The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Automatic Generation of Text Descriptive Comments for Code Blocks

Yuding Liang, Kenny Q. Zhu*
liangyuding@sjtu.edu.cn, kzhu@cs.sjtu.edu.cn
Department of Computer Science and Engineering
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai, China 200240

Abstract

We propose a framework to automatically generate descriptive comments for source code blocks. While this problem has been studied by many researchers previously, their methods are mostly based on fixed template and achieves poor results. Our framework does not rely on any template, but makes use of a new recursive neural network called Code-RNN to extract features from the source code and embed them into one vector. When this vector representation is input to a new recurrent neural network (Code-GRU), the overall framework generates text descriptions of the code with accuracy (Rouge-2 value) significantly higher than other learning-based approaches such as sequence-to-sequence model. The Code-RNN model can also be used in other scenario where the representation of code is required. [1]

1 Introduction

Real-world software development involves large source code repositories. Reading and trying to understand other people's code in such repositories is a difficult and unpleasant process for many software developers, especially when the code is not sufficiently commented. For example, if the Java method in Fig. 1 does not have the comment in the beginning, it will take the programmer quite some efforts to grasp the meaning of the code. However, with a meaningful sentence such as "calculates dot product of two points" as a descriptive comment, programmer's productivity can be tremendously improved.

```
/* Calculates dot product of two points.
 * @return float */
public static float ccpDot(final CGPoint v1, final
     CGPoint v2) {
   return v1.x * v2.x + v1.y * v2.y;
}
```

Figure 1: source code example

A related scenario happens when one wants to search for a piece of code with a specific functionality or meaning. Ordinary keyword search would not work because expressions in programs can be quite different from natural languages. If methods are annotated with meaningful natural language comments, then keyword matching or even fuzzy semantic search can be achieved.

Even though comments are so useful, programmers are not using them enough in their coding. Table 1 shows the number of methods in ten actively developed Java repositories from Github, and those of which annotated with a descriptive comment. On average, only 15.4% of the methods are commented.

To automatically generate descriptive comments from source code, one needs a way of accurately representing the semantics of code blocks. One potential solution is to treat each code block as a document and represent it by a topic distribution using models such as LDA (Blei, Ng, and Jordan 2003). However, topic models, when applied to source code, have several limitations:

- a topic model treats documents as a bag of words and ignores the structural information such as programming language syntax and function or method calls in the code;
- the contribution of lexical semantics to the meaning of code is exaggerated;
- comments produced can only be words but not phrases or sentences.

One step toward generating readable comments is to use templates (McBurney and McMillan 2014; Sridhara et al. 2010). The disadvantage is that comments created by templates are often very similar to each other and only relevant to parts of the code that fit the template. For example, the comment generated by McBurney's model for Fig. 1 is fairly useless: *"This method handles the ccp dot and returns a float. ccpDot() seems less important than average because it is not called by any methods."*

To overcome these problems, in this paper, we propose to use Recursive Neural Network (RNN) (Socher et al. 2011a; 2011b) to combine the semantic and structural information from code. Recursive NN has previously been applied to parse trees of natural language sentences, such as the example of two sentences in Fig. 2. In our problem, source codes can be accurately parsed into their parse trees, so recursive NN can be applied in our work readily. To this end, we design a new recursive NN called Code-RNN to extract the features from the source code.

[1] Kenny Q. Zhu is the contact author.

Table 1: Ten Active Projects on Github

Project	Description	# of bytes	# of Java Files	# of Methods	# Methods Commented
Activiti	a light-weight workflow and Business Process Management (BPM) Platform	168M	2939	15875	1080
aima-java	Java implementation of algorithms from "Artificial Intelligence - A Modern Approach"	182M	889	4078	1130
neo4j	the worlds leading Graph Database.	270M	4125	24529	1197
cocos2d	cocos2d for android, based on cocos2d-android-0.82	78M	512	3677	1182
rhino	a Java implementation of JavaScript.	21M	352	4610	1195
spring-batch	a framework for writing offline and batch applications using Spring and Java	56M	1742	7936	1827
Smack	an open source, highly modular, easy to use, XMPP client library written in Java	41M	1335	5034	2344
guava	Java-based projects: collections, caching, primitives	80M	1710	20321	3079
jersey	a REST framework that provides JAX-RS Reference Implementation and more.	73M	2743	14374	2540
libgdx	a cross-platform Java game development framework based on OpenGL (ES)	989M	1906	18889	2828

A comment here refers to the description at the beginning of a method, with more than eight words.

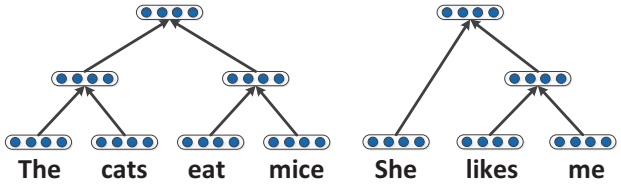

Figure 2: The Recursive Neural Networks of Two Sentences

Using Code-RNN to train from the source code, we can get a vector representation of each code block and this vector contains rich semantics of the code block, just like word vectors (Mikolov et al. 2013). We then use a Recurrent Neural Network to learn to generate meaningful comments. Existing recurrent NN does not take good advantage of the code block representation vectors. Thus we propose a new GRU (Cho et al. 2014) cell that does a better job.

In sum, this paper makes the following contributions:

- by designing a new Recursive Neural Network, *Code-RNN*, we are able to describe the *structural information* of source code;
- with the new design of a GRU cell, namely *Code-GRU*, we make the best out of code block representation vector to effectively generate comments for source codes;
- the overall framework achieves remarkable accuracy (Rouge-2 value) in the task of generating descriptive comments for Java methods, compared to state-of-the-art approaches.

2 Framework

In this section, we introduce the details of how to represent source code and how to use the representation vector of source code to generate comments.

2.1 Code Representation

We propose a new kind of recursive neural network called Code-RNN to encapsulate the critical structural information of the source code. Code-RNN is an arbitrary tree form while other recursive neural nets used in NLP are typically binary trees. Fig. 3 shows an example of Code-RNN for a small piece of Java code.

In Code-RNN, every parse tree of a program is encoded into a neural network, where the structure of the network is exactly the parse tree itself and each syntactic node in the parse tree is represented by a vector representation. [2] One unique internal node "CombineName" indicates a compound identifier that is the concatenation of several primitive words, for example, "allFound" can be split into "all" and "found". More on the semantics of identifier will be discussed later in this section.

There are two models for the Code-RNN, namely *Sum Model* and *Average Model*:

1. Sum Model

$$V = V_{node} + f(\mathbf{W} \times \sum_{c \in C} V_c + \mathbf{b}) \quad (1)$$

2. Average Model

$$V = V_{node} + f(\mathbf{W} \times \frac{1}{n} \sum_{c \in C} V_c + \mathbf{b}) \quad (2)$$

Here V is the vector representation of sub-tree rooted at N; V_{node} is the vector that represents the syntactic type of N itself, e.g., *IfStatement*; C is the set of all child nodes of N; V_c is the vector that represents a subtree rooted at c, one of N's children. During the training, W and b are tuned. V, V_{node} and V_c are calculated based on the structure of neural network. f is *RELU* activation function.

These equations are applied recursively, bottom-up through the Code-RNN at every internal node, to obtain the vector representation of the root node, which is also the vector of the entire code piece.

Identifier Semantics In this work, we adopt two ways to extract the semantics from the identifiers. One is to split all the long forms to multiple words and the other one is to recover the full words from abbreviations.

Table 2 shows some example identifiers and the results of splitting. Many identifiers in the source code are combination of English words, with the first letter of the word in upper case, or joined together using underscores. We thus define simple rules to extract the original English words accordingly. These words are further connected by the "CombineName" node in the code-RNN.

Table 3 shows some abbreviations and their intended meaning. We can infer the full-versions by looking for

[2] We use JavaParser from https://github.com/javaparser/javaparser to generate parse tree for Java code in this paper.

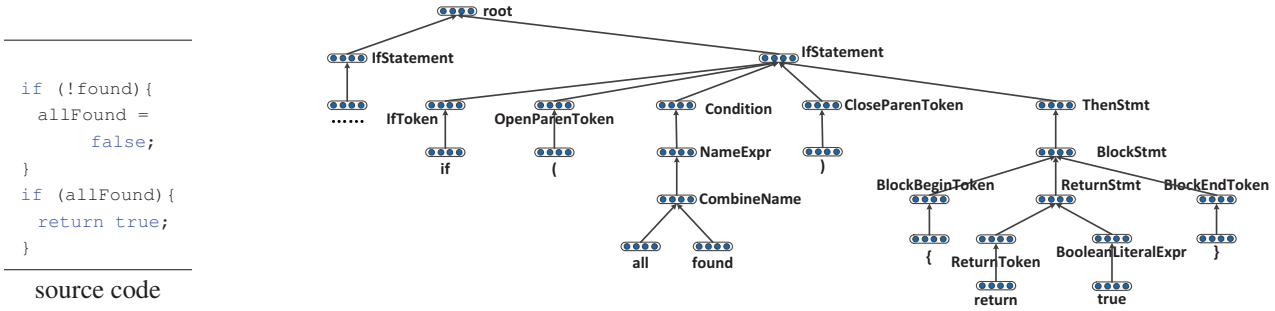

Figure 3: Code-RNN Example

longer forms in the context of the identifier in the code. Specifically, we compare the identifier with the word list generated from the context of the identifier to see whether the identifier's name is a substring of some word from the list, or is the combination of the initial of the words in the list. If the list contains only one word, we just check if the identifier is part of that word. If so, we conclude that the identifier is the abbreviation of that word with higher probability. If the list contains multiple words, we can collect all the initials of the words in the list to see whether the identifier is part of this collection. Suppose the code fragment is

```
Matrix dm = new DoubleMatrix(confusionMatrix);
```

We search for the original words of "dm" as follows. Since "dm" is not the substring of any word in the context, we collect the initials of the contextual words in a list: "m" "dm" and "cm". Therefore, "dm" is an abbreviation of "DoubleMatrix".

Table 2: Example of Split Identifiers

Identifier	Words
contextInitialize	context, initialize
apiSettings	api, settings
buildDataDictionary	build, data, dictionary
add_result	add, result

Table 3: Example of Abbreviation

Abbreviation	Origin	Context
val	value	key.value()
cm	confusion, matrix	new ConfusionMatrix()
conf	configuration	context.getConfiguration()
rnd	random	RandomUtils.getRandom()

Training Each source code block in the training data has a class label. Our objective function is:

$$\arg\min CrossEntropy(softmax(W_s V_m + b_s), V_{label}) \quad (3)$$

where V_m is the representation vector of source code, V_{label} is an one-hot vector to represent the class label. W_s and b_s are parameters for softmax function and will be tuned during training. We use AdaGrad (Duchi, Hazan, and Singer 2011) to apply unique learning rate to each parameter.

2.2 Comment Generation

Existing work (Elman 1990; Sutskever, Martens, and Hinton 2011; Mikolov et al. 2010) has used Recurrent Neural Network to generate sentences. However, one challenge to utilize the code block representation vector in Recurrent NN is that we can not feed the code block representation vector to the Recurrent NN cell directly. We thus propose a variation of the GRU based RNN. Fig. 4 shows our comment generation process.

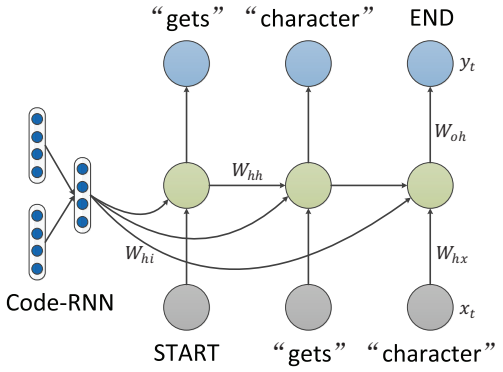

Figure 4: Comment Generation

We use pre-trained model Code-RNN to get the representation vector of the input code block V_m. This vector V_m is fixed during training of comment generation model. Then we feed code block vector into the RNN (Recurrent Neural Network) model at every step. For example in Fig. 4, we input the START token as the initial input of model and feed the code block vector into the hidden layer. After calculating the output of this step, we do the back-propagation. Then at step two, we input the word "gets" and feed the code block vector V_m into hidden layer again, and receive the h_{t-1} from the step one. We repeat the above process to tune all parameters. The equations of comment generation model are listed below.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \quad (4)$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \quad (5)$$
$$c_t = \sigma(W_c \cdot [h_{t-1}, x_t]) \quad (6)$$
$$\tilde{h_t} = tanh(W \cdot [r_t * h_{t-1}, c_t * V_m, x_t]) \quad (7)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t} \quad (8)$$
$$y_t = softmax(W_{oh} h_t + b_o) \quad (9)$$

where V_m is the code block representation vector, h_{t-1} is the previous state and x_t is the input word of this step.

To better use the code block vectors, our model differs from existing RNNs, particularly in the definition of c_t in the Equation 6 and 7. The new RNN cell, illustrated in Fig. 5, aims to strengthen the effect of code block vectors. This modified GRU is hereinafter called *Code-GRU*. Code block vector contains all information of code block but not all information is useful at all steps. Therefore, we add a new gate called choose gate to determine which dimension of code block vector would work in Code-GRU. In Fig 5, the left gate is the choose gate, and the other two gates are the same as the original GRU.

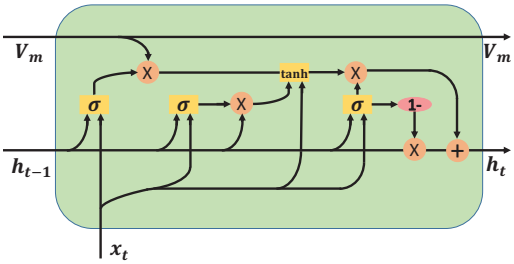

Figure 5: Structure of Code-GRU

During test time, we input the "START" token at first and choose the most probable word as the output. Then from the second step the input words of every step are the output words of previous one step until the output is "END" token. So that we can get an automatically generated comment for code blocks in our model.

To gain better results, we also apply the beam search while testing. We adopt a variant of beam search with a length penalty described in (Wu et al. 2016). In this beam search model, there are two parameters: beam size and weight for the length penalty. We tune these two parameters on the validation set to determine which values to use. Our tuning ranges are:

- beam size: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- weight for the length penalty: [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

3 Evaluation

Our evaluation comes in two parts. In the first part, we evaluate Code-RNN model's ability to classify different source code blocks into k known categories. In the second part, we show the effectiveness of our comment generation model by comparing with several state-of-the-art approaches in both quantitative and qualitative assessments. The source code of our approach as well as all data set is available at https://adapt.seiee.sjtu.edu.cn/CodeComment/.

3.1 Source Code Classification

Data Set The goal is to classify a given Java method (we only use the body block without name and parameters) into a predefined set of classes depending on its functionality. Our data set comes from the Google Code Jam contest (2008∼2016), which there are multiple problems, each associated with a number of correct solutions contributed by programmers. [3] Each solution is a Java method. The set of solutions for the same problem are considered to function identically and belong to the same class in this work. We use the solutions (10,724 methods) of 6 problems as training set and the solutions (30 methods) of the other 6 problems as the test set. Notice that the problems in the training data and the ones in the test data do not overlap. We specifically design the data set this way because, many methods for the same problem tend to use the same or similar set of identifiers, which is not true in real world application. The details of training set and test set are shown in Table 4.

Table 4: Data Sets for Source Code Clustering

	Problem	Year	# of methods
Training Set	Cookie Clicker Alpha	2014	1639
	Counting Sheep	2016	1722
	Magic Trick	2014	2234
	Revenge of the Pancakes	2016	1214
	Speaking in Tongues	2012	1689
	Standing Ovation	2015	2226
Test Set	All Your Base	2009	5
	Consonants	2013	5
	Dijkstra	2015	5
	GoroSort	2011	5
	Osmos	2013	5
	Part Elf	2014	5

Baselines We compare Code-RNN with two baseline approaches. The first one is called language embedding (*LE*) and only treats the source code as a sequence of words, minus the special symbols (e.g., "$", "(", "+", $\cdots$). All concatenated words are preprocessed into primitive words as previously discussed. Then the whole code can be represented by either the sum (*LES*) or the average (*LEA*) of word vectors of this sequence, trained in this model.This approach basically focuses on the word semantics only and ignores the structural information from the source code.

The second baseline is a variant of Code-RNN, which preprocesses the code parse tree by consistently replacing the identifier names with placeholders before computing the overall representation of the tree. This variant focuses on the structural properties only and ignores the word semantics.

[3] All solutions are available at http://www.go-hero.net/jam/16.

Result of Classification At test time, when a method is classified into a class label, we need to determine which test problem this class label refers to. To that end, we compute the accuracy of classification for all possible class label assignment and use the highest accuracy as the one given by a model.

Table 5 shows the purity of the produced classes, the F1 and accuracy of the 6-class classification problem by different methods. It is clear that Code-RNN (avg) perform better uniformly than the baselines that use only word semantics or only structural information. Therefore, in the rest of this section, we will use Code-RNN(avg) model to create vector representation for a given method to be used for comment generation. The F1 score for each individual problem is also included in Table 6.

Table 5: Purity, Average F1 and Accuracy

	Purity	F1	Accuracy
LEA	0.400	0.3515	0.3667
LES	0.3667	0.2846	0.3667
CRA(ni)	0.4667	0.4167	0.4667
CRS(ni)	0.4667	0.4187	0.4667
CRA	**0.533**	**0.4774**	**0.5**
CRS	0.4667	0.3945	0.4333

LEA = Language Embedding Average model; LES = Language Embedding Sum model; CRA = Code-RNN Average model; CRS = Code-RNN Sum model; (ni) = no identifier.

Table 6: F1 scores of individual problems

	Dijkstra	Part Elf	All Your Base	GoroSort	Consonants	Osmos
LEA	0.25	0.33	0.43	0.33	0.4	0.36
LES	0.33	0	0.53	0	0.53	0.31
CRA(ni)	0.6	0	0.44	0.40	0.56	0.5
CRS(ni)	0.62	0.29	0.67	0.5	0.44	0
CRA	0.67	0	0.6	0.57	0.53	0.52
CRS	0.73	0	0.44	0.55	0.4	0.25

3.2 Comment Generation Model

Data Set We use ten open-source Java code repositories from GitHub for this experiment (see Table 1). In each of these repositories we extract descriptive comment and the corresponding method pairs. Constructor methods are excluded from this exercise. These pairs are then used for training and test. Notice that all the method names and parameters are excluded from training and test.

Baselines We compare our approach with four baseline methods.

- *Moses*[4] is a statistical machine translation system. We regard the source codes as the source language and the comments as the target, and use Moses to translate from the source to the target.
- *CODE-NN* (Iyer et al. 2016) is the first model to use neural network to create sentences for source code. In this model author used LSTM and attention mechanism to generate sentences. The original data set for CODE-NN are StackOverFlow thread title and code snippet pairs. [5]. In this experiment, we use the comment-code pair data in place of the title-snippet data.
- We apply the *sequence-to-sequence (seq2seq)* model used in machine translation (Britz et al. 2017) and treat the code as a sequence of words and the comment as another sequence.
- A. Karpathy and L. Fei-Fei (Karpathy and Fei-Fei 2015) proposed a meaningful method to generate image descriptions. It also used Recurrent NN and representation vector, so we apply this method to comment generation model. The main equations are:

$$b_v = W_{hi} V_m \tag{10}$$
$$h_t = f(W_{hx} x_t + W_{hh} h_{t-1} + b_h + b_v) \tag{11}$$
$$y_t = softmax(W_{oh} h_t + b_o) \tag{12}$$

  where $W_{hi}, W_{hx}, W_{hh}, W_{oh}, x_i$ and b_h, b_o are parameters to be learned, and V_m is the method vector. We call this model *Basic RNN*.

Moses and CODE-NN has its own terminate condition. Seq2Seq, Basic RNN and our model run 800 epochs during training time. For one project, we separate the commented methods into three parts: training set, validation set and test set. We tune the hyper parameter on the validation set. The results of ten repositories are shown in Table 7.

Evaluation Metric We evaluate the quality of comment generation by the Rouge method(Lin 2004). Rouge model counts the number of overlapping units between generated sentence and target sentence. We choose Rouge-2 score in this paper where word based 2-grams are used as the unit, as it is the most commonly used in evaluating automatic text generation such as summarization.

Table 7: Rouge-2 Values for Different Methods

	neo4j	cocos2d	jersey	aima-guava java	Smack	Activiti	spring-batch	libgdx	rhino
MOSES	0.076	0.147	0.081	0.1440.134	0.145	0.104	0.147	0.212	0.082
CODE-NN	0.077	0.136	0.105	0.1240.153	0.135	0.103	0.184	0.208	**0.171**
Seq2seq	0.039	0.115	0.183	0.1080.152	0.109	0.158	0.171	**0.247**	0.169
Basic RNN*	0.133	0.152	0.214	0.2070.156	0.150	**0.203**	**0.237**	0.218	0.163
Code-GRU*	**0.141**	**0.158**	**0.230**	**0.2090.164**	**0.162**	0.200	0.213	0.233	0.165

*: both models use the method representation vector from Code-RNN.

Examples of Generated Comment Fig. 6 shows the comments generated by the competing methods for three example Java methods coming from different repositories. Because we delete all punctuation from the training data, the generated comments are without punctuation. Nonetheless, we can see that comments by our Code-GRU model are generally more readable and meaningful.

[4]Home page of Moses is http://www.statmt.org/moses/.

[5]Data comes from https://stackoverflow.com/. Source code of CODE-NN is available from https://github.com/sriniiyer/codenn.

```
Project: jersey
   public long tryConvertToReadLock(long stamp) {
      long a = stamp & ABITS, m, s, next; WNode h;
      while (((s = state) & SBITS) == (stamp & SBITS)) {
         if ((m = s & ABITS) == 0L) {
            if (a != 0L) break;
            else if (m < RFULL) {
               if (U.compareAndSwapLong(this, STATE, s,
                    next = s + RUNIT))
                  return next;}
            else if ((next = tryIncReaderOverflow(s)) !=
                 0L) return next;}
         else if (m == WBIT) {
            if (a != m) break;
            state = next = s + (WBIT + RUNIT);
            if ((h = whead) != null && h.status != 0)
               release(h);
            return next;}
         else if (a != 0L && a < WBIT) return stamp;
         else break;}
      return 0L;}
```

Gold	if the lock state matches the given stamp performs one of the following actions if the stamp represents holding a write lock releases it and UNK a read lock or if a read lock returns it or if an optimistic read acquires a read lock and returns a read stamp only if immediately available this method returns zero in all other cases
MOSES	if the lock state matches the given if the lock state matches the given gets of processing sbits state matches the given sbits string of the lock hold abits l l break component of rfull that runs sets of processing of runit or create a new pattern if this inc reader overflow l create a human readable description of component of wbit break if the lock of processing of wbit runit h whead by the given status release the given action if the sum associated with the given component l lock state matches the given action wbit get returns break l
CODE-NN	returns code true if the lock is not a link org glassfish jersey server mvc
Seq2Seq	UNK a new item to the list of superclass timeout version
Basic RNN*	get a UNK to a link javax ws rs core UNK
Code-GRU*	if the lock state matches the given stamp performs one of the following actions if the stamp represents holding a write lock returns it or if a read lock if the write lock is available releases the read lock and returns a write stamp or if an optimistic read returns

```
project: cocos2d
   public static float ccpDot(final CGPoint v1, final
        CGPoint v2) {
      return v1.x * v2.x + v1.y * v2.y; }
```

Gold	Calculates dot product of two points
MOSES	subtract another subtract another the given vector
CODE-NN	rotates two points
Seq2Seq	returns the closest long to the specified value
Basic RNN*	calculates cross product of two points
Code-GRU*	calculates cross product of two points

```
project: libgdx
   public static IntBuffer allocate (int capacity) {
     if (capacity < 0) {
      throw new IllegalArgumentException();
     }
     return BufferFactory.newIntBuffer(capacity);}
```

Gold	creates an int buffer based on a newly allocated int array
MOSES	based on the creates a new backing buffer
CODE-NN	creates a byte buffer based on a newly allocated char array
Seq2Seq	creates a float buffer based on a newly allocated float array
Basic RNN*	creates a char buffer based on a newly allocated char array
Code-GRU*	creates a long buffer based on a newly allocated long array

Figure 6: Examples of generated comments and corresponding code snippets

In the first example, we can see that CODE-NN, Seq2Seq and Basic RNN's results are poor and have almost nothing to do with the Gold comment. Even though both MOSES produces a sequence of words that look similar to the Gold in the beginning, the rest of the result is less readable and does not have any useful information. For example, "if the lock state matches the given" is output repeatedly. MOSES also produces strange terms such as "wbit" and "runit" just because they appeared in the source code. In the contrast, Code-GRU's result is more readable and meaningful.

In the second example, there is not any useful word in the method body so the results of MOSES, CODE-NN and Seq2Seq are bad. Code-RNN can extract the structural information of source code and embed it into a vector, so both models that use this vector, namely Basic RNN and Code-GRU, can generate the relevant comments.

In the third example, although all results change the type of the value, that is, Basic RNN changes "int" to "char" while Code-GRU changes to "long". "long" and "int" are both numerical types while "char" is not. Thus Code-GRU is better than Basic RNN. For the result of Seq2Seq, although "float" is also a numerical type, it is for real numbers, and not integers.

4 Related Work

Mining of source code repositories becomes increasingly popular in recent years. Existing work in source code mining include code search, clone detection, software evolution, models of software development processes, bug localization, software bug prediction, code summarization and so on. Our work can be categorized as code summarization and comment generation.

Sridhara et al. (Sridhara et al. 2010) proposed an automatic comment generator that identifies the content for

the summary and generates natural language text that summarizes the methods overall actions based on some templates. Moreno et al. (Moreno et al. 2013) also proposed a template based method but it is used on summarizing Java classes. McBurney and McMillan (McBurney and McMillan 2014) presented a novel approach for automatically generating summaries of Java methods that summarize the context surrounding a method, rather than details from the internals of the method. These summarization techniques (Murphy 1996; Sridhara, Pollock, and Vijay-Shanker 2011; Moreno et al. 2013; Haiduc et al. 2010) work by selecting a subset of the statements and keywords from the code, and then including information from those statements and keywords in the summary. To improve them, Rodeghero et al. (Rodeghero et al. 2014) presented an eye-tracking study of programmers during source code summarization, a tool for selecting keywords based on the findings of the eye-tracking study.

These models are invariably based on templates and careful selection of fragments of the input source code. In contrast, our model is based on learning and neural network. There are also some models that apply learning methods to mine source code.

Movshovitz-Attias and Cohen (Movshovitz-Attias and Cohen 2013) predicted comments using topic models and n-grams. Like source code summarization, Allamanis et al. (Allamanis et al. 2015) proposed a continuous embedding model to suggest accurate method and class names.

Iyer et al. (Iyer et al. 2016) proposed a new model called CODE-NN that uses Long Short Term Memory (LSTM) networks with attention to produce sentences that can describe C# code snippets and SQL queries. Iyer et al.'s work has strong performance on two tasks, code summarization and code retrieval. This work is very similar to our work, in that we both use the Recurrent NN to generate sentences for source code. What differs is that we propose a new type of Recurrent NN. Adrian et al. (Kuhn, Ducasse, and Gírba 2007) utilized the information of identifier names and comments to mine topic of source code repositories. Punyamurthula (Punyamurthula 2015) used call graphs to extract the metadata and dependency information from the source code and used this information to analyze the source code and get its topics.

In other related domains of source code mining, code search is a popular research direction. Most search engines solve the problem by keyword extraction and signature matching. Maarek et al. (Maarek, Berry, and Kaiser 1991) used keywords extracted from man pages written in natural language and their work is an early example of approaches based on keywords. Rollins and Wing (Rollins and Wing 1991) proposed an approach to find code with the signatures present in code. Mitchell (Mitchell 2008) combined signature matching with keyword matching. Then Garcia et al. (Garcia-Contreras, Morales, and Hermenegildo 2016) focused on querying for semantic characteristics of code and proposed a new approach which combines semantic characteristics and keyword matching.

Cai (Cai 2016) proposed a method for code parallelization through sequential code search. That method can also be used for clone detection. Williams and Hollingsworth (Williams and Hollingsworth 2005) described a method to use the source code change history of a software project to drive and help to refine the search for bugs. Adhiselvam et al. (Adhiselvam, Kirubakaran, and Sukumar 2015) used MRTBA algorithm to localize bug to help programmers debug. The method proposed in this paper can also benefit natural language search for code fragments.

5 Conclusion

In this paper we introduce a new Recursive Neural Network called *Code-RNN* to extract the topic or function of the source code. This new Recursive Neural Network is the parse tree of the source code and we go through all the tree from leaf nodes to root node to get the final representation vector. Then we use this vector to classify the source code into some classes according to the function, and classification results are acceptable. We further propose a new kind of GRU called *Code-GRU* to utilize the vector created from Code-RNN to generate comments. We apply Code-GRU to ten source code repositories and gain the best result in most projects. This frame work can also be applied to other programming languages as long as we have access to the parse tree of the input program.

As future work, we can add call graphs into our model, so that Code-RNN can contain invocation information and extract more topics from source code.

Acknowledgement

This work was supported by Oracle-SJTU Joint Research Scheme, NSFC Grant No. 9164620571421002 and 61373031, and SJTU funding project 16JCCS08. Hongfei Hu contributed to the identifier semantics part of the work.

References

Adhiselvam, A.; Kirubakaran, E.; and Sukumar, R. 2015. An enhanced approach for software bug localization using map reduce technique based apriori (mrtba) algorithm. *Indian Journal of Science and Technology* 8(35).

Allamanis, M.; Barr, E. T.; Bird, C.; and Sutton, C. 2015. Suggesting accurate method and class names. In *ESEC/FSE*, 38–49. ACM.

Blei, D. M.; Ng, A. Y.; and Jordan, M. I. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3(Jan):993–1022.

Britz, D.; Goldie, A.; Luong, T.; and Le, Q. 2017. Massive Exploration of Neural Machine Translation Architectures. *ArXiv e-prints*.

Cai, B. 2016. Code parallelization through sequential code search. In *ICSE-C*, 695–697. ACM.

Cho, K.; Van Merriënboer, B.; Bahdanau, D.; and Bengio, Y. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimiza-

tion. *Journal of Machine Learning Research* 12(Jul):2121–2159.

Elman, J. L. 1990. Finding structure in time. *Cognitive science* 14(2):179–211.

Garcia-Contreras, I.; Morales, J. F.; and Hermenegildo, M. V. 2016. Semantic code browsing. *arXiv preprint arXiv:1608.02565*.

Haiduc, S.; Aponte, J.; Moreno, L.; and Marcus, A. 2010. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, 35–44. IEEE.

Iyer, S.; Konstas, I.; Cheung, A.; and Zettlemoyer, L. 2016. Summarizing source code using a neural attention model. In *ACL*, 2073–2083.

Karpathy, A., and Fei-Fei, L. 2015. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3128–3137.

Kuhn, A.; Ducasse, S.; and Gírba, T. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49(3):230–243.

Lin, C.-Y. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out: Proceedings of the ACL-04 workshop*, volume 8. Barcelona, Spain.

Maarek, Y. S.; Berry, D. M.; and Kaiser, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering* 17(8):800–813.

McBurney, P. W., and McMillan, C. 2014. Automatic documentation generation via source code summarization of method context. In *ICPC*, 279–290. ACM.

Mikolov, T.; Karafiát, M.; Burget, L.; Cernockỳ, J.; and Khudanpur, S. 2010. Recurrent neural network based language model. In *Interspeech*, volume 2, 3.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119.

Mitchell, N. 2008. Hoogle overview. *The Monad. Reader* 12:27–35.

Moreno, L.; Aponte, J.; Sridhara, G.; Marcus, A.; Pollock, L.; and Vijay-Shanker, K. 2013. Automatic generation of natural language summaries for java classes. In *ICPC*, 23–32. IEEE.

Movshovitz-Attias, D., and Cohen, W. W. 2013. Natural language models for predicting programming comments.

Murphy, G. C. 1996. *Lightweight structural summarization as an aid to software evolution*. Ph.D. Dissertation.

Punyamurthula, S. 2015. *Dynamic model generation and semantic search for open source projects using big data analytics*. Ph.D. Dissertation, Faculty of the University Of Missouri-Kansas City in partial fulfillment Of the requirements for the degree MASTER OF SCIENCE By SRAVANI PUNYAMURTHULA B. Tech, Jawaharlal Nehru Technological University.

Rodeghero, P.; McMillan, C.; McBurney, P. W.; Bosch, N.; and D'Mello, S. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE*, 390–401. ACM.

Rollins, E. J., and Wing, J. M. 1991. Specifications as search keys for software libraries. In *ICLP*, 173–187. Citeseer.

Socher, R.; Lin, C. C.; Manning, C.; and Ng, A. Y. 2011a. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 129–136.

Socher, R.; Pennington, J.; Huang, E. H.; Ng, A. Y.; and Manning, C. D. 2011b. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*, 151–161. Association for Computational Linguistics.

Sridhara, G.; Hill, E.; Muppaneni, D.; Pollock, L.; and Vijay-Shanker, K. 2010. Towards automatically generating summary comments for java methods. In *ASE*, 43–52. ACM.

Sridhara, G.; Pollock, L.; and Vijay-Shanker, K. 2011. Generating parameter comments and integrating with method summaries. In *ICPC*, 71–80. IEEE.

Sutskever, I.; Martens, J.; and Hinton, G. E. 2011. Generating text with recurrent neural networks. In *ICML*, 1017–1024.

Williams, C. C., and Hollingsworth, J. K. 2005. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering* 31(6):466–480.

Wu, Y.; Schuster, M.; Chen, Z.; Le, Q. V.; 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 preprint arXiv:1609.08144*.