Improving source code suggestion with code embedding and enhanced convolutional long short-term memory

Yasir Hussain | Zhiqiu Huang | Yu Zhou

1College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing, China
2State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Correspondence
Zhiqiu Huang, College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing 211106, China.
Email: zqhuang@nuaa.edu.cn

Abstract
Source code suggestion is the utmost helpful feature in the integrated development environments that helps to quicken software development by suggesting the next possible source code tokens. The source code contains useful semantic information but is ignored or not utilised to its full potential by existing approaches. To improve the performance of source code suggestion, the authors propose a deep semantic net (DeepSN) that makes use of semantic information of the source code. First, DeepSN uses an enhanced hierarchical convolutional neural network combined with code-embedding to automatically extract the top-notch features of the source code and to learn useful semantic information. Next, the source code’s long and short-term context dependencies are captured by using long short-term memory. We extensively evaluated the proposed approach with three baselines on ten real-world projects and the results are suggesting that the proposed approach surpasses state-of-the-art approaches. On average, DeepSN achieves 7.6% higher accuracy than the best baseline.

1 | INTRODUCTION

Source code suggestion is one of the vital tools of modern integrated development environments (IDE). Source code suggestion helps to accelerate software development by suggesting the next likely source code token. However, traditional source code suggestion tools rely on the limited information already present in the IDE which could not capture human’s programming patterns well. Recently, source code modelling research has gained much attention by using statistical language models and learning from large-scale codebases (e.g., Github). N-gram is one of the most widely used source code language model [1, 4]. More recently, deep learning-based approaches [3, 5–7] have revealed excessive potential in source code modelling.

Hindle et al. [2] have revealed the naturalness of source code by employing an n-gram statistical language model. Further, they have demonstrated the effectiveness of their approach with the source code suggestion task. The n-gram-based models work well with smaller context size and do not consider semantic information of the source code. Recently, White et al. [7] have revealed the recurrent neural network (RNN) can significantly improve the performance of source code suggestion models as compared to n-gram models. Although the RNN-based neural language models [3, 7] showed their effectiveness in the modelling of source code, they suffer from vanishing gradient problem. Further, recurrent neural models do not consider spatiotemporal information which can be helpful in the source code modelling. Moreover, the source code contains useful semantic information that is ignored or not utilised to its full potential by these approaches.

We propose a deep semantic net (DeepSN) that makes use of semantic information of the source code to defy the above apprehensions and to improve the performance of source code suggestions. First, we use an enhanced hierarchical convolutional neural network (CNN) to automatically extract the top-notch features of the source code combined with code-embedding to deep learn the semantics of the source code. CNN can extract local features by using multiple convolving filters and achieves sound performance on the tasks related to semantic parsing [8]. Thus, DeepSN adopts CNN to extract top-notch features of the source code and learns their deep semantic relationships by using the code-embedding technique. Moreover, DeepSN learns the source code’s long and short-term context dependencies by using long short-term-memory (LSTM). We extensively evaluated the proposed approach on ten projects (dataset) and the outcome suggests that the
anticipated approach significantly surpasses state-of-the-art methods.

- We propose DeepSN, an enhanced hierarchical convolutional neural network combined with code-embedding technique to extract top-notch features of the source code and to learn the deep semantics of source code. Additionally, it learns the source code's long and short-term context.
- The proposed approach is extensively evaluated with a real-world dataset and the outcome suggests that the anticipated approach outmatches the state-of-the-art approaches. Further, we make the pre-trained code-embeddings of all ten publicly available projects that could be used for different tasks without requiring to train them.

The work of the authors is organised as follows. Section 2 discusses a brief overview of related work. Section 3 discusses the preliminaries. Section 4 discusses the proposed model in detail. In Sections 5 and 6, we discuss the experimental setup and evaluated our approach in comparison with existing approaches. Section 7 discusses the threats to validity. Section 8 is the conclusion, in which we summarise our work.

2 | RELATED WORK

Recently, deep learning methods have shown excessive potential for various source code modelling tasks and showed their usefulness for various problems such as code generation, code completion, API mining, code migration, code categorisation and syntax error correction.

Tu et al. [4] have proposed an n-gram language model consist of a cache. Hellendoorn et al. [9] have presented a cache-based model that showed improvements over the simple n-gram-based approach by introducing nested locality. Similarly, Franks et al. [10] have proposed an eclipse plugin (CACHECA) that combines the default suggestions with suggestions returned by a cache language model [4]. A common limitation of n-gram-based approaches is that they work well with a small size of context and lack to capture semantical similarities. Raychev et al. [11] utilised statistical language techniques to synthesise source code by providing completions for a program. They have utilised the RNN and n-gram language models for the task of synthesising source code. Recently, White et al. [7] have utilised a neural network-based approach to provide completions for the source code. They have shown that the RNN can outperform the cache-based n-gram models. They have shown the usefulness of their method on the task of the source code completion.

Gupta et al. [5] have presented an approach based on deep learning to fix C programming language errors. Similarly, Santos et al. [6] have proposed LSTM-based neural model for the detection and correction of syntax errors presented in programs. Our focus is on code completion, rather than correcting syntax errors presented in a file. Mou et al. [11] have given tree-based structure utilising CNN for the classification of source code programs. Yin et al. [12] have proposed a syntax-driven neural network for the task of source code generation. Maddison et al. [13] have proposed a method for the task of source code generation via structured generation. Similarly, Rabinovich et al. [14] have proposed an abstract syntax network-based approach for the generation of source code. Sethi et al. [15] have proposed a method for the generation of the source code from research papers by employing neural learning. These works are more focused on the source code generation task while the authors work is focused on the prediction of the next token.

Allamanis et al. [16] have proposed probabilistic context-free grammars (PCFG) based model to mine source code idioms. They demonstrated that source code idioms occur across software projects. Chan et al. [17] have proposed a method that uses sub-graph for API usage search by using simple text phrases. Bielik et al. [18] have proposed a probabilistic decision tree model for the purpose of API elements prediction. Thung et al. [19] have proposed a method recommendation system for feature requests stored in bug repositories. Pham et al. [20] have projected a method that utilises byte code for the recommendation of API usage. Nguyen et al. [21] proposed a graph-based statistical language model named GraLan for the task of API code recommendation. Different from Nguyen et al. [21], the concentration of anticipated work is on source code suggestion tasks rather than API code recommendation.

Raychev et al. [22] have proposed a decision tree-based approach for the task of code completion for dynamic type languages. They formulate the problem of learning a probabilistic model over abstract syntax trees of code as learning a decision tree in a domain-specific language. Li et al. [23] utilise different neural networks that leverage token level information and structural information on dynamic type languages. Similarly, Bhoochand et al. [24] have proposed a sparse pointer network-based approach for code completion tasks for Python programming language (dynamic type languages). They utilise a filtered view of a memory of a previous identifier that helps capture long-range dependencies.

The work of authors is closely related to these works [2, 3, 7]. Hindle et al. [2] showed that even n-gram-based statistical language techniques can help improve the source code suggestion task implying source code is natural. The n-gram based models work well with smaller context size and do not consider semantic information of the source code. Recently, White et al. [7] showed that the RNN can significantly improve the performance of source code models. Although the RNN-based neural learning based approaches [3, 7] have shown their effectiveness in the modelling of source code, they suffer from vanishing gradient and do not consider spatiotemporal information which can be helpful in source code modelling. Furthermore, beneficial semantic information is present in the source code which is ignored or not utilised to its full potential by existing source code suggestion approaches.

3 | PRELIMINARY

3.1 | Convolutional neural network

Initially CNN was developed for the task of image recognition. The internal representation of the image is learned by the
model for the purpose of classification [25]. The convolution layers are utilised to correlate between spatially adjacent points by transmutation of the local patterns with connected neurons. The same approach can be adopted for the text classification [8]. Normally, the CNN consists of two different kinds of layers. First, the layer is convolutional which utilises the filters to transforms the input to a feature mapping in which features correspond to filters. Second, the layer is the pooling layer. The purpose of this layer is to help lessen the spatial dimension of the feature mapping.

3.2 | Long short-term memory

LSTM network [26] is an advanced version of RNN that helps overcome the vanishing and exploding gradient problem which normally resides in vanilla RNN. The LSTM consists of a memory cell to accumulate the state information to learn long term dependencies. The LSTM is formalised as,

\[ i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \]  
\[ f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \]  
\[ c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \]  
\[ o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \]  
\[ h_t = o_t \cdot \tanh(c_t) \]

where \( i_t \) is the input gate, \( f_t \) is the forget gate, \( c_t \) is cell state, \( o_t \) is the output gate and \( h_t \) is the hidden state at time \( t \). Similarly, \( f_{t-1} \) is the forget gate, \( i_{t-1} \) is the input gate, \( c_{t-1} \) is cell state, \( h_{t-1} \) is the hidden state at time \( t-1 \) and \( o_{t-1} \) is the output gate. \( \sigma \) is sigmoid and \( \tanh \) is hyperbolic tangent function. \( W \) is the weight and \( b \) is bias.

3.3 | BATCH NORMALISATION

Batch normalisation [27] can mitigate the problem of internal covariate shift and reduces its sensitivity to initialisation. Batch normalisation is used to regulate the input layer by adjusting and scaling the activation which results in improving the performance, and stability of neural networks. It can be expressed as

\[ x_i = \frac{x_i - \mu}{\sqrt{\phi^2 + \epsilon}} \]  
\[ BN(x_i) = rx_i + \beta \]  

where \( x_i \) is the batch with \( j \) values. \( \mu \) and \( \phi \) are the mean and variance of the batch. \( \epsilon \) is a constant, \( r \) and \( \beta \) are learned during model training.

4 | PROPOSED APPROACH

This section describes the proposed DeepSN that suggests the next source code token. The purpose here is to improve the performance of the source code suggestion task by learning the source code context and semantics for real-world historical code-bases. Figure 1 illustrates the general framework of the projected approach. First, we discuss the preprocessing of the source code, which involves, regularisation, tokenisation, and vectorization.

DeepSN uses the vectorized source code files as input to the enhanced hierarchical CNNs combined with code-embedding to automatically learn source code context (\( \tau \)) and semantic relationships. Next, the extracted features are used as input to the LSTM to learn their long and short-term context dependencies. Given a source code program, the trained model is used to automatically suggest the next source code token.

**Figure 1** Overview of the proposed approach
The suggestions for the succeeding source code token can be defined as,

\[ Y = f(\tau) \]  

(9)

Here, \( Y \) describes the suggestions for succeeding source code token, \( f \) describes the function utilized for classification to provide predicts for the succeeding token and \( \tau \) describes the input context. We utilise real software applications (projects) source code obtained from Github\(^2\) to train and test the models for the next source code suggestion task. The collected data set (\( D \)) can be formalised as,

\[
D = < P_1, P_2, \ldots, P_k >
\]  

(10)

\[
P_i = < F_{i1}, F_{i2}, \ldots, F_{ij} >
\]  

(11)

where \( i = 1 \) to \( k \) and \( j \geq 1 \) and \( i1 < = i \). Here, \( < P_1, P_2, \ldots, P_k > \) represents the \( k \) number of projects in \( D \) and \( < F_{i1}, F_{i2}, \ldots, F_{ij} > \) represents the \( j \) number of files in \( i\)-th project.

### 4.1 Preprocessing

Preprocessing is an important process in machine learning/deep learning-based language modelling. Following common practices, we carry out regularisation, feature extraction and vocabulary building. Each of the preprocessing steps is discussed in the following subsections.

#### 4.1.1 Regularisation

Commonly, data set comprises unnecessary features that are not significant for a specific task and these type of features will deeply affect the model’s performance. To avoid this issue, we take out all block-level comments, inline comments and blank lines from source code files. Moreover, we change constant values with their general types (e.g. 7 = IntVal, 9.2 = FloatVal, ‘Hello World’ = StringVal).

#### 4.1.2 Tokenisation

Next, we perform tokenisation in which the terms/words are extracted from the data set (\( D \)). To achieve this goal, we split each source code program (file) into a sequence of space-separated terms. Then each sequence is subdivided into multiple sequences with a fixed size context (\( \tau \)) 20 by employing a sliding window approach.

#### 4.1.3 Vectorization

It is necessary to construct a vocabulary system to train a neural language model. To achieve this, we substitute singleton terms with a unique token UNK which is a common process of building a global vocabulary system for statistical language models. Next, we construct the vocabulary (\( V \)) in which all unique tokens (terms) are abstracted from the dataset. Finally, all source code tokens are substituted with positive integer value correspondent to the vocabulary index for the preparation of sequences into a state that is acceptable to train a neural language model. The abstracted context vectors found in a software system (project) can be expressed as

\[
P_{i}^\tau = < \tau_1, \tau_2, \ldots, \tau_k >
\]  

(12)

where \( \tau_1, \tau_2, \ldots, \tau_k \) describes the number (\( k \)) of context vectors in \( i\)-th project.

### 4.2 Deep semantic net (DeepSN)

The vectorized source code files are then used to automatically extract the hidden features (deep features) by learning source code’s semantics along with their long and short-term context dependencies. The model first transforms the source code tokens into a high dimensional vector space (code-embedding) in which similar code tokens are put closer. In code-embedding, source code tokens are embodied by high dimensional vectors that contain their precise semantic relationships. An important ingredient of our approach is simultaneous learning of code-embedding while training DeepSN that helps learn source code’s semantics by utilising the deep neural net. Figure 2 illustrates an example of transforming source code tokens into code-embedding. We use the source code tokens instead of integer encoded in the presented illustration (Figure 2) for ease of understanding. If there are \( n \)-tokens then code-embedding is \( n \times K \) where \( K \) is the size of the vector.

Next, this high dimensional code-embedding is feed into enhanced CNN to automatically learn the deep features of the source code. The model adopts multiple filters with varying sizes to learn multiple representations of the features simultaneously. Usually, these learned features are pooled by max-over-time pooling [8]. Apart from traditional max-over-time pooling, we employ the batch normalisation on the learned features. We use hierarchical CNN to learn the deep features (deep semantics) of the source code by enforcing batch

![FIGURE 2](https://www.github.com/)

An example of transforming source code tokens into code vectors.
normalisation on the learned features. The features learned from source code are considered the number of filters ($f_n$) and the filter size ($f_s$). The choice of hierarchies ($h_n$) in CNN, the number of $f_n$, and the size of $f_s$ will be discussed in Section 6.2. Next, the source code’s long and short-term context dependencies are learned by employing LSTM. Figure 3 illustrates the net structure of DeepSN. Moreover, we consider the model to provide a low cross-entropy and having a high probability for the accurate suggestions. It can be expressed as

$$H(T, p) \approx -\sum_{i=1}^{N} \frac{1}{N} \log p(x_i)$$  \hspace{1cm} (13)$$

where $T$ is the training set and $N$ is the size of the testing set, $p(x_i)$ is the probability of event $x_i$ estimated from the training set.

5 | EVALUATION

We conduct several experiments to evaluate the effectiveness of DeepSN. This section discusses the evaluation setup in detail. We discuss the research questions in Section 5.1. We describe the data set used to train and evaluate the models in Section 5.2. Section 5.3 discusses model training and testing. Section 5.4 discusses the evaluation metrics in detail. The results are discussed in Section 6.

5.1 | Research questions

The goal of the study is to improve the performance of the source code suggestion task. We decompose the evaluation process of DeepSN into three major parts and aim to answer the following research questions (RQs):

- **RQ1:** How effective is DeepSN as compared to state-of-the-art approaches?
  
  We compare DeepSN against three state-of-the-art approaches to validate its effectiveness. We choose these baselines [2, 7] because they are close to the target task of source code suggestion and considered as state-of-the-art. We train n-gram [2], RNN [7] and CNN [8] -based model as baseline.

- **RQ2:** What is the impact of hierarchical feature extraction on DeepSN?
  
  Before building DeepSN, we analyse the impact of hierarchical feature extraction by varying the number of convolutional layers $h_n$, the number of filters $f_n$, and the filter size $f_s$.

- **RQ3:** What is the impact of enhanced CNN on DeepSN’s performance?
  
  We conduct several experiments to evaluate the effectiveness of enhanced CNN on DeepSN performance against traditional CNN.

5.2 | Dataset

Note that we use the same dataset used in previous studies [2, 28] of this evaluation of this work. We build the dataset by extracting the intersecting projects from these studies with their recent versions available on their corresponding GitHub master branch at the time of collection. This dataset comprises ten real-world java projects extracted from Github3. Table 1 presents each project's version, lines of code in each project (LOC), overall token count, and distinctive code tokens count for each project. The experiments are repeated on each project separately to effectively evaluate the anticipated approach. Each project is randomly divided into ten foldings from which one fold is utilised for validation, one fold is utilised for testing,
and the rest are used for model training purposes. Models are trained using the training set, then parameter tuning is done by using the validation set to achieve the optimal performance and finally the performance evaluation is done by using the test set. It is worth noting that the testing set was not utilised during model training or validation and was exclusively utilised for the purpose of model evaluation. The validation set was used for the model's parameter optimisation to achieve optimal performance. A brief description of these projects is provided here:

- ANTLR is a structured text or binary file manipulation parser generator.
- Ant is a Java-based build tool similar to `make`.
- POI is used to read and write Microsoft Office binary and OOXML file formats.
- Maven is a software project management and comprehension tool.
- JGit is a java-based Git version control system.
- Batik is a Java-based image manipulation tool for scalable vector graphics (SVG) format.
- Cassandra is a highly-scalable partitioned row store.
- db4o is a java variant of object-oriented database.
- JTS is a Java library for generating and manipulating vector geometry.
- iText is a java-based tool to create, edit and enhance PDF documents.

### 5.3 Training and prediction

Table 2 shows the overall architecture of the DeepSN. We use dual-layer CNN $hs = 2$ with constant $fn = 200$ and $fs = 3$, $4$ for first layer and second layer, respectively. We utilise 300 hidden layer neurons with the context size of 20 as extensively studied by White et al. [7]. We utilize Adam optimizer [29] with its default parameter value of 0.001. We use dropout [30] for regularisation and to prevent the model from over-fitting. The model training was done until its converge by utilizing early-stop with three hit(patience) on the validation loss. All the models are trained utilizing tensorflow\(^4\) v1.14 on Intel server Xeon Silver 4110 having CPU 2.10 GHz consisting of 32 cores with 128 GB RAM, equipped with NVIDIA RTX 2080 GPU. The operating system used is Ubuntu 18.04.2 LTS. It is worth noting that the model training is offline thus, doesn't impact the prediction time. The prediction time for the next source code token takes approximately 20 milliseconds on an ordinary computer/laptop.

The suggestions for the succeeding source code token provided a source code file $f$, the trained model expects the context $c$ before the current suggestion position $y$ as input and preprocesses it as discussed in Section 4.1. Next, the preprocessed context vector is used as input to trained DeepSN to provide predictions for most probable succeeding source code tokens $Y$ for the given context $c$.

### 5.4 Metrics

We evaluated the effectiveness of DeepSN by using well-known assessment metrics such as accuracy, precision, recall and f-measure. We measure the Accuracy@k, the percentage of correct next source code token within the top-k ranks. The general formalisation of these metrics are:

$$\text{Accuracy} = \frac{TP + TN}{TP + FN + FP + TN} \quad (14)$$

$$\text{Precision} = \frac{TP}{TP + FP} \quad (15)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (16)$$

$$F – \text{measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (17)$$

\(^4\)Accessed on 1/12/2019: www.tensorflow.org
where $TP$ is the total quantity of examples that are properly detected as positive, $TN$ is the total quantity of examples that are properly detected as negative, $FP$ is the total quantity of examples that are mistakenly detected as positive and $FN$ is the total quantity of examples that are mistakenly detected as negative.

Further, we utilise mean reciprocal rank (MRR) ranking metric for the evaluation purpose in which predictions are ranked higher than those occurring later. The MRR metric is formalised as,

$$MRR(\tau) = \frac{1}{|\tau|} \sum_{i=1}^{|\tau|} \frac{1}{y^i}$$

$MRR(\tau)$ is the average of computed source code contexts ranks in the test set where $\tau$ is the source code context and $y^i$ represents the first relevant prediction index.

6 | RESULTS

6.1 | RQ1: DeepSN performance for source code suggestion

Table 3 presents the Accuracy@k for $k = 1, 5, 10$ and MRR results of all models (N-Gram, RNN, CNN and DeepSN). DeepSN outperforms the other models for all ten projects, in both Accuracy@k and MRR. Compared to the best baseline (RNN), on average DeepSN achieves 7.6%, 5.08%, and 4.37% higher accuracy for Accuracy@1, Accuracy@5 and Accuracy@10, respectively.

Figure 4 presents the MRR results of all four models on all ten projects. From the presented results we can observe that the proposed approach’s min MRR score is higher than the best baseline's (RNN) max MRR score. On average DeepSN (highlighted with a green line) achieves the MRR score of 0.8146 which is higher as compared to N-Gram's 0.4600, RNN's 0.7482, and CNN's 0.7104. DeepSN shows evident improvements, which is significant for source code suggestion because developers prefer to have fewer suggestions ranking the correct ones on top. The results suggest that the proposed approach is capable of suggesting the correct prediction on its top index four out of five times.

Figure 5 presents the Accuracy@k ($k = 1, 2, 3, 5, 10$) results of all models (N-Gram, RNN, CNN and DeepSN) on POI project. DeepSN achieves the accuracy score of 78.02%, 84.44%, 86.82%, 89.15% and 91.36% where $k = 1, 2, 3, 5, 10$, respectively. In comparison, n-gram achieves Accuracy@k 44.16%, 48.90%, 50.38%, 51.65%, 52.83%, RNN achieves 64.81%, 73.17%, 76.43%, 79.53%, 82.80%, and CNN achieves 60.74%, 71.70%, 76.30%, 80.51%, 84.23% where $k = 1, 2, 3, 5, 10$, respectively. Figure 6 illustrates the learned code-embedding of the POI project by DeepSN for qualitative evaluation. The highlighted regions present semantically similar code tokens put closer where region highlighted with green circle present arrays, red circle present conditional operators, blue circle present conditions, and orange circle present loops. From the illustration, we can observe that DeepSN effectively

| Table 3 | Accuracy and MRR comparison of all models |
|----------------|------------------|------------------|------------------|------------------|
|             | State-of-the-art | N-Gram | RNN | CNN | DeepSN |
| antlr4      | Accuracy@1       | 39.04%  | 72.90% | 69.23% | 78.62% |
|             | Accuracy@5       | 44.57%  | 86.73% | 87.31% | 90.40% |
|             | Accuracy@10      | 45.20%  | 89.67% | 90.62% | 92.66% |
|             | MRR              | 0.4144  | 0.7851 | 0.7657 | 0.8347 |
| Ant         | Accuracy@1       | 44.88%  | 68.46% | 62.83% | 74.19% |
|             | Accuracy@5       | 51.54%  | 83.92% | 82.54% | 88.00% |
|             | Accuracy@10      | 52.57%  | 87.17% | 86.49% | 90.74% |
|             | MRR              | 0.4785  | 0.7740 | 0.7074 | 0.7983 |
| Bank        | Accuracy@1       | 44.26%  | 69.10% | 62.99% | 76.82% |
|             | Accuracy@5       | 50.25%  | 84.16% | 82.70% | 89.95% |
|             | Accuracy@10      | 51.24%  | 87.42% | 86.56% | 91.37% |
|             | MRR              | 0.4962  | 0.7518 | 0.7087 | 0.8179 |
| Cassandra   | Accuracy@1       | 43.99%  | 65.27% | 56.86% | 74.15% |
|             | Accuracy@5       | 52.79%  | 82.09% | 79.03% | 87.85% |
|             | Accuracy@10      | 54.03%  | 85.53% | 83.18% | 90.51% |
|             | MRR              | 0.4793  | 0.7209 | 0.6576 | 0.7977 |
| db4o        | Accuracy@1       | 45.06%  | 69.13% | 61.52% | 76.05% |
|             | Accuracy@5       | 52.80%  | 84.33% | 83.10% | 88.86% |
|             | Accuracy@10      | 53.77%  | 87.33% | 86.84% | 91.44% |
|             | MRR              | 0.48546 | 0.7530 | 0.7022 | 0.8127 |
| Itext       | Accuracy@1       | 35.43%  | 67.99% | 61.67% | 74.98% |
|             | Accuracy@5       | 41.40%  | 83.53% | 82.41% | 88.41% |
|             | Accuracy@10      | 42.17%  | 86.84% | 86.42% | 91.06% |
|             | MRR              | 0.3808  | 0.7422 | 0.7002 | 0.8046 |
| Jgit        | Accuracy@1       | 43.66%  | 68.94% | 62.49% | 77.17% |
|             | Accuracy@5       | 51.11%  | 84.40% | 83.21% | 89.55% |
|             | Accuracy@10      | 52.14%  | 87.70% | 87.10% | 91.97% |
|             | MRR              | 0.4699  | 0.7520 | 0.7086 | 0.8222 |
| Jts         | Accuracy@1       | 40.39%  | 67.71% | 62.36% | 74.12% |
|             | Accuracy@5       | 46.69%  | 84.03% | 83.85% | 88.29% |
|             | Accuracy@10      | 47.61%  | 87.58% | 87.81% | 91.10% |
|             | MRR              | 0.4317  | 0.7428 | 0.7098 | 0.7990 |
| Maven       | Accuracy@1       | 47.69%  | 72.14% | 68.03% | 78.36% |
|             | Accuracy@5       | 56.25%  | 86.47% | 86.78% | 90.49% |
|             | Accuracy@10      | 57.62%  | 89.36% | 89.87% | 92.87% |
|             | MRR              | 0.5153  | 0.7793 | 0.7568 | 0.8333 |
| Poi         | Accuracy@1       | 44.16%  | 64.81% | 60.74% | 78.02% |
|             | Accuracy@5       | 51.65%  | 79.53% | 80.51% | 89.15% |
|             | Accuracy@10      | 52.83%  | 82.80% | 84.23% | 91.36% |
|             | MRR              | 0.4755  | 0.7079 | 0.6872 | 0.8256 |

(Continues)
learned the source code semantics and arrange the related code tokens close to each other. From the results, it is evident that the proposed approach is capable of capturing the source code semantics by using the code-embedding technique and is able to learn long and short-term contexts, resulting in significant performance boost.

Figures 7 and 8 illustrate the precision, recall and f-measure scores of all four approaches (N-Gram, RNN, CNN and DeepSN). The precision and recall scores are presented as bars and f-measure is illustrated with a line chart. Here again, the proposed approach outperforms all other approaches. The source code suggestion precision of the N-gram model is high with very low recall which means it is random guessing, resulting in poor performance (low f-measure). Compared to RNN and CNN, DeepSN achieves high precision and returning the majority of results in positive (high recall) for all ten projects, attaining good model performance (high f-measure).

6.2 | RQ2: Impact of hierarchical feature extraction

This section empirically evaluated the impact of hierarchical feature extraction on model performance by varying the number of filters \( f_n \), the number of layers in CNN \( b_n \) and the filter size \( f_s \) in each layer. To limit the experimental run time, we randomly divide our data set into two groups, Group A (maven, cassandra, antlr4, db4o, jts) and Group B (ant, poi, igit, batik, jtext) each containing five projects. Group A studies the impact of the number of filters on model performance and Group B studies the impact of CNN layers and the size of the filters in each layer.

In Group A experiments, we varied the number for filters from \( f_n = 50, 100, 200, 300 \). Figure 9 presents the results of Group A experiments. From the illustrated results, we can observe a significant performance boost when the number of filters is increased from 50 to 100. Further, we can observe that increasing the number of filters more than 200 does not necessarily improve the model performance. From the results, we conclude that the optimal performance is achieved when we use the number of filters \( f_n = (100, 200) \).

In Group B experiments, we varied the CNN layers with various filter sizes. In hierarchical CNN, the learned features are convolved over to capture deeper source code semantics. Based on our preceding findings, we use the number of filters \( f = 100 \) as a control variable in each hierarchy. We start from a single-layered hierarchy (CNN-1) with a filter size of \( f_s = 3 \). Next, we experimented with the dual-layered hierarchy (CNN-2) where the first layer is similar to CNN-1 and 2nd layer uses the filter size \( k = 4 \). Similarly, we experimented with the triple-layered hierarchy (CNN-3) where the first and second layers are similar to CNN-1 and CNN-2 respectively, and the third layer uses the filter size \( k = 5 \). Figure 10 illustrates the performance comparison between CNN-1, CNN-2 and CNN-3. From the results, we can observe that the CNN-2 performs better as compared to CNN-3 and a significant performance boost can be observed when compared to CNN-1. Further, we found that using the number of layers greater than two does not necessarily improve the model performance and, it is prone to under-fitting instead. From the results, we conclude that the optimal performance is achieved with the filter size \( f_s = 3 \) and \( f_b = (3, 4) \) for single-layered and dual-layered CNN, respectively.

6.3 | RQ3: Impact of enhanced CNN on DeepSN

This section evaluated the impact of enhanced CNN and traditional CNN on DeepSN's performance. We train two different versions of enhanced CNN and traditional CNN. The first version is trained by using single-layered CNN with the number of filters \( f = 100 \) and filter size \( k = 3 \). The second version is trained by using dual-layered CNN with the number of filters \( f = 200 \) with filter size \( k = (3, 4) \). Table 4 presents the comparative Accuracy@k scores of enhanced CNN and traditional CNN. We find that the performance of enhanced CNN is significantly improved as compared to traditional CNN in both cases. We can also observe in some cases (ant, antlr4, jts) when using traditional CNN the model performance drops catastrophically. Further, we observe when using the dual-layered setup with traditional CNN, the model is prone to under-fitting, whereas enhanced CNN was able to learn deeper source code semantics.
FIGURE 6  Visualisation of code-embedding on POI project by t-SNE [31]. The output is restricted to 300 code tokens for better visualisation.
To validate the statistical significance of different approaches (enhanced CNN and traditional CNN), we conduct the ANOVA one-way statistical test on the dual-layered setup. Table 5 presents the results of the AVOVA test where the target was Accuracy@k for k = 1, 2, 3, 5, 10. From the results, we reject the null hypothesis $P-value < (alpha = 0.05)$ and $f > f - crit$, reflecting that a significant statistically difference present between enhanced CNN and traditional CNN. From Tables 4 and 5, a significant performance boost in DeepSN can be observed when using enhanced CNN. Further, we can
observe that the optimal performance is achieved with the dual-layered setup.

We can observe that the dual-layer setup achieves the optimal results in all cases. On average, the performance of DeepSN is similar regardless of the project sizes. For example, the accuracy@1 of antlr and POI projects are similar to a marginal difference of 0.6% regardless of the huge difference in their sizes. Similarly, when comparing the traditional and enhanced CNN we have observed that the performance of DeepSN with traditional CNN is relatively poor when trained on a smaller amount of information. We can perceive that traditional CNN achieves very low accuracy scores of 10.10%, 8.47%, and 8.31% for ant, antlr4, and its projects in single-layered setup, respectively, while the DeepSN with enhanced CNN achieves optimal performance with a significant improvement in modeling accuracy (ant = 74.14%, antlr4 = 78.31%, jts = 74.14%).

6.4 | Suggesting code with DeepSN

We use the DeepVS [32] plugin interface in Visual Studio Code IDE that interacts with the trained DeepSN model to provide predictions for the task of source code suggestion. Given the example provided in Figure 11 in which a software developer is writing a bubble sort algorithm. The DeepSN takes the context before the cursor position from the plugin interface and preprocesses it by following the steps already discussed in Section 4.1. Next, the preprocessed context vector is used as input to trained DeepSN for possible predictions of the next source code token. Here, we can observe that DeepSN provides top-5 suggestions [<, >, <=, >, ==] based on their contextual and semantical relevance, ranking the ; at its first index which is the most relevant logical operator in this scenario. Moreover, correlating this example with Figure 6 one can observe that DeepSN effectively arranges semantically similar code tokens (conditional operators highlighted in red circle) closer resulting in better modelling performance. Extending the bubble sort example (Figure 12) we can observe that it provides suggestions [i, j, IntLiteral, pos] that are closely related, ranking j on its first index which is the most probable next source code token given the context.

6.5 | DISCUSSION

The experimental results show that the proposed approach performs better than the other baseline approaches. There are several reasons that helped DeepSN to achieve optimal performance.

- First, it simultaneously learns the different representations of the source code features by using multiple filters.
- Second, by using hierarchical feature mapping and by capturing the long and short term-context dependencies of the source code helped to improve model generality and applicability.
- Third, by adopting the batch normalisation on each phase of feature mapping helped the model to further improve its performance.
- Finally, the training and optimisation of DeepSN along with simultaneous learning of code-embedding helped the model to learn deeper source code semantics.

7 | THREATS TO VALIDITY

7.1 | Limitations

Although the proposed approach improves the modelling performance, still several limitations need to be addressed. The evaluation of the proposed approach is done on each project independently; it may produce different results when suggesting cross-project/domain predictions. Moreover, neural language models usually suffer from the out-of-vocabulary issue that reduces the modelling accuracy which implies that there are cases where correct predictions are not even within top-ten suggestions. These limitations could be overcome by training the model with more information. Another solution to this problem is to use local cache
| Project       | DeepSN single layer | Ant | antlr4 | batik | cassandra | db4o | itext | jgit | jts | maven | poi | 
|---------------|---------------------|-----|--------|-------|-----------|------|-------|------|-----|-------|-----| 
| Accuracy@1    | 10.10%              | 8.47% | 72.54% | 70.00% | 71.76%    | 71.16% | 68.89% | 8.31% | 10.52% | 73.17% | 
| Accuracy@2    | 19.96%              | 17.73% | 80.04% | 78.46% | 80.00%    | 79.26% | 77.07% | 17.91% | 20.69% | 80.62% | 
| Accuracy@3    | 27.36%              | 25.98% | 83.12% | 81.73% | 83.19%    | 82.48% | 80.44% | 27.46% | 30.62% | 83.46% | 
| Accuracy@5    | 39.44%              | 36.16% | 85.01% | 84.66% | 86.17%    | 85.45% | 83.75% | 38.41% | 41.49% | 86.16% | 
| Accuracy@10   | 51.79%              | 50.18% | 88.58% | 87.58% | 89.04%    | 88.46% | 87.08% | 51.24% | 52.24% | 88.81% | 
| Enhanced DeepSN single layer |                  |     |        |       |           |      |       |      |     |       |     | 
| ant           | 74.14%              | 78.31% | 76.79% | 73.99% | 74.80%    | 72.07% | 71.17% | 74.14% | 72.17% | 
| antlr4        | 74.14%              | 82.14% | 85.29% | 88.03% | 80.78%    | 79.39% | 78.99% | 83.30% | 81.62% | 
| batik         | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| cassandra     | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| db4o          | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| itext         | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| jgit          | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| jts           | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| maven         | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| poi           | 74.14%              | 82.14% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| Enhanced DeepSN dual layer |              |     |        |       |           |      |       |      |     |       |     | 
| ant           | 74.14%              | 78.62% | 76.79% | 73.99% | 74.80%    | 72.07% | 71.17% | 74.14% | 72.17% | 
| antlr4        | 74.14%              | 82.17% | 85.29% | 88.03% | 80.78%    | 79.39% | 78.99% | 83.30% | 81.62% | 
| batik         | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| cassandra     | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| db4o          | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| itext         | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| jgit          | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| jts           | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| maven         | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 
| poi           | 74.14%              | 82.17% | 88.03% | 80.78% | 83.30%    | 81.62% | 84.14% | 85.30% | 84.11% | 

The bold values represent the best performance compared to others.
along with neural learner. We take into account these limitations and intend to provide vigorous methodology in the future.

### 7.2 Construct validity

Note that a threat to construct validity is an alteration in model settings or utilising a different set of training, validation and testing set which might produce dissimilar results. An additional threat to construct validity is the appropriateness of the metrics that are used for evaluation. The Accuracy@$k$ [2, 7] and MRR [6, 28] metrics are usually utilised to evaluate the neural language-based source code models. Moreover, the anticipated approach is evaluated with precision, recall, f-measure, and ANOVA statistical test to alleviate this threat and to validate the effectiveness.

### 7.3 Internal Validity

Here, a threat to internal validity is the implementation of baselines. The purpose of re-implementation was to build a generalised baseline as the dataset is an intersection of previous
studies with the recent versions available on GitHub. So, we implemented our version by following the same procedure as in their original manuscripts. To avoid this risk we doubly check the implementations and the results; nonetheless, there may some unobserved flaws. The other threat to internal validity is the learning of code-embedding. There are several other techniques (e.g. CBOW [33] and skip-gram [34]) to learn the code-embedding. It would be interesting to test other code-embedding techniques that may enhance DeepSN or employ the presented approach for the cross-domain recommendation [35–37].

7.4 | External validity

The generality of the results is a threat to external validity. The data set utilised in the anticipated work is gathered from GitHub which is a renowned platform for source code repositories. The software projects that are used in the anticipated study does not necessarily represent Java code completely or additional programming languages. Moreover, the selection of hyper-parameters is another threat to external validity. There is no common methodology to acquire the optimal parameters, thus is mainly experimental.

8 | CONCLUSION

The existing approaches for source code suggestion ignore or underutilise the semantic information present in the source code. To improve the performance of source code suggestion, we proposed a DeepSN that makes use of semantic information of the source code. First, DeepSN uses an enhanced hierarchical CNN to automatically extract the top-notch features of the source code combined with code-embedding to learn useful semantic information. Next, the source code's long and short-term context dependencies are captured by using LSTM. We extensively evaluated the proposed approach with three baselines on ten real-world projects and results have suggested that the anticipated approach surpasses state-of-the-art approaches. Further, we make the pre-trained code-embeddings of all ten projects publicly available that can be used for different source code modelling tasks without requiring to train them. The approach presented here is general and can be extended to other source code modelling tasks. In our future work, we consider evaluating our approach for other source code modelling tasks such as syntax error correction, bug prioritizing etc.

ACKNOWLEDGEMENTS

The work of authors was supported by the National Key R&D (Grant no. 2018YFB1003902), Natural Science Foundation of Jiangsu Province (No. BK20201292), National Natural Science Foundation of China (No. 61972197, No. 61802179) and Qing Lan Project.

REFERENCES

1. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp. 213–222 (2009)
2. Abram Hindle et al.: On the naturalness of software. In: 2012 34th International conference on software engineering (ICSE), pp. 837–847. IEEE (2012)
3. Raychev, V., Veech, M., Yahav, E.: Code completion with statistical language models ACM Sigplan Notices, 49, pp. 419–428. ACM (2014)
4. 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, pp. 269–280. ACM (2014)
5. Gupta, R., Kanade, A. Shevade, S.: Deep reinforcement learning for programming language correction (2018) arXiv preprint arXiv:1801.10467
6. Eddie, A. S.: Syntax and sensibility: using language models to detect and correct syntax errors. In: IEEE 25th international conference on
software analysis, evolution and reengineering (SANER), pp. 311–322. IEEE (2018)
7. White, M. et al: Towards deep learning software repositories. In: Proceedings of the 12th working conference on mining software repositories, pp. 334–345. IEEE Press (2015)
8. Yoon, K.: Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1746–1751. Doha, Qatar (2014). https://www.aclweb.org/anthology/D14-1181
9. Vincent, J., Devanbu, P: Are deep neural networks the best choice for modelling source code? In: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, pp. 763–773 (2017)
10. Frankle, C. et al: A cache language model based code suggestion tool. IEEE/ACM 37th IEEE Intl. Conf. Softw. Eng. 2(2015), 705–708 (2015)
11. Mou, L. et al: Convolutional neural networks over tree structures for programming language processing In: Thirtieth AAAI conference on artificial intelligence (2016)
12. Yin, P., Graham N.: A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 440–450. Association for Computational Linguistics, Vancouver (2017). https://www.aclweb.org/anthology/P17-1041
13. Maddison, C., Tarlow, D: Structured generative models of natural source code. In: International conference on machine learning, pp. 649–657 (2014)
14. Rabinovich, M., Stern, M, Klein, D: Abstract syntax networks for code generation and semantic parsing. arXiv preprint arXiv:1704.07535, 2017
15. Sethi, A. et al: Dlipaper2code: auto-generation of code from deep learning research papers. In: Thirty-Second AAAI conference on artificial intelligence (2018)
16. Allamanis, M., Sutton, C.: Mining idioms from source code. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp. 472–483. ACM (2014)
17. Chan, W-K., Cheng, H., Lo, D: Searching connected api subgraph via text phrases. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, pp. 10. ACM (2012)
18. Bielić, P., Raychev, V., Veeche, M.: Phog: probabilistic model for code. In: International conference on machine learning, pp. 2933–2942 (2016)
19. Thung, F. et al: Automatic recommendation of api methods from feature requests. In: Proceedings of the 28th IEEE/ACM international conference on automated software engineering, pp. 290–300. IEEE Press (2013)
20. Hung, V.P, et al: Learning api usages from bytecode: a statistical approach. In: Proceedings of the 38th International Conference on Software Engineering (ICSE ’16), pp. 416–427. Association for Computing Machinery, NY (2016). https://doi.org/10.1145/2884781.2884873
21. Anh, T. N., Nguyen, T. N.: Graph-based statistical language model for code 2015 IEEE/ACM 37th IEEE Intl. Conf. Softw. Eng., 1, pp. 858–868. IEEE (2015)
22. Raychev, V., Bielić, P., Veeche, M: Probabilistic model for code with decision trees. ACM SIGPLAN Not. 51(10), 731–747 (2016)
23. Chang, L., et al: Neural code completion (2016)
24. Bhoopchand, A. et al: Learning python code suggestion with a sparse pointer network (2016) arXiv preprint arXiv:1611.08307
25. Krizhevsky, A., Sutskever, I., Hinton, G. E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105 (2012)
26. Hochreiter, S., Jürgen Schmidhuber: Long short-term memory. Neural Comput. 9(8), 1735–1780 (1997)
27. Sergey Ioffe, Szegedy, C.: Batch normalisation: accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on Machine Learning vol. 37, pp. 448-456 (2015). Available from http://proceedings.mlr.press/v37/ioffe15.html
28. Anh, T. N. et al: A deep neural network language model with contexts for source code. In: IEEE 25th international conference on software analysis, evolution and reengineering (SANER), pp. 323–334. IEEE (2018)
29. Kingma, D.P., Jimmy Ba: Adam: a method for stochastic optimization. In: ICLR 2015 : International Conference on Learning Representations 2015. arXiv preprint arXiv:1412.6980, 2015
30. Amit, G. et al: An exploration of dropout with rns for natural language inference. In: International conference on artificial neural networks, pp. 157–167. Springer (2018)
31. van der Maaten, L., Hinton, G: Visualising data using t-sne. J. Machine Learn. Res. 9(Nov), 2579–2605 (2008)
32. Hussain, Y. et al: Deepvs: an efficient and generic approach for source code modelling usage. Electron Lett. 56 (12), 604–607 (2020)
33. Mikolov, T. et al: Efficient estimation of word representations in vector space. In: ICLR (Workshop Poster). arXiv preprint arXiv:1301.3781, 2013
34. Mikolov, T. et al: Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp. 3111–3119 (2013)
35. Fu, W. et al: Deeply fusing reviews and contents for cold start users in cross-domain recommendation systems Proc. AAAI Conf. Artificial Intel. 33, pp. 94–101 (2019)
36. Song, T. et al: Based cross-domain recommendation through joint tensor factorization. In: International conference on database systems for advanced applications, pp. 525–540. Springer (2017)
37. Wang, X. et al: Cross-domain recommendation for cold-start users via neighbourhood based feature mapping. In: International conference on database systems for advanced applications, pp. 158–165.Springer (2018)

How to cite this article: Hussain Y, Huang Z, Zhou Y. Improving source code suggestion with code embedding and enhanced convolutional long short-term memory. IET Soft. 2021;15:199–213. https://doi.org/10.1049/sfw.2017.12017