1-2019

Deep code comment generation with hybrid lexical and syntactical information

Xing HU
Peking University

Ge LI
Peking University

Xin XIA
Monash University

David LO
Singapore Management University, davidlo@smu.edu.sg

Zhi JIN
Peking University

Citation

HU, Xing; LI, Ge; XIA, Xin; LO, David; and JIN, Zhi. Deep code comment generation with hybrid lexical and syntactical information. (2019). *Empirical Software Engineering*. 25, (3), 2179-2217.
Available at: https://ink.library.smu.edu.sg/sis_research/4407

Published in Empirical Software Engineering,
2019
DOI 10.1007/s10664-019-09730-9

Deep code comment generation with hybrid lexical and syntactical information

Xing Hu[1,2] · Ge Li[1,2] · Xin Xia[3] · David Lo[4] · Zhi Jin[1,2]

Abstract

During software maintenance, developers spend a lot of time understanding the source code. Existing studies show that code comments help developers comprehend programs and reduce additional time spent on reading and navigating source code. Unfortunately, these comments are often mismatched, missing or outdated in software projects. Developers have to infer the functionality from the source code. This paper proposes a new approach named Hybrid-DeepCom to automatically generate code comments for the functional units of Java language, namely, Java methods. The generated comments aim to help developers understand the functionality of Java methods. Hybrid-DeepCom applies Natural Language Processing (NLP) techniques to learn from a large code corpus and generates comments from learned features. It formulates the comment generation task as the machine translation problem. Hybrid-DeepCom exploits a deep neural network that combines the lexical and structure information of Java methods for better comments generation. We conduct experiments on a large-scale Java corpus built from 9,714 open source projects on GitHub. We evaluate the experimental results on both machine translation metrics and information retrieval metrics. Experimental results demonstrate that our method Hybrid-DeepCom outperforms the state-of-the-art by a substantial margin. In addition, we evaluate the influence of out-of-vocabulary tokens on comment generation. The results show that reducing the out-of-vocabulary tokens improves the accuracy effectively.

Keywords Program comprehension · Comment generation · Deep learning

1 Introduction

During software development and maintenance, developers spend around 59% of their time on program comprehension activities (Xia et al. 2017). Previous studies have shown that

Communicated by: Chanchal Roy, Janet Siegmund, and David Lo

✉ Ge Li
lige@pku.edu.cn

✉ Zhi Jin
zhijin@pku.edu.cn

Extended author information available on the last page of the article.

good comments are important to program comprehension, since developers can understand the meaning of a piece of code by reading the natural language description of the comments (Sridhara et al. 2010). Unfortunately, due to tight project schedule and other reasons, code comments are often mismatched, missing or outdated in many projects. Automatic generation of code comments can not only save developers' time in writing comments, but also help in source code understanding.

Many approaches have been proposed to generate comments for methods (Sridhara et al. 2010; McBurney and McMillan 2014) and classes (Moreno et al. 2013) of Java, which is the most popular programming language in the past 10 years.[1] Their techniques vary from the use of manually-crafted (Moreno et al. 2013) to Information Retrieval (IR) (Haiduc et al. 2010a, b). Moreno et al. (2013) defined heuristics and stereotypes to synthesize comments for Java classes. These heuristics and stereotypes are used to select information that will be included in the comment. Haiduc et al. (2010a, b) applied IR approaches to generate summaries for classes and methods. IR approaches such as Vector Space Model (VSM) and Latent Semantic Indexing (LSI) usually search comments from similar code snippets. Although promising, these techniques have two main limitations: first, they fail to extract accurate keywords used to identify similar code snippets when identifiers and methods are poorly named. Second, they rely on whether similar code snippets can be retrieved and how similar the snippets are.

Recently, there is an emerging interest in building probabilistic models for large-scale source code. Hindle et al. (2012) have addressed the naturalness of software and demonstrated that code can be modeled by probabilistic models. Several subsequent studies have developed various probabilistic models for different software tasks (Gu et al. 2016; Loyola et al. 2017; Wang et al. 2016; White et al. 2016). When applied to code summarization, different from IR-based approaches, existing probabilistic-model-based approaches usually generate comments directly from code instead of synthesizing them from keywords. One of such probabilistic-model-based approaches is by (Iyer et al. 2016) who proposed an attention-based Recurrent Neural Network (RNN) model called CODE-NN. CODE-NN built a language model for natural language comments and aligns the words in comments with individual code tokens directly by the attention component. CODE-NN recommended code comments given source code snippets extracted from Stack Overflow. Experimental results demonstrated the effectiveness of probabilistic models on code summarization. These studies provided principled methods for probabilistically modeling and resolving ambiguities both in natural language descriptions and in the source code.

In this paper, to better utilize the advantage of deep learning techniques, we propose a new approach Hybrid-DeepCom to generate descriptive comments for Java methods which are functional units of Java language. Hybrid-DeepCom builds upon advances in Neural Machine Translation (NMT). NMT is normally used to automatically translate from one language (e.g., Chinese) to another language (e.g., English), and it has been shown to achieve great success for natural language corpora (Bahdanau et al. 2014; Sutskever et al. 2014). Intuitively, generating comments can be considered as a variant of the NMT problem, where source code written in a programming language needs to be translated to text in natural language. Different from CODE-NN which only built a language model for comments, the NMT model builds language models for both source code and comments. The words in comments align with the RNN hidden states which involve the semantics of code tokens. Hybrid-DeepCom generates comments by automatically learning from features (e.g., identifier names, formatting, semantics, and syntax features) extracted from a large-scale Java corpus.

[1] https://www.tiobe.com/tiobe-index/

Compared to traditional machine translation, our task is more challenging since:

1. **Source code is structured:** In contrast to natural language text which is weakly structured, programming languages are formal languages and source code written in them are unambiguous and structured (Allamanis et al. 2017). Many probabilistic models used in NMT are sequence-based models that need to be adapted to structured code analysis. The main challenge and opportunity is how to take advantage of rich and unambiguous structure information of source code to boost effectiveness of existing NMT techniques.
2. **Vocabulary:** In natural language (NL) corpora used for NMT, the vocabulary is usually limited to the most common words, e.g., 30,000 words, and words outside the vocabulary are treated as unknown words – often marked as ⟨*UNK*⟩. It is effective for such NL corpora because words outside the dominant vocabulary are rare. In code corpora, the vocabulary consists of keywords, operators, and identifiers. It is common for developers to define various new identifiers, and thus they tend to proliferate. In our dataset, we get 794,711 unique tokens after replacing numerals and strings with generic tokens ⟨*NUM*⟩ and ⟨*STR*⟩. In a codebase used to build probabilistic models, there are likely to be many out-of-vocabulary identifiers. As Table 1 illustrates, there are 794,621 unique identifiers in our dataset. If we use most common 30,000 tokens as the code vocabulary, about 95 % identifiers will be regarded as ⟨*UNK*⟩. Hellendoorn and Devanbu (2017) have demonstrated that it is unreasonable for source code to use such a vocabulary.

To address these issues, we propose a new approach Hybrid-DeepCom that customizes a sequence-based language model to analyze the source code and Abstract Syntax Trees (AST) at the same time. It learns both the lexical and syntactic information from the source code and the AST respectively. The ASTs are converted into sequences before they are fed into Hybrid-DeepCom. It is generally accepted that a tree cannot be restored from a sequence generated by classical traversal methods such as pre-order traversal and post-order traversal. To better present the structure of ASTs, and keep the sequences unambiguous, Hybrid-DeepCom designs a new structure-based traversal (SBT) method to traverse ASTs. Using SBT, a subtree under a given node is included into a pair of brackets. The brackets represent the structure of the AST and we can restore a tree unambiguously from a sequence generated using SBT. In addition, we leverage a hybrid attention component to fuse the lexical and syntactic information.

Moreover, to address the vocabulary challenge, we split the identifiers into multiple subtokens. Most identifiers consist of multiple words according to the camel naming convention, e.g., getIndex→ {get, index}. These words usually represent the functionality of the methods or variables. Hybrid-DeepCom generates comments word-by-word from both source code and AST sequences. We train and evaluate Hybrid-DeepCom on the Java dataset that consists of 9,714 Java projects from GitHub.

This paper extends our preliminary study which appears as a research paper in ICPC 2018 (Hu et al. 2018a). In particular, we extend our preliminary work in the following direction:

Table 1 Statistics for code tokens in our dataset

#Methods	#All Tokens	# All Identifiers	# Unique Tokens	#Unique Identifiers
588,108	44,378,497	13,779,297	794,711	794,621

1. We propose Hybrid-DeepCom that is an extended version of DeepCom proposed in our preliminary work (Hu et al. 2018a). There are three major differences between Hybrid-DeepCom and DeepCom: (1) In DeepCom, we directly generate comments from the traversed AST sequences. In Hybrid-DeepCom, we combine the source code and the traversed AST sequences together to generate the comments. (2) In DeepCom, we use the node "type" to represent the out-of-vocabulary tokens. In Hybrid-DeepCom, we split the identifiers into multiple words according to the camel casing naming convention. (3) In DeepCom, the comments are generated word by word, while in Hybrid-DeepCom, we leverage the beam search (Koehn 2004) while generating the code comments. The motivation of these modifications are discussed in Section 3. Our experiments show that Hybrid-DeepCom outperforms the baselines including CODE-NN and DeepCom.
2. We strengthen the experiments by adding more evaluation metrics, including corpus-level BLEU score, METEOR, precision, recall, F-score, and F-mean. In addition, we leverage the smoothing techniques to better compute the sentence-level BLEU score.
3. We further discuss how performance differs considering varying code lengths and comment lengths on different metrics.
4. We illustrate the influence of the out-of-vocabulary tokens of source code on comment generation. We also evaluate the ability to ease the out-of-vocabulary problem by splitting identifiers into subtokens.
5. We conduct the 10-fold-cross-validation to evaluate the generalization ability of our trained model on new projects.
6. Moreover, we conduct a human evaluation to evaluate the quality of the automatically generated code comments.

Our contributions, which form a super-set of those in our preliminary study, are as follows:

- We formulate code comments generation task as a machine translation task.
- We customize a sequence-based model to learn the lexical and the structural information at the same time to generate comments for Java methods. In particular, we propose a new AST traversal method (namely structure-based traversal) to represent the structure information better.
- We leverage a simple but effective approach to reduce the out-of-vocabulary tokens in the source code.

Paper Organization The remainder of this paper is organized as follows. Section 2 presents background materials on language models and NMT. Section 3 elaborates on the details of Hybrid-DeepCom. Section 4 presents the experiment setup and results. Section 5 presents the human evaluation of Hybrid-DeepCom. Section 6 discusses strengths of Hybrid-DeepCom, and threats to validity. Section 7 surveys the related work. Finally, Section 8 concludes the paper and points out potential future directions.

2 Background

2.1 Language Models

Our work is inspired by the machine translation problem in the Natural Language Processing (NLP) field. We exploit the language models learning from a large-scale source code corpus. The models generate code comments from the learned features. Language models learn the probabilistic distribution over sequences of words. They work tremendously well on a large variety of problem (e.g., machine translation (Bahdanau et al. 2014), speech recognition (Chelba et al. 2012), and question answering (Yin et al. 2015)).

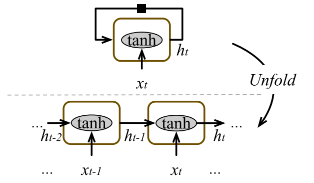
For a sequence $x = (x_1, x_2, ..., x_n)$ (e.g., a statement), the language model aims to estimate the probability of it. The probability of a sequence is computed via each of its tokens. That is,

$$P(x) = P(x_1)P(x2|x_1)...P(x_n|x_1...x_{n-1}) \tag{1}$$

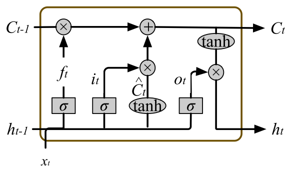
In this paper, we adopt a language model based on the deep neural network called Gated Recurrent Unit (GRU) (Cho et al. 2014). GRU is one of the state-of-the-art RNNs and is a variant of Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997). GRU outperforms general RNN because it is capable of learning long-term dependencies. It is a natural model to use for source code which has long dependencies (e.g., a class is used far away from its import statement). The details of RNN, LSTM and GRU are shown in Fig. 1.

2.1.1 Recurrent Neural Networks

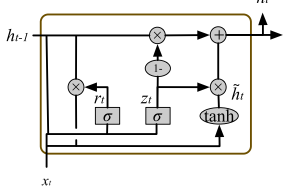
RNNs are intimately related to sequences and lists because of their chain-like natures. It can in principle map from the entire history of previous inputs to each output. At each time step t, the unit in the RNN takes not only the input of the current step but also the hidden state outputted by its previous time step $t-1$. As Fig. 1a illustrates, the hidden state of time step t is updated according to the input vector x_t and its previous hidden state h_{t-1}, namely, $h_t = \tanh(Wx_t + Uh_{t-1} + b)$ where W, U, and b are the trainable parameters which are updated while training, and tanh is the activation function: $\tanh(z) = (e^z - e^{-z})/(e^z + e^{(-z)})$. Generally, these parameters are tuned by back-propagation technique. Back-propagation is widely used in neural networks to find the gradient (partial derivatives) of the error with respect to the parameters. Those derivatives are then used by the gradient descent algorithm to adjust the parameters to decrease the error. A prominent drawback of the standard RNN model is that gradients may explode or vanish during the back-propagation. Exploding gradients are a problem where large error gradients accumulate and result in very large updates to neural network model weights during training. On the other hand, if the gradient have small values, they shrink exponentially until they vanish and make it impossible for the model to learn, this is the vanishing gradient problem. These issues arise during training of the RNN when the gradients are being propagated back in time all the way to the initial layer. Because the layers and time steps of deep neural networks relate to each other through multiplication, derivatives are susceptible to vanishing or exploding. Exploding gradients



(a) Standard RNN model and its unfolded architecture through time step



(b) The LSTM unit



(c) The GRU unit

Fig. 1 An illustration of basic RNN, LSTM and GRU

cause every weights' gradients become saturated on the high end; Vanishing gradients can become too small for computers to work with or for networks to learn. These phenomena often appear when long dependencies exist in the sequences. To address these problems, some researchers have proposed several variants to preserve long-term dependencies. These variants include LSTM and Gated Recurrent Unit (GRU). In this paper, we adopt the GRU which has achieved success on many NLP tasks (Cho et al. 2014).

2.1.2 Long Short-Term Memory

LSTM introduces a structure called the *memory cell* to solve the problem that ordinary RNN is difficult to learn long-term dependencies in the data. The LSTM is trained to selectively "forget" information from the hidden states, thus allowing room to take in more important information (Hochreiter and Schmidhuber 1997). LSTM introduces a gating mechanism to control when and how to read previous information from the memory cell and write new information. The memory cell vector in the recurrent unit preserves long-term dependencies. In this way, LSTM handles long-term dependencies more effectively than vanilla RNN. LSTM has been widely used to solve semantically related tasks and has achieved convincing performance. Figure 1b illustrates a typical LSTM unit and for more details of LSTM, please refer to (Hochreiter and Schmidhuber 1997; Chung et al. 2014).

2.1.3 Gated Recurrent Unit

GRU (Cho et al. 2014) is another popular RNN that aims to solve the vanishing gradient problem. GRU can be considered as a variation of LSTM because both are designed similarly and, in some cases, produce equally excellent results. It also uses the gating mechanism to learn long-term dependencies. The key difference between a GRU and an LSTM is that a GRU has two gates (reset and update gates) whereas an LSTM has three gates (namely input, output and forget gates). Compared to LSTM, GRU is much simpler to compute and implement. The reset gate allows the hidden state to drop information that is found to be irrelevant in the future. The update gate controls how much information from the previous hidden state will carry over to the current hidden state. The details of the GRU is illustrated as Fig. 1c. It is similar to the memory cell in LSTM and helps RNN to capture the long-term information. As GRU does not need memory units, it is easier and faster to train. These advantages motivate us to exploit GRU for building models for source code and comments.

2.2 Neural Machine Translation

NMT (Wu et al. 2016) is an end-to-end learning approach for automated translation. It is a deep learning based approach and has made rapid progress in recent years. NMT has shown impressive results surpassing those of phrase-based systems while addressing shortcomings such as the need for hand engineered features. It typically consists of two RNNs, one to consume the input text sequences and the other one to generate the translated output sequences. It is often accompanied by an attention mechanism that aligns target with source tokens (Bahdanau et al. 2014).

NMT bridges the gap between different natural languages. Generating comments from the source code is a variant of machine translation problem between the source code and the natural language. We explore whether the NMT approach can be applied to comments generation. In this paper, we follow the common Sequence-to-Sequence (Seq2Seq) (Sutskever et al. 2014) learning framework with attention (Bahdanau et al. 2014) which helps cope effectively with the long source code.

2.3 Sequence-to-Sequence Model

Sequence-to-Sequence (Seq2Seq) model consists of two RNNs, namely, the encoder and the decoder. Let $X = x_1, x_2, ..., x_m$ denote a sequence of source code snippet, $Y = y_1, y_2, ..., y_l$ denote a sequence of generated words. When translating the code snippet X into natural language description Y, the encoder transforms the code snippet X into a set of hidden states $(s_1, s_2, ..., s_m)$ with an RNN, while the decoder uses another RNN to generate one word y_{t+1} at a time in the target sequence.

Encoder The encoder is an RNN that has a hidden state, which is a fixed-length vector. At time step t, the encoder computes the hidden state s_t by:

$$s_t = f(x_t, s_{t-1}) \tag{2}$$

where f is the hidden layer which mainly has two main options, i.e., LSTM and GRU.

Decoder The decoder aims to generate the target sequence $Y = y_1, y_2, ..., y_l$. At the time step t, decoder computes the hidden state h_t and the conditional distribution of the next symbol y_{t+1} by:

$$p(y_{t+1}|y_t) = g(h_t, c_t) \tag{3}$$

where g is used to estimate the probability of the word y_t. c_t is the context vector for y_t, computed by:

$$c_t = \sum_{j=1}^{m} \alpha_{t,j} s_j \tag{4}$$

where $\alpha_{t,j}$ is the attention weight of y_t on s_j Bahdanau et al. (2014).

2.4 Out-of-Vocabulary Issue

When building the language models, the available vocabulary must first be defined. However, to learn all words in the given corpus is hard, especially for the rare words. The learning-based models require adequate example size, and they can hardly learn the semantic of the rare words. These words are usually proper nouns, such as people names or city names. The Out-of-Vocabulary (OOV) problem refers to words contained in the input sequences but are not part of the available vocabulary lexicon. One of the most common methods to deal with this OOV issue is to use a specific symbol $\langle UNK \rangle$ to represent these words. This works for NLP tasks, since words outside the dominant vocabulary are so rare. In source code, although each language only has a fixed set of keywords and operators, new identifier names tend to proliferate (Hellendoorn and Devanbu 2017).

3 Proposed Approach

The transition process between source code and comments is similar to the translation process between different natural languages. Existing research has applied machine translation methods translating code from one source language (e.g., Java) to another (e.g., C#) (Gu et al. 2017). A few studies adopt machine translation method for generating natural language descriptions from the source code. Oda et al. (2015) present a machine translation approach to generate natural language Pseudo-code of the source code at the statement level. In this paper, Hybrid-DeepCom translates the source code to a high-level description at the method level. Hybrid-DeepCom combines the source code and the AST sequences to generate the code comments. On one hand, the words in the comments are usually extracted from the lexical information of source code, such as method names and variable names. On the other hand, the syntactical information can be effectively captured from the AST sequences which are composed by the "type" of AST nodes. Hybrid-DeepCom learns both the lexical and syntactical information to generate the comments.

The overall framework of Hybrid-DeepCom is illustrated in Fig. 2. Hybrid-DeepCom consists of three stages: the data processing, the model training, and the online testing. The source code we obtained from GitHub is parsed and preprocessed into a parallel corpus of Java methods and their corresponding comments. In order to learn the structural information, the Java methods are converted into AST sequences by a special traversal approach before input into the model.

With the parallel corpus of the source code, AST sequences and comments, we build and train generative neural models based on the idea of NMT. There are three challenges during training process:

- How to represent ASTs to preserve the structural information and keep the representation unambiguous while traversing the ASTs?
- How to learn both the lexical information and the syntactic information and then fuse them?
- How to reduce the out-of-vocabulary tokens in the source code?

In the following paragraphs, we will introduce the details of the model and the approaches we propose to resolve the above-mentioned challenges.

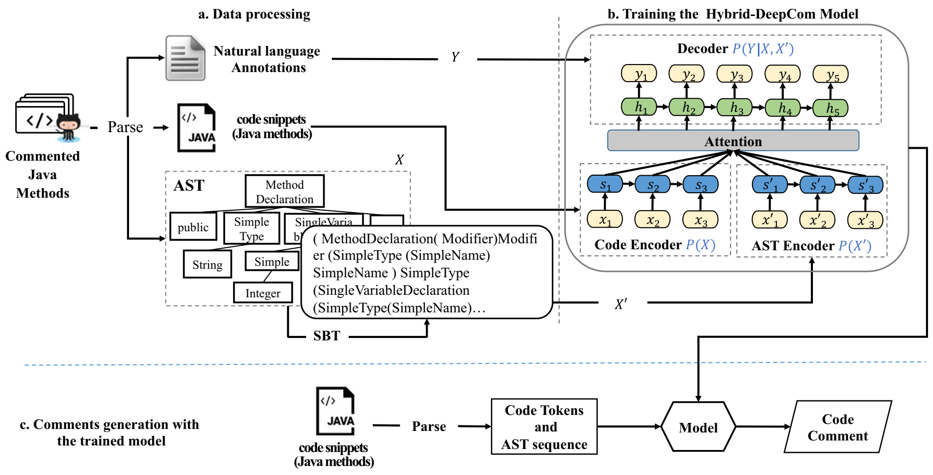

a. Data processing
b. Training the Hybrid-DeepCom Model
Natural language Annotations
Y
Commented Java Methods
Parse
JAVA
code snippets (Java methods)
X
AST
Method Declaration
public
Simple Type
SingleVaria...
String
Simple
Integer
(MethodDeclaration(Modifier)Modifier (SimpleType (SimpleName) SimpleName) SimpleType (SingleVariableDeclaration (SimpleType(SimpleName)…
SBT
X′
Decoder $P(Y|X,X')$
y_1 y_2 y_3 y_4 y_5
h_1 h_2 h_3 h_4 h_5
Attention
s_1 s_2 s_3
s'_1 s'_2 s'_3
x_1 x_2 x_3
x'_1 x'_2 x'_3
Code Encoder $P(X)$
AST Encoder $P(X')$
c. Comments generation with the trained model
code snippets (Java methods)
Parse
Code Tokens and AST sequence
Model
Code Comment

Fig. 2 Overall framework of Hybrid-DeepCom

3.1 The Deep Neural Network for Code Comment Generation

Different from the vanilla Seq2Seq model we have introduced in Section 2, we extend the model by integrating the structure information of source code. It is effective for Seq2Seq model to generate the target sentence from a given sentence. However, the lexical and syntactical information is different modalities of the source code, thus we need two input sequences (the code sequence and structure sequence) that are fed into the model. The vanilla Seq2Seq model has to be adapted to learn the lexical and syntactical information at the same time. Figure 3 illustrates the detailed Hybrid-DeepCom model for code comment generation.

3.1.1 Encoders

Hybrid-DeepCom uses two encoders to encode the source code and AST sequences respectively. One encoder learns the lexical information in the source code and the other encoder learns the structure information in the AST sequences.

Code Encoder The code encoder encodes the tokens of Java method and aims to learn the lexical information within it. The encoder is GRU that we describe in Section 2. For a code snippet $X = x_1, ..., x_m$, the GRU unit maps the code tokens into hidden states. At each time step t, it reads one token x_t of the sequence, then updates and records the current hidden state s_t, namely,

$$s_t = f(x_t, s_{t-1}) \tag{5}$$

where f is a GRU unit that maps a word of source language x_t into a hidden state s_t. The hidden states of the encoded source code are $s = [s_1, ..., s_m]$.

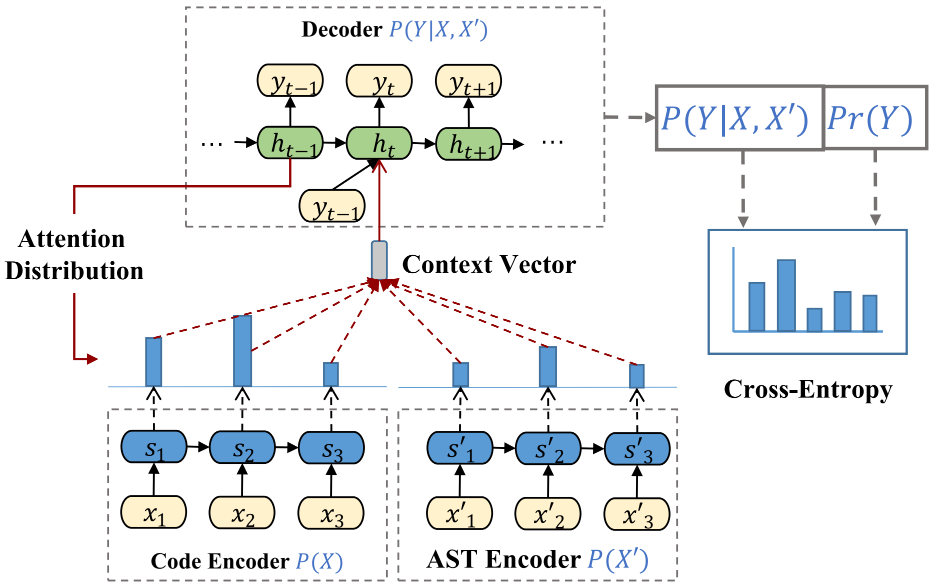


Fig. 3 The detailed Hybrid-DeepCom model

AST Encoder Hybrid-DeepCom uses another encoder to learn the structure information from the AST sequences. For an AST sequence $X' = x'_1, ..., x'_n$ where x'_i represents an AST node, the GRU unit maps an AST node into hidden state. Similarly, at each time step t, it reads an AST node x'_t of the sequence, then updates and records the current hidden state s'_t, namely,

$$s'_t = f'(x'_t, s'_{t-1}) \tag{6}$$

where f' is another GRU unit. At last, the AST sequence is encoded into hidden states $s' = [s'_1, ..., s'_n]$.

The two encoders learn latent features from source code and AST sequences. Then, the features are encoded into the context vector C. These latent features include the identifiers naming conventions, the control structures, and etc. In this paper, Hybrid-DeepCom adopts the attention mechanism to compute the context vector C and fuse the lexical and syntactical information.

3.1.2 Attention

When translating a sentence, people pay attention to the words around the word that is presently translating. Neural networks can achieve the same behavior using the attention mechanism. Attention mechanism is widely used to select the important parts from the input sequence for each target word. It focuses on a subset of the information of the source code tokens. For example, the token "whether" in comments usually aligns with the "if" statements in the source code. The generation of each word is guided by a classic attention method proposed by (Bahdanau et al. 2014). Different from the attention mechanism used in the vanilla Seq2Seq model, Hybrid-DeepCom needs to pay attention to both the source code and the AST sequence. The attention mechanism used in Hybrid-DeepCom needs to project the hidden states of all encoders into a shared space and then compute the distribution over the projections.

It defines individual c_i for predicting each target word y_i as a weighted sum of all hidden states in two encoders and c_i is computed as

$$c_i = \sum_{j=1}^{m} \alpha_{ij} s_j + \sum_{j=1}^{n} \alpha'_{ij} s'_j \tag{7}$$

in which α and α' are attention distributions of the source code and the AST sequences respectively. The weight α_{ij} of each hidden state s_j is computed as

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{ik})} \tag{8}$$

and

$$\alpha'_{ij} = \frac{\exp(e'_{ij})}{\sum_{k=1}^{m} \exp(e'_{ik})} \tag{9}$$

where

$$e_{ij} = a(h_{i-1}, s_j) \tag{10}$$

and

$$e'_{ij} = a(h_{i-1}, s'_j) \tag{11}$$

are alignment models which score how well the inputs around position j and the output at position i match.

3.1.3 Decoder

The Decoder aims to generate the target sequence y by sequentially predicting the probability of a word y_i conditioned on the context vector c_i and its previous generated words $y_1, ..., y_{i-1}$, i.e.,

$$p(y_i|y_1, ..., y_{i-1}, x) = g(y_{i-1}, h_i, c_i) \tag{12}$$

where g is used to estimate the probability of the word y_i. The goal of the model is to minimize the cross-entropy, i.e., minimize the following objective function:

$$H(y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{l} logp(y_j^{(i)}) \tag{13}$$

where N is the total number of training instances, and l is the length of each target sequence. $y_j^{(i)}$ means the jth word in the ith instance. The cross-entropy describes how much the predicted probability diverges from the ground truth. Through optimizing the objective function using optimization algorithms such as gradient descendant, the parameters can be estimated.

3.1.4 Beam Search

Hybrid-DeepCom uses Beam Search (Koehn 2004), a heuristic search strategy, to find comments that have the least cost (computed using (13)). Beam Search expands upon the greedy search and returns a list of most likely output sequences. It searches comment tokens produced at each step one by one. At each time step, it selects k tokens with the least cost, where k is the beam-width. It then prunes off the remaining branches and continues selecting the possible tokens that follow on until it meets the end-of-sequence symbol (i.e., $\langle End \rangle$).

Figure 4 shows an example of a beam search (beam-width=2) for generating comment "creates a new request". First, the token $\langle Start \rangle$ is selected as the first token in the generated comment. Then, it estimates the probabilities of all possible tokens that follow on according to the language model and computes their costs following (13). In this example, it selects

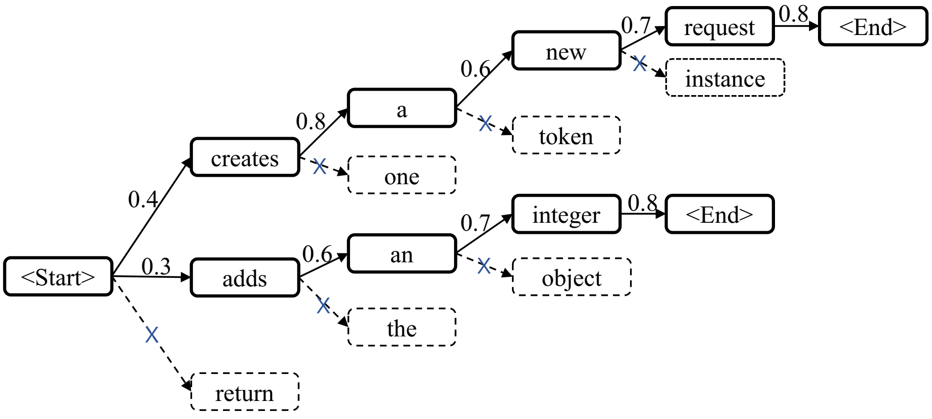


Fig. 4 An illustration of beam search (beam size=2)

the tokens with maximum probabilities, i.e., “creates” and “adds”. Then, it ignores branches of other tokens and continue estimating possible tokens after “creates” and “adds”. For example of the “creates” branch, the sentences that include “creates a ...” are more than those that include “creates one...” in the training corpus. Thus, “a” has a much higher probability that “one” given its preceding the word “creates”. Once it selects an end-of-sequence symbol (i.e., $\langle End \rangle$) as the next token, it stops that branch and the branch is selected as a generated comment. Finally, Hybrid-DeepCom produces k comments for each Java method. We rank the generated comments according to their average probabilities during the beam search procedure. In this paper, we select the Top 1 comment as the final result.

3.2 Abstract Syntax Tree with SBT traversal

Translation between source code and natural language is challenging due to the structure of source code. One simple way to model source code is to just view it as plain text. However, in such way, the structure information will be omitted, which will cause inaccuracies in the generated comments. The structure information can be learned from the AST of the source code. To better learn the structure information by the NMT model, the AST is traversed into sequences. Sequences obtained by classical traversal methods (e.g., pre-order traversal) are lossy since the original ASTs cannot unambiguously be reconstructed back from them. This ambiguity may cause different Java methods (each with different comments) to be mapped to the same sequence representation. It is confusing for the neural network if there are multiple labels (in our setting, comments) given to a particular input. For addressing this problem, we propose a Structure-based Traversal (SBT) method to traverse the AST. The details are presented in Algorithm 1. Figure 5 illustrates a simple example of SBT to traverse a tree and the detailed procedure is as follows:

- From the root node, we first use a pair of brackets to represent the tree structure and put the root node itself behind the right bracket, that is (1)1, shown in Fig. 5.
- Next, we traverse the subtrees of the root node and put all root nodes of subtrees into the brackets, i.e., (1(2)2(3)3)1.
- Recursively, we traverse each subtree until all nodes are traversed and the final sequence (1(2(4)4(5)5(6)6)2(3)3)1 is obtained.

Algorithm 1 Structure-based traversal.

```
1:  procedure SBT($r$)                          ▷ Traverse a tree from root $r$
2:      $seq \leftarrow \varnothing$            ▷ $seq$ is the sequence of a tree after traversal
3:      if $!r.hasChild$ then
4:          $seq \leftarrow (r)r$               ▷ Add brackets for terminal nodes
5:      else
6:          $seq \leftarrow (r$                 ▷ Add left bracket for non-terminal nodes
7:          for $c$ in $childs$ do
8:              $seq \leftarrow seq + SBT(c)$
9:          end for
10:         $seq \leftarrow seq + )r$   ▷ Add right bracket for non-terminal nodes after traversing all
    their children
11:     end if
12:     return $seq$
13: end procedure
```

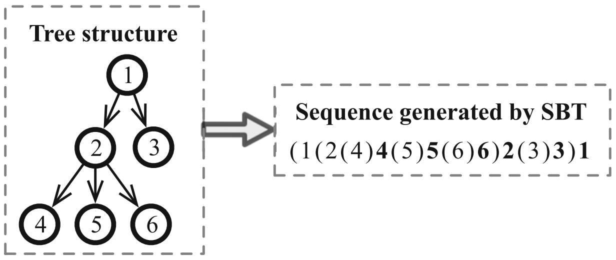


Fig. 5 An example of sequencing an AST to a sequence by SBT. (For a number, the bold font number after bracket indicates node itself and the number in brackets denotes the tree structure by taking it as the root node)

Hybrid-DeepCom processes each AST into a sequence following the SBT algorithm. For example, the AST sequence of the following Java method extracted from project Eclipse Che[2] is shown in Fig. 6:

```
public String extractFor(Integer id){
  LOG.debug("Extracting method with ID:{}", id);
  return requests.remove(id);
}
```

The left part of Fig. 6 is the AST of the method. The non-terminal nodes (those without boxes) illustrate the structural information of source code. They have the feature "type" which is a fixed set (e.g., IfStatement, Block, and ReturnStatement). The terminal nodes (those within boxes) not only have "type" but also have "value" (token within brackets). The "value" is the concrete token occurring in the source code and "type" indicates the type of the token. In this paper, we exploit the source code and its structure representation to generate comments. The structure information is related to the "type" of the nodes. Therefore, we keep the "type" of the nodes while traversing a particular AST. The right part of the figure is the sequenced constructed by traversing the AST. A subtree is included in a pair of brackets and we can restore the AST from the given sequence. In this way, we can keep the structural information and make the representation lossless – the original AST can be unambiguously reconstructed from the sequence.

3.3 Reduce Out-of-Vocabulary Tokens

Vocabulary is another challenge to model source code (Hellendoorn and Devanbu 2017). In natural language, studies usually limit vocabulary to the most common words (e.g., top 30,000) during data processing. The out-of-vocabulary tokens are usually replaced by a special unknown token, e.g., $\langle UNK \rangle$. It is effective for NLP because words outside vocabulary are rare. However, this method is arguably inappropriate when it comes to source code. In addition to fixed operators and keywords, there are user-defined identifiers which take up the majority of code tokens (Broy et al. 2005). These identifiers have a substantial influence on the vocabulary of language models. If we keep a regular vocabulary size for source code, there will be many unknown tokens. If we want the occurrences of $\langle UNK \rangle$ tokens to be as few as possible, the vocabulary size will increase a lot. A large vocabulary size will make it difficult to train a deep learning model since it requires more training data, time, and

[2]https://github.com/eclipse/che

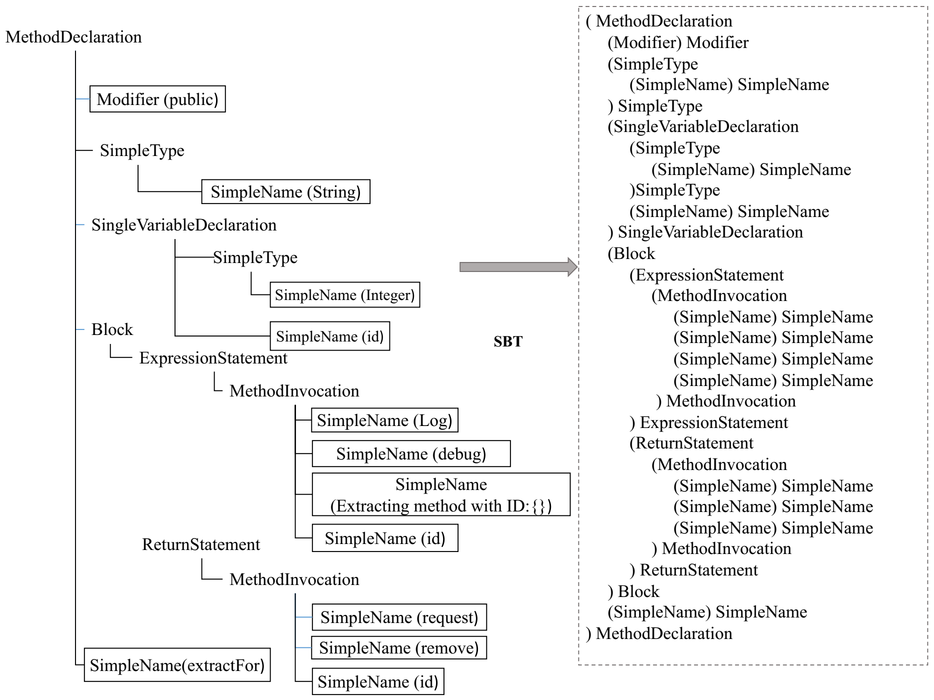


Fig. 6 AST of the Java method named *extractFor*

memory. To achieve optimal and stable results, models need to run a larger number of iterations to tune the parameters for each word in the vocabulary.

After manual observation, we find that most identifiers are composed of several words. These words are usually common words and are used repeatedly. Hence, we split the identifiers into several words to reduce the out-of-vocabulary tokens in the source code. After that, the number of unique tokens in the training set decreases from 542,680 to 47,939.

3.4 Differences between Hybrid-DeepCom and DeepCom

Note that Hybrid-DeepCom is an extended version of DeepCom proposed in our preliminary work (Hu et al. 2018a). The major differences between Hybrid-DeepCom and DeepCom are as follows:

1. The combination of lexical and syntactical information. In DeepCom, the input of the model is the traversed AST sequence which is composed of the "type_value" pairs of the nodes. It exploits the vanilla seq2seq model with attention to generate comments from these AST sequences. The model contains an encoder, a decoder and the attention layer. It is limited for DeepCom to learn the lexical information as it is more relevant to the source code. In Hybrid-DeepCom, we use two different encoders to learn the lexical and syntactical information respectively. The lexical information is learned from the source code tokens and the structure information is learned from the AST sequence which contains the nodes represented by their "type". In addition, we leverage a combined attention mechanism to fuse the lexical and syntactical information to predict comment words. The attention mechanism learns the alignment between the generated words and

the source code or AST sequences. In this way, Hybrid-DeepCom can more effectively learn the relationship between comments and features (i.e., lexical and syntactical) of source code.

2. Identifier processing. DeepCom uses the "type" of a node to represent its out-of-vocabulary "value" in the AST sequences. We find that the "type" for most out-of-vocabulary "value" tokens is the same, i.e., "SimpleName" which is the "type" for identifiers. In other words, we replace most out-of-vocabulary tokens by another special token "SimpleName" instead of ⟨*UNK*⟩. It causes that the improvement to be limited as these out-of-vocabulary tokens are still represented by the same token and the model can not distinguish them. In addition, the "type" mainly expresses the structure information, thus the semantic is missing after replacing the identifiers.

   In Hybrid-DeepCom, we leverage a simple but effective approach to reduce the out-of-vocabulary tokens. We split the identifiers according to the camel casing naming convention and the semantic of the identifiers can be represented as these subtokens. The number of tokens in the source code reduces significantly and accuracy improves.
3. Decoding with Beam Search. In DeepCom, the tokens are generated word by word according to a language model. While in Hybrid-DeepCom, at each step, the token is generated following Beam Search. Beam Search is better at exploring the search space of all possible comments by keeping around a small set of top candidates.
4. Much cleaner dataset. In DeepCom, we process the dataset roughly and it causes there is a lot of noise in the dataset, such as, override methods, much longer methods or comments, and etc. The noise leads to the limited ability of these approaches. In Hybrid-DeepCom, we define more rules to get a much cleaner dataset on the statistics of the collected samples.

4 Experiments

In this section, we evaluate different approaches by measuring their accuracy on generating Java methods' comments. Specifically, we mainly focus on the following research questions:

- RQ1: How effective is Hybrid-DeepCom compared with the state-of-the-art baselines?
- RQ2: What is the impact of source code and comments with different lengths on the performance of Hybrid-DeepCom?
- RQ3: What is the impact of the vocabulary size on the performance of Hybrid-DeepCom?
- RQ4: How effective is Hybrid-DeepCom to generate comments for new projects?

In RQ1, we evaluate whether and to what extent our Hybrid-DeepCom outperforms the baselines. In RQ2 and RQ3, we investigate the impact of various parameters of Hybrid-DeepCom on its performance. In RQ4, we evaluate the performance of Hybrid-DeepCom considering cross-project evaluation setting whereas Hybrid-DeepCom is trained on a set of projects and applied to another projects not in the training set.

In the following subsections, we describe the experimental setup and results. We introduce baselines that we compare against our proposed approach (Section 4.1.1), the dataset that we used in our experiments (Section 4.1.2), the details of the training process from which we build the neuronal networks (Section 4.1.3), metrics that we use to evaluate our approach and the baselines (Section 4.1.4), and experiment results that answer the research questions (Sections 4.2.1 – 4.2.4).

4.1 Experimental Setup

4.1.1 Baselines

We compare Hybrid-DeepCom with CODE-NN (Iyer et al. 2016) which is the state-of-the-art code summarization approach and also a deep learning based method. CODE-NN is an end-to-end generation system to generate summaries for code snippets. It exploits an RNN with attention to generate summaries by integrating the token embeddings of source code instead of building language models for source code. We do not use IR approaches as baselines, because the results in CODE-NN has shown that CODE-NN outperforms the IR based approaches.

We also compare Hybrid-DeepCom with the basic Seq2Seq model, the attention based Seq2Seq model, and DeepCom with a classical traversal method (i.e., pre-order traversal). The Seq2Seq model and the attention based Seq2Seq model take the source code as inputs. They aim to evaluate the effectiveness of NMT approaches for comments generation. To evaluate the effectiveness of SBT, we compare SBT with one of the most ordinary traversal methods – pre-order traversal. Since many of the code snippets in the dataset that CODE-NN proposed are incomplete and hard to be parsed into ASTs, we do not compare these approaches on their dataset.

Different from the average sentence-level BLEU score used in (Hu et al. 2018a), we use the smoothing function (method 4) (Chen and Cherry 2014) to avoid the non-overlapping grams between the generated comments and ground truth. In addition, we apply Corpus-level BLEU score, METEOR and IR metrics to evaluate effectiveness of different approaches.

4.1.2 Dataset

The dataset was collected from GitHub's Java repositories created from 2015 to 2016. Following (Nguyen et al. 2016; Leitner and Bezemer 2017; Amann et al. 2016) we consider only those having more than 10 stars to filter out low quality repositories. The dataset is available online.[3] We follow the following steps to prepare the dataset.

Preprocessing First, we extract Java methods and their corresponding Javadoc from these Java projects. We use the first sentences of the Javadoc as the target comments because they typically describe the functionalities of Java methods according to Javadoc guidance.[4] This follows (Gu et al. 2016) who also used the first sentence of Javadoc as a short summary of a method. The methods without Javadoc are omitted. Second, we filter out some ⟨Method, Comment⟩ pairs collected from previous step. Pairs that have just one-word Javadoc descriptions are filtered out in this work as these comments can not sufficiently express the Java method functionalities. We also exclude the setter, getter, constructor and test methods (marked with @Test annotation), since it is trivial to generate comments for such methods. After the preprocessing of the dataset, we get 588,108 ⟨method, comment⟩ pairs. Table 1 shows the statistics of the corpus after preprocessing. Figure 7 shows the distribution of the number of methods and classes in each project and the distribution of the number of methods in each class.

[3] https://github.com/xing-hu/EMSE-DeepCom

[4] http://www.oracle.com/technetwork/articles/java/index-137868.html

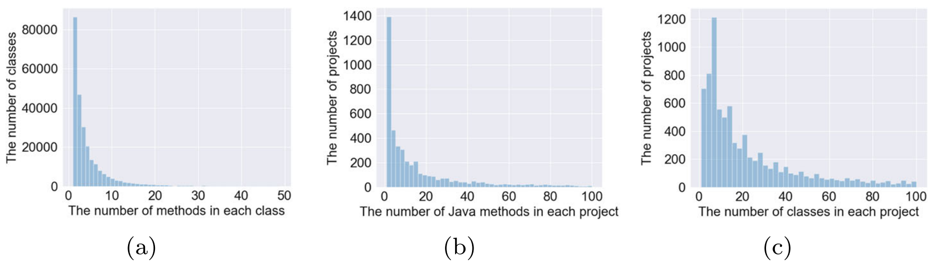


Fig. 7 Distributions of the number of methods and classes in the projects: **a** distribution of number of methods per class; **b** distribution of number of methods per project; **c** distribution of number of classes per project

Cleaning the Dataset After the preprocessing of the dataset, we further manually check the collected samples and filter out some samples to get a cleaner dataset. First, we filter out Java methods with @SmallTest, @LargeTest, and @MediumTest annotations. Second, we exclude the overridden Java methods as the overridden methods usually implement the same functionality and will cause unwanted repetition.

Setting Maximum Lengths for Source Code and Comments A maximum sequence length for the source and target sequences need to be set for neural networks to work well, c.f., (Bahdanau et al. 2014). Table 2 illustrates the statistics for code lengths and comments lengths after the dataset is cleaned following the previous step. We also give the distributions of method lengths and comment lengths as shown in Fig. 8. The average length of Java methods and comments are 93.93 and 11.24 tokens in this corpus. We find that more than 95% of code comments have no more than 30 words and about 90% of Java methods are no longer than 200 tokens. Based on these observations and following (Jiang et al. 2017), the Java methods and comments that have more than 200 tokens and 30 tokens are excluded according to the data length distribution. Because the ability of the neural network models is limited, the long sequences are hard to learn. In addition, many long code and comments are irregular in natural and some are not typical natural language descriptions (shown as Fig. 9). We also filter out the pairs whose comments are less than 4 words for calculating the BLEU score accurately. Computation of BLEU score is impossible for such cases as we cannot get 4-grams from a sentence if its length is less than 4. At last, we get 485,812 pairs of Java methods and comments.

Generating Training/Validation/Test Sets for RQ1-RQ3 We randomly select 20,000 pairs for testing and validation respectively and the rest of 445,812 pairs are used for training. As the total pairs are less than the ones used by DeepCom, we try to ensure that the training

Table 2 Statistics for code lengths and comments lengths

Method lengths					
Avg	Mode	Median	<100	<150	<200
93.93	10	45	74.86 %	83.80 %	88.63 %
Comments lengths					
Avg	Mode	Median	<20	<30	<50
11.24	8	10	91.14 %	96.96%	98.41 %

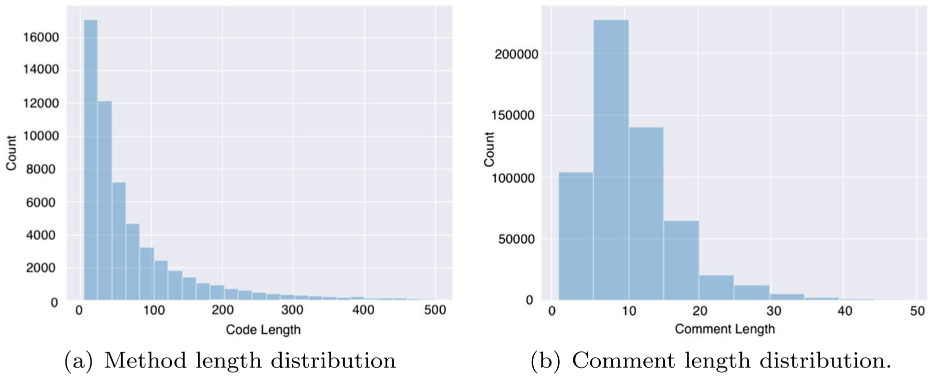


Fig. 8 Length distribution of the dataset

set has enough data to learn the model well. In many prior works, the test set typically has about 1%-20% of the total dataset (Bahdanau et al. 2014; Jiang et al. 2017). In this paper, we use about 5% (20,000 pairs) of the total instances to test the learned model. A number of previous studies also validate their deep learning based techniques by splitting their dataset into training, validation, and test sets. For example, Jiang et al. (2017) randomly selected 3k samples for testing, 3k samples for validation, and the remaining 26k samples for training.

Cross-Project Setting for RQ4 First, we randomly split the 9,714 projects into 10 folds, where each fold has around 971 projects. Second, we extract the ⟨method, comment⟩ pairs following the same procedure introduced above. Then, each fold is used as a testing dataset to evaluate the prediction model built on the other nine folds (i.e., training dataset).

Building Vocabularies The source code and comments are parsed into tokens by javalang and NLTK[5] respectively. Then, all tokens are converted to lowercase. During the training, the numerals and strings are replaced with generic tokens ⟨*NUM*⟩ and ⟨*STR*⟩ respectively. The maximum length of source code and AST sequences is set to 200 and 500 respectively. We use a special symbol ⟨*PAD*⟩ to pad the shorter sequences and the longer sequences will be cut into sequences with 500 tokens. We add special tokens ⟨*START*⟩ and ⟨*EOS*⟩ to the decoder sequences during training. ⟨*START*⟩ is the start of the decoding sequence and the ⟨*EOS*⟩ means the end of it. The maximum comment length is set to 30. The out-of-vocabulary tokens are replaced by a special token ⟨*UNK*⟩.

For RQ1, RQ2, and RQ4, the vocabulary sizes for comments and the source code are both 30,000. For RQ3, we create five different vocabularies for source code by varying the vocabulary size for source code to be 5,000, 10,000, 30,000, 50,000, and whole vocabulary after splitting. This setting results in different out-of-vocabulary rates. The vocabulary size of the source code after splitting based on the camel casing conversion is 47,939.

[5] https://www.nltk.org/

Fig. 9 The example comment with a long comment that is not typical natural language descriptions

```
/**
* version 1:
*
* --------------
* | key | value |
* --------------
*
* version 2
*
* ----------------------------------------
* | key | value | timestamp | persistent |
* ----------------------------------------
*/
```

4.1.3 Training Details

During training, the model is validated every 2,000 minibatches on the validation set by BLEU score (Papineni et al. 2002) which is a commonly used evaluation metric for NMT. The maximum number of epochs is 50, the maximum number of minibatches is 500,000, and early stopping is used. The validation set is used to estimate how well the model has been trained and to estimate model properties. Finally, the accuracy of the model on test set gives a realistic estimate of the performance of the model on unseen data. However, the validation process takes extra time. Taking time and model capacity into consideration, we validate every 2,000 minibatches and select the model which performs best on the validation dataset as the best model. The best model is then evaluated on the test set by computing average BLEU scores and the results will be discussed in Section 4.

For implementation, we build our model using the Tensorflow which is an open-source deep learning framework.[6] The time of the training lasts about 80 hours for each model. The hyperparameters are shown as follows:

- The SGD (with minibatch size 128 randomly chosen from training instances) is used to train the parameters.
- Hybrid-DeepCom uses one-layered GRU with 256 dimensions of the hidden states and 256 dimensional word embeddings.
- The learning rate is set to 0.5 and we clip the gradients norm by 5. The learning rate is decayed using the rate 0.99.
- The beam width is set to 5.

4.1.4 Evaluation Measures: IR Metric and MT Metric

In this paper, we use Information Retrieval (IR) metrics and Machine Translation (MT) metrics to evaluate our method. For IR metrics, we report the precision, recall, F-sore and F-mean of different approaches. For machine translation metrics, we use BLEU and METEOR to evaluate our approach. The details of the six metrics are introduced as follows.

Precision Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. We calculate the precision according to the approach proposed by (Denkowski and Lavie 2014). It gives different weights for content and function words. Content words usually give us information to understand the sentences. Function words are necessary words for grammar. In other words, content words give us the most important

[6]https://www.tensorflow.org/

information while function words are used to stitch those words together. Therefore, the content words and function words have different contributions on the evaluation.

The content and function words in the generated comment and reference are (h_c, h_f) and (r_c, r_f) respectively. For each of the matchers m_i, count the number of content and function words covered by matches of this type in the hypothesis $(m_i(h_c), m_i(h_f))$ and reference $(m_i(r_c), m_i(r_f))$. The weighted precision is calculated as follows

$$P = \frac{\sum_i w_i \cdot (\delta \cdot m_i(h_c) + (1-\delta) \cdot m_i(h_f))}{\delta \cdot |h_c| + (1-\delta) \cdot |h_f|} \tag{14}$$

where δ means the content-function word weight and is set to 0.75 according to (Denkowski and Lavie 2014).

Recall Similar to the precision, we calculate the weighted recall to evaluate approaches.

$$R = \frac{\sum_i w_i \cdot (\delta \cdot m_i(r_c) + (1-\delta) \cdot m_i(r_f))}{\delta \cdot |r_c| + (1-\delta) \cdot |r_f|} \tag{15}$$

F-score A summary measure that combines both Precision and Recall - it evaluates if an increase in Precision (Recall) outweighs a reduction in Recall (Precision). In many cases, high Recall indicates the sacrifice of Precision, and vice versa.

$$F_{score} = 2 \cdot \frac{P \cdot R}{P + R} \tag{16}$$

F-mean F-mean measures the parameterized harmonic mean of precision and recall.

$$F_{mean} = \frac{P \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R} \tag{17}$$

where α is set to 0.85 (Denkowski and Lavie 2014).

METEOR METEOR (Denkowski and Lavie 2014) is a machine translation metric that we use to evaluate our proposed approach. METEOR evaluates translation hypotheses by aligning them to reference translations and calculating sentence-level similarity scores. The METEOR is calculated:

$$Score = (1 - Pen)F_{mean} \tag{18}$$

The penalty (Pen) is calculated according to the number of chunks (ch) and the number of matches (m):

$$Pen = \gamma * \left(\frac{ch}{m}\right)^{\beta} \tag{19}$$

METEOR is a recall-oriented metric for machine translation and is widely used for machine translation and code summarization. It accepts two parameters, namely, β and γ. They are set to 0.20 and 0.60 respectively following (Denkowski and Lavie 2014).

BLEU score Hybrid-DeepCom uses machine translation evaluation metric BLEU score (Papineni et al. 2002) to measure the quality of generated comments. BLEU score is a widely-used accuracy measure for NMT (Klein et al. 2017) and has been used in software tasks evaluation (Gu et al. 2016; Jiang et al. 2017). It calculates the similarity between the generated sequence and reference sequence (usually a human-written sequence). The BLEU score ranges from 1 to 100 as a percentage value. The higher the BLEU, the closer the candidate is to the reference. If the candidate is completely the same to the reference, the BLEU becomes 100%. Jiang et al. (2017) exploit it to evaluate the generated summaries for commit messages. Gu et al. (2016) use BLEU to evaluate the accuracy of generated

API sequences from natural language queries. Their experiments show that BLEU score is reasonable to measure the accuracy of generated sequences.

In detail, BLEU computes the n-gram precision of a candidate sequence to the reference. The score is computed as:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{20}$$

where p_n is the ratio of length n subsequences in the candidate that are also in the reference. In this paper, we set N to 4, which is the maximum number of grams. BP is brevity penalty,

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-r/c)} & if\ c \leq r \end{cases} \tag{21}$$

where c is the length of the candidate translation and r is the effective reference sequence length.

In this paper, we regard a generated comment as a candidate and a programmer-written comment (extracted from Javadoc) as a reference. We compare different approaches by both sentence-level BLEU score and corpus-level BLEU score. To avoid the non overlapping N-grams in the sentence, we use the smoothing-4 method proposed in (Chen and Cherry 2014) to compute the sentence-level BLEU score. The sentence-level BLEU score is evaluated by the NLTK with smoothing-4 method and the corpus-level BLEU score is computed by the script "multi-bleu.perl" that is provided by Moses.[7]

4.2 Results

4.2.1 RQ1: Hybrid-DeepCom vs. Baseline

We measure the gap between automatically generated comments and human-written comments. The difference is evaluated by two machine translation metrics (i.e., BLEU score and METEOR) and four IR metrics (i.e., Precision, Recall, F-score, and F-mean). Table 3 illustrates the average sentence-level BLEU scores, corpus-level BLEU score, and METEOR of different approaches to generating comments for Java methods. The accuracy of machine translation model Seq2Seq substantially outperforms CODE-NN. CODE-NN fails to learn the semantic of the source code when it generates comments from token embeddings of source code directly. Seq2Seq model exploits RNN to build a language model for the source code and effectively learns the lexical of Java methods. The BLEU score increases further while integrating the structural information. Compared to DeepCom with the pre-order traversal, the SBT based model is much more capable of learning syntactic information within Java methods. When combining the lexical and syntactical information together, the Hybrid-DeepCom outperforms the baselines. The average BLEU score of DeepCom improves about 41.71% compared to CODE-NN.

Table 4 demonstrates the results considering IR metrics. Different from the results considering the MT metrics, CODE-NN achieves the highest precision and lowest recall. Many comments generated by CODE-NN are shorter which cause the high precision and for longer comments CODE-NN generates less matched words. When comparing Seq2Seq model with DeepCom, the precision of Seq2Seq model is significantly higher than DeepCom. It shows that the lexical of the source code contributes to the precision improvement.

[7] http://www.statmt.org/moses/?n=Moses.SupportTools

Table 3 BLEU and METEOR score for our approach compared with baselines

Approaches	S-BLEU* (%)	C-BLEU* (%)	METEOR
CODE-NN	27.88	21.31	18.04
Seq2Seq	34.57	28.29	21.16
Seq2Seq(Attention)	35.51	29.85	22.04
DeepCom (Pre-order)	37.13	31.49	21.86
DeepCom (SBT)	38.22	32.94	22.63
Hybrid-DeepCom	39.51	34.51	24.46

* S-BLEU means average Sentence-level BLEU score; C-BLEU means Corpus-level BLEU score

When it comes to the recall, the DeepCom outperforms Seq2Seq models. It demonstrates the importance of the structure information on more comprehensive comment generation. The precision and recall of Hybrid-DeepCom outperforms other approaches by a substantial margin. Hybrid-DeepCom learns the lexical and syntactical information at the same time. The combined information is helpful to generate more comprehensive and accurate comments.

To better illustrate the results, we evaluate the generated comments by different smoothing methods. Sentence-level BLEU score is problematic as it can easily become zero. Because it computes a geometric mean of N-gram precisions, if a higher order N-gram precision (eg. n = 4) of a sentence is 0, then the BLEU score of the entire sentence is 0. Therefore, some smoothing techniques are proposed to solve the problem. In this paper, we evaluate results by smoothing techniques introduced by Chen and Cherry (2014). Chen and Cherry (2014) introduced 7 smoothing techniques that work better for sentence-level evaluation. We compare different approaches by using different smoothing techniques and the results are shown in Fig. 10. As the method 6 only works when the precision for bigrams is non-zero, it is not applicable to evaluate the generated comments in this paper.

Through the evaluation, we have verified that comments generation task is very similar to machine translation except that the structural information in source code needs to be taken into account. The sentence-level BLEU score improves from 27.88% to 34.57% when exploiting the neural machine translation model, i.e., Seq2Seq model. Hybrid-DeepCom can generate more informative comments by integrating the structure information. Compared to the model without AST, i.e., Seq2Seq (Attention), the BLEU score of Hybrid-DeepCom increases about 11%. When compared to the AST-based model

Table 4 Precision, Recall, and F-score for our approach compared with baselines

Approaches	Precision	Recall	F-score	F-mean
CODE-NN	53.97	35.07	42.51	37.01
Seq2Seq	48.25	41.54	44.64	42.42
Seq2Seq(Attention)	48.47	43.05	45.60	43.79
DeepCom(Pre-order)	44.81	42.53	43.64	42.86
DeepCom(SBT)	45.68	43.77	44.70	44.04
Hybrid-DeepCom	51.11	47.03	48.99	47.60

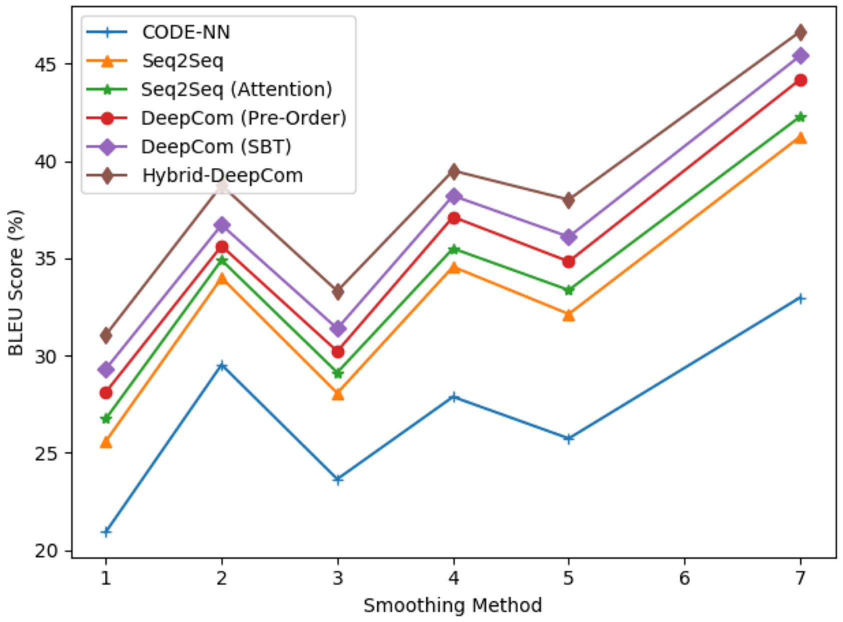
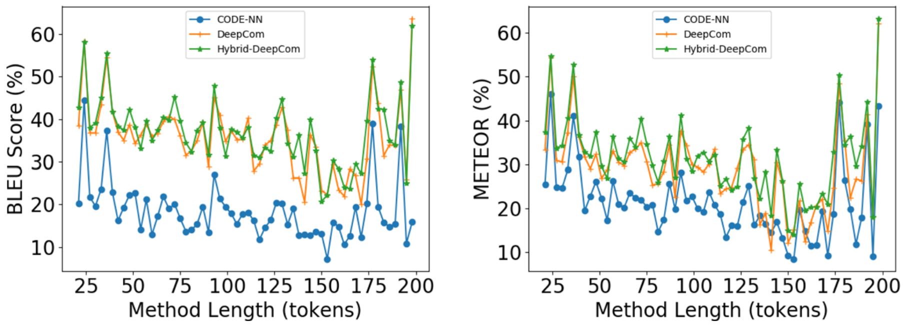


Fig. 10 The Sentence-Level BLEU score with different smoothing methods

DeepCom, Hybrid-DeepCom increases about 3%. Experimental results indicate that combining the lexical and syntactical information is important for translating text in structured languages to unstructured ones.

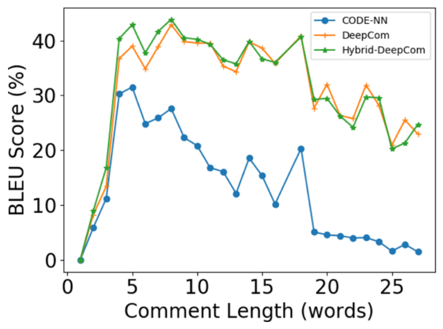
4.2.2 RQ2: Metrics for Source Code and Comments of Different Lengths

We further analyze the prediction accuracy for Java methods and comments of different lengths. Figures 11 and 12 present the average sentence-level BLEU scores and METEOR

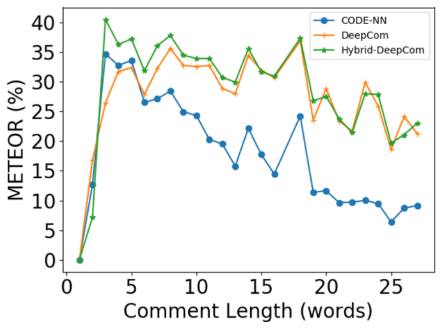


(a) BLEU scores for different method lengths (b) METEOR scores for different method lengths

Fig. 11 Experimental results of our proposed method and the baseline on different metrics for varying code length



(a) BLEU scores for different comment lengths



(b) METEOR scores for different comment lengths

Fig. 12 Experimental results of our proposed method and the baseline on different metrics for varying comment length

of Hydrid-DeepCom, DeepCom, and CODE-NN for source code and generate comments of varying lengths.

As Fig. 11 illustrates, the average BLEU scores tend to be lower for higher source code lengths. When the code length is between 25 and 50, the models achieve higher accuracy. The neural network models can learn the complete semantic of the source code when the Java methods are short. The performance of these models decreases while code length increases. The performance decline for CODE-NN is much more obvious than Hybrid-DeepCom. Hybrid-DeepCom outperforms the CODE-NN regardless of the code lengths. We find that the accuracy fluctuates greatly when the source code has more than 175 tokens. Because there are few Java methods with more than 175 tokens thus it is limited for Hybrid-DeepCom to learn the them well. It causes great fluctuation when code has more than 175 tokens. Overall, we can note that the performance of CODE-NN is significantly lower than our proposed approaches.

Figure 12 illustrates the performance of Hybrid-DeepCom, DeepCom, and CODE-NN considering various comment lengths. The performance of Hybrid-DeepCom is better than DeepCom slightly. Both Hybrid-DeepCom and DeepCom are better than CODE-NN significantly. When the comment length increases, the performance of Hybrid-DeepCom and DeepCom decreases slightly. However, the performance of CODE-NN decreases sharply when code comment length increases. When the code comment lengths are greater than 20 tokens, the BLEU-score of CODE-NN decreases to less than 10%. On the other hand, DeepCom and Hybrid-DeepCom still perform well when we need to generate comments consisting of more than 20 words.

4.2.3 RQ3: The Influence of Code Vocabulary Size on Comment Generation

To evaluate the influence of Out-of-Vocabulary tokens of source code on the comment prediction, we conduct experiments for approaches with different vocabulary size. We create four specific vocabulary by varying the vocabulary size for source code to be 5,000, 10,000, 30,000, 50,000, resulting in different out-of-vocabulary rates. We get 542,680 unique tokens from the source code training set and 47,939 tokens if the identifiers are split. Table 5 illustrates the sentence-level BLEU score with different vocabulary settings. The column

Table 5 Accuracies (BLEU score) on comment generation with different vocabulary sizes. The Out-of-Vocabulary (OoV) rate denotes the percentage of code tokens that are out of the vocabulary

Vocabulary Size	5,000	10,000	30,000	50,000	camel
OoV Rate	99%	98%	94%	90%	0
CODE-NN	21.60	24.24	27.88	30.88	28.38
Seq2Seq	27.78	29.53	34.57	35.73	35.79
Seq2Seq(Attention)	30.11	32.42	35.51	36.25	38.03
DeepCom(Pre-order)	31.48	34.78	37.13	37.60	–
DeepCom(SBT)	31.08	33.62	38.22	39.71	–
Hybrid-DeepCom	32.91	34.66	39.51	40.26	41.86

"camel" means that the vocabulary contains all tokens after splitting identifiers. The identifiers cannot be split for the approaches DeepCom(Pre-order) and DeepCom(SBT) since they just exploit the traversed AST sequences to generate code comments.

The results show that the BLEU score increases along with the decreasing of out-of-vocabulary rate. The semantic of a piece of source code cannot be inferred well when there are many out-of-vocabulary words in the source code. Compared to the accuracy of the 5,000 vocabulary size, the accuracy of camel split setting improves about 27%. The results motivate us to decrease the out-of-vocabulary rate as much as possible to ensure the semantic of source code.

4.2.4 RQ4: Cross-Project Validation

We evaluate the performance of our approach using 10-fold cross-validation considering cross-project setting. The reason is that we would like to investigate whether our prediction model built on pairs coming from a set of projects can be generalized to pairs from other projects not in the set. The entire process is repeated ten times to alleviate possible sampling bias in random shuffle and sampling, and we record the average evaluation results. Tables 6 and 7 illustrates the average results of 10-fold cross-validation. When applying to the new projects, the accuracy decreases slightly as the new projects may have different characteristics from the existing projects. Compared to the CODE-NN, Hybrid-DeepCom still performs better on generating comments for Java methods. The model that Hybrid-DeepCom trained from existing projects can generalize better to new Java projects. Overall, for this experiment setting, Hybrid-DeepCom improves the sentence-level BLEU score over

Table 6 Comparison results of different approaches for 10-fold cross-validation of MT metrics

Approach	S-BLEU(%)	C-BLEU(%)	METEOR(%)
CODE-NN	21.61	14.55	13.34
Seq2Seq	30.37	21.30	15.93
Seq2Seq(Attention)	31.64	21.90	16.73
DeepCom(Pre-order)	31.76	23.44	16.92
DeepCom(SBT)	32.87	24.11	17.82
Hybrid-DeepCom	34.82	27.62	20.81

Table 7 Comparison results of different approaches for 10-fold cross-validation of IR metrics

Approach	Precision(%)	Recall(%)	F-score(%)
CODE-NN	43.02	26.82	33.04
Seq2Seq	31.68	32.65	32.14
Seq2Seq(Attention)	31.65	34.20	32.87
DeepCom(Pre-order)	33.80	34.17	33.97
DeepCom(SBT)	34.36	35.94	35.12
Hybrid-DeepCom	39.89	40.81	40.33

CODE-NN and DeepCom by 61% and 6% respectively. In addition, CODE-NN achieves the highest precision and the lowest recall which is consistent with the results of RQ1. The recall and F-score of Hybrid-DeepCom outperform the baselines' by a substantial margin. It demonstrates that the lexical and syntactical information learned from a dataset can be generalized to another unseen dataset. The combined information is much helpful to generate a more comprehensive comment for new projects.

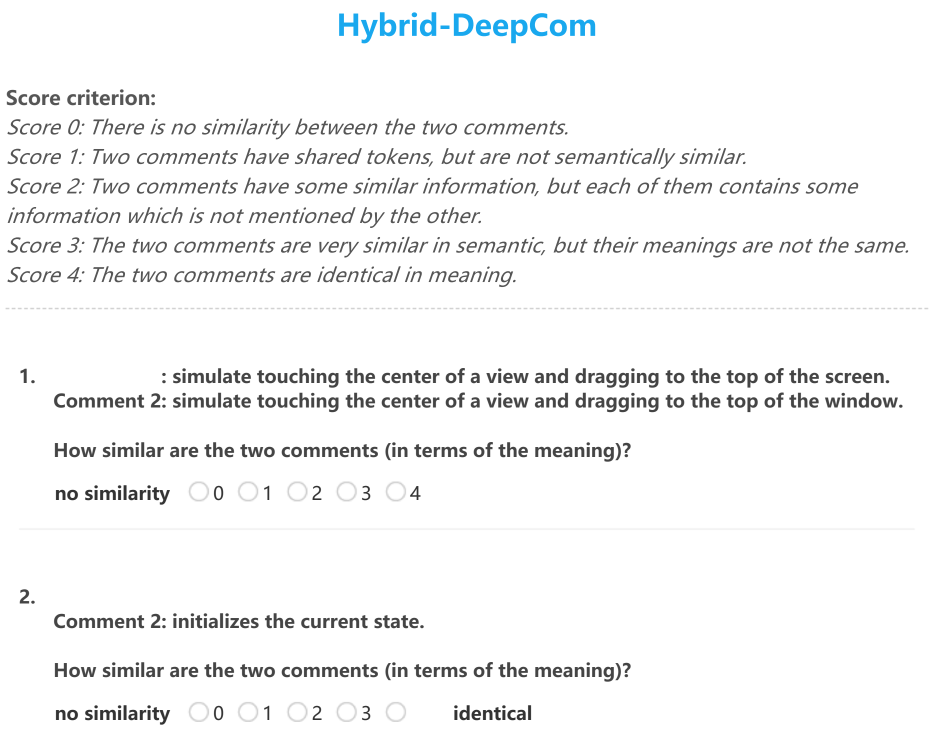
5 Human Evaluation

In this section, we performed a manual verification to evaluate the quality of the automatically generated code comments by the Hybrid-DeepCom and CODE-NN. Since automatic metrics do not always agree with the actual quality of the results, we perform human evaluation studies to measure the similarity of the output of our system and the ground truth. We invite 8 participants for 30 minutes each to evaluate the comments in a survey-based study. The 8 participants are computer science Ph.D students and are not co-authors. They all have Java programming experience ranging from 2 to 5 years. In the rest of this section, we describe the process of conducting the survey and the survey results.

5.1 Survey Procedure

We randomly select 200 samples consisting of the generated comments and their references. 100 of the comments are generated by Hybrid-DeepCom and the other 100 comments are generated by the baseline CODE-NN. The 200 samples are evenly divided into four groups and we make a questionnaire for each group. Similar to (Jiang et al. 2017) and (Liu et al. 2018), we ask participants to give score between 0 to 4 to measure the semantic similarities between reference comment and the generated comment. Score 0 means there is no similarity between the two comments and score 4 means two comments are identical in meaning. Figure 13 shows the design of the questionnaire with pairs of comments. Note that the two comments(reference/generated comment) are listed randomly. The participants do not know which comment is the generated one. In addition, the participants also do not know which approach generates the generated comment.

Each comment group is evaluated by 2 participants. As shown in Fig. 13, the score criterion is listed in the beginning of each questionnaire to guide participants. We follow the score criterion defined by (Liu et al. 2018) which is used to measure the similarities between commit messages. Participants are allowed to search the Internet for related information and unfamiliar concepts.



Hybrid-DeepCom

Score criterion:
Score 0: There is no similarity between the two comments.
Score 1: Two comments have shared tokens, but are not semantically similar.
Score 2: Two comments have some similar information, but each of them contains some information which is not mentioned by the other.
Score 3: The two comments are very similar in semantic, but their meanings are not the same.
Score 4: The two comments are identical in meaning.

1. : simulate touching the center of a view and dragging to the top of the screen.
   Comment 2: simulate touching the center of a view and dragging to the top of the window.

   How similar are the two comments (in terms of the meaning)?

   no similarity ◯0 ◯1 ◯2 ◯3 ◯4

2.
   Comment 2: initializes the current state.

   How similar are the two comments (in terms of the meaning)?

   no similarity ◯0 ◯1 ◯2 ◯3 ◯ identical

Fig. 13 The score criterion and the example questions in our survey

5.2 Results

We obtain 400 scores from our human evaluation in which 200 are scores for Hybrid-DeepCom and the other 200 are scores for CODE-NN. Same as (Liu et al. 2018), we regard a score of 0 and 1 as low quality, a score of 2 as medium quality, and a score of 3 and 4 as high quality. Table 8 illustrates the results of our user study. The proportion of high-quality Hybrid-DeepCom comments is significantly higher than that of CODE-NN comments. On the other hand, the proportion of low-quality Hybrid-DeepCom comments is much lower than that of CODE-NN. Moreover, the mean score of Hybrid-DeepCom is higher than that of CODE-NN. These results show that Hybrid-DeepCom outperforms the baseline CODE-NN by a substantial margin. In addition, we confirmed the dominance of our proposed approach Hybrid-DeepCom using a Mann Whitney U test; the improvement of Hybrid-DeepCom over other approaches is statistically significant with a p-value of 0.002.

Table 8 The results of our user study

Approach	Low	Medium	High	Mean score
CODE-NN	53.5%	13.0%	33.5%	1.73
Hybrid-DeepCom	39.5%	11.0%	49.50%	2.21

6 Discussion

As shown in previous sections, our model can achieve a promising performance in generating comments for Java methods. However, there are some other observations worth for further investigation. In this section, we will report other observations including:(1)when Hybrid-DeepCom generates comments with high BLEU score? (2) why the automatically generated comments receive low BLEU scores? (3)why there are unknown words in the generated comments?

To analyze the reason why Hybrid-DeepCom generates comments with high BLEU score, low BLEU score, and unknown words, the first author and one Ph.D student from Peking University jointly figure out the labels for the reasons. The detailed steps are as below:

- We first split the generated comments into three distinct sets, namely, the high BLEU score comments(more than 70%), the low BLEU score comments(less than 20%), and comments with $\langle UNK \rangle$. From each set, we randomly select 100 samples and each sample includes the Java method and the generated comment.
- For each set, the first author and one Ph.D student independently label the reasons why comments with high BLEU score/low BLEU score/$\langle UNK \rangle$.
- The overall agreement of the two labelers is about 78%, which indicates substantial agreement between the labelers. After completing the manual labeling process, the two labelers discussed their disagreements to reach a common decision.

6.1 Investigating the Comments with High BLEU Score

There are many comments generated by Hybrid-DeepCom with high BLEU score and we are interested in which kinds of comments with high BLEU score. There are 4,171 (about 21%) generated comments the same as the references. Most of these Java methods have clear functionality and the code conventions are universal. Hybrid-DeepCom can generate exactly correct comments from the source code (Case 1 in Table 12), which validate the capability of our approach to encode Java methods and decode comments. The comment generated by CODE-NN is related to the source code, but it losses a lot of functionalities of Java methods. In addition, we randomly select 100 samples whose BLEU scores are more than 70% and analyze the functionality of these Java methods. The first author and one Ph.D student label the functionalities of these methods, e.g., strings process, exceptions, and SQL process. Table 9 illustrates the functionalities of the Java methods. We find that Java methods aiming to create an object, implement algorithms, and process strings take the majority of the high score results.

Table 9 The goal of the Java methods

	# Samples
Create an object	19
Algorithm implementation	13
String process	14
Exception	4
File	9
SQL	3
Other	38

For example, as Case 2 in Table 12 shows, the method "sort" aims to sort an array using quick sort, Hybrid-DeepCom captures the correct functionality and generates the correct comment. However, CODE-NN ignores much important information and just generates one keyword "sort".

6.2 Investigating the Comments with Low BLEU Score

In addition to the performance of Hybrid-DeepCom on comment generation, we are also interested in the reason of automatically generated comments with low BLEU score. We randomly select 100 samples whose BLEU scores of the comments generated by Hybrid-DeepCom are less than 20%. The first author and one Ph.D student measure these samples respectively. For each sample, participants are give the generated comment, the reference, and the Java method.

The evaluation criteria are shown as follows:

- Good. The generated comment is accurate.
- Related. The generated comment is related to the Java method.
- Bad. The generated comment is not accurate, adequate, or useful at describing Java method.

Table 10 illustrates the results of the generated comments with low BLEU scores. We find that 27% of generated comments are accurate at describing the Java methods although there is little matched words. These comments can be committed as the comments of the Java methods. About 20% comments are related to the Java methods to a certain degree and can be committed as comments with little fixes.

The comments which are tagged as bad and related are mainly divided into two types, meaningless sentences, and sentences with clear semantics. The former mainly contains empty sentences and results with too many repetitive words (shown in case 3 in Table 12). We conjecture the problems come from out-of-vocabulary words in original comments or mismatch between the Java methods and comments in the original dataset. The lacking of samples with some symbols (@Nullable in Case 3) is another reason for lower BLEU score. As we have remove Java methods having a number of annotations, there are fewer samples with annotations. The model can not learn their semantic well with such fewer samples.

In the latter ones, most of them are irrelevant to original comments in their semantics. The comments tagged as good usually hold relevant semantics (shown in Case 2 and Case 5 in Table 12). The automatically generated and manual comments may describe similar functionalities but with different words or order.

6.3 Investigating the Reasons for Unknown Words in Generated Comments

There are unknown words in the generated comments and these words have no meaning for describing the Java methods. Similar to the evaluation of the above discussion, we also randomly select 100 generated comments with unknown words. The first author and one Ph.D student analyze the accurate words that are replaced by unknown token $\langle UNK \rangle$. Table 11

Table 10 The quality of the generated comments with low BLEU scores

	Good	Related	Bad
# Samples	27	21	52

Table 11 The accurate words of the unknown words

	# Samples
Method name	26
Class name	11
Variable name	8
Other	55

illustrates the accurate words in the reference are replaced by the unknown word in the generated comments. We find that more than 26% of the generated unknown tokens represent the method names appeared in the references. The class names and variable names account for 11% and 8% respectively. The method, class, and variable names are all identifiers which are defined by the programmers. In other words, half of the unknown words in the generated comments due to the identifiers in the references.

As Case 4 in Table 12 shows, Hybrid-DeepCom fails to predict the token "SeparatorPainter" which is the method name defined by developers. Hybrid-DeepCom is not good at learning the method or identifiers names occurred in comments. Developers define various names while programming and most of these tokens appearing at most once in the comments. During the training process, we split the identifiers in the source code whereas we do not split the identifiers occur in the comments. We can easily distinguish a token is an identifier by using a parser to analyze the source code. However, we can not determine tokens of comments are identifiers from source code or ordinary natural language words. It is hard for Hybrid-DeepCom to learn these user-defined tokens in comments that have been replaced by the unknown token $\langle UNK \rangle$ during data processing. However, CODE-NN is better than Hybrid-DeepCom on identifier prediction. As shown in Case 4, although the identifier predicted by CODE-NN is incorrect, it can predict a related identifier which improves the semantic of the generated comment. Because CODE-NN generates the comment by integrating the embedding of source code tokens, it is better at learning the identifiers than Hybrid-DeepCom.

6.4 Strengths of Hybrid-DeepCom

A major challenge for generating comments from code is the semantic gap between code and natural language descriptions. Existing approaches are based on manually crafted templates or information retrieval and lack a model to capture the semantic relationship between source code and natural language. Hybrid-DeepCom, a variant machine translation model, has the ability to bridge the gap between two languages, i.e., programming language and natural language.

6.4.1 Probabilistic Model Connecting Semantics of Code and Comments

One advantage of Hybrid-DeepCom is generating comments directly by learning source code instead of synthesizing comments from keywords or searching similar code snippets' comments.

Synthesizing comments from keywords usually uses some manually crafted templates. The procedure of templates definition is time-consuming and the quality of keywords depends on the quality of a given Java method. They fail to extract accurate keywords when the identifiers and methods are poorly named. The IR based approaches usually search the

Table 12 Examples of generated comments by Hybrid-DeepCom

Case ID	Example
1	`public PaymentDataException(String message, Reason reason){` `super(message);` `setMessageKey(getMessageKey()+reason.toString());` `}` **CODE-NN:** constructs a new exception with the specified detail message. **Hybrid-DeepCom:** constructs a new exception with the given detail message and appends the specified reason to the message key. **Human-written:** constructs a new exception with the given detail message and appends the specified reason to the message key.
2	`public static void sort(int[] data, int start, int end, IntegerComparator comp){` `quickSort(data, start, end, comp);` `}` **CODE-NN:** sorts the specified range. **Hybrid-DeepCom:** sort the contents of the given array using the given comparator. **Human-written:** sort the array using the given comparator.
3	`public static boolean isValid(@Nullable EncodedImage encodedImage){` `return encodedImage != null && encodedImage.isValid();` `}` **CODE-NN:** returns true if. **Hybrid-DeepCom:** checks if the encoded bytes is valid compatible with the encoded encoded encoded bytes. **Human-written:** checks if the encoded image is valid
4	`public SeparatorPainter(Which state){` `super();` `this.state = state;` `this.ctx = new PaintContext(CacheMode.FIXED_SIZES);` `}` **CODE-NN:** creates a new buttonpainter object. **Hybrid-DeepCom:** creates a new ⟨UNK⟩ object. **Human-written:** creates a new separatorpainter object.
5	`public boolean contains(int key){` `return rank(key) != -1;` `}` **CODE-NN:** If the object exists **Hybrid-DeepCom:** Checks whether the given object is contained within the given set **Human-written:**Is the key in this set of integers?

These samples are necessarily limited to short methods because of space limitations similar code snippets and take their comments as the final results. They rely on whether similar code snippets can be retrieved and how similar the snippets are.

Hybrid-DeepCom builds language models for code and natural language descriptions. The language models are able to handle the uncertainty in the correspondence between code and text. Hybrid-DeepCom learns common patterns from a large-scale source code and the encoder itself is a language model which remembers the likelihood of different Java methods. The decoder of Hybrid-DeepCom learns the context of source code which bridges the gap between natural language and code. Furthermore, the attention mechanism helps align code tokens and natural language words.

6.4.2 Generation Assisted by Structural Information

Programming languages are formal languages which are more structure dense than text and have formal syntax and semantics. It is difficult for models to learn semantic and syntax information at the same time just given code sequences. Existing approaches usually analyze source code directly and omit its syntax representation.

In contrast to traditional NMT models, Hybrid-DeepCom takes advantage of rich and unambiguous code structures. In this way, Hybrid-DeepCom bridges the gap between code and natural language with the assistance of structure information within the source code. From the evaluation results, we find that the structural information improves the quality of comments. The improvements for methods implementing standard algorithms are much more obvious. Java methods realizing the same algorithm may define different variables while their ASTs are much more similar.

6.4.3 The Impact of Each Proposed Change

We also analyze the impact of each proposed change in our proposed approach. Table 13 illustrates the improvement of each proposed change in our approach. The sentence-level BLEU score improves from 27.88% to 34.57% when exploiting the neural machine translation model, i.e., Seq2Seq model. The improvement shows that formulating the comment generation problem as the machine translation task has the biggest impact on the improvement. The attention mechanism improves the Seq2Seq model by 3%. Hybrid-DeepCom can generate more informative comments by integrating the structure information. Compared to the model without AST, i.e., Seq2Seq (Attention), the BLEU score of Hybrid-DeepCom increases BLEU score by 11%. When compared to the AST-based model DeepCom,

Table 13 The improvement of each proposed change

Approach	S-BLEU(%)	Improvement
CODE-NN	27.88	-
Seq2Seq	34.57	+6.69
Seq2Seq(Attention)	35.51	+0.94
DeepCom (Pre-order)	37.13	+1.62
DeepCom (SBT)	38.22	+1.09
Hybrid-DeepCom	38.72	+0.5
Hybrid-DeepCom + Beam Search	39.51	+0.79
Hybrid-DeepCom + camel	41.86	+2.35

Hybrid-DeepCom increases by 1.3%. The BLEU score increased to 39.51% when using Beam Search while generating comments. The BLEU score improves to 41.86% by exploiting the camel case splitting. Experimental results indicate that the comments generation task is very similar to machine translation task. The learning of both lexical and structure information and the camel case splitting benefits the code comment generation.

7 Threats to Validity

7.1 Internal validity

Threats to internal validity relates to the errors in our code and bias in replication of the CODE-NN model. To reduce errors in our code, we have double checked and fully tested our code, still there could be errors that we did not notice. To reduce the impact of undetected errors in our code, we also published our source code and dataset to enable other researchers replicate and extend our work. We replicated the CODE-NN by running the source code provided by authors on our dataset. Although we use the same experiment setting as Iyer et al. (2016) work, their parameters' setting could not suitable for our dataset. Thus, we believe there is little threat to internal validity.

7.2 External Validity

Threats to external validity relates to the quality of our dataset and generalizability of our findings. We have collected a large number of Java projects from GitHub. The neural network regards the corpus as the ground truth. We have tried our best to reduce the noise samples in our dataset. Although we have defined heuristic rules to decrease the noise in the corpus, there could be noise that we did not notice. In future work, we plan to collect more data from GitHub and filter out more noise samples. Also, we collected Java projects according to their stars and did not consider the domains of the projects, which might introduce bias to our experimental results. We used cross-project prediction to evaluate our approach, and the results show that our approach can be trained from a set of project and performs well when applied to a project outside the training set. In this work, we focus on Javadoc comments and ignore inlined comments. As a result, some high-quality ⟨method, comment⟩ pairs for which the comment part appears as inlined comments may be omitted in our experiments. In future work, it would be interesting to also consider inlined comments.

7.3 Construct Validity

The construct validity relates to the suitability of our evaluation measures. We use 6 evaluation measures, namely BLEU, METEOR, Precision, Recall, F-score, and F-mean. We use BLEU and METEOR because they are widely used in the machine translation tasks and have been used in many software engineering studies (Gu et al. 2016; Jiang et al. 2017). In addition to the machine translation metrics, we also use IR metrics Precision, Recall, F-score, and F-mean because these metrics are widely used in software engineering tasks. Thus, we believe there is little threat to construct validity.

8 Related Work

8.1 Code Summarization

As a critical task in software engineering, code summarization aims to generate brief natural language descriptions for source code. Automatic code summarization approaches vary from manually-crafted template (Sridhara et al. 2011; Sridhara et al. 2010; McBurney and McMillan 2014), IR (Wong et al. 2013; Haiduc et al. 2010a, b) to learning-based approaches (Iyer et al. 2016; Movshovitz-Attias and Cohen 2013; Allamanis et al. 2016).

Creating manually-crafted templates to generate code comments is one of the most common code summarization approaches. Sridhara et al. (2010) use the Software Word Usage Model (SWUM) to create a rule-based model that generates natural language descriptions for Java methods. Moreno et al. (2013) predefine heuristic rules to select information and generate comments for Java classes by combining the information. These rule-based approaches have been expanded to cover special types of code artifacts such as test cases (Zhang et al. 2011) and code changes (Buse and Weimer 2010). Human templates usually synthesize comments by extracting keywords from the given source code.

IR approaches are widely used in summary generation and usually search comments from similar code snippets. Haiduc et al. (2010b) apply the Vector Space Model (VSM) and Latent Semantic Indexing (LSI) to generate term-based comments for classes and methods. Their works are replicated and expanded by Eddy et al. (2013) which exploit a hierarchical topic model. Wong et al. (2015) apply code clone detection techniques to find similar code snippets and use the comments from similar code snippets. The work is similar to their previous work AutoComment (Wong et al. 2013) which mines human-written descriptions for automatic comment generation from Stack Overflow.

Recently, some studies try giving natural language summaries by deep learning approaches. Iyer et al. (2016) present RNN networks with attention to produce summaries that describe C# code snippets and SQL queries. It takes source code as plain text and models the conditional distribution of the summary. Allamanis et al. (2016) apply a neural convolutional attentional model to the problem that extremely summarizes the source code snippets into short, name-like summaries. Hu et al. (2018b) leverage the API sequence within the source code to assist the code comment generation. Their approach TL-CodeSum fuses the source code and the API sequences within it for better comment generation. Chen and Zhou (2018) propose a framework BVAE which uses two Variational AutoEncoders (VAEs) to model bimodal data: C-VAE for source code and L-VAE for natural language. BVAE learns vector representations for both code and description and generate completely new descriptions for arbitrary code snippets. These learning-based approaches mainly learn the latent features from source code, such as semantics, formatting, and etc. The comments are generated according to these learned features. The experimental results of them have proved the effectiveness of deep learning methods on code summarization. In this paper, Hybrid-DeepCom integrates the structure information which is verified important for comments generation.

8.2 Language Models for Source Code

Recently, thanks to the insight of Hindle et al. (2012), there is an emerging interest in building language models of source code. These language models vary from n-gram model (Nguyen et al. 2013; Allamanis et al. 2014), bimodal model (Allamanis et al. 2015b), and RNNs (Gu et al. 2016; Iyer et al. 2016). Hindle et al. (2012) first propose to explore N-gram to model the source code and demonstrate that most software is also natural and find regularities in natural code. Some studies build the models to bridge the gap between the programming language and natural language descriptions. Allamanis et al. (2014) develop a framework to learn the code conventions of a codebase and the framework exploits N-gram model to name Java identifiers. Allamanis et al. (2015a) and Raychev et al. (2015) suggest names for variables, methods, and classes. Mou et al. (2016) present a tree-based convolutional neural networks to model the source code and classify programs. Gu et al. (2016) present a classic encoder-decoder model to bridge the gap between the Java API sequences and natural language. Yin and Neubig (2017) build a data-driven syntax-based neural network model for generating code from natural language.

Learning from source code is applied to various software engineering tasks, e.g., fault detection (Ray et al. 2016), code completion (Nguyen et al. 2013; Mou et al. 2015), code clone (Svajlenko and Roy 2016) and code summarization (Iyer et al. 2016). In this paper, we explore the combination of deep learning methods and source code features to generate code comments. Compared to the previous works, Hybrid-DeepCom explains the code summarization procedure from a machine translation perspective. The experimental results also prove the ability of Hybrid-DeepCom to generate comments by learning lexical and structural information from source code.

9 Conclusion and Future Work

This paper formulates code summarization task as a machine translation problem which translates source code written in a programming language to comments in natural language. We point out two challenges while learning the source code, that is, the code structure learning and reducing out-of-vocabulary tokens. We propose Hybrid-DeepCom, a variant of attention-based Seq2Seq model, to generate comments for Java methods. Hybrid-DeepCom leverages both the source code and its AST structure to generate the code comment. These ASTs are converted to specially formatted sequences using a new structure-based traversal (SBT) method. SBT can express the structural information and keep the representation lossless at the same time. To reduce the out-of-vocabulary tokens, we split the identifier according to the camel casing naming convention. Hybrid-DeepCom outperforms the state-of-the-art approaches and achieves better results on both machine translation metrics and information retrieval metrics. In future work, we plan to improve the effectiveness of our proposed approach by introducing more domain-specific customizations. We also plan to apply our proposed approach to other software engineering tasks that can be mapped to a machine translation problem (e.g., code migration, etc.).

Acknowledgements This research is supported by the National Basic Research Program of China (the 973 Program) under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No.61620106007 and No. 61751210. Zhi Jin and Ge Li are corresponding authors.

References

Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 281–293

Allamanis M, Barr ET, Bird C, Sutton C (2015a) Suggesting accurate method and class names. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. ACM, pp 38–49

Allamanis M, Tarlow D, Gordon A, Wei Y (2015b) Bimodal modelling of source code and natural language. In: International conference on machine learning, pp 2123–2132

Allamanis M, Peng H, Sutton C (2016) A convolutional attention network for extreme summarization of source code. In: International conference on machine learning, pp 2091–2100

Allamanis M, Barr ET, Devanbu P, Sutton C (2017) A survey of machine learning for big code and naturalness. arXiv:170906182

Amann S, Nadi S, Nguyen HA, Nguyen TN, Mezini M (2016) Mubench: a benchmark for api-misuse detectors. In: 2016 IEEE/ACM 13Th working conference on mining software repositories, MSR. IEEE, pp 464–467

Bahdanau D, Cho K, Bengio Y (2014) Neural machine translation by jointly learning to align and translate. Comp Sci

Broy M, Deißenböck F, Pizka M (2005) A holistic approach to software quality at work. In: Proc. 3rd world congress for software quality (3WCSQ)

Buse RP, Weimer WR (2010) Automatically documenting program changes. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, pp 33–42

Chelba C, Bikel D, Shugrina M, Nguyen P, Kumar S (2012) Large scale language modeling in automatic speech recognition. arXiv:12108440

Chen B, Cherry C (2014) A systematic comparison of smoothing techniques for sentence-level bleu. In: Proceedings of the ninth workshop on statistical machine translation, pp 362–367

Chen Q, Zhou M (2018) A neural framework for retrieval and summarization of source code. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. ACM, pp 826–831

Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv:14061078

Chung J, Gulcehre C, Cho K, Bengio Y (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:14123555

Denkowski M, Lavie A (2014) Meteor universal: language specific translation evaluation for any target language. In: Proceedings of the ninth workshop on statistical machine translation, pp 376–380

Eddy BP, Robinson JA, Kraft NA, Carver JC (2013) Evaluating source code summarization techniques: replication and expansion. In: 2013 IEEE 21st international conference on program comprehension (ICPC). IEEE, pp 13–22

Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 631–642

Gu X, Zhang H, Zhang D, Kim S (2017) Deepam: migrate apis with multi-modal sequence to sequence learning. arXiv:170407734

Haiduc S, Aponte J, Marcus A (2010a) Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 2. ACM, pp 223–226

Haiduc S, Aponte J, Moreno L, Marcus A (2010b) On the use of automated text summarization techniques for summarizing source code. In: 2010 17th working conference on reverse engineering (WCRE). IEEE, pp 35–44

Hellendoorn VJ, Devanbu P (2017) Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. ACM, pp 763–773

Hindle A, Barr ET, 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 (2018a) Deep code comment generation. In: Proceedings of the 26th conference on program comprehension. ACM, pp 200–210

Hu X, Li G, Xia X, Lo D, Lu S, Jin Z (2018b) Summarizing source code with transferred api knowledge. In: IJCAI, pp 2269–2275

Iyer S, Konstas I, Cheung A, Zettlemoyer L (2016) Summarizing source code using a neural attention model. In: ACL (1)

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering. IEEE Press, pp 135–146

Klein G, Kim Y, Deng Y, Senellart J, Rush AM (2017) Opennmt: open-source toolkit for neural machine translation. arXiv:170102810

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

Leitner P, Bezemer CP (2017) An exploratory study of the state of practice of performance testing in java-based open source projects. In: Proceedings of the 8th ACM/SPEC on international conference on performance engineering. ACM, pp 373–384

Liu Z, Xia X, Hassan AE, Lo D, Xing Z, Wang X (2018) Neural-machine-translation-based commit message generation: how far are we? In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. ACM, pp 373–384

Loyola P, Marrese-Taylor E, Matsuo Y (2017) A neural architecture for generating natural language descriptions from source code changes. arXiv:170404856

McBurney PW, McMillan C (2014) Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd international conference on program comprehension. ACM, pp 279–290

Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: 2013 IEEE 21st international conference on program comprehension (ICPC). IEEE, pp 23–32

Mou L, Men R, Li G, Zhang L, Jin Z (2015) On end-to-end program generation from user intention by deep neural networks. arXiv:151007211

Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: AAAI, vol 2, p 4

Movshovitz-Attias D, Cohen WW (2013) Natural language models for predicting programming comments

Nguyen TD, Nguyen AT, Nguyen TN (2016) Mapping api elements for code migration with vector representations. In: 2016 IEEE/ACM 38Th international conference on software engineering companion, ICSE-C, IEEE, pp 756–758

Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN (2013) A statistical semantic language model for source code. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM, pp 532–542

Oda Y, Fudaba H, Neubig G, Hata H, Sakti S, Toda T, Nakamura S (2015) Learning to generate pseudo-code from source code using statistical machine translation (t). In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 574–584

Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting on association for computational linguistics. Association for Computational Linguistics, pp 311–318

Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu P (2016) On the naturalness of buggy code. In: Proceedings of the 38th international conference on software engineering. ACM, pp 428–439

Raychev V, Vechev M, Krause A (2015) Predicting program properties from big code. In: ACM SIGPLAN Notices, vol 50. ACM, pp 111–124

Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM international conference on automated software engineering. ACM, pp 43–52

Sridhara G, Pollock L, Vijay-Shanker K (2011) Automatically detecting and describing high level actions within methods. In: Proceedings of the 33rd international conference on software engineering. ACM, pp 101–110

Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: Advances in neural information processing systems, pp 3104–3112

Svajlenko J, Roy CK (2016) A machine learning based approach for evaluating clone detection tools for a generalized and accurate precision. Int J Softw Eng Knowl Eng 26(09n10):1399–1429

Wang S, Liu T, Tan L (2016) Automatically learning semantic features for defect prediction. In: Proceedings of the 38th international conference on software engineering. ACM, pp 297–308

White M, Tufano M, Vendome C, Poshyvanyk D (2016) Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM, pp 87–98

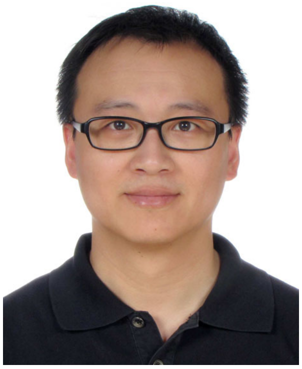
Wong E, Yang J, Tan L (2013) Autocomment: mining question and answer sites for automatic comment generation. In: Proceedings of the 28th IEEE/ACM international conference on automated software engineering. IEEE Press, pp 562–567

Wong E, Liu T, Tan L (2015) Clocom: mining existing source code for automatic comment generation. In: 2015 IEEE 22nd international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 380–389

Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K et al (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv:160908144

Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2017) Measuring program comprehension: a large-scale field study with professionals. IEEE Trans. Softw. Eng.
Yin J, Jiang X, Lu Z, Shang L, Li H, Li X (2015) Neural generative question answering. arXiv:151201337
Yin P, Neubig G (2017) A syntactic neural model for general-purpose code generation. arXiv:170401696
Zhang S, Zhang C, Ernst MD (2011) Automated documentation inference to explain failed tests. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering. IEEE Computer Society, pp 63–72

Xing Hu is currently a Ph.D. student in the School of Electronics Engineering and Computer Science, Peking University, China. Her research interests include source code analysis and deep learning in the field of software engineering.



Ge Li is an associate professor in the Department of Computer Science and Technology, School of EECS. He obtained his Ph.D from Peking University in 2006, and had been a visiting associate professor at Stanford University in 2013-2014. He is currently the deputy secretary general of CCF Software Engineering Society and the founder of the Software Program Generation Study Group. Dr. Ge Li was one of the earliest researchers engaged in the study of the computer program language model based on deep neural network, and the study of end-to-end program code generating techniques. His current research mainly concerns applications of probabilistic methods for machine learning, including program language process, natural language process, and software engineering.

Xin Xia is a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received both of his Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. More information at: https://xin-xia.github.io/.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has more than 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation and Lee Kong Chian Fellow for Research Excellence from the Singapore Management University in 2009 and 2018, and a number of international research and service awards including multiple ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).

Zhi Jin received the PhD degree in computer science from Changsha Institute of Technology, China, in 1992. She is currently a professor of computer science at Peking University. She is deputy director of Key Lab of High Confidence Software Technologies (Ministry of Education) at Peking University. Her research interests include software engineering, requirements Engineering, knowledge engineering, and machine learning. She is/was principle investigator of more than 10 national competitive grants, including the chief scientist of a national basic research project (973 project) of the Ministry of Science and Technology of China. She is currently a senior member of the IEEE, a standing board member of China Computer Federation (CCF), the director of CCF Technical Committee of Software Engineering and was elected to CCF fellow in 2012.

Affiliations

Xing Hu[1,2] · **Ge Li**[1,2] · **Xin Xia**[3] · **David Lo**[4] · **Zhi Jin**[1,2]

Xing Hu
huxing0101@pku.edu.cn

Xin Xia
xin.xia@monash.edu

David Lo
davidlo@smu.edu.sg

1 Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China

2 Institute of Software, EECS, Peking University, Beijing, China

3 Faculty of Information Technology, Monash University, Melbourne, Australia

4 School of Information Systems, Singapore Management University, Singapore, Singapore