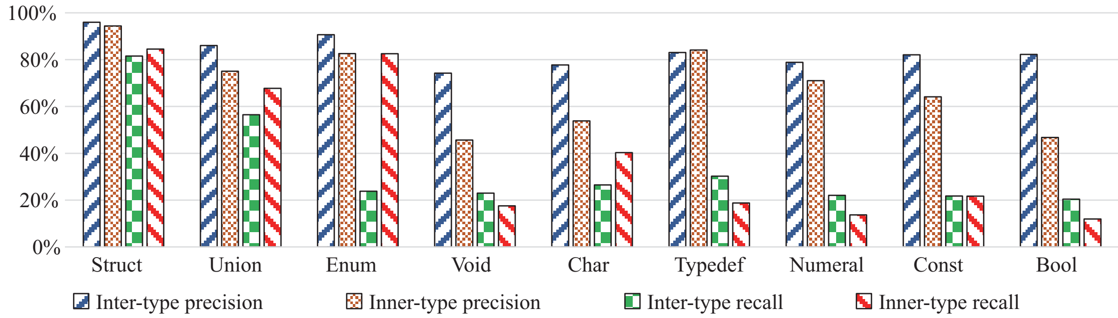

Figure 5 (Color online) Comparison between inner and inter-type recommendation.

recommending such type of arguments is that a large amount of such arguments are literals, complex expressions, or type conversion. The proposed approach is determined to fail when arguments are literals, complex expressions, or type conversion because such arguments are not on the list of candidates from which the proposed approach recommends the most likely one. For example, up to 27.7% of numeral arguments are literals, complex expressions, or type conversion. Consequently, the precision and recall on recommending numeral arguments are as low as 76.83% and 22.07%, respectively. In contrast, only 0.13% of struct arguments are literals, complex expressions, or type conversion, and thus the precision and recall increase dramatically to 94.91% and 81.51%, respectively.

We conclude from the analysis that the performance of the proposed approach varies significantly as the types of parameters change. Consequently, if high precision is required, users may improve the precision by ignoring some categories of arguments, e.g., numeral arguments, that are more challenging to recommend.

5.6.4 *RQ4: Comparison between inner and inter-type recommendation*

To investigate research question RQ4, we compare the inner-type and inter-type recommendation via ten-fold validation.

Evaluation results are presented in Table 3 and Figure 5. From Figure 5, we observe that compared to the inter-type recommendation, the inner-type recommendation often results in reduced precision. It suggests that training the neural network with different types of arguments helps to improve the precision in most cases. The reason why the inter-type model outperforms the inner-type model may be that the former is trained with much larger dataset compared to the latter.

From Table 3, we also observe that for some types of parameters, the inner-type recommendation increases the recall dramatically. For example, for enum parameters, the recall increases dramatically from 23.77% to 82.52% whereas the precision reduces slightly from 88.6% to 80.54%. For such parameters, e.g., enum parameters, inner-type recommendation may be employed if users are more sensitive to recall than precision.

We conclude from the analysis that the inter-type recommendation is often the better choice whereas the inner-type recommendation could be employed to increase recall for some kinds of parameters at the cost of reduced precision.

5.6.5 *RQ5: Influence of vectorization approaches*

To investigate the influence of vectorization approaches, we replace the Paragraph Vector employed in Subsection 4.3 with one-hot vectorization and repeat the evaluation. Evaluation results are presented in Table 4. We also conduct analysis of variance (ANOVA) on the resulting precision, recall and F1, and present the results in Figures 6–8. From Table 4 and Figures 6–8, we make the following observations.

• First, vectorization significantly influences the performance of the proposed approach, especially recall. The analysis on variance on recall presented in Figure 7 ($F > F_{\rm crit}$ and P-value < 0.05 where $\alpha = 0.05$) suggests that the single factor (different vectorization approaches) does significantly influence the resulting recall. The same is true for F1 as presented in Figure 8. However, the analysis on variance on precision presented in Figure 6 ($F < F_{\rm crit}$ and P-value > 0.05 where $\alpha = 0.05$) suggests that different vectorization approaches lead to essentially the same precision. One of the reasons for its minor influence on precision is that the precision varies dramatically even with the same vectorization approach. As presented in Figure 6, the SS (sum of squares of deviation from mean) within groups (0.137) is much bigger than that between groups (0.015).

• Second, Paragraph Vector results in better performance than one-hot. Replacing Paragraph Vector with one-hot reduces the average precision, recall and F1 by 6.6% = (83.37%−77.87%)/83.37%, 17.8% = (62.49%−51.35%)/62.49%, and 14.1% = (0.71−0.61)/0.71, respectively. One of the reason for the reduction in performance is that Paragraph Vector can learn the order of the words and the semantics of the words, whereas one-hot is a bag-of-words approach.

We conclude from the analysis that vectorization approaches influence significantly the performance (especially recall) of the proposed approach, and Paragraph Vector is a better choice for the proposed approach than one-hot.

5.6.6 *RQ6: Influence of evaluation models*

To investigate research question RQ6, we compare the cross-project model with the within-project model. On each project, we randomly partition arguments into ten equally sized folds noted as F_i ($i = 1, \ldots, 10$). For the ith fold validation, we consider all arguments in F_i as the testing data, and those not in F_i as the training data. Evaluation results are presented in Table 5. From this table, we observe that cross-project model significantly outperforms the within-project model: the precision and recall of within-project model decrease from 83.37% to 58.25%, and 62.49% to 47.55%, respectively. The reason why within-project model is not comparable to cross-project model may be that within-project model is trained with limited data from one project, whereas cross-project model can learn from a large amount of data from different projects.

SUMMARY

Groups	Count	Sum	Average	Variance
Paragraph Vector approach	10	8.3372	0.83372	0.008352704
One-hot approach	10	7.7866	0.77866	0.006917574

ANOVA

Source of variation	SS	df	MS	F	P-value	F_{crit}
Between groups	0.015158018	1	0.015158018	1.985296957	0.175874948	4.413873419
Within groups	0.1374325	18	0.007635139			
Total	0.152590518	19				

Figure 6 Anova analysis on precision.

SUMMARY

Groups	Count	Sum	Average	Variance
Paragraph Vector approach	10	6.2486	0.62486	0.006287627
One-hot approach	10	5.1349	0.51349	0.012021572

ANOVA

Source of variation	SS	df	MS	F	P-value	F_{crit}
Between groups	0.062016385	1	0.062016385	6.774341548	0.017994771	4.413873419
Within groups	0.164782793	18	0.0091546			
Total	0.226799178	19				

Figure 7 Anova analysis on recall.

ANOVA analysis on f1 measure

SUMMARY

Groups	Count	Sum	Average	Variance
Paragraph2vector A p	10	7.13	0.713	0.004867778
One-hot Approach	10	6.13	0.613	0.008223333

ANOVA

Source of variation	SS	df	MS	F	P-value	F_{crit}
Between Groups	0.05	1	0.05	7.638771007	0.012789036	4.413873419
Within Groups	0.11782	18	0.006545556			
Total	0.16782	19				

Figure 8 Anova analysis on F1 measure.

Table 4 Influence of vectorization approaches

Ten-fold	Paragraph Vector			One-hot approach		
	Precision (%)	Recall (%)	F1	Precision (%)	Recall (%)	F1
1#	87.77	52.63	0.66	86.11	45.32	0.59
2#	79.26	65.44	0.72	69.46	55.29	0.62
3#	92.92	55.38	0.69	87.78	48.61	0.63
4#	93.97	68.74	0.79	85.69	64.62	0.74
5#	65.30	56.90	0.61	59.95	34.11	0.43
6#	84.23	70.22	0.77	78.00	54.77	0.64
7#	84.42	60.53	0.71	77.54	46.93	0.58
8#	84.42	57.96	0.69	79.30	45.96	0.58
9#	89.89	78.07	0.84	78.08	72.61	0.75
10#	71.54	58.99	0.65	76.75	45.27	0.57
Avg	**83.37**	**62.49**	**0.71**	**77.87**	**51.35**	**0.61**

We employ cross-project validation because it is more similar to real application scenarios of the proposed approach. We expect the proposed approach could be useful even at the beginning of an application, instead of only useful at the end of the project. To this end, the proposed approach cannot rely on the samples (training data) from the same applications. Otherwise, it could not take effect until a large part of the project has been implemented. However, if the proposed approach could be trained

Table 5 Influence of evaluation models

Evaluation model	Precision (%)	Recall (%)
Cross-project evaluation	83.37	62.49
Within-project evaluation	48.25	37.55

Table 6 Influence of unsupported arguments

Evaluation data	Precision (%)	Recall (%)
Including unsupported arguments	83.37	62.49
Excluding unsupported arguments	92.69	72.80

with samples from other applications, it could be employed at the beginning of the new project, making the proposed approach more useful.

5.6.7 *RQ7: Unsupported arguments*

As explained in Subsection 4.2, the proposed approach cannot recommend literals, complex expressions, or type conversions as arguments. Such unsupported arguments account for 14.86% (= 144539/972660) of the arguments involved in the evaluation.

The unsupported arguments have negative impact on the performance (precision and recall) of the proposed approach because it always fails when the correct argument is one of the unsupported arguments. The performance of the approach when unsupported arguments are counted in has been presented on Table 1 and discussed in Subsection 5.6.1. To evaluate the influence of the unsupported arguments, we remove all unsupported arguments from the subject applications, and repeat the evaluation as described in Subsection 5.5.

Evaluation results are presented on Table 6. From this table, we observe that excluding the unsupported arguments increases the recall of the purposed approach significantly from 62.49% to 72.80% and increases the precision of the approach slightly from 83.37% to 92.69%. As a conclusion, the unsupported arguments do have significant impact on the recall of the proposed approach whereas their impact on precision is trivial.

A potential way to reduce the negative impact (especially the impact on recall) of unsupported arguments is to add some common literals to the list of candidate arguments. For example, by taking true and false as candidate arguments, recall of inner-type recommendation on bool arguments increases significantly from 13.59% to 44.50%.

5.6.8 *RQ8: Response time*

Like other deep learning based approaches, the proposed approach may require quite a long time to train the neural network proposed in Subsection 4.4. In our evaluation, it takes approximately 63 min to train the neural network with 972660 items (arguments and their context) from 85 applications on a graphics workstation: 2.6 GHz Intel Xeon E5-2680 processor, 64G RAM, NVIDIA Tesla P4 GPU, Ubuntu 16.04.2 LTS, and Keras. However, it will not make the IDEs where the approach is integrated unresponsive because the training could be done in advance outside the IDEs.

What does matter is the time the approach takes to make recommendation (or decide not to make any recommendation) for a given call site. According to our evaluation, the trained neural network makes decisions quickly, and it takes around 9 ms on average to make recommendation for a given call site on graphics workstation: 3.4 GHz Core i7-6700 processor, 16G RAM, 64-bit Windows 7, and Keras. On personal computers without GPU, the time increases to around 12 ms. Considering that the proposed approach is expected to work while developers are typing, it is likely that the approach would not cause any perceptible delay in user interaction.

We conclude from the analysis that the proposed approach responds quickly without any perceptible delay, and thus it could be integrated into IDEs to make instant on-line recommendation.

5.7 Threats to validity

A threat to external validity is that conclusion drawn on a set of sample applications might not hold on other applications. To reduce the threat, we select the most popular open-source applications from the well-known open-source community GitHub, which yields 85 applications and 972660 non-API arguments from various application domains. Furthermore, we reduce this threat via ten-fold validation as described in Subsection 5.5. While training and testing data change during the ten-fold validation, the conclusion keeps unchanged.

Another threat to external validity is that conclusion drawn on C applications may not hold on applications in other programming languages. In future, it should be interesting to investigate whether the proposed approach works for other languages, e.g., Java.

A threat to construct validity is that the LSTM structure for inner-type recommendation is not optimized, and thus the performance may have been underrated. To facilitate casual analysis, we employ identical neural networks in both inter-type and inner-type recommendation. Replacing it with a simpler network (e.g., with fewer layers) may improve the performance of inner-type recommendation because the size of its training data is much smaller than that of inter-type recommendation. However, the difference in networks may complicate casual analysis.

The second threat to construct validity is that during the evaluation existing arguments in the involved application are assumed to be correct. We take such an assumption because of the following reasons. First, it is challenging, if not impossible, to manually determine whether the arguments are correct. Second, such applications are popular and well-known, and thus it is likely that most of the arguments are correct. However, it is also likely that some of them are incorrect. Consequently, while determining whether recommendations are correct by comparing recommended arguments against current arguments, we may make mistakes. To reduce the threat, we select well-known and popular applications only where incorrect arguments are less likely to appear.

A threat to the validity of the case study in Section 2 is that the respondents are students instead of developers from the industry. Experienced developers from the industry may find better ways to argument selection, and thus have greater chance to make correct selection and figure out quantitative selection rules.

6 Discussion

6.1 Hybrid recommendation

The proposed approach is not to replace existing API targeted approaches [4–6]. Such approaches have been proved to be accurate for popular APIs. Consequently, a practical argument recommendation system may contain a series of API-specific recommendation tools besides our non-API recommendation tool. For each popular API library, we can train an API-specific recommendation tool and employ it whenever methods from this library is called. For a given method invocation, the hybrid recommendation system may employ our non-API recommendation approach if there is no API-specific recommendation tool associated with the given invocation.

6.2 Top-N recommendation and split menus

Most of the argument recommendation tools recommend multiple candidates so that developers can frequently find the wanted arguments from such candidates. Although the proposed approach is trained to recommend (select) the most likely one only from a list of candidates, Lar (an implementation of the approach) does recommend a long list of candidates as other tools do. The recommendation list is composed of two parts. On the first part (on the top of the list) is the most likely candidate argument recommended by the proposed approach. The second part (bottom of the list) is composed of other syntactically correct candidates sorted alphabetically (convenient for location). The design of the split list is inspired by split menus [16]. A split menu is composed of two parts as well: The first part contains

items that are selected frequently, and the second part is composed of items selected rarely. Split menus have been proved to be more convenient than traditional menus [16].

6.3 Limitations

The first limitation of the proposed approach is that it is destined to failure if the correct arguments are literals, complex expressions, or type conversion. Its impact and potential solutions have been discussed in Subsection 5.6.5.

The second limitation of the proposed approach is that the approach is a black box. The limitation comes from the nature of neural network: It is hard to look into the network and figure out exactly what it has learnt, and it is also challenging, if not impossible, to explain the explicit logical relationship between the input and output of the neural networks.

The third limitation of the approach is that it takes the assumption that identifiers follow the popular camel case or snake case naming conventions. To reduce the limitation, in future, we may employ smarter approaches [17] that can decompose identifiers correctly even if they do not follow the naming conventions.

The fourth limitation of the proposed approach is that the approach may be less effective for dynamically typed languages. The approach, as well as the one proposed by Liu et al. [3], retrieves a set of syntactically correct candidate arguments for a given call site based on static source code analysis. Static type-checking makes it possible to determine by static analysis whether a given candidate argument is type compatible with the corresponding formal parameter, and thus we can exclude type incompatible candidates. However, for dynamically typed languages, e.g., Javascript, we cannot infer the types of candidate arguments via static source code analysis, and thus the number of syntactically correct candidate arguments would increase significantly. As a result, it becomes more challenging for the approach to select the correct one from a large number of candidates. In future, it would be interesting to investigate how to reduce the number of candidate arguments for dynamically typed languages.

The fifth limitation of the proposed approach is that we only compare the proposed approach against one-hot vectorization. In future, it would be interesting to investigate the performance of other vectorization techniques, such as word2vector [18], GloVe [19], and FastText [20].

Finally, we have not yet compared the proposed approach against the argument recommendation approaches specific for API method invocations. The reason is that we cannot get the implementation of such approaches [4,6], or that the implementation cannot work on C applications directly [5]. In future, it would be interesting to implement such approaches for C applications and compare the proposed approach against them.

7 Related work

7.1 Recommendation of arguments

The importance of code completion has been well recognized. Murphy [2] investigated how Java developers use Eclipse IDE where code completion is integrated. By analyzing usage data collected from 41 Java developers, they suggested that code completion is one of the top ten commands in Eclipse IDE used by developers.

Although argument recommendation is widely used in IDEs, there are few academic papers concerning argument recommendation and most of them focus on a specific category of arguments: API arguments. The first one is proposed by Zhang et al. [4], called Precise. The key idea behind the approach is that we may recommend API arguments according to the rich invocation history of the invoked API as well as the context of the call site. They retrieve invocation history of different APIs, and generate a parameter usage instance for each of the involved arguments. For a given call site, they generate its usage instance first, and retrieve the most similar k usage instances (k-nearest neighbor algorithm, KNN) that have been generated in advance. Based on the resulting instances, they generate recommendation (a list of candidates) with different strategies that are specific to different types of usage instances. They evaluate

the approach on SWT[11)], and results suggest that for 53% of the time the correct argument is contained in the top 10 of the recommendation list. Asaduzzaman et al. [5] improved Precise by exploiting the localness of argument usage. While building usage instances for arguments, they add the method names, class names, interface names and keywords that appear within m lines prior to the argument. Such names are used to represent the localness of the argument. This is the major difference between their approach and Zhang's approach [4]. Raychev et al. [6] proposed an approach based on statistical language models (called SLANG) to recommend API methods as well as API arguments based on the sequences of API method calls and their associated arguments. Their key idea is that statistical natural language models, e.g., n-gram and neutral network (RNN) language models, could be employed to predict the next API invocation, including the API method and its arguments.

The proposed approach differs from such API targeted approaches [4–6] in that such approaches rely heavily on rich invocation history of the invoked method and thus work for popular API invocations only. APIs are used widely and thus we can retrieve a large number of similar invocation on the same API method. However, for non-API invocations, it is often difficult to collect a large number of similar invocation on the same method. As a result, their approach may not work for such method invocations. Although both SLANG and our approach employ neural network and deep learning, there are some essential differences. SLANG employs neural network as a language model to predict the next term (argument) by searching an extremely large space: the whole vocabulary. In contrast, our approach turns argument recommendation into the selection of correct argument from a small number of syntactically correct candidates, which dramatically reduces the size of search space. Evaluation results in Section 5 suggest that the size of search space could significantly influence the performance of the neural network.

Liu et al. [3] carried out an empirical study on the lexical similarity between arguments and parameters. They found that arguments are often lexically similar to their corresponding parameters. They also found that parameter names frequently associated with low similar arguments in one application are often associated with low similar arguments in other applications as well. As an illustrating application of these findings, they proposed a similarity-based lightweight approach to recommending arguments. They build a list of low similarity parameter names first on sample applications. For a given call site, if the corresponding parameter name is on the list, they refuse to make any recommendation. Otherwise, they retrieve all syntactically correct potential arguments via static source code analysis, and recommend to use the one with the greatest lexical similarity with the corresponding parameter. The proposed approach differs from their approach in that we use LSTM and deep learning to select useful features automatically whereas they select the feature (lexical similarity between actual arguments and formal parameters) manually in advance and rely solely on the lexical similarity. Evaluation results in Subsection 5.6.1 suggest that the proposed approach outperforms theirs and improves the precision by 16.7%.

7.2 Statistical language models and neural networks for code completion

Recently, statistical language models have been successfully applied to infer source code, e.g., sequences of calls and identifiers. Hindle et al. [21] exploited statistical language models to predict next token based on the preceding n tokens with an n-gram model. Their research results suggest that source code could be successfully modeled by statistical language models and such models can be leveraged to support software engineers. Tu et al. [22] improved the n-gram model by adding a cache, and called this new model cache language model. The cache is essentially an n-gram model based on the source file where the n-gram model based prediction is applied. Allamanis et al. [23] exploited statistical language models to predict variable names. Later, they extended this approach to recommend method names and class names with the help of subtoken context models of code [24]. Raychev et al. [25] predicted names of local variables in JavaScript applications, based on statistical language models and conditional random fields [26].

Neural network and deep learning are also applied to facilitate software engineering tasks, e.g., code completion. White et al. [27] introduced deep learning for software language modeling and illustrated how it outperforms n-gram models. Murali et al. [28] generated API-heavy Java code by training a neural

11) The Standard Widget Toolkit. 2018. http://www.eclipse.org/swt/.

generator on program sketches. It can predict the entire body of method given just a few API calls or data types that appear in the method. Wang et al. [29] proposed a novel semantic program embedding that is learned from program execution traces. They exploited recurrent neural networks to model natural fit of program semantic and used the model for program repair.

Le and Mikolov [8] proposed a neural network based approach (Paragraph Vector) to convert paragraphs into fixed-length feature vectors that could be fed directly into neural networks. Compared to bag-of-words (BOW) [30], Paragraph Vector exploits the order of the words and the semantics of the words. The empirical results presented in [8] suggest that Paragraph Vector outperforms other techniques for text representations. Consequently, in this study we convert identifier names into fix-length vectors by Paragraph Vector, and such vectors could be used as the input to neural network proposed in this study. We also calculate lexical similarity between identifier names based on the vectors generated by Paragraph Vector. Evaluation results in Section 5 suggest that the proposed Paragraph Vector vectorization approach outperforms one-hot vectorization approach.

Hellendoorn et al. [7] proposed a nested and cached n-gram based approach SLP-Core to modeling and completing source code. By adding cache mechanism to n-gram models, they assign higher probability to tokens most recently used based on the findings that identifiers in source code have a high degree of localness. Their evaluation results suggest that SLP-Core significantly outperforms existing approaches in modeling source code, including RNN and LSTM based approaches. Argument recommendation is a kind of code completion, consequently SLP-Core may be used to recommend arguments as well. This is one of the reasons why we compare the proposed approach against this approach. The proposed approach differs from their approach in that: (1) we use LSTM and deep learning to recommend arguments, whereas they exploit enhanced n-gram language model to suggest next token; (2) our approach is based the context information of the current invocation, whereas they assign higher probability to recently used tokens. Evaluation results in Subsection 5.6.1 suggest that the proposed approach outperforms theirs and improves the precision by 135%.

8 Conclusion and future work

In this study, we propose the first LSTM-based approach to recommending arguments for non-API methods. For each actual argument from open-source applications, we extract its context, including the invoked method, the corresponding formal parameter, and a list of syntactically correct candidate arguments for the given call site. The context is then converted into a fixed-length vector with a neural network called Paragraph Vector. With arguments and their context from open-source applications, we train an LSTM network to select correct arguments from candidates. In prediction phase, we extract the context of the given call site where an argument should be recommended, and feed it into the trained LSTM network. The output of the network indicates which candidate should be used. Results of the ten-fold validation on 85 open-source applications suggest that the proposed approach outperforms the state-of-the-art approaches.

A potential way to improve the proposed approach in future is to replace the Euclidean distance in Subsection 4.2 with more advanced similarity metrics. The approach converts identifier names into vectors via Paragraph Vector, and computes the similarity between identifiers based on the Euclidean distance between the resulting vectors. Although we have tried some alternative metrics, e.g., cosine similarity, and found that it has little influence on the performance, it is likely that replacing the metrics with more advanced metrics may improve the performance of the proposed approach.

Acknowledgements The work was supported by National Natural Science Foundation of China (Grant Nos. 61772071, 61690205, 61832009) and National Key R&D Program (Grant Nos. 2018YFB1003904).

References

1 Robillard M, Walker R, Zimmermann T. Recommendation systems for software engineering. IEEE Softw, 2010, 27: 80–86

2 Murphy G C, Kersten M, Findlater L. How are Java software developers using the Eclipse IDE? IEEE Softw, 2006, 23: 76–83
3 Liu H, Liu Q, Staicu C A, et al. Nomen est omen: exploring and exploiting similarities between argument and parameter names. In: Proceedings of the 38th International Conference on Software Engineering. New York: ACM, 2016. 1063–1073
4 Zhang C, Yang J, Zhang Y, et al. Automatic parameter recommendation for practical API usage. In: Proceedings of the 2012 International Conference on Software Engineering. Piscataway: IEEE Press, 2012. 826–836
5 Asaduzzaman M, Roy C K, Monir S, et al. Exploring API method parameter recommendations. In: Proceedings of 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015. 271–280
6 Raychev V, Vechev M, Yahav E. Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2014. 419–428
7 Hellendoorn V J, Devanbu P. Are deep neural networks the best choice for modeling source code? In: Proceedings of Joint Meeting on Foundations of Software Engineering, 2017. 763–773
8 Le Q, Mikolov T. Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning, Beijing, 2014. 1188–1196
9 Kuoa R J, Chen Z Y, Tien F C. Integration of particle swarm optimization and genetic algorithm for dynamic clustering. Inf Sci, 2012, 195: 124–140
10 Pradel M, Sen K. Deepbugs: a learning approach to name-based bug detection. In: Proceedings of the ACM on Programming Languages, 2018. 1–25
11 Liu H, Xu Z, Zou Y. Deep learning based feature envy detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. New York: ACM, 2018. 385–396
12 Liu H, Jin J, Xu Z, et al. Deep learning based code smell detection. IEEE Trans Softw Eng, 2019. doi: 10.1109/TSE.2019.2936376
13 Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput, 1997, 9: 1735–1780
14 Wu D, Chi M. Long short-term memory with quadratic connections in recursive neural networks for representing compositional semantics. IEEE Access, 2017, 5: 16077–16083
15 Theano Development Team. Theano: a Python framework for fast computation of mathematical expressions. 2016. ArXiv: 1605.02688
16 Sears A, Shneiderman B. Split menus: effectively using selection frequency to organize menus. ACM Trans Comput-Human Interaction, 1994, 1: 27–51
17 Butler S, Wermelinger M, Yu Y, et al. Improving the tokenisation of identifier names. In: Proceedings of European Conference on Object-Oriented Programming. Berlin: Springer, 2011. 130–154
18 Mikolov T, Chen K, Corrado G, et al. Efficient estimation of word representations in vector space. 2013. ArXiv: 1301.3781
19 Pennington J, Socher R, Manning C. Glove: global vectors for word representation. In: Proceedings of Conference on Empirical Methods in Natural Language Processing, 2014. 1532–1543
20 Joulin A, Grave E, Bojanowski P, et al. Fasttext: compressing text classification models. 2016. ArXiv: 1612.03651
21 Hindle A, Barr E T, Gabel M, et al. On the naturalness of software. Commun ACM, 2016, 59: 122–131
22 Tu Z, Su Z, Devanbu P. On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014. 269–280
23 Allamanis M, Barr E T, Bird C, et al. Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014. 281–293
24 Allamanis M, Barr E T, Bird C, et al. Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2015. 38–49
25 Raychev V, Vechev M, Krause A. Predicting program properties from "big code". In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2015. 111–124
26 Lafferty J, McCallum A, Pereira F C. Conditional random fields: probabilistic models for segmenting and labeling sequence data. In: Proceedings of the 18th International Conference on Machine Learning. New York: ACM, 2001. 282–289
27 White M, Vendome C, Linares-Vásquez M, et al. Toward deep learning software repositories. In: Proceedings of the 12th Working Conference on Mining Software Repositories. Piscataway: IEEE Press, 2015. 334–345
28 Murali V, Qi L, Chaudhuri S, et al. Neural sketch learning for conditional program generation. 2017. ArXiv: 1703.05698
29 Wang K, Singh R, Su Z. Dynamic neural program embedding for program repair. 2017. ArXiv: 1711.07163
30 Harris Z S. Distributional structure. Word, 1954, 10: 146–162