The Journal of Systems & Software 196 (2023) 111553

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Boosting source code suggestion with self-supervised Transformer Gated Highway[☆]

Yasir Hussain [a,*], Zhiqiu Huang [a], Yu Zhou [a], Senzhang Wang [b]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing 211106, China
[b] Central South University, China

ARTICLE INFO

Article history:
Received 4 April 2022
Received in revised form 21 September 2022
Accepted 29 October 2022
Available online 5 November 2022

Keywords:
Deep learning
Transformer models
Source code modeling
Source code suggestion

ABSTRACT

Attention-based transformer language models have shown significant performance gains in various natural language tasks. In this work, we explore the impact of transformer language models on the task of source code suggestion. The core intention of this work is to boost the modeling performance for the source code suggestion task and to explore how the training procedures and model architectures impact modeling performance. Additionally, we propose a transformer-based self-supervised learning technique called Transformer Gated Highway that outperforms recurrent and transformer language models of comparable size. The proposed approach combines the Transformer language model with Gated Highway introducing a notion of recurrence. We compare the performance of the proposed approach with transformer-based BERT (*CodeTran*), RoBERTa (*RoBERTaCode*), GPT2 (*TravTrans*), *CodeGen* and recurrent neural language-based LSTM (*CodeLSTM*) models. Moreover, we have experimented with various architectural settings for the transformer models to evaluate their impact on modeling performance. The extensive evaluation of the presented approach exhibits better performance on two programming language datasets; *Java* and *C#*. Additionally, we have adopted the presented approach for the syntax error correction task to predict the correct syntax token to render its possible implications for other source code modeling tasks.

1. Introduction

Source code suggestion that is also known as code completion is the most frequently used feature of Integrated Development Environments (IDEs). This feature offers predictions for the next possible token to the software developers, allowing the developers to speed up the software development process. The effectiveness of automatically providing code suggestions requires the next possible code token to be predicted in the top ranks. Traditional code suggestion tools rely on the code context already written in the IDEs and order the predictions based on the counts or arrange them into alphabetical order which significantly limits their capabilities.

This motivated the use of machine learning for the task of source code suggestion, as machine learning methods not only learn source code context effectively but also serve as a code sighting method. These approaches can be trained on real-world codebases that are capable of providing predictions that have not been observed by the IDE yet. Early approaches adapted n-gram language models on sequences of source code tokens (Hindle et al., 2012; Franks et al., 2015). Later, Recurrent Neural Networks (RNNs) (White et al., 2015) and their variants (Hussain et al., 2020, 2021) have been adopted to improve their accuracy. Although predictive models cannot be expected to be perfect, the accuracy of the current state-of-the-art methods leaves a substantial margin to be improved.

Recently, attention-based transformer model (Vaswani et al., 2017) and their variants (Devlin et al., 2018; Liu et al., 2019a) have revolutionized NLP domain by removing recurrent layers completely. The transformer-based models allow parallelization and achieve a new state-of-the-art in translation and other NLP tasks. The architecture such as BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2018) that are based on transformer architecture, have shown superior performance as compared to RNNs. These models are trained on a large amount of data and with high-performance computational machines having several GPUs.

To this end, we present an empirical study to explore the capabilities of neural language models for the task of source code suggestion. Here, we consider the source code suggestion task as a classification task. Among the various neural language

[☆] Editor: Lingxiao Jiang.
[*] Corresponding author.
E-mail addresses: yaxirhuxxain@nuaa.edu.cn (Y. Hussain), zqhuang@nuaa.edu.cn (Z. Huang), zhouyu@nuaa.edu.cn (Y. Zhou), szwang@nuaa.edu.cn (S. Wang).

https://doi.org/10.1016/j.jss.2022.111553
0164-1212/© 2022 Elsevier Inc. All rights reserved.

models proposed in the literature, we focus on traditional recurrent neural language models and revolutionized transformer-based language models. Specifically, we experimented with the transformer-based *CodeTran* (BERT), *RoBERTaCode* (RoBERTa), *TravTrans* (GPT2), *CodeGen*, and LSTM-based *CodeLSTM* models. We propose a methodology called Transformer Gated Highway which employs a self-supervised learning approach along with an enhanced encoder–decoder model that outperforms the recurrent and transformer baselines. The Gated Highway introduces a notion of recurrence, resulting in better modeling performance. We experimented with two different self-supervised learning approaches to study their impact on modeling performance. Additionally, we show the generalization of our approach by employing it for a different source code modeling task of syntax error correction. For the syntax error correction task, our target is to predict the correct syntax token (abstract token) rather than locating the position of the syntax error. The presented approach is evaluated extensively with two programming language datasets; *Java* and *C#*. The results have suggested that on average the proposed approach achieves around 95% accuracy rate for the correct syntax token prediction within the top three ranks and 90% for the source code suggestion task within the top-10 ranks for *Java*. The core intentions of this work are to explore the impact of learning strategies and model architecture design in the modeling of source code for enhanced performance. Specifically, we are interested in studying how various factors (learning strategies and model architecture) impact modeling performance.

The main contributions of this work are as follows:

1. We propose a methodology called Transformer Gated Highway (TGH) with self-supervised learning techniques that outperform four baselines including one recurrent-based and four transformer-based. Using a customized decoder along with a specifically tailored learning procedure, the presented Transformer Gated Highway exhibits enhanced performance compared to other baselines.
2. We evaluate the proposed approach with two different programming language datasets, *Java*, and *C#*, for the task of source code suggestion. Additionally, we employed the proposed methodology for the prediction of correct syntax tokens to reflect its generalization and discuss other possible implications.
3. Additionally, we experimented with different model architecture settings to evaluate their impact on the modeling performance. Specifically, we experimented with data size and model architecture designs for performance evaluation. We make the material used in this work publicly available.[1]

As mentioned earlier, the intent here is to boost the modeling performance for the source code suggestion task. Additionally, we explore how training procedures (i.e., encoding, self-supervised learning) and model architectures impact modeling performance. Specifically, we aim to explore the following research questions during this study.

1. **How well does the proposed approach perform compared to the transformer models for the task of source code suggestion?** To explore and answer this question, we first trained four transformer-based models as baselines for the task of source code suggestion. Specifically, we experimented with *CodeTran* which employs BERT, *ROBERTACode* which employs RoBERTa, *TravTrans* which employs GPT2, and *CodeGen* language models to evaluate the modeling performance. Additionally, we experiment with different self-supervised learning techniques and explore their impact on modeling performance. The intent here is to have a baseline and push that baseline for enhanced performance. Finally, we compare the enhanced baselines with the proposed approach for performance evaluation.
2. **What is the modeling performance of traditional Recurrent Neural Networks (RNNs) compared to the proposed approach for the Source code suggestion task?** In this research question, we explore the capabilities of traditional RNN-based models compared to the proposed approach. Here, we aim to study if the recurrent language models are completely dominated by transformer language models or if their performance can still be improved. Specifically, we aim to evaluate how different learning procedures with similar model architecture will impact the modeling performance for the task of source code suggestion.
3. **Exploring the impact of model design, data size, and the generalization of the proposed approach.** Here, we study the impact of model design by altering the encoder and decoder architectures. We also aim to study the impact of data size on modeling performance to learn how data size impacts modeling performance. We also aim to study if similar learning procedures with different models have any significant impact on the modeling performance for the task of source code suggestion. Finally, to validate the generalization we employ the proposed approach for the prediction of correct syntax tokens and discuss its applicability for other related tasks.

The rest of the paper is organized in the following manner. We discussed the related background in Section 2. The proposed methodology and the model architecture are discussed in detail in Sections 3 and 4. We discussed the training and evaluation procedures in Section 5 and the results are discussed in Section 6. We discuss the threats to validity in Section 7 followed by the related work discussed in Section 8. Finally, we conclude the work in Section 9.

2. Background

In this section, we have detailed briefly the vanilla RNN, LSTM, and transformer models.

2.1. Recurrent neural networks

Recurrent neural networks are the most widely adopted methods in the domain of source code modeling (White et al., 2015; Raychev et al., 2014; Hussain et al., 2020). Generally, an RNN applies a multi-layer Elman network (Elman, 1990) with an activation function non-linearly to an input sequence. For each element in the input sequence, each layer computes the following function:

$$ht = tanh(W_{ih}xt + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \tag{1}$$

$$tanh(inp) = \frac{exp(inp) + exp(-inp)}{exp(inp) - exp(-inp)} \tag{2}$$

where ht is hidden state at time t, xt is input at time t, and h_{t-1} is previous hidden state or initial state at $t = 0$. The ih, hh, W, and b represent input-to-hidden, hidden-to-hidden, weight, and bias respectively.

The vanilla RNN suffers from the vanishing gradient problem, LSTM (Hochreiter and Schmidhuber, 1997) was introduced to overcome this issue. LSTM works similarly to RNN, the difference between the two is how they calculate the hidden state. The LSTM

1 https://github.com/yaxirhuxxain/SS-TGH.

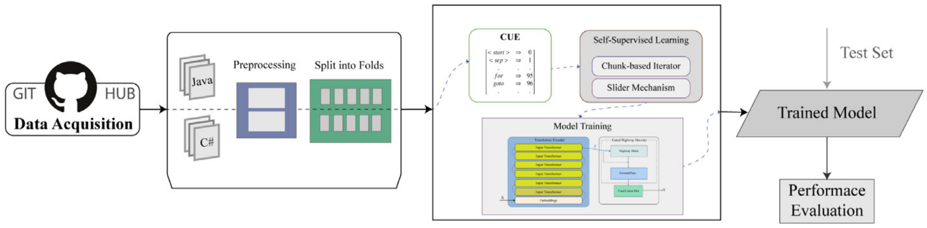


Fig. 1. Overall workflow of proposed approach.

is composed of several gates (i, f, g, o are the input, forget, cell, and output gates, respectively) controlling the context. For each element in the input sequence, each layer computes the following function:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{3}$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{4}$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{5}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{6}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{7}$$

$$h_t = o_t \odot \tanh(c_t) \tag{8}$$

where ct is cell state at time t. σ is sigmoid activation function and $\odot$ is element-wise multiplication. In our experiments, we utilize a single layer LSTM referred to as *CodeLSTM*. The input to *CodeLSTM* is passed through an embedding layer, and the last hidden state $\underset{\sim}{s}$ is passed through a linear layer for classifying the next possible *tokens*.

$$tokens = \underset{\sim}{s}W + b \tag{9}$$

2.2. Transformers

In this section, we briefly review the transformer model as well as its BERT variant. The transformer model was introduced in "*Attention is all you need*" (Vaswani et al., 2017). The original architecture is composed of two main blocks, an encoder block, and a decoder block where each block is repeated n-times. The backbone of each block is an attention function that can be described as mapping a query (qu) and a set of key–value (ke, va) pairs to an output (ou). The general formulation of the attention function can be described as:

$$Attention(qu,\ ke,\ va) = softmax(\frac{qu \ \cdot \ ke^T}{sv})\ va \tag{10}$$

here, sv is used to scale the dot product to avoid really small gradients. Both the encode and decoder blocks similarly utilize the attention mechanism. The decoder block further enforces a masking strategy to restrict the decoder to attend the next tokens. For more in-depth detail on the transformer model, we refer the readers to the original article (Vaswani et al., 2017) and other related links.[2] [3]

The BERT model uses a similar architectural design as by Transformer, omitting the decoder block entirely. It uses a masked language modeling approach to model pre-training and then predicts masked tokens. Later on, the pre-trained model can be fine-tuned for different tasks such as language translation, sentiment classification, etc. This reveals two characteristics of the BERT model. The first is the ability to pre-train to enhance modeling performance. The second is the fine-tuning phase for several tasks. In the pre-training phase, the model is trained from scratch on a large corpus to have a general abstraction of knowledge. Later on, pre-trained models can be adapted for a particular task or dataset, boosting the modeling performance without requiring the model training from scratch. Additionally, fine-tuning could be beneficial when a task has a limited amount of data. One could learn a general abstraction of knowledge during pre-training and can attain good modeling performance during fine-tuning even with a small task-specific dataset. For more in-depth detail on the BERT model, we refer readers to the original article (Devlin et al., 2018) and other related links.[4] [5]

In this study, we did not enforce pre-training on any of the models (the baselines and the proposed) for two reasons. First, the fine-tuning task of predicting code tokens would be very similar to the pre-training phase of predicting masked tokens as in BERT. Second, for a reasonable comparison among different approaches. In our experiments, we utilize the Transformer-based BERT model with a classification head as a baseline that we referred to as *CodeTran*.

3. Proposed methodology

In this section, we define the source code suggestion task as a classification task and comprehensively describe the proposed methodology. The task of syntax error correction is closely related to the task of source code suggestion which aims to predict the correct syntax token given the source code context. The overall workflow of the proposed approach is illustrated in Fig. 1. In the first step, we collect open-source datasets for two different programming languages; *Java* and *C#*. Next, we perform pre-processing to remove noise followed by splitting the pre-processed data into folds. Then, we use code unit encoding to transform code tokens into a form suitable for model training. Finally, we train Transformer Gated Highway with a self-supervised learning technique and evaluate its performance.

3.1. Problem definition

We can describe the source code suggestion as a classification task where the target is to predict the next code token provided a partial program at the current position. Let $p(tokens \mid context)$ be the probability distribution of code *tokens* given the partial program *context*. The intent here is to learn the probability p by employing a machine learning model (ML) which can be defined as:

$$p(tokens \mid context) = ML(context,\ \theta) \tag{11}$$

[2] https://jalammar.github.io/illustrated-transformer/.
[3] https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634.

[4] https://jalammar.github.io/illustrated-bert/.
[5] https://towardsdatascience.com/understanding-bert-bidirectional-encoder-representations-from-transformers-45ee6cd51eef.

here, θ represents the model parameters that are learned during model training by minimizing the cross-entropy loss over the *realtokens* and predicted *tokens*. This lines up with the general concept of language modeling, where given previous tokens $(x_1, x_2, \dots, x_{i-1})$ as the context in a sequence, the model tries to predict the next token (x_i):

$$p(x_i \mid x_1, x_2, \dots, x_{i-1}) \tag{12}$$

Similarly, we consider a program before the incorrect syntax token position as *context* and utilize the trained model to help produce the correct syntax token.

3.2. Data acquisition and preliminary processing

The first step in this context is data acquisition for model training, validation, and testing. For this intent, we collect data from GitHub for two different programming languages; *Java* and *C#*. GitHub is an open-source repository provider from where we can get the source code of real-world software systems. We collect the top 100 projects (software systems) for each programming language sorted by the star count which can be considered as the popularity measure. From the collected projects, we remove all non-language files as they are irrelevant to our target task. For example, from *Java* projects, we only keep files that end with *.java* extension containing the source code for that project. We remove projects that have zero bytes of data after the removal of non-language files and preprocessing.

Training a model with imprecise and invalid data will underperform and will possibly lead to incorrect predictions for the next code token classification. To alleviate this issue, we preprocess the collected projects by removing invalid source code files from each collected project. Specifically, we first parse the source code files by using ANTLR (ANother Tool for Language Recognition)[6] which leaves us with valid source code files. Second, we tokenize the source code files by replacing literals with their base types and space separating each source code token (i.e. *if(i==0)* $\Rightarrow$ *if (i == IntVal)*). These space-separated code units (tokens) are then combined into a single sequence corresponding to a file in a project. We combine all sequences found in a project with a new line directive and dump it into a single file corresponding to the project name it belongs to. There is a possibility of having duplicate source code files in projects which may impact the modeling performance. This percentage is most often reduced when splitting across projects (Allamanis, 2019) thus, we randomly split the collected projects into five folds containing an equal number of projects in each fold.

3.3. Code Unit Encoder (CUE)

Pre-processed projects cannot be directly utilized for modeling purposes as neural language models are inept to handle plain text. We first need to build an encoder–decoder system that will transform the source code units into a form that is suitable for neural learning. For this intent, we first build an abstract vocabulary (V_s) by collecting common programming keywords defined by their language grammars (i.e. Java[7], C#[8]) and some special tokens (*<start>*, *<sep>*, *<end>*, *<idf>*, *<pad>*, *<mask>*) for generalization purpose. Next, we build a code unit count over the training data for each fold where we count the number of occurrences for each code token seen in the training set. We sort the collected code units count in a series V_f that begins with the largest count and ends with the least count values. After the sorting process, we remove all singleton code units from V_f (White et al., 2015). Finally, we build a dictionary (V) of key–value pairs by combining the V_s and V_f. Having the V, we simply encode the provided input context by replacing each code unit with the value corresponding to the key as shown in listing 1. A reversal method can be adapted to decode the encoded code units back to their original form.

Algorithm 1: Self-supervised learning algorithms

```
F_s ⇒ Total token length of a file
C_s ⇒ Context Size
Function ChunkBasedIterator(P):
    for file in Projects do
        for i in range(0, F_s, C_s) do
            yield file[i : i + C_s]
        end
    end
End Function
Function SliderMechanism(P):
    for file in Projects do
        for i in range(0, F_s − C_s) do
            yield file[i : i + C_s]
        end
    end
End Function
```

Listing 1: Example of code units and their encoding.

$$\begin{bmatrix} <start> & \Rightarrow & 0 \\ <sep> & \Rightarrow & 1 \\ . & . & . \\ for & \Rightarrow & 95 \\ goto & \Rightarrow & 96 \\ . & . & . \\ try & \Rightarrow & 112 \\ void & \Rightarrow & 113 \\ . & . & . \end{bmatrix}$$

3.4. Self-supervised learning

We adopt self-supervised learning-based model training to educate the models. Specifically, we use two different variations of self-supervised learning techniques, a chunk-based approach, and a slider mechanism. The pseudo-codes for these methods are presented in algorithm 1. In the chunk-based approach, we simply divide source code files into fixed-size context (C_s) chunks and learn to predict the next token provided the chunk. This matches the max-length-based learning procedure described in the transformers language modeling training scripts.[9][10] In the slider mechanism, we first pad each source code file with a special token *<pad>* with the size equal to one less than C_s. Next, we view the source code files with a fixed size window by iterating over the file, increasing one index at a time.

[6] https://www.antlr.org/.
[7] https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html.
[8] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/lexical-structure.
[9] https://github.com/huggingface/transformers/blob/master/examples/pytorch/language-modeling/run_mlm.py.
[10] https://github.com/huggingface/transformers/blob/master/examples/pytorch/language-modeling/run_plm.py.

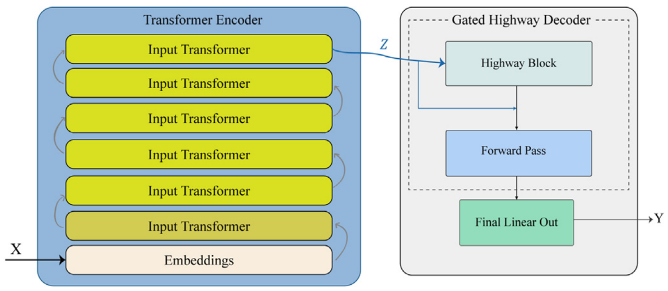


Fig. 2. Overall architecture design of Transformer Gated Highway.

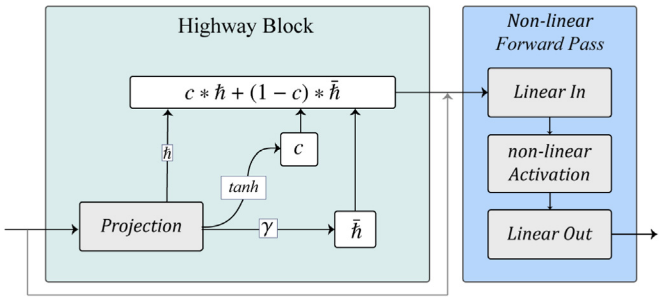


Fig. 3. Overall flow of Gated Highway Decoder.

4. Transformer Gated Highway

In this section, we describe the architecture of the Transformer Gated Highway, shown in Fig. 2. The model maps an input sequence context of tokens $X = (x_1, x_2, \ldots, x_n)$ into a sequence of continuous representations $Z = (z_1, z_2, \ldots, z_n)$. Given Z, the model classifies the next code token Y. The Transformer Gated Highway uses an attention-based transformer model to map the input tokens into a continuous representation and uses the Gated Highway Decoder to classify the next token.

4.1. Transformer encoder

Provided the encoded input context sequence X, our goal is to transform it into a hidden state Z. For this intention, we employ a variant of the transformer-based model (Vaswani et al., 2017; Raffel et al., 2019) that transforms X into Z, which is composed of six encoder blocks where the first block contains relative attention. Our transformer encoder block works similarly to the originally proposed (Raffel et al., 2019). First, an encoded input sequence embedding is passed into the encoder. Each block contains two sub-blocks: a self-attention block and a feed-forward block. Layer normalization is applied to the input of each sub-blocks where the activation is re-scaled and no bias, followed by a skip connection (He et al., 2016) and dropout (Srivastava et al., 2014) is utilized on input, output, skip connection, attention weights, and within the feed-forward network.

4.2. Gated Highway Decoder

The Gated Highway Decoder is used to map the state Z for classifying the next token Y. Having the Z, we take the hidden state $\hbar$ corresponding to the first token, and pass it through a Highway Block followed by a non-linear activated forward pass as shown in Fig. 3. The highway block can be described as:

$$\bar{\hbar} = \gamma\ (\hbar) \tag{13}$$

$$c = \tanh(\hbar) \tag{14}$$

$$Highway = c * \hbar + (1-c) * \bar{\hbar} \tag{15}$$

Here, $\hbar$ is the input hidden state, γ is non-linear activation (Agarap, 2018), c is the carry gate, and $*$ is element-wise multiplication. Moreover, we make use of skip connection with highway block output. Finally, the output of the gated highway is passed through a non-linear activated forward pass that is composed of two linear transformations with non-linear activation γ in between. We refer to this block as Gated Highway Decoder (GHD) and it can be formulated as:

$$GHD(\hbar) = \gamma(Highway(\hbar)W1)W2 \tag{16}$$

$$\gamma(x) = max(0\ ,\ x) \tag{17}$$

where $W1$ and $W2$ represent weights for linear transformations.

4.3. Embedding and classification

Corresponding to general NLP classification models, we make use of embeddings to convert an encoded input to a vector of H_{dim} dimension that is an input to our encoder block. For the classification of the next code token, we simply impose a linear transformation having the input dimension of H_{dim} and output dimension of vocabulary size containing prediction probabilities for the possible next code token.

5. Evaluation

In this section, we describe the training procedure for our proposed approach and baseline models.

5.1. Hardware and system settings

The experiments of this study were carried out on Intel server Xeon Silver 4110 having 32 cores and 128 GB of RAM running ubuntu 18.04.5 LTS. We make use of parallel computing by utilizing half of the number of CPU cores during dataset preparation. The system is equipped with a single Nvidia RTX 2080 GPU with 8 GB of memory. The models are built with Python[11] version 3.9.0 and Pytorch[12] version 1.9.0 with CUDA support. One important thing to mention here is that the training and testing are offline and thus have no impact on prediction time. These models are capable of providing predictions for the next token suggestion in milliseconds.

5.2. Dataset and encoder

We utilize the dataset described in Section 3.2. We divide each fold into train, valid, and test sets with a ratio of 50%–25%–25%. The splits are done at the project level rather than at the file level. During the split, we ensure that each project belongs to only one of the folds and one of the train, valid, or test sets in that specific fold. This helps ensure that the proposed approach is capable of providing cross-project predictions and reduces the percentage of duplicate source code. The overall statistics of the dataset are presented in Table 1. It provides the total project count before and after pre-processing, total file count, total token count, and average (rounded) unique token count between projects.

For sequence encoding, we utilize the data encoder described in Section 3.3 for *Transformer Gated Highway* model. The *CodeTran* utilizes the byte-pair encoding (BPE) (Sennrich et al., 2015) which is commonly adopted (Vaswani et al., 2017; Liu et al., 2019a; Kim et al., 2021). We train a combined byte-pair encoder for both languages. We restrict the vocabulary size to 30 K code units for both, the byte-pair encoder and the encoder discussed in Section 3.3.

11 https://www.python.org/.
12 https://pytorch.org/.

Table 1
Dataset statistics.

	Java	*C#*
Num projects	100	100
After pre-processing	95	98
Total files	155,303	140,338
Total token	110,243,044	86,921,848
Average unique	15,422	14,050

5.3. Model parameters and baselines

We use the hyperparameters described throughout the paper, specified otherwise. We utilize *AdamW optimizer* (Loshchilov and Hutter, 2017) a variant of Adam optimizer (Kingma and Ba, 2014) with its default parameter value of 0.001 and employed dropout with the value of 0.5. All models are trained with an upper limit of twenty epochs. The Transformer models are built and trained by using Transformers library[13] version 4.9.1 with their default settings (specified otherwise). Further, models are trained matching the architecture as follows: $d_{model} = 300$, $layers = 6$, $heads = 6$, and $FF = d_{model}*4$ which is similar to Kim et al. (2021). For transformer models, we use the warm-up steps equal to the step-per-epoch integer division by five. Each model is evaluated twice an epoch and logged to save the model state.

CodeLSTM: We train the recurrent neural language model-based LSTM model as a baseline which we refer to as *CodeLSTM*. The baseline model is similar to the one that has been employed by other studies (Kim et al., 2021; Yang, 2019).We use single-layer LSTM with the hidden dimension $H_{dim} = 300$. We trained the models with early stopping imposed on validation loss with three hits and three warm-up epochs.

CodeTran: We train the transformer-based BERT (Devlin et al., 2018) model as a baseline that we refer to as CodeTran. BERT has been employed (Feng et al., 2020; Guo et al., 2020) for various related tasks such as code document generation and code search, etc. Unlike these studies, we employed BERT for the next token classification task. We have adopted the public implementation of BERT for classification[14] and trained it with a chunk-based iterator approach along with BPE. Furthermore, we experimented with an enhanced version of *CodeTran* that we refer to as *CodeTran+* by replacing the chunk-based iterator with the slider mechanism.

TravTrans: Another transformer-based baseline that we adopt is *TravTrans* (Kim et al., 2021) which employs the GPT2 model. We used the original model implementation provided by the authors.[15] We use the same hyper-parameters as in the original article and train the model similar to the*CodeTran+*.

RoBERTaCode: We train the transformer-based RoBERTa (Liu et al., 2019b) model as a baseline utilized in a recent study (Ciniselli et al., 2021a) that we refer to as *RoBERTaCode*. Here, we employ *RoBERTaCode* for the classification of the next token instead of predicting masked tokens. We train *RoBERTaCode* similar to *CodeTran+* and *TravTrans*.

CodeGen: We also train a more recent transformer-based model named *CodeGen* (Nijkamp et al., 2022). We adopt the original model implementation provided by the authors and train it similarly to *codeTran+*, *TravTrans*, and *RoBERTaCode*.

5.4. Evaluation process and metrics

Preferably, we would manually evaluate the results, but the manual evaluation is very difficult to scale. Thus, we have adopted the measures used by previous works (White et al., 2015; Hindle et al., 2012; Santos, 2018; Alon et al., 2019) that includes *Accuracy@k*, *MRR@k*, *Precision*, *Recall*, and F_1. We divide these metrics into two categories. First, exact match-based metrics that include *Precision*, *Recall*, and F_1 in which we compute the scores based on the first prediction. Second, rank-based metrics that include *Accuracy@k* and *MRR@k* in which we compute the scores based on top predicted ranks k for a given input. Once the model is trained, we utilize the test set to evaluate the performance of the trained model. Provided the *context*, we compare the outcome produced by the model with the actual token in the test set and report the computed metrics for performance evaluation. The exact match-based metrics can be defined as:

$$Precision = \frac{TP}{TP + FP} \tag{18}$$

$$Recall = \frac{TP}{TP + FN} \tag{19}$$

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{20}$$

Here, TP are the predictions that are suggested correctly, FP are the incorrect predictions and FN are false-negative predictions. Having the prediction probabilities ($\hat{Y}$) along with the real tokens (Y), we can compute the rank-based metrics as:

$$Accuracy@k = \frac{1}{N}\sum_{i}^{N} 1(Y_i == \hat{Y}_i) \tag{21}$$

$$MRR@k = \frac{1}{N}\sum_{i}^{N} \frac{1}{|Y_i|}\sum_{j=1}^{|Y_i|} \frac{1}{\hat{Y}^j} \tag{22}$$

Here, Y is the list of real tokens from the test set, $\hat{Y}$ is the list of probabilities predicted by the trained model, N is the size of the test set, and K is the number of top-k predictions over which the metric is computed.

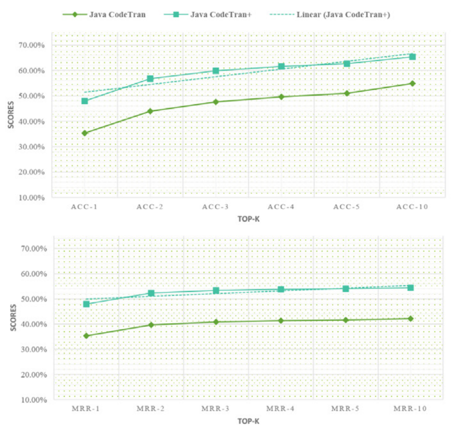
6. Results

6.1. Exploring the impact of learning procedures and transformers on the source code suggestion task

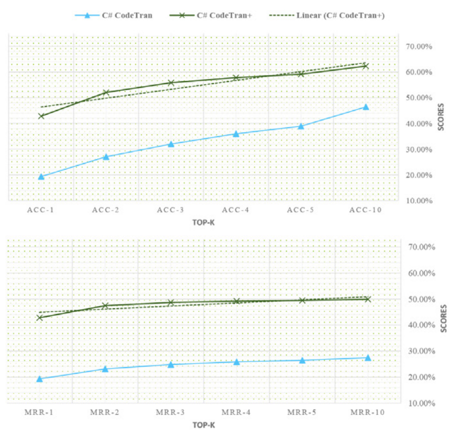
In this section, we evaluate the performance of *CodeTran*, *CodeRoberta*, *TravTrans*, and *CodeGen* on the source code suggestion task. Additionally, we explore how different learning procedures can impact modeling performance. The intent here is to build baselines and push for enhanced performance. During our experiments, we observe that *CodeTran* is under-performing and early stopping was triggered on early epochs (3∼5) during model training. This may be because the model is unable to learn most of the source code *contexts* and naively triggers early stopping during model training. To further study this aspect, we lift the early stopping on *CodeTran* and train the models again. In this setting, the *CodeTran* model is trained for the maximum number of epochs whereas *CodeTran+* is trained for an optimum number of epochs by stopping model training when early stop criteria meet. The results of *CodeTran* and *CodeTran+* are presented in Fig. 4. The x-axis in this plot provides information about the accuracy and MRR scores for top-k ranks ranging between $k = [1, 2, 3, 4, 5, 10]$ and the y-axis provides information about the scores obtained for each specific rank in percentage for ease of understanding.

Based on the results (Fig. 4), we can observe that the *CodeTran* baseline is still underperforming and the performance is poor, especially in the case of *C#*. On average, the accuracy score of *CodeTran* ranges between 35∼54% and MRR score ranges between

13 https://huggingface.co/.
14 https://huggingface.co/transformers/model_doc/bert.html.
15 https://github.com/facebookresearch/code-prediction-transformer.

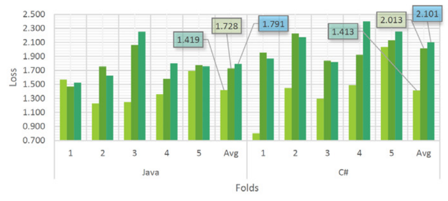


(a) Accuracy and MRR scores on Java.

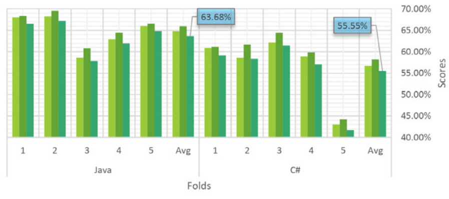


(b) Accuracy and MRR scores on C#.

Fig. 4. Performance evaluation of *CodeTran* and *CodeTran+*.



(a)



(b)

Fig. 5. Performance evaluation of proposed approach on different programming languages (*Java* and *C#*).
(a) Analysis of loss during model training, validation and testing (lower is better).
(b) Analysis of modeling Precision, Recall rate and F1 score on test set (higher is better).

35∼42% for *Java*. Where the average accuracy for *C#* ranges between 19∼46% and MRR ranges between 19∼27%. The enhanced *CodeTran+* baseline achieves a significant boost in modeling performance as compared to *CodeTran*. On average, the modeling accuracy ranges between 47∼65% and ranking capability (MRR) ranges between 47∼54% for *Java*. The average performance gain $((current - previous)/previous)$ in accuracy and MRR score for top-1 is 35.82% and 121.34% for *Java* and *C#*, respectively.

Analyzing the results presented in Table 2, we can see that on average the *CodeTran* achieves the Recall rate of 35.31% with 28.58% Precision for *Java* and 19.34% Recall with 13.18% Precision for *C#*. Compared to *CodeTran*, *CodeTran+* achieves better performance by attaining the 47.96% Recall with 47.55% Precision and 42.80% Recall with 42.00% Precision for *Java* and *C#*, respectively. Analyzing the results further, we notice that the average F1 score for *CodeTran+* is 46.16% for *Java* and 40.70% for *C#*, which are comparatively closer to each other (within the range of 40%). The results for the slider mechanism *(CodeTran+)* are all within the same bracket of 40% where the results of *CodeTran* are more diverse and the difference between them is relatively large when comparing *Java* with *C#*. The *CodeTran+* uses a similar modeling procedure and architecture design with the sole difference in learning the contexts, yet consistently outperforms *CodeTran* with a significant margin. This raises the question about the relative importance of pre-processing details like the encoding and sequence learning that we explore in this work. While comparing the results of baselines, the modeling performance is closely related and the difference is marginal. In the case of Java, *CodeTran+* achieves precision score of 47.55% as compared to *TravTrans* achieves precision score of 48.29%, *RoBERTaCode* achieves precision score of 45.58% and *CodeGen* achieves precision score of 47.25% whereas for *C# CodeTran+* achieves precision score of 42.00% compared to *TravTrans'* 42.68%, *RoBERTaCode's* 39.99% and *CodeGen's* 40.75%. From the results, it is evident that the learning procedure impacts significantly on the modeling performance and the performance gain between different model architectures with similar learning procedures is marginal.

6.2. Comparative analysis of proposed approach with baselines

The results of rank-based metrics are presented in Table 3. We report the accuracy (Acc@K) and Mean Reciprocal Rank (MRR@K) where the values of K are [1, 3, 5, 10]. Results exhibit a significant performance boost where the proposed approach achieved the average accuracy (Acc@10) of 90.10% with ranking capability (MRR@10) 75.13% and 86.05% accuracy (Acc@10) rate with MRR@10 rate of 68.66% for *Java* and *C#*, respectively. The proposed approach outperform *TravTrans* and *CodeTran+* for all cases. The average performance gain for top-1 ranks (MRR) is 37.48% for *Java* and 36.12% for *C#* compared with *TravTrans*, where it gains 38.15% for *Java* and 36.08% for *C#* compared with *CodeTran+*.

Fig. 5 illustrate the results of our experiments. The left-hand side plot illustrates the loss during model training, validation, and testing whereas the right-hand plot presents the modeling precision and recall rate. Moreover, it presents the harmonic meaning (F1) of precision and recall. The average train loss of the proposed approach is closely related for both programming languages (0.419% = *Java*, 0.413% = *C#*) reflecting good modeling performance during model training. While comparing the average validation and test loss for *Java* programming language, the proposed approach exhibits a better modeling performance (1.728% = Valid, 1.791 = Test) as compared to *C#* (2.013% = Valid, 2.101 = Test). The proposed approach achieved 64.77% precision with 64.48% Recall rate for *Java* programming language whereas *C#* achieved 56.71% precision with 58.26% Recall rate. A performance dip can be noticed for *C#*'s fold five which can be anomalous since

Table 2
Performance evaluation of *CodeTran*, *CodeTran+*, *TravTrans*, *RoBERTaCode* and *CodeGen*.

	Fold	Java			C#		
		Precision	Recall	F1	Precision	Recall	F1
CodeTran	1	33.08%	37.45%	34.05%	28.16%	32.74%	28.75%
	2	33.83%	39.24%	34.99%	0.55%	7.01%	1.02%
	3	22.40%	29.87%	23.54%	27.85%	33.62%	29.24%
	4	19.76%	29.82%	22.46%	8.91%	17.37%	11.20%
	5	33.81%	40.19%	35.23%	0.44%	5.95%	0.82%
	Avg	**28.58%**	**35.31%**	**30.06%**	**13.18%**	**19.34%**	**14.21%**
CodeTran+	1	51.66%	49.63%	48.78%	44.49%	44.61%	42.67%
	2	44.93%	47.57%	44.93%	41.28%	43.38%	40.92%
	3	39.45%	42.36%	39.37%	48.03%	48.10%	46.37%
	4	47.89%	47.65%	46.17%	43.25%	42.31%	40.80%
	5	53.83%	52.60%	51.55%	32.93%	35.60%	32.71%
	Avg	**47.55%**	**47.96%**	**46.16%**	**42.00%**	**42.80%**	**40.70%**
TravTrans	1	50.44%	48.51%	47.31%	43.65%	43.14%	41.32%
	2	49.30%	48.34%	46.94%	42.52%	43.33%	41.26%
	3	41.12%	42.04%	40.01%	48.28%	47.68%	46.06%
	4	47.56%	47.47%	45.48%	43.40%	42.75%	41.09%
	5	53.01%	52.30%	50.85%	35.56%	37.27%	34.74%
	Avg	**48.29%**	**47.73%**	**46.12%**	**42.68%**	**42.83%**	**40.89%**
CodeRoberta	1	47.44%	50.00%	47.33%	40.71%	42.03%	39.96%
	2	53.86%	52.60%	51.36%	44.00%	45.32%	43.01%
	3	42.02%	45.36%	42.32%	38.81%	40.89%	37.91%
	4	43.19%	45.00%	42.45%	41.63%	42.92%	40.46%
	5	41.39%	44.53%	41.32%	34.81%	39.29%	35.61%
	Avg	**45.58%**	**47.50%**	**44.96%**	**39.99%**	**42.09%**	**39.39%**
CodeGen	1	51.32%	52.33%	50.27%	40.91%	42.76%	40.51%
	2	53.68%	52.55%	51.26%	44.44%	45.86%	43.81%
	3	44.14%	46.79%	44.08%	38.90%	41.58%	38.54%
	4	43.82%	45.65%	42.91%	43.19%	44.42%	42.09%
	5	43.27%	45.72%	43.05%	36.30%	40.31%	36.87%
	Avg	**47.25%**	**48.61%**	**46.31%**	**40.75%**	**42.99%**	**40.36%**

it contains the least amount of training examples among both programming languages. The results exhibit that the modeling performance among folds and programming languages varies but with comparatively related modeling performance where *Java* exhibits good modeling performance by achieving 63.68% F1 score and *C#* attaining 55.55% F1 score.

We present the results of modeling Precision, Recall, and F1 as a violin plot in Fig. 6. The markers in each violin represent the score for each fold and the red line intersecting the violins represents the average scores. From the plots, we can observe that the minimum precision and recall rate of the proposed approach is higher compared with the *TravTrans* baseline's highest score in the case of *Java*. On average, the modeling precision of the proposed approach is 64.77% with the recall rate of 65.94% reflecting good modeling performance (F1 = 63.68%) where the baseline achieves a modeling precision of 47.55% with a recall rate of 47.96% exhibiting comparatively lower modeling performance (F1 = 46.16%).

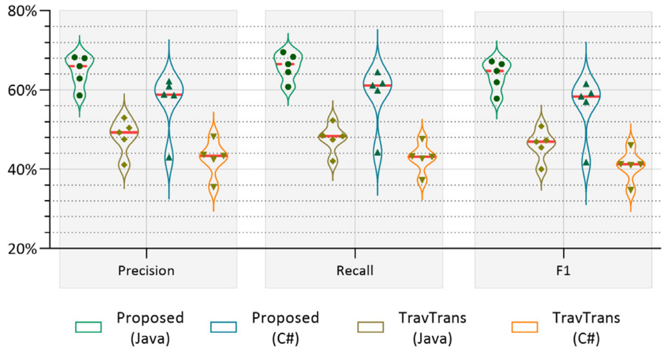


Fig. 6. Precision, Recall, F1 score comparison between proposed approach and *TravTrans* baseline.

6.3. *Exploring the limits and capabilities of RNNs in contrast to transformer language models*

In this section, we explore the impact of different models by employing similar training procedures. The intent here is to explore how similar model training impacts modeling performance with different model architectures. Here, we present our findings on *Java* programming language only, partly for the sake of simplicity and partly due to its superior modeling performance exhibited in our previous experiments. In this regard, we train two different variants of LSTM-based models named *CodeLSTM+* and *CodeLSTM++*. Both models are trained with a single-layer LSTM-based recurrent neural network (as described in Section 2.1). We train *CodeLSTM+* similar to *CodeTran+* by replacing the transformer with LSTM and train *CodeLSTM++* by adopting our proposed approach. The results of our experiments are presented in Fig. 7. In all cases, the performance of the recurrent neural network is closely related with a marginal difference of 1∼2% (±) among folds. When comparing *LSTM+* against *LSTM++* it is yet again observed that the learning procedure impacts significantly on the modeling performance. The average performance gain from the *LSTM+* to *LSTM++* is around ∼37% for Precision and ∼39% for Recall rate. The *CodeLSTM++* performs significantly better as compared to *CodeTran+* and *CodeLSTM+*, where the proposed approach still attains the best performance among all.

From the results, we can observe that the best recurrent baseline (*CodeLSTM++*) achieves a significant performance boost with a modeling precision of 62.24% with the recall rate of 63.50% attaining good modeling performance (F1 = 60.75%) compared to

Table 3
Comparative analysis of proposed approach with baselines. We report the scores for K = [1, 3, 5, 10] and the MRR scores are represented in percentage for the sake of simplification. Both *TravTrans* and *CodeTran+* are trained by employing BPE and utilizes slider mechanism. Proposed approach adopts the same slider mechanism and follows the procedure described in Section 3 with Transformer Gated Highway described in Section 4.

		Fold	Acc@1	MRR@1	Acc@3	MRR@3	Acc@5	MRR@5	Acc@10	MRR@10
CodeTran+	Java	1	49.63%	49.63%	62.41%	55.52%	65.79%	56.30%	68.68%	56.70%
		2	47.57%	47.57%	58.54%	52.57%	61.05%	53.15%	63.46%	53.48%
		3	42.36%	42.36%	54.79%	47.98%	57.73%	48.66%	60.53%	49.05%
		4	47.65%	47.65%	59.49%	53.02%	62.32%	53.67%	65.05%	54.04%
		5	52.60%	52.60%	64.13%	57.89%	66.59%	58.45%	69.04%	58.79%
		AVG	**47.96%**	**47.96%**	**59.87%**	**53.40%**	**62.70%**	**54.05%**	**65.35%**	**54.41%**
	C#	1	44.61%	44.61%	58.31%	50.81%	61.60%	51.57%	64.80%	52.01%
		2	43.38%	43.38%	55.69%	48.95%	58.71%	49.65%	61.86%	50.08%
		3	48.10%	48.10%	60.99%	53.95%	63.86%	54.61%	66.92%	55.03%
		4	42.31%	42.31%	55.67%	48.32%	59.48%	49.20%	62.85%	49.66%
		5	35.60%	35.60%	48.46%	41.40%	52.07%	42.22%	55.24%	42.65%
		AVG	**42.80%**	**42.80%**	**55.83%**	**48.69%**	**59.14%**	**49.45%**	**62.33%**	**49.88%**
TravTrans	Java	1	48.51%	48.51%	63.38%	55.21%	66.15%	55.85%	68.43%	56.17%
		2	48.34%	48.34%	60.19%	53.73%	62.78%	54.33%	65.12%	54.65%
		3	42.04%	42.04%	54.97%	47.87%	58.14%	48.60%	60.88%	48.97%
		4	47.47%	47.47%	60.32%	53.29%	63.18%	53.95%	65.65%	54.29%
		5	52.30%	52.30%	65.04%	58.13%	67.58%	58.72%	69.89%	59.03%
		AVG	**47.73%**	**47.73%**	**60.78%**	**53.65%**	**63.57%**	**54.29%**	**66.00%**	**54.62%**
	C#	1	43.14%	43.14%	57.36%	49.54%	60.98%	50.37%	64.24%	50.82%
		2	43.33%	43.33%	57.07%	49.56%	60.36%	50.32%	63.42%	50.74%
		3	47.68%	47.68%	61.45%	53.94%	64.58%	54.66%	67.52%	55.06%
		4	42.75%	42.75%	56.83%	49.13%	60.51%	49.98%	63.79%	50.43%
		5	37.27%	37.27%	51.19%	43.53%	54.59%	44.32%	57.56%	44.73%
		AVG	**42.83%**	**42.83%**	**56.78%**	**49.14%**	**60.20%**	**49.93%**	**63.30%**	**50.35%**
Proposed	Java	1	68.34%	68.34%	86.58%	76.32%	90.09%	77.14%	92.30%	77.44%
		2	69.57%	69.57%	85.82%	76.97%	88.99%	77.70%	91.38%	78.03%
		3	60.79%	60.79%	80.04%	69.51%	84.22%	70.48%	87.44%	70.93%
		4	64.48%	64.48%	83.23%	72.96%	87.07%	73.85%	89.95%	74.25%
		5	66.53%	66.53%	82.23%	73.70%	85.91%	74.55%	89.42%	75.03%
		AVG	**65.94%**	**65.94%**	**83.58%**	**73.89%**	**87.26%**	**74.74%**	**90.10%**	**75.13%**
	C#	1	61.14%	61.14%	81.21%	70.26%	86.07%	71.38%	89.74%	71.88%
		2	61.65%	61.65%	80.27%	70.06%	84.45%	71.03%	87.84%	71.49%
		3	64.43%	64.43%	82.37%	72.59%	86.14%	73.46%	89.25%	73.89%
		4	59.85%	59.85%	78.18%	68.23%	82.68%	69.27%	86.29%	69.78%
		5	44.23%	44.23%	66.40%	54.24%	72.21%	55.58%	77.13%	56.26%
		AVG	**58.26%**	**58.26%**	**77.69%**	**67.08%**	**82.31%**	**68.14%**	**86.05%**	**68.66%**

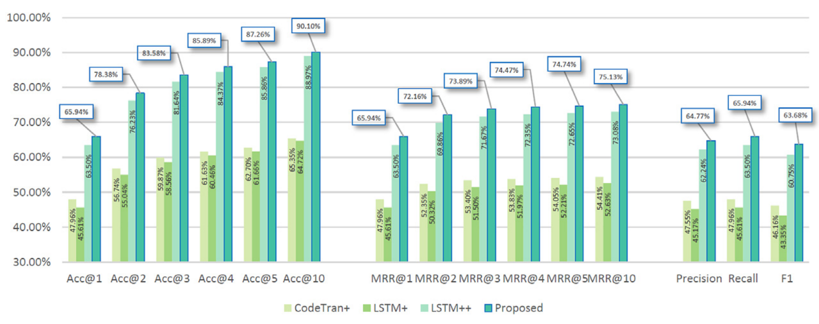


Fig. 7. Performance comparison between the proposed approach and enhanced recurrent neural baselines.

CodeTran+. The *CodeLSTM++* losses performance when comparing the modeling performance with the proposed approach. On average, the performance loss is 4.8% (F1) with a decrease of 4.1% and 3.8% in modeling precision and recall rate, respectively. The results reflect that different model training procedures can significantly impact the modeling performance whereas the different model architectures have a comparatively small impact on the modeling performance.

6.4. Impact of data size and model architecture on proposed approach

In this section, we present the results of different model settings that we have tried during our experiments. Further, we studied the impact of data size on modeling performance. Table 4 exhibits the data size and modeling performance for both programming languages for all folds. The table provides information regarding the total number of contexts in the train, valid, and test

Table 4
Performance evaluation: Data size versus modeling performance.

	Fold	Contexts			Test			Loss		
		Train	Valid	Test	Precision	Recall	F1	Train	Valid	Test
Java	1	14,466,490	4,699,302	1,784,628	68.06%	68.34%	66.52%	1.567	1.468	1.526
	2	12,683,807	22,651,300	3,122,207	**68.26%**	**69.57%**	**67.23%**	1.228	1.755	1.624
	3	6,119,912	4,954,229	11,196,844	58.64%	60.79%	57.86%	1.245	2.061	2.250
	4	5,551,185	7,767,036	4,576,393	62.90%	64.48%	61.94%	1.359	1.580	1.801
	5	9,359,732	514,377	4,007,726	66.00%	66.53%	64.82%	1.693	1.775	1.757
C#	1	31,304,342	1,612,646	3,364,220	60.91%	61.14%	59.16%	0.803	1.953	1.866
	2	5,232,070	1,970,909	1,700,173	58.60%	61.65%	58.36%	1.448	2.225	2.173
	3	19,353,862	1,676,922	404,619	**62.18%**	**64.43%**	**61.47%**	1.294	1.837	1.819
	4	6,377,324	1,694,579	1,554,743	58.85%	59.85%	57.01%	1.488	1.921	2.399
	5	3,765,669	6,627,242	2,962,394	43.02%	44.23%	41.74%	2.032	2.130	2.251

Table 5
Exploring the impact of different model settings on modeling performance.

	Acc@1	Acc@5	Acc@10	MRR@1	MRR@5	MRR@10	Precision	Recall	F1
Setting 1	50.66%	73.76%	76.33%	50.66%	60.43%	60.78%	47.68%	50.66%	46.25%
Setting 2	65.62%	85.81%	89.11%	65.62%	73.91%	74.37%	65.65%	65.62%	64.19%
Setting 3	52.80%	73.45%	77.02%	52.80%	61.35%	61.82%	49.99%	52.80%	49.05%
Setting 4	66.29%	84.96%	87.64%	66.29%	74.10%	74.47%	65.38%	66.29%	64.52%
Setting 5	65.29%	84.39%	87.48%	65.29%	73.21%	73.63%	65.24%	65.29%	63.65%
Setting 6	66.31%	**86.14%**	89.34%	66.31%	74.48%	74.92%	65.80%	66.31%	64.55%
Proposed	**66.53%**	85.91%	**89.42%**	**66.53%**	**74.55%**	**75.03%**	**66.00%**	**66.53%**	**64.82%**

Abbrivations	Settings
OH = Original Highway (Srivastava et al., 2015)	Setting 1: Encoder = Section 4.1 Decoder = OH + MFF + LT
CH = Custom Highway 4.2	Setting 2: Encoder = Section 4.1 Decoder = CH + MFF + LN + LT
MFF = Feed-Forward with Mish (Misra, 2019) activation	Setting 3: Encoder = Section 4.1, Decoder = FF
FF = Feed-Forward with no activation	Setting 4: Encoder = Section 4.1 Decoder = LT
LT = Linear layer with Tanh activation	Setting 5: Encoder = Section 2.2 Decoder = Section 4.2
LN = Layer Norm	Setting 6: Encoder = Section 4.1 Decoder = CH + MFF
	Proposed: Encoder = Section 4.1 Decoder = Section 4.2

sets. We only present Precision, Recall, and F1 scores calculated on the test sets for simplicity. Further, we present the modeling loss for train, valid, and test sets. From the presented results, we can observe that the modeling performance improves when the train and valid sets are relatively large. In the case of *Java*, fold 4 exhibits superior performance compared to other folds, and fold 3 in the case of *C#*. Analyzing the results further, one can notice that fold 1 for both programming languages is the largest one, but still, their performance is comparatively low (compared to fold 4 of *Java* and fold 3 of *C#*). In the majority of cases, the modeling performance is good when training and valid data sizes are relatively large.

In addition to different model training approaches, we also experimented with different variations of model architectures to see their impact on modeling performance. In particular, we experimented by changing the encoder and decoder blocks for TGH. Additionally, we trialed different combinations of layers in the decoder block to exhibit how they impact the modeling performance. The results of these experiments are represented in Table 5. The top section of the table presents the results of each setting and the bottom section of the table provides added details regarding each setting. From the results, we can observe that our customized highway decoder significantly improves the modeling precision, where the difference is around 18% (Setting-1 vs. Setting 4–6). When comparing the type of encoder used (Setting 5 vs. Proposed), the difference is comparatively small (F1 ∼1.18%). The variation of different layers in the customized highway decoder also has a relatively small impact on modeling precision ranging around 1∼2%.

6.5. Performance evaluation of proposed approach on syntax error correct task

To verify the generality of the proposed approach, we utilize our best model and train it to help predict the correct syntax token provided a source code context. The intent here is to produce the correct syntax token by assuming the provided context as the location before the syntax error token. We train the model by employing a similar procedure by enforcing the model to restrict the predictions to abstract vocabulary (discussed in Section 3.3) in other words the syntax tokens. Here, we evaluate the syntax error correction capability of our model by predicting the correct syntax token provided by the source code context. Having a test set, we compare the predicted token produced by the model with the correct token in the test set and report the results here. The results of our experiments are presented in Table 6. From the results, we can observe that the average accuracy of the correct syntax token prediction is around 95% within the top-3 ranks. The proposed approach exhibits that the modeling precision, recall rate, and F1 score are all above 75%. To further evaluate the ranking capability of the proposed approach for syntax error correction, we employed the MRR metric. The result of MRR exhibits that the ranking ability of the proposed approach is above 85% within the top three ranks, reflecting good modeling performance.

6.6. Discussion and utility applications

The presented approach works well due to two main reasons: first, the specifically tailored learning strategy significantly boosts the modeling performance, and second, the customized highway decoder provides further support exhibiting performance improvements. We believe the source code contains vital long- and short-term information that is retained better with our proposed approach resulting in better modeling performance. Further, the applicability of the proposed approach is not limited to the tasks discussed earlier. We further discussed two possible extensions, the first extension is related to the code generation task in which the target is to produce a whole code statement or block of code instead of a single next token. This can be achieved by

Table 6
Performance evaluation on syntax error correction task.

Fold	Acc@1	Acc@3	Acc@5	Acc@10	MRR@1	MRR@3	MRR@5	MRR@10	Precision	Recall	F1
1	77.84%	95.04%	98.47%	99.34%	77.84%	85.86%	86.68%	86.80%	79.09%	77.84%	76.74%
2	79.60%	96.53%	98.65%	99.61%	79.60%	87.43%	87.93%	88.07%	79.30%	79.60%	78.46%
3	74.56%	93.78%	97.00%	98.78%	74.56%	83.34%	84.09%	84.34%	73.78%	74.56%	73.09%
4	76.29%	95.39%	98.20%	99.40%	76.29%	85.06%	85.71%	85.88%	75.74%	76.29%	74.96%
5	79.01%	95.77%	98.14%	99.32%	79.01%	86.74%	87.29%	87.46%	77.59%	79.01%	77.38%
AVG	**77.46%**	**95.30%**	**98.09%**	**99.29%**	**77.46%**	**85.69%**	**86.34%**	**86.51%**	**77.10%**	**77.46%**	**76.13%**

progressively generating subsections of a partial code with beam search strategies (Koehn, 2004; Freitag and Al-Onaizan, 2017). The second extension is related to syntax error correction in which a complete solution can be provided which can locate and correct syntax errors at the same time. We consider these two extensions as our future work. Another pathway for the improvement of this work is better preprocessing strategies such as including the token type information, improving the quality of the dataset, parameter optimization, and better architecture designs.

7. Threats to validity

Although the proposed approach improves the modeling performance, several limitations still need to be addressed. The evaluation of the proposed approach is done on each programming language and folds independently. It may exhibit different modeling performances for cross-fold or language evaluation. This limitation could be overcome by utilizing transfer learning where we could utilize the knowledge of these models and evolve it for cross-fold or language evaluation. Moreover, neural modeling approaches typically suffer from out-of-vocabulary issues, reducing the modeling performance, and implying that there will be cases where the produced token is not correct (even within the top ranks). This limitation could be overcome by expanding the vocabulary size by utilizing enhanced techniques or by using a local cache. We consider these confines and aim to deliver enhanced methodology in the future.

The threat to construct validity is the utilization of different sets for training, validation, and testing which might produce dissimilar results. Additionally, the suitability of the metrics that are used for evaluation raises a threat in this regard. Preferably, we would manually evaluate the results, but the manual evaluation is very difficult to scale. Thus, we have adopted the measures used by previous works (White et al., 2015; Hindle et al., 2012; Santos, 2018; Alon et al., 2019) that includes *Accuracy@k*, *MRR@k*, *Precision*, *Recall*, and F_1 to validate the effectiveness and modeling performance.

An important threat concerning internal validity is the choice of encoding method, model parameters, and architecture design. To address this, we have chosen well-studied parameters (White et al., 2015; Santos, 2018; Kim et al., 2021) for our models and utilize similar parameters across all models, partly for fair comparison and partly due to our intention of presenting a study that is generalized for researchers and could fit into a single GPU. To further address this issue, we have performed an exploratory study, experimenting with different model architectures and their impact on modeling performance. Modeling performance could be further enhanced with a more suitable encoding strategy and design choices that could be explored, but we omit that because training costs are prohibitively large.

Although our work is evaluated extensively with two programming language datasets and on two source code modeling tasks still, it may be affected by the threat to external validity related to its generalization. It is unknown how the results of the presented approach will generalize for other projects of studied languages or other programming languages and tasks. Another threat in this regard is the adoption of *Pytorch* and *Transformer* libraries for the preparation of model training and building. These libraries are well maintained and widely adopted by the research community (Ciniselli et al., 2021b; Liu et al., 2019a) but still different experimental environments may impact the re-producibility and generalization.

8. Related work

In this section, we detailed the literature that is related to the task of source code suggestion. Specifically, we are interested in approaches that are aimed at completing the next code token. Due to lack of space, we omit approaches that are related to other source code modeling tasks such as code readability classification (Mi et al., 2018a,b), API recommendation (Nguyen et al., 2016; Zhou et al., 2021; Svyatkovskiy et al., 2021; Chen et al., 2021), code and comment generation (Zhou et al., 2019; Hu et al., 2020; Svyatkovskiy et al., 2020), bug localization (Huo and Li, 2017; Harer et al., 2018; Xiao et al., 2018; Bader et al., 2019) and other related studies (Bajracharya et al., 2006; McMillan et al., 2011).

Hindle et al. (2012) established that statistical language models could be applied to software, employing n-gram language models that provide predictions for the next token, provided a statement. Further, they shed light on the "naturalness of source code". Nguyen et al. (2012) proposed an approach named GraPacc to provide context-sensitive code completion for API usage patterns. A similar approach was later proposed by Niu et al. (2017) for API completion in Android: Given an API method as a query, their approach recommends a set of relevant API usage patterns. Raychev et al. (2014) presented an approach by employing statistical language models that are trained on a large codebase to predict API calls. Tu et al. (2014) have proposed an approach based on the cache component combined with an n-gram language model that takes advantage of the "localness of code". Their enhanced approach shows that localized information can help to improve the modeling performance. Similarly, an Eclipse autocompletion plugin named CACHECA was presented by Franks et al. (2015) that makes use of the cache language model (Tu et al., 2014), exhibiting superior performance compared to the Eclipse builtin plugin.

Raychev et al. (2014) have proposed a method to synthesize source code by using RNN and n-gram language models to provide completions for a program. Assuming a source code program with holes, their approach provides completion for the holes with the probable method calls. Similarly, White et al. (2015) have proposed a neural language-based modeling approach to provide the source code completions by using the RNN language model and have shown that it can outperform the cache-based n-gram language models. Moreover, they have conducted detailed experiments to study the impact of the size and the context on the modeling performance. Liu et al. (2016) have utilized neural language techniques on a large corpus of JavaScript (dynamically typed language) programming language for code completion. Moreover, they utilized different neural language models with the token level information along with structural information and evaluated their performance. Hellendoorn and Devanbu (Xiao et al., 2018) proposed further improvements to the cached models aimed at considering specific characteristics of code (e.g., unlimited, nested, and scoped vocabulary) exhibiting superior performance and questioning *"Are deep neural networks the best choice for modeling source code?"*. Later, Karampatsis and Sutton (2019) exhibited and suggested *"Maybe deep neural networks are the best choice for modeling source code"*. Sui et al. (2020) proposed Flow2Vec, a different code embedding approach to conserving intraprocedural program dependence by approximating value-flows with asymmetric transitivity. They extensively evaluated their approach for the tasks of code summarization and code classification using open-source C/C++ projects. Wan et al. (2019) proposed a Multi-Modal Attention Network (MMAN) for semantic source code retrieval. They have developed a methodology to represent the structure and unstructured features of source code and have shown that their approach can accurately retrieve code snippets and outperform state-of-the-art methods. In contrast to these two works, our target task is to help predict the next possible code token that is providing source code suggestions.

Transformer-based language models have recently been adopted for various software engineering tasks. Here, we mainly discuss studies that adopted transformers and are related to the task of source code suggestion. We choose to omit studies that are related to program-language understanding and generation (Ahmad et al., 2021; Ding et al., 2021; Kanade et al., 2020) and other related studies such as CodeBERT (Feng et al., 2020) that is used for code document generation and code search, GraphCodeBERT (Guo et al., 2020) that is used for code search, clone detection, code translation and code refinement. Alon et al. (2020) proposed a methodology based on LSTMs and Transformers that utilizes the syntax trees to complete a partial statement. Svyatkovskiy et al. (2020) presented "IntelliCode Compose, a general-purpose multilingual code completion tool" achieving a perplexity of 1.82 that can generate an entire statement. Liu et al. (2020) have proposed an architecture pre-trained to incorporate both code understanding and generation tasks. Another recent work proposed by Nijkamp et al. (2022) introduces a variant of the transformer-based model named CodeGen for the task of program synthesis. Kim et al. (2021) proposed an architecture that utilizes the code AST to capture the syntactic structure information for code completion. Indeed, all these studies pushed the capabilities of the source code suggestion task but still, there is a substantial margin for improvement. In this work, we aim to explore the capabilities of traditional (RNNs) and evolved (Transformers) neural language models in the context of source code suggestion. We partly focus on the learning procedure (i.e., encoding and self-supervised learning) adopted for model training and partly on the adoption of model architectures, ultimately aiming to study their impact on modeling performance and enhancing the modeling precision.

9. Conclusion

In this work, we have explored the transformer language models for the task of source code suggestion to enhance the modeling performance. We have proposed a self-supervised Transformer Gated Highway approach that outperforms traditional recurrent and transformer-based language models of comparable size. Further, we have shown the generalization of our approach by employing it for a different source code modeling task for syntax error correction. The core intentions of this work were to explore how different learning procedures and model architectures could impact and aid in enhancing the modeling performance. We have also experimented with different settings to study the impact of model design and data size on modeling performance. The presented approach is evaluated extensively with two programming language datasets: *Java* and *C#*. The results have suggested that on average proposed approach achieves around 90% accuracy for the source code suggestion task within the top-10 ranks and 95% for correction syntax token prediction within the top-3 ranks for *Java*.

CRediT authorship contribution statement

Yasir Hussain: Conceptualization, Methodology, Investigation, Writing – original draft, Visualization, Funding acquisition. **Zhiqiu Huang:** Supervision, Administration, Resources, Funding acquisition. **Yu Zhou:** Supervision, Validation, Reviewing. **Senzhang Wang:** Validation, Reviewing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Links to related materials are provided in appropriate places.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62150410441), the Natural Science Foundation of Jiangsu Province (No. BK20201292), National Natural Science Foundation of China (No. 61972197, No. 61802179), and Joint Funds of the National Natural Science Foundation of China (No. U2241216).

References

Agarap, A.F., 2018. Deep learning using rectified linear units (relu). arXiv preprint arXiv:1803.08375.

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.-W., 2021. Unified pre-training for program understanding and generation. arXiv preprint arXiv:2103.06333.

Allamanis, M., 2019. The adverse effects of code duplication in machine learning models of code. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 143–153.

Alon, U., Sadaka, R., Levy, O., Yahav, E., 2020. Structural language models of code. pp. 245–256, URL https://proceedings.mlr.press/v119/alon20a.html.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2Vec: Learning distributed representations of code. In: Proceedings of the ACM on Programming Languages, Vol. 3. ACM, pp. 1–29. http://dx.doi.org/10.1145/3290353, POPL. arXiv:arXiv:1803.09473v5.

Bader, J., Scott, A., Pradel, M., Chandra, S., 2019. Getafix: Learning to fix bugs automatically. In: Proceedings of the ACM on Programming Languages, Vol. 3. ACM New York, NY, USA, pp. 1–27, OOPSLA.

Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C., 2006. Sourcerer: a search engine for open source code supporting structure-based search. In: Companion To the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications. pp. 681–682.

Chen, C., Peng, X., Xing, Z., Sun, J., Wang, X., Zhao, Y., Zhao, W., 2021. Holistic combination of structural and textual code information for context based API recommendation. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2021.3074309.

Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Di Penta, M., Bavota, G., 2021a. An empirical study on the usage of BERT models for code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, pp. 108–119.

Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Di Penta, M., Bavota, G., 2021. An empirical study on the usage of BERT models for code completion. XX(X), 108–119. http://dx.doi.org/10.1109/msr52588.2021.00024. arXiv:2103.07115.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Ding, Y., Buratti, L., Pujar, S., Morari, A., Ray, B., Chakraborty, S., 2021. Contrastive learning for source code with structural and functional properties. arXiv preprint arXiv:2110.03868.

Elman, J.L., 1990. Finding structure in time. Cogn. Sci. 14 (2), 179–211.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Franks, C., Tu, Z., Devanbu, P., Hellendoorn, V., 2015. Cacheca: A cache language model based code suggestion tool. In: Proceedings of the 37th International Conference on Software Engineering-Volume 2. IEEE Press, pp. 705–708.

Freitag, M., Al-Onaizan, Y., 2017. Beam search strategies for neural machine translation. arXiv preprint arXiv:1702.01806.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., McConley, M.W., Opper, J.M., Chin, P., Lazovich, T., 2018. Automated software vulnerability detection with machine learning. arXiv:1803.04497. URL http://arxiv.org/abs/1803.04497.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778.

Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 837–847.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2020. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 25 (3), 2179–2217. http://dx.doi.org/10.1007/s10664-019-09730-9.

Huo, X., Li, M., 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: IJCAI International Joint Conference on Artificial Intelligence. pp. 1909–1915.

Hussain, Y., Huang, Z., Zhou, Y., 2021. Improving source code suggestion with code embedding and enhanced convolutional long short-term memory. IET Softw. 15 (3), 199–213. http://dx.doi.org/10.1049/sfw2.12017.

Hussain, Y., Huang, Z., Zhou, Y., Wang, S., 2020. CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. Inf. Softw. Technol. 125, 106309. http://dx.doi.org/10.1016/j.infsof.2020.106309, arXiv:1903.00884.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and evaluating contextual embedding of source code. In: International Conference on Machine Learning. PMLR, pp. 5110–5121.

Karampatsis, R.-M., Sutton, C., 2019. Maybe deep neural networks are the best choice for modeling source code. arXiv preprint arXiv:1903.05734.

Kim, S., Zhao, J., Tian, Y., Chandra, S., 2021. Code prediction by feeding trees to transformers. 1, 150–162. http://dx.doi.org/10.1109/icse43902.2021.00026. arXiv:2003.13848.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Koehn, P., 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In: Conference of the Association for Machine Translation in the Americas. Springer, pp. 115–124.

Liu, F., Li, G., Zhao, Y., Jin, Z., 2020. Multi-task learning based pre-trained language model for code completion. In: Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020. Institute of Electrical and Electronics Engineers Inc., pp. 473–485. http://dx.doi.org/10.1145/3324884.3416591.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019a. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692. URL http://arxiv.org/abs/1907.11692.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019b. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Liu, C., Wang, X., Shin, R., Gonzalez, J.E., Song, D., 2016. Neural code completion.

Loshchilov, I., Hutter, F., 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101.

McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., Xie, Q., 2011. Exemplar: A source code search engine for finding highly relevant applications. IEEE Trans. Softw. Eng. 38 (5), 1069–1087.

Mi, Q., Keung, J., Xiao, Y., Mensah, S., Gao, Y., 2018a. Improving code readability classification using convolutional neural networks. Inf. Softw. Technol. 104 (November 2017), 60–71. http://dx.doi.org/10.1016/j.infsof.2018.07.006.

Mi, Q., Keung, J., Xiao, Y., Mensah, S., Mei, X., 2018b. An inception architecture-based model for improving code readability classification. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. pp. 139–144.

Misra, D., 2019. Mish: A self regularized non-monotonic neural activation function. 4, 2. arXiv preprint arXiv:1908.08681.

Nguyen, A.T., Hilton, M., Codoban, M., Nguyen, H.A., Mast, L., Rademacher, E., Nguyen, T.N., Dig, D., 2016. API code recommendation using statistical learning from fine-grained changes. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016. pp. 511–522. http://dx.doi.org/10.1145/2950290.2950333, URL http://dl.acm.org/citation.cfm?doid=2950290.2950333.

Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J., Nguyen, T.N., 2012. Graph-based pattern-oriented, context-sensitive source code completion. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 69–79.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C., 2022. A conversational paradigm for program synthesis. arXiv preprint arXiv:2203.13474.

Niu, H., Keivanloo, I., Zou, Y., 2017. API usage pattern recommendation for software development. J. Syst. Softw. 129, 127–139.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683.

Raychev, V., Vechev, M., Eth, Z., Yahav, E., 2014. Code completion with statistical language models. In: Acm Sigplan Notices, Vol. 49. ACM, pp. 419–428.

Santos, e.a., 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In: 25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings, Vol. 2018-March. IEEE, pp. 311–322. http://dx.doi.org/10.1109/SANER.2018.8330219.

Sennrich, R., Haddow, B., Birch, A., 2015. Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909.

Srivastava, R.K., Greff, K., Schmidhuber, J., 2015. Highway networks. arXiv preprint arXiv:1505.00387.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.

Sui, Y., Cheng, X., Zhang, G., Wang, H., 2020. Flow2vec: Value-flow-based precise code embedding. In: Proceedings of the ACM on Programming Languages, Vol. 4. ACM New York, NY, USA, pp. 1–27, OOPSLA.

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. IntelliCode compose: Code generation using transformer. In: ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, Inc, pp. 1433–1443. http://dx.doi.org/10.1145/3368089.3417058, arXiv:2005.08025.

Svyatkovskiy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J.V., Allamanis, M., 2021. Fast and memory-efficient neural code completion. In: Proceedings - 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021. Institute of Electrical and Electronics Engineers Inc., pp. 329–340. http://dx.doi.org/10.1109/MSR52588.2021.00045.

Tu, Z., Su, Z., Devanbu, P., 2014. On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014. ACM, pp. 269–280. http://dx.doi.org/10.1145/2635868.2635875, URL http://dl.acm.org/citation.cfm?doid=2635868.2635875.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems. pp. 5998–6008.

Wan, Y., Shu, J., Sui, Y., Xu, G., Zhao, Z., Wu, J., Yu, P., 2019. Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 13–25.

White, M., Vendome, C., Linares-Vasquez, M., Poshyvanyk, D., Linares-Vásquez, M., Poshyvanyk, D., 2015. Toward deep learning software repositories. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE Press, pp. 334–345. http://dx.doi.org/10.1109/MSR.2015.38, arXiv:arXiv:1404.7828v1, URL http://ieeexplore.ieee.org/document/7180092/.

Xiao, Y., Keung, J., Bennin, K.E., Mi, Q., 2018. Machine translation-based bug localization technique for bridging lexical gap. Inf. Softw. Technol. 99 (August 2017), 58–61. http://dx.doi.org/10.1016/j.infsof.2018.03.003.

Yang, Y., 2019. Improve language modelling for code completion through statement level language model based on statement embedding generated by bilstm. arXiv preprint arXiv:1909.11503.

Zhou, Y., Yan, X., Yang, W., Chen, T., Huang, Z., 2019. Augmenting java method comments generation with context information based on neural networks. J. Syst. Softw. 156, 328–340. http://dx.doi.org/10.1016/j.jss.2019.07.087.

Zhou, Y., Yang, X., Chen, T., Huang, Z., Ma, X., Gall, H.C., 2021. Boosting API recommendation with implicit feedback. IEEE Trans. Softw. Eng..

Yasir Hussain received his B.Sc. degree from Bahauddin Zakariya University (BZU), Pakistan, in 2013 and the Master's degree in computer science from the Virtual University of Pakistan in 2015. He received his Ph.D. degree in Computer Science from Nanjing University of Aeronautics and Astronautics (China) in 2020. Currently, he is conducting his PostDoc research in software engineering at Nanjing University of Aeronautics and Astronautics (China). He is particularly interested in software engineering, source code modeling, machine learning, deep learning, recommendation systems, and predictive modeling.

Huang Zhiqiu is a full professor at Nanjing University of Aeronautics and Astronautics. He received his BSc. and M.Sc. degrees in computer science from the National University of Defense Technology of China. He received his Ph.D. degree in computer science from Nanjing University of Aeronautics and Astronautics of China. His research interests include big data analysis, cloud computing, and web services.

Zhou YU is a full professor at Nanjing University of Aeronautics and Astronautics. He received his B.Sc. degree in 2004 and his Ph.D. degree in 2009, both in Computer Science from Nanjing University China, and conducted his PostDoc research in software engineering at Politechnico di Milano, Italy. In 2015–2016, he visited the SEAL lab at the University of Zurich Switzerland, where he was also an adjunct researcher. His research interests mainly include software evolution analysis, mining software repositories, software architecture, and reliability analysis.

Senzhang Wang is a full professor at the College of Computer Science and Technology, Central South University. He received his B.Sc. degree from Southeast University, Nanjing, China, in 2009, and his Ph.D. degree from Beihang University, Beijing, China, in 2016. His main research interests include data mining, social computing, and urban computing. He has published more than 80 referred conference and journal papers.