A Neural-Network based Code Summarization Approach by Using Source Code and its Call Dependencies

Bohong Liu*
513925617@qq.com
National University of Defense Technology
Changsha, Hunan, China

Tao Wang
taowang2005@nudt.edu.cn
National University of Defense Technology
Changsha, Hunan, China

Xunhui Zhang
zhangxunhui@nudt.edu.cn
National University of Defense Technology
Changsha, Hunan, China

Qiang Fan
fanqiang09@nudt.edu.cn
National University of Defense Technology
Changsha, Hunan, China

Gang Yin
jack_nudt@163.com
National University of Defense Technology
Changsha, Hunan, China

Jinsheng Deng
jsdeng@nudt.edu.cn
National University of Defense Technology
Changsha, Hunan, China

ABSTRACT

Code summarization aims at generating natural language abstraction for source code, and it can be of great help for program comprehension and software maintenance. The current code summarization approaches have made progress with neural-network. However, most of these methods focus on learning the semantic and syntax of source code snippets, ignoring the dependency of codes. In this paper, we propose a novel method based on neural-network model using the knowledge of the call dependency between source code and its related codes. We extract call dependencies from the source code, transform it as a token sequence of method names, and leverage the Seq2Seq model for code summarization using the combination of source code and call dependency information. About 100,000 code data is collected from 1,000 open source Java proejects on github for experiment. The large-scale code experiment shows that by considering not only the code itself but also the codes it called, the code summarization model can be improved with the BLEU score to 33.08.

CCS CONCEPTS

• **Software and its engineering** → **General programming languages**.

KEYWORDS

Code Summarization, Neural Network, Open Source, Call Dependency

ACM Reference Format:
Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. 2019. A Neural-Network based Code Summarization Approach by Using Source Code and its Call Dependencies. In *Internetware '19: Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware '19), October 28–29, 2019, Fukuoka, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3361242.3362774

1 INTRODUCTION

The rapid growth of open source projects provide large number of reusable components for software development [24]. However, a large proportion of OSS projects are lack of natural language comments. For example, the comment rate for the popular deep learning framework Tensorflow is only 9.2%, and that for Hadoop is 13.1%. This may result in obstacles for software comprehension and increase the cost for the future maintenance [7]. Automatic code summary generation is a hot topic in this field.

Traditional automatic summary techniques are based on Information Retrieval [8, 10, 17].In recent years, machine learning methods are leveraged to solve this problem. The basic idea is regarding summarizing codes process as language translation tasks, view the source code and its structure information as input and translate source codes into natural language with its functional description [2, 3, 13]. However, most of these methods mainly focus on the first level of code structure, ignoring the valuable information about call graph of source codes.

"I set the brake up by connecting up rod and lever. - Yes, given the whole of the rest of the mechanism." [23]. For programming language, the situation is the same. Programmers assign meaning to variable names, functions and classes following the basic rules of programming language. A function is completed, that means not only the codes of this function is done, but also all codes that called by it must have been completed.

For instance, if we want to implement the function *replace()*, the method invocations in it such as *substitute()* and *toString()* must be implemented first. In other words, programming language has different layers. To implement a high level function, the basic levels it used must be implemented first.

```
/* Replaces all the occurrences of variables with
     their matching values from the resolver using the
      given source string as a template.*/

source code:
public String replace(final LogEvent event, final
     String source, final int offset, final int length
     )
{
    if (source == null) {  return null; }
    final StringBuilder buf = new StringBuilder(length
         ).append(source, offset, length);
  if (!substitute(event, buf, 0, length)) {
      return source.substring(offset, offset + length)
          ;
       }
  return buf.toString();
}

related code:
String toString(){
    return "StrSubstitutor(" + variableResolver.
         toString() + ")";
}
```

Code summarization is targeted at finding the meaning of code snippets. Exploiting the meaning of related codes called by source code can serve the target. The code summarization showed above is "*Replaces all the occurrences of variables with their matching values from the resolver using the given source string as a template*". However, one cannot get any information about "*variables with their matching values from the resolver*" by only analysing the source code in *replace()*. There are no such method names or variable names related to it. If the information about the related code *toString()* can be taken into consideration, it can be inferred that *variableResolver.toString()* makes contribution to generating "*variables with their matching values from the resolver*".

One of the reasons why previous comment generation methods do not consider related calls is that they only save the *(codes,annotation)* pairs when collecting data. They can neither know the position of the code in a project nor the relationship between these code snippets when processing these data. We redesign the data collection processing by taking care of the call relations between codes.

In this paper, we design a new code summarization approach based on neural-network algorithm, which employs both the source code information and call dependencies[1].

We implement a tool to extract JAVA call dependencies of codes based on JAVA Abstract Syntax Tree(AST) analysis [21]. The method invocation names between caller and callees are extracted as the representation of call dependency information. Then the call dependency information is used to assist the generating from source code to natural language description

[1]Data and code are available at https://github.com/yorhaz40/CallNN

Our experiments are based on a large-scale JAVA open source repository for more than 1,000 projects which are collected from github. We extract 100,000 *(code, call dependency, comment)* tuples from the repository as data. Divide them with the proportion of 8:1:1 as train, validation and test data. The experiment results show that call dependency information can improve the effect of the model and make contribution to code summarization task.

The contributions of our work are as follows:

- We propose a new approach to summarize codes with call dependency information based on neural-network sequence model.
- We implement a tool that can extract call dependency on source code within project.
- Extensive experiments are conducted on more than 1,000 projects. The results suggest the effectiveness of our approach.

2 RELATED WORKS

With the development of software, many studies have focused on the summarization of software artifacts including source code, commit messages, bug reports ad etc. [18] Besides the software artifacts above, most of the researchers focus on the summarization of source code. It is directly related to the software development and reuse.

Traditional summarization approaches are based on information retrieval(IR) methods. Bacchelli et al. [5] linked emails and source code by generating technical term vectors based on vector space model. Haiduc et al. [10] generated text summaries based on technical terms and latent semantic indexing method. Though IR-based approaches are easy and effective, they are limited by the text information. It is difficult to generate suitable summaries if there is not much natural language regarding to the source code.

Recently, the development of natural machine translation brings new breakthroughs in code summarization area. These machine translation models are based on the model called seq2seq, which aims to encode an input sequence of tokens to a vector and decode it to generate target sequence [6, 11]. A way to use this model in code summarization is to substitute input sequence with code sequence, and target the natural description of codes as output. The basic hypothesis is that some hidden information exists between source codes and its summarization. The model learns hidden information on source code with encoder structure, then decodes the information to generate summarization [15]. By using machine learning model, code summarization tools can know the meaning of codes.

Programming language code has a huge difference between natural language. It should be structured, made no confuse on semantics, be analyzed by parser and executed by computer [1]. Some works are done to add extended information to fit model on code-to-nl situation. DeepCom [13] proposes an approach to analyze structural information on Java methods for better generation. TL-CodeSum [14] and DeepAPI [9] captures the API knowledge in the source code to help model training. Code2seq implements an other way to present codes as vectors using a simple neural network. They treat the codes as a set of structured paths. Their model focuses on the structural information about the codes [4].

All these machine learning based methods try to explore useful information contained only in the source code. However, they ignore the fact that the function of source code is relying on other codes dependent by the source code for programming language. The paper pays attention on exploiting useful information contained in call dependencies on codes to assist generating code summarization.

3 OUR APPROACH

This section gives a description of the details of our approach.

3.1 Framework of Code Summarization Approach

In this paper, we propose a code summarization method based on neural-network model considering both source codes and call dependencies. We extract call dependency from source codes, transform it as a token sequence of method names, then leverage the Seq2Seq model for code summarization using the combination of source code and call dependency information.

As shown in Figure 1, the whole process can be divided into three parts: data processing, model training and summary generation with trained model. The main process of these three steps are as follows:

- **Data Processing:** Extract call dependency between source code and its related codes. Present the call dependency as a sequence of tokens.
- **Model Training:** Set up two encoders, one for the source codes, one for the call dependency sequence. Integrate the two outputs and decode them to the target natural language summarization.
- **Code Summary Generation:** As for a new snippet of code, extract its call dependency first. Then put the source code and its call dependency sequence in the trained model to generate its code summarization.

The granularity of code snippets can be alternative, such as a class,a method or a statement. The method level codes are always not too long. They have their own independent function that can be described as natural language. So we choose method level JAVA codes as our source data in this paper.

The key idea of this paper is to use call dependencies as an assistance to generate code summarization. To achieve this goal, there are two major tasks should be depicted in detail. First, how to extract call dependencies on a given code snippet and how to present them. Second, how to implement the model to make use of the call dependency information to train the model.

Algorithm 1 Call Dependency Extraction Process

```
Input: Java method $J$, source file $F_s$, related files $F_r$
Output: sequence of call dependency $C_{seq}$
  Build Java abstract syntax tree from method $J$, source file $F_s$ and its related files $F_r$.
  Get the AST of $J$ as $J_t$
  Get the set of ASTs of all related files in $F_r$ as $F_{rt}$
  Get the AST of $F_s$ as $F_{st}$
  Generate a variance pool $V_p$ from $F_r$
  Generate a Java method declaration pool $M_p$ from $F_r$
  Extract method invocations $M_i$ from $J$
  for $m$ in $M_i$ do
      add $m$ to call dependency list $C_{seq}$
      Try to find $m$ in $M_p$ and use $V_p$ to check the parameter of $m$
      if $m$ is found then
          Check whether it is a loop call
          if $m$ is not a loop call then
              Do Call Dependency Extraction Process with $m$, its source file and its relatd files
              Merge the result $C_{seq'}$ from Call Dependency Extraction Process with $C_{Seq}$
          end if
      end if
  end for
  Return $C_{seq}$
```

3.2 Call dependency extraction

The call dependency extraction tool is based on JAVA abstract syntax tree [19]. It is a tree structure representation of source code abstract syntax, whose leaf nodes are source code tokens and non-leaf nodes are types of the tokens. With the help of AST, we can extract method invocations from given codes.

The call dependency extraction process can be demonstrated as Alg 1. It is a recursive algorithm calling itself at "*Do Call Dependency Extraction Process with m, its source file and its related files*". The algorithm takes a Java method as input to analyse. To get to know what methods are called by J, J's source file and its related files must be considered.

The files in a project's folder are regarded as related files. These files do not include third parted library calls and system calls. They are difficult to get by only using static analysis method. Besides, there are also call relations between third parted libraries. It can make the third parted library analysis expand to a huge scale. So our tool have not considered them yet. All method declarations of source file and related files are included in the method pool M_p. It contains all method declarations with their body codes. It is prepared for searching call dependency. We can directly get the called codes when a method invocation in source code is also found in this pool. The variance pool V_p records variance names and their types. It is used to identify function overloading when two methods have the same name. The algorithm stops when it cannot find any method in its method pool. To avoid

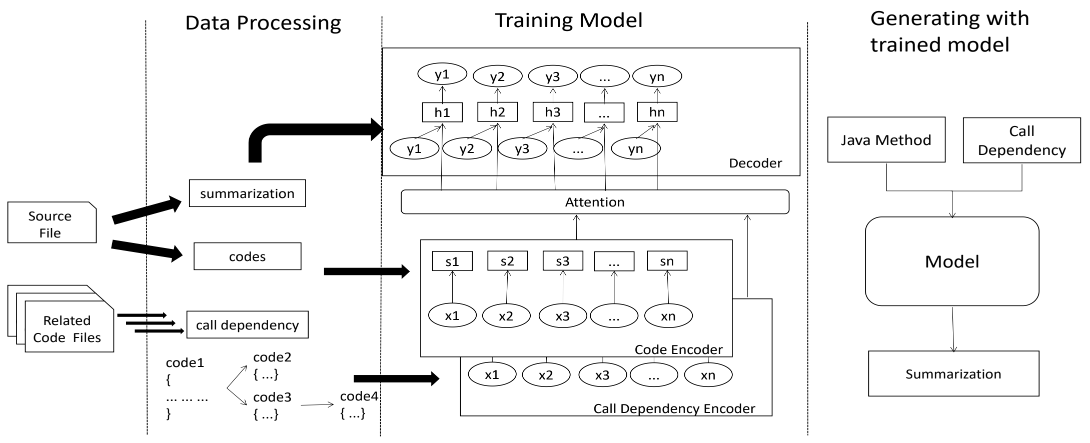
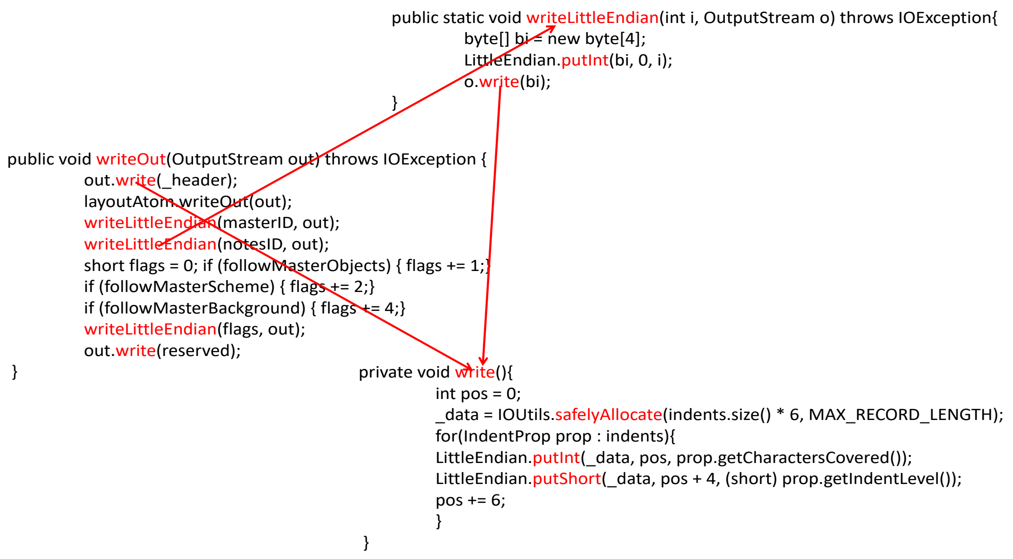


Figure 1: Structure of CallNN



```
public static void writeLittleEndian(int i, OutputStream o) throws IOException{
        byte[] bi = new byte[4];
        LittleEndian.putInt(bi, 0, i);
        o.write(bi);
}

public void writeOut(OutputStream out) throws IOException {
        out.write(_header);
        layoutAtom.writeOut(out);
        writeLittleEndian(masterID, out);
        writeLittleEndian(notesID, out);
        short flags = 0; if (followMasterObjects) { flags += 1;}
        if (followMasterScheme) { flags += 2;}
        if (followMasterBackground) { flags += 4;}
        writeLittleEndian(flags, out);
        out.write(reserved);
}

private void write(){
        int pos = 0;
        _data = IOUtils.safelyAllocate(indents.size() * 6, MAX_RECORD_LENGTH);
        for(IndentProp prop : indents){
        LittleEndian.putInt(_data, pos, prop.getCharactersCovered());
        LittleEndian.putShort(_data, pos + 4, (short) prop.getIndentLevel());
        pos += 6;
        }
}
```

Figure 2: Call dependency of *WriteOut()*

endless loop caused by circular call between codes, "*Check whether it is a loop call*" must be done. We use a loop call pool to save the methods that have been called. A method has been analysed will not be analysed again.

C_{seq} is the result call dependency which is represented by a sequence of method names that invoked by the source code. To keep the structure and layer information about call dependency sequence, we format the C_{seq} as $methodname(call_1, call_2, (\dots), call_3, \dots)$. '(' and ')' serve the target of divide different function levels when merging the different layers of $C_{seq'}$ and C_{seq} . It is like the format used in DeepCom [13] to split different level of AST. For example, Figure 2 shows that given a Java method $writeOut()$, how we express the call dependency sequence of it. The call dependency about $writeOut()$ is: ***writeOut (writeOut writeLittleEndian (write (putShort safelyAllocate putInt) putInt) write (putShort safelyAllocate putInt))***

3.3 The code summarization model

We name Our code summarization model as CallNN for using call dependency information. Figure 3 shows the structure of the model. It is based on the seq2seq model, which has achieved great success in natural machine translation area [22]. To make use of the call dependency information, an extra encoder structure is extended. There are two encoders in the model, one for source code, another for call dependency. The decoder uses combined information about these two encoders to generate natural language description. The model structure is similar to TL-CodeSum [14] with an extended encoder to learn API sequences.

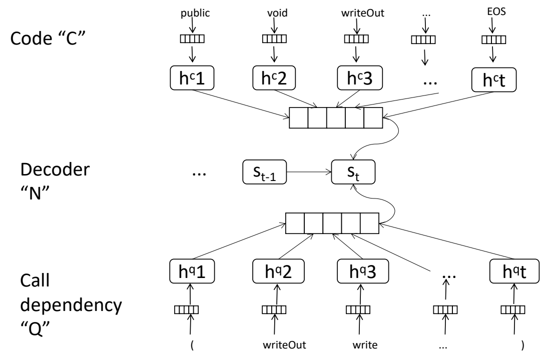


Figure 3: Summarization model with call dependency

Let $C = \{C_i\}$ be the set of code snippets sequences. Then $C_i = [c_1, c_2, \ldots, c_m]$ is one of the code snippet with c_j as code token. Each C_i is paired with a $Q_i = [q_1, q_2, \ldots, q_n]$, which denotes the call dependency sequence of C_i. Q is the set of call dependency sequences. The (C_i, Q_i) pair is corresponded to one natural language description $N_i = [n_1, n_2, \ldots, n_o]$. The model targets to find a map $C, Q \mapsto N$ as showed in Figure 3.

C and Q have independent encoder structures. Each encoder uses RNN units to read sequence tokens one by one. The hidden states of C and Q at step t denote as h_t^c and h_t^q.

$$h_t^c = f(c_t, h_{t-1}^c) \tag{1}$$

$$h_t^q = f(q_t, h_{t-1}^q) \tag{2}$$

The non-linear function f generates t step hidden state h_t using the information about previous hidden state h_{t-1} and the t token in input sequence. In this paper, we use Gated Recurrent Units as f. Decoder structure is trained to predict conditional probability of the next word d_t. To better capture the latent information between encoders and decoder, we use attention mechanism [6].

$$p(d_t|d_1, d_2, \ldots, d_{t-1}, C, Q) = g(d_{t-1}, s_t, A_t) \tag{3}$$

$$s_t = f(s_{t-1}, d_{t-1}, A_t) \tag{4}$$

$$A_t = \sum_{j=1}^{m} \alpha_{ij}^c h_j^c + \sum_{j=1}^{n} \alpha_{ij}^q h_j^q \tag{5}$$

g is a non-linear function which outputs the probability of d_t. s_t is RNN hidden state for time step t in decoder. A_t is the attention context vector calculated by a combination of hidden states from encoder C and encoder Q. The weight of hidden states is computed as:

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{m} exp(e_{ik})} \tag{6}$$

and

$$e_{ij} = a(s_{i-1}, h_j) \tag{7}$$

is a score function which evaluates the matching degree between inputs around position j and outputs at position i.

4 EXPERIMENT SETUPS

4.1 Dataset

To get the call dependency, related source code files must be included during analysis. As discussed in 3.2, the call dependencies should be extracted within a project. The experimental data is extracted on more than 1,000 java projects in github. We filter 100,000 code snippets from the database. All these codes are method level codes and have their independent functions with comments. We apply a regulation on codes to substitute all the String type to "_STR" and all the numbers to "_NUM". In Javadoc, the first sentence of comments usually describes the function of methods. We regard the first sentence of the comments as our target sentence.

Table 1: Data details

Code					
Count	< 100	< 150	< 200	> 200	***Avg***
102,577	68,616	14,679	7,226	12,056	106
Comment					
Count	< 10	< 50	< 100	> 100	***Avg***
102,577	26,477	75,005	962	133	14
Call dependency					
Count	< 10	< 50	< 100	> 100	***Avg***
102,577	57,558	40,667	2,982	1,370	18

The details of data are in Table 1. The lengths of data vary. The average code snippet length is 106, and there are about 90% code snippets have less than 200 tokens. We set the max length of code encoder to 200 tokens. The average call dependency sequence length is 18, though there are still some sequences have more than 200 length. To get as much full call dependencies as possible, the max length of call dependency encoder is set to 200 too. As for the decoder, the max length is set to 30, which is enough for the length of natural language description.

4.2 Experiment Settings

The dimension of GRU hidden states in our model is 128. We set the embedding of code snippet, call dependency and natural language summary to 128. The optimize algorithm

Table 2: Summarization examples

Examples

Code

```
public void writeOut(OutputStream out)
     throws IOException{
    out.write(_header);
    layoutAtom.writeOut(out);
    writeLittleEndian(masterID, out);
    writeLittleEndian(notesID, out);
    short flags = _NUM;
    if (followMasterObjects)
    {flags += _NUM;}
    if (followMasterScheme)
    { flags += _NUM; }
    if(followMasterBackground)
    { flags += _NUM;  }
    writeLittleEndian(flags, out);
    out.write(reserved);}
 }
```

```
public static boolean isGiftCard( String
     stPassed ){
if ( isOFBGiftCard(stPassed) ){ return true;
      }
else if ( isValueLinkCard(stPassed)){ return
      true ; }
return false ; }
```

Call dependency	writeOut (writeOut writeLittleEndian (write (putShort safelyAllocate putInt) putInt) write (putShort safelyAllocate putInt))	tisGiftCard (isOFBGiftCard (stripCharsInBag (append charAt indexOf toString length) length isEmpty (length) sumIsMod10) isValueLinkCard (startsWith stripCharsInBag (append charAt indexOf toString length) length isEmpty (length)))
Generated	Write the contents of the record back, so it can be written to disk .	Check to see if a card number is a valid OFB Gift Card .
Human-written	Write the contents of the record back, so it can be written to disk .	Check to see if a card number is a supported Gift Card .

Code

```
 public void testScannerThrowsException
 WhenCoprocessorThrowsDNRIOE()throws
     IOException, InterruptedException{
  reset();
  IS_DO_NOT_RETRY.set(true);
  TableName tableName = TableName.valueOf(
      name.getMethodName());
  try(Table t = TEST_UTIL.createTable(
      tableName , FAMILY)){
  TEST_UTIL.loadTable(t , FAMILY , false);
  TEST_UTIL.getAdmin().flush(tableName);
  inject();
  TEST_UTIL.countRows(t,new Scan().addColumn
      (FAMILY , FAMILY));
  fail(_STR );
  }
  catch(DoNotRetryIOException expected)
  { }
  assertTrue(REQ_COUNT.get()>_NUM );
 }
```

```
public static boolean injectCriteria( String
      klassName ){
    boolean trigger = false;
    if(generator.nextFloat() <
        getProbability(klassName))
    { trigger = true; }
    return trigger;
}
```

Call dependency	testScannerThrowsExceptionWhenCoprocessor ThrowsDNRIOE (inject (set) getAdmin countRows reset (set) loadTable valueOf assertTrue flush set createTable (waitUntilAllRegionsAssigned createTable) fail)	injectCriteria (getProbability (isDebugEnabled getFloat equals debug set getProperty) nextFloat)
Generated	Tests the case where a coprocessor throws the same data in a seek to the directory .	<UNK> method to check if we have reached the point of injection .
Human-written	Tests the case where a coprocessor throws a DoNotRetryIOException in the scan .	Simplistic method to check if we have reached the point of injection .

is stochastic gradient descent (SGD), and batch size is set to 32. For decoder, we use beam search. The beam size is 5. To limit the vocabulary size, we ordered the tokens in data by their frequency and chose the most frequent tokens as vocabulary. The vocabularies of code, call dependency and summary are 50,000, 50,000 and 30,000. The model is trained using Tensorflow on GPU.

4.3 Evaluation Metrics

We leverage the IR metrics used in TL-CodeSum [14] and DeepCom [13] to evaluate the code summarization performance: precision, recall and F-score.

Denote m as the number of mapped unigrams between the generated string output and target string. The total number of unigrams in the output string is o and total number of unigrams in target string is t. The precision is defined as

$$P = \frac{m}{o} \tag{8}$$

and recall is defined as

$$R = \frac{m}{t} \tag{9}$$

Precision shows the ratio of matching words in the generated comments. Recall shows the ratio of matching words in target comments. F-score is calculated as

$$F = \frac{2 * PR}{P + R} \tag{10}$$

which is an integration metric with precision and recall.

In addition, we also use machine translation metric BLEU [20] to evaluate our model. It computes the n-gram precision of hypothesis sequence according to the reference which is widely used in nmt area. The score reflects the similarity between the generated sequence and reference sequence, and is computed as:

$$BLEU = BP * exp(\sum_{n=1}^{N} w_n log p_n) \tag{11}$$

p_n is the ratio of length n subsequences in the generated sequence that also appear in reference. N is the maximum number of grams. It is set to 4 in experiment. BP is brevity penalty,

$$BP = \begin{cases} 1 & if c > r \\ e^{(1-r/c)} & if c \leq r \end{cases} \tag{12}$$

The BLEU score has been used in DeepApi [9] to evaluate the generated API sequences. Jiang et al. [16] uses it to evaluate generated summaries for commit messages. It can be inferred that BLEU is reasonable to measure generated summarization in our experiment.

5 EXPERIMENT RESULTS

5.1 Generated summarization

Table 2 shows some summarization generated by our model. In the first case, the generated summarization is the same as human-written. We check that this code snippet of {writeOut()} is not appeared in train data. Both of human-written and model generation understand the code and make a good description of the function of *writeOut()*. However, there are some cases the model generating summarization different from human-written summarization. In the second case, human summaries the function of *isGiftCard()* as "Check to see if a card number is a supported Gift Card". But the model learned the function of this code as "Check to see if a card number is a valid OFB Gift Card". It can be inferred that the model can generate summarization with its knowledge of call dependency "tisGiftCard (isOFBGiftCard ...".

There are also some bad generations. In the third cases, the model generated summarization is lack of the information about Exception. The model knows the method *testScannerThrowsExceptionWhenCoprocessorThrowsDNRIOE()* is to test the case where a coprocessor throws something. But it does not understand this test function throws "DoNotRetryIOException". Another bad situation is that some '<UNK>' tokens are generated. In the last case, a '<UNK>' token is generated as the first token of the sentence. That can make the summarization confused to read. We conjecture the reason for these bad generations is caused by the limitation of vocabulary. There are huge amount of different variable names, class names and method names in the source codes. We cut off the vocabulary to limit the size, which may cause the model cannot understand some extremely rare tokens, such as "testScannerThrowsExceptionWhenCoprocessorThrowsDNRIOE" and "DoNotRetryIOException".

5.2 Baseline

The experiments of TLCodeSum [14] show that by using the knowledge of api as input, the model outperforms using the embeddings of tokens directly (CODE-NN) [15]. So we compared our work with the TL-CodeSum [14] model which use api in source code which is the state-of-the-art work in code summarization area.

Table 3: Metrics compared with baseline

Approach	*Precision*	*Recall*	*F-score*	*BLEU*
TL-CodeSum(Code+Api)	0.4186	0.4152	0.4026	32.46
TL-CodeSum(Tranfer)	0.4214	0.4192	0.4062	32.55
CallNN	**0.4269**	**0.4244**	**0.4121**	**33.08**

5.3 Result discussion

As discussed above, TL-CodeSum only save the *(codes,annotation)* pairs when collecting data. We cannot directly use their data for lake of call dependency information. So we recollect our own dataset from projects on github and extract call dependency sequence. The experiment is set on exactly same data collected by us to compare the three models. TL-CodeSum(Code+Api) uses api and code as input and TL-CodeSum model(Transfer) is with a pre-trained api-to-nl model as transfer knowledge. CallNN uses call dependency sequence information. The experiment result from Table 3 shows that our CallNN with call dependency information gets the higher score of all these metrics. But the improvement is

Table 4: Comparison between CallNN and TL-CodeSum generation

	Examples	
1	**Call dependency**	handleParseResult (asList printHelpIfRequested (printVersionHelp (println) usage (usage) isVersionHelpRequested isUsageHelpRequested) emptyList)
	Api sequence	Collections.emptyList Arrays.asList
	Target sentence	Prints help if requested and otherwise executes the top level Runnable or Callable command
	TL-CodeSum(Transfer) gen	runnable executed and unlike most of any order or all non - level option or all Throwable mechanism
	CallNN gen	Prints help if requested and otherwise executes the top level command and all subcommands as Runnable or Callable
2	**Call dependency**	getKeyComparator (getInstance compare (compare))
	Api sequence	Collator.getInstance Collator.compare
	Target sentence	Gets a comparator for comparing state entry keys
	TL-CodeSum(Transfer) gen	Gets a new instance of the given String and returns the name entry
	CallNN gen	Gets a comparator for comparing state keys
3	**Call dependency**	getFloat (get (substituteVars) parseFloat)
	Api sequence	Float.parseFloat
	Target sentence	Get the value of the name property as a float
	TL-CodeSum(Transfer) gen	Get the value of the name property as a float
	CallNN gen	Get the private float value of its name , useful for loading flags into the value .
4	**Call dependency**	toString (toString)
	Api sequence	Date.toString
	Target sentence	Generates a string representation of the date
	TL-CodeSum(Transfer) gen	Generates a string representation of the date
	CallNN gen	Generates a JavaScript fragment that will be used for sending of connections when this server was successful

not as much as we thought. We analyse the summarizations of TL-CodeSum(Transfer) and CallNN to find out the reason.

There are 10257 code snippets in test data. The summarizations generated by TL-CodeSum(Transfer) and CallNN with same code snippet are not always similar to each other. Table 4 shows four different generated examples of the two models. CallNN model generates better summarizations for the first two cases but gets lower BLEU scores in the last two cases.

The target sentences in the first two cases have specific functional descriptions. The specific functional descriptions are related to the functional requirements within that project such as “Prints help if requested” and “Gets a comparator”. They are not general functions. The CallNN model generates “Gets a comparator ”, but TL-CodeSum generates a more general description “Gets a new instance”. CallNN generations are better in this situation. The target sentences in the last two cases are general functional description such as “Get the value of the name property as a float” and “Generates a string representation of the date”. TL-CodeSum with api performs better on these general meaning summarizations. Our CallNN generates redundant words and wrong words in these two cases.

According to the analysis above, we can conclude:

- CallNN generates better summarizations when the function is specific related to the meaning within a project.
- Comparing with TL-CodeSum, CallNN is not good at generating general meaning summarizations.
- Yet, on the whole, the higher score of CallNN denotes that call dependency information is to some extent more important than apis only included in source code.

It can be inferred that call dependency can extend the semantic information when generating summarizations. But the call dependency we extract now is limited within project. The third parted library calls and system calls are not taken into consideration. We conjecture it is why CallNN does not perform better on general meaning summarizations than TL-CodeSum. A more completed extraction of call dependency should be done in the future.

6 QUALITY ANALYSIS

This section, we demonstrate the ability of our model in learning latent information about code. The token embeddings and attention weights of the model are analysed to visualize the "meaning" of call dependency. We also discuss the reason for effect of our model.

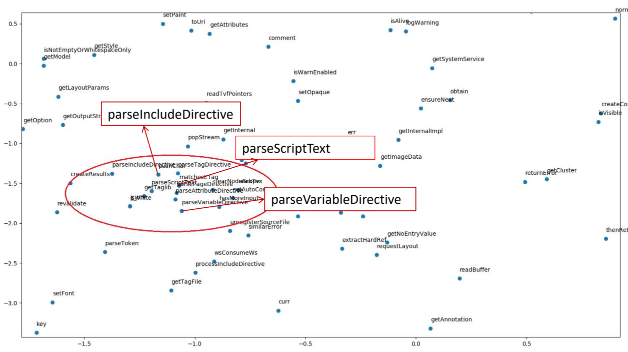


Figure 4: Embedding of Call dependency

6.1 Embedding Quality

Embeddings are projected vectors of words of vocabulary. It reflects the "meaning" of tokens. We analyse the embeddings of call dependency tokens. The dimension of call dependency embedding is 128. It is projected to 2 using t-sne for visualization. Figure 4 shows part of embeddings in our model. The call dependency on analogous function aggregates with each other. For instance, the"parseIncludeDirective", "parseScriptText" and "parseVariableDirective" are related to the function of a parser for JSP page in tomcat7. In this aspect, the embeddings learned by our model have effects to express the meaning of call dependency. It denotes our model learned the latent information about call dependency.

6.2 Attention of call dependency

As the input of the model is a sequence of tokens, each token makes different contribution to the generation task. To generate a hypothetical summarization, the model pays different attention on these input tokens. It can be visualized by the attention weight in the model. Figure 5 shows an example of the call dependency attention on a generation of code in Figure 2. The higher weights are with brighter colors. To generate this summary, the model pays more attention to the token "putint". It is a method invocation in the codes of related method *write()* but not in the source method *writeOut()*. To generate the summarization, the model not only concentrates on the source code information, but also makes use of the information about related codes. It can be inferred that the call dependency is important to learn the meaning of programming language.

The call dependency makes contribution to code understanding. It is according to our experiences of writing codes.

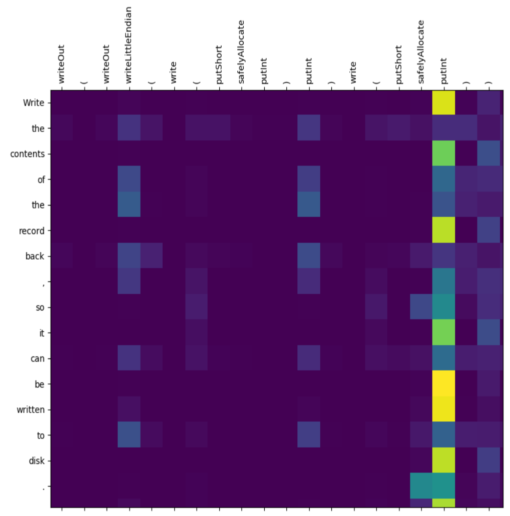


Figure 5: Attention vector of *writeOut()*

```
public void writeOut(OutputStream out) throws
    IOException {
  out.write(_header);
  layoutAtom.writeOut(out);
  writeLittleEndian(masterID, out);
  writeLittleEndian(notesID, out);
  short flags = 0;
    ...
  writeLittleEndian(flags, out);
  out.write(reserved);
 }

private void write(){
  int pos = 0;
  _data = IOUtils.safelyAllocate(indents.size() * 6,
      MAX_RECORD_LENGTH);
  for(IndentProp prop : indents){
  LittleEndian.putInt(_data, pos, prop.
      getCharactersCovered());
  LittleEndian.putShort(_data, pos + 4, (short) prop.
      getIndentLevel());
  pos += 6;
  }
 }

 public static void writeLittleEndian(int i,
     OutputStream o) throws IOException{
  byte[] bi = new byte[4];
  LittleEndian.putInt(bi, 0, i);
  o.write(bi);
 }
```

Take the *writeOut()* codes for example. If one want to know the exact function of *writeOut()*, only use the information about codes in *writeOut()*, he can not really meet his target. He can know the function *writeOut()* call other functions such as *wirte()* and *wirteLittleEndian()*. But he cannot know the details of these method invocations. Only if he can also get the information about *write()* and *writeLittleEndian()*, the function of these related codes can be understood. At this moment, the code reader can have a more comprehensive understanding about the function *writeOut()*. This is why we usually jump from method invocations to method definitions between files when we read codes. It is also part of the same thing that a compiler does when it compiles a program. Human design the caller-callee structure of programmig language, so the compiler must toe the mark. But they follow the compiler rules conversely when human wants to comprehend their codes.

7 CONCLUSION

This paper extracts call dependencies between source code and its related codes from large-scale open source Java projects on github. The call dependency information is expressed as a sequence of method name tokens to assist code summarization. A model based on Seq2Seq is used to generate code summarization from source code using the call dependency information. As far as we know, it is the first attempt to consider call dependency information about code in code summarization area with machine learning method. Experiments show the effect of our method. By using call dependency information, the CallNN can improve the performance of code summarization.

However, the call dependency extracted in this paper is limited with project, ignoring library calls and system calls. The extraction tool should be improved to extract more completed call dependency in the future. Meanwhile, the expression of call dependency on this paper is simply and basic as a sequence of method names. There is still much information about related codes which can be taken advantage of. In the future, we will dig out more comprehensive information about call dependency to help the generation task.

ACKNOWLEDGMENT

This work was supported by National Grand R&D Plan (Grant No. 2018YFB1003903), the National Natural Science Foundation of China (Grant No.61432020) and the Laboratory of Software Engineering for Complex Systems.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.

[3] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[4] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[5] Alberto Bacchelli, Michele Lanza, and Romain Robbes. 2010. Linking e-mails and source code artifacts. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 375–384.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[7] Themistoklis Diamantopoulos, Georgios Karagiannopoulos, and Andreas Symeonidis. 2018. Codecatch: extracting source code snippets from online sources. In *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, 21–27.

[8] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 13–22.

[9] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[10] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[11] Hany Hassan, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, et al. 2018. Achieving human parity on automatic chinese to english news translation. *arXiv preprint arXiv:1803.05567* (2018).

[12] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 232–242.

[13] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[14] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).

[15] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

[16] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 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, 135–146.

[17] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[18] Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology* 31, 5 (2016), 883–909.

[19] Federico Tomassetti Nicholas Smith, Danny van Bruggen. [n.d.]. JAVAPARSER FOR PROCESSING JAVA CODE. https://javaparser.org/.

[20] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 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, 311–318.

[21] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2017. JavaParser: visited. *Leanpub, oct. de* (2017).

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[23] Eike von Savigny. 2010. *Ludwig Wittgenstein: Philosophische Untersuchungen*. Vol. 13. Walter de Gruyter.

[24] Gang Yin, Tao Wang, Huaimin Wang, Qiang Fan, Yang Zhang, Yue Yu, and Cheng Yang. 2015. OSSEAN: mining crowd wisdom in open source communities. In *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE, 367–371.