Summarizing Source Code with Transferred API Knowledge

Xing Hu[1,2], **Ge Li**[1,2*], **Xin Xia**[3], **David Lo**[4], **Shuai Lu**[1,2], **Zhi Jin**[1,2*]
[1] Keylaboratory of High Confidence Software Technologies (Peking University), Ministry of Education
[2] Institute of Software, EECS, Peking University, Beijing, China
[3] Faculty of Information Technology, Monash University, Australia
[4] School of Information Systems, Singapore Management University, Singapore
[1,2] {huxing0101, lige, shuai.l, zhijin}@pku.edu.cn, [3] xin.xia@monash.edu, [4] davidlo@smu.edu.sg

Abstract

Code summarization, aiming to generate succinct natural language description of source code, is extremely useful for code search and code comprehension. It has played an important role in software maintenance and evolution. Previous approaches generate summaries by retrieving summaries from similar code snippets. However, these approaches heavily rely on whether similar code snippets can be retrieved, how similar the snippets are, and fail to capture the API knowledge in the source code, which carries vital information about the functionality of the source code. In this paper, we propose a novel approach, named TL-CodeSum, which successfully uses API knowledge learned in a different but related task to code summarization. Experiments on large-scale real-world industry Java projects indicate that our approach is effective and outperforms the state-of-the-art in code summarization.

1 Introduction

As a critical task in software maintenance and evolution, code summarization aims to generate functional natural language description for a piece of source code (e.g., method). Good summaries improve program comprehension and help code search [Haiduc *et al.*, 2010]. The code comment is one of the most common summaries used during software developments. Unfortunately, the lack of high-quality code comments is a common problem in software industry. Good comments are often absent, unmatched, and outdated during the evolution. Additionally, writing comments during the development is time-consuming for developers. To address these issues, some studies have tried to give summaries for source code automatically [Haiduc *et al.*, 2010; Moreno *et al.*, 2013; Iyer *et al.*, 2016; Hu *et al.*, 2018]. Generating code summaries automatically can help save the developers' time in writing comments, program comprehension, and code search.

Previous works have exploited Information Retrieval (IR) approaches and learning-based approaches to generate summaries. Some IR approaches search comments from similar code snippets as summaries [Haiduc *et al.*, 2010; Eddy *et al.*, 2013], while some approaches extract keywords from the given code snippets as summaries [Moreno *et al.*, 2013]. However, these IR-based approaches have two main limitations. First, they fail to extract accurate keywords when the identifiers and methods are poorly named. Second, they cannot output accurate summaries if no similar code snippet exists.

Recently, some studies have adopted deep learning approaches to generate summaries by building probabilistic models of source code [Iyer *et al.*, 2016; Allamanis *et al.*, 2016; Hu *et al.*, 2018]. [Hu *et al.*, 2018] combine the neural machine translation model and the structural information within the Java methods to generate the summaries automatically. [Allamanis *et al.*, 2016] proposes a convolutional model to generate name-like summaries, and their approach can only produce summaries with an average of 3 words. [Iyer *et al.*, 2016] presents an attention-based Recurrent Neural Networks (RNN) named CODE-NN to generate summaries for C# and SQL code snippets collected from Stack Overflow. Their experimental results have proved the effectiveness of deep learning approaches on code summarization. Although deep learning techniques are successful in the first step toward automatic code summary generation, the performance is limited since they treat source code as plain text. There is much latent knowledge in source code, e.g., identifier naming conventions and Application Programming Interface (API) usage patterns.

Intuitively, the functionality of a code snippet is related to its API sequences. Developers often invoke a specific API sequence to implement a new feature. Compared to source code with different coding conventions, API sequences tend to be regular. For example, we usually use the following API sequence of Java Development Kit (JDK): *FileRead.new*, *BufferReader.new*, *BufferReader.read*, and *BufferReader.close* to implement the function "Read a file". We conjecture that knowledge discovery in API sequence can assist the generation of code summaries. Inspired by the transfer leaning [Pan and Yang, 2010], the code summarization task can be fine tuned by using the API knowledge learned in a different but related task. In order to verify our conjecture, we conduct an experiment on generating summaries for Java methods which are functional units of Java programming language.

In this paper, we propose a novel approach called TL-CodeSum, which generates summaries for Java methods with

*Corresponding Authors

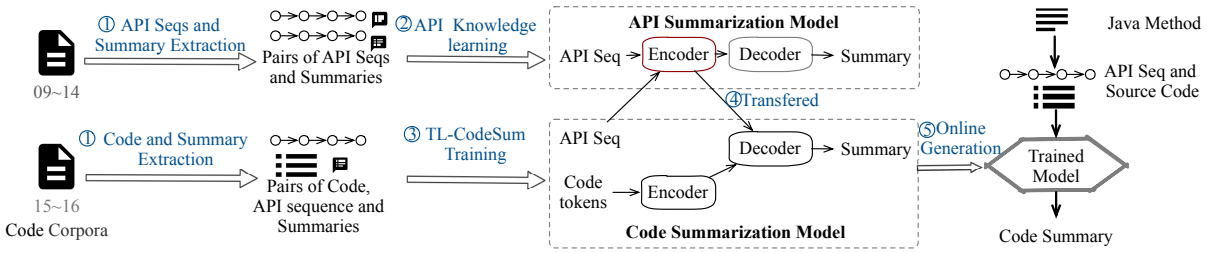


Figure 1: The overall architecture of TL-CodeSum

the assistance of transferred API knowledge from another task of API sequences summarization. We conduct the code summarization task on the Java projects which are created from 2015 to 2016 in GitHub. The API sequences summarization task aims to build the mappings between API knowledge and the corresponding natural language descriptions. The corpus for API sequences summarization consists of $\langle$*API sequence*, *summary*$\rangle$ pairs extracted from a large-scale Java projects which are created from 2009 to 2014 in GitHub. The experimental results demonstrate that TL-CodeSum significantly outperforms the state-of-the-art on code summarization.

The contributions of our work are shown as follows:

- We propose a novel approach named TL-CodeSum that summarizes Java methods with the assistance of the learned API knowledge.
- We design a framework to learn API knowledge from API summarization task and use it to assist code summarization task.

2 Related Work

As an integral part of software development, code summaries describe the functionalities of source code. IR approaches [Haiduc *et al.*, 2010; Wong *et al.*, 2015] and learning-based approaches [Iyer *et al.*, 2016; Allamanis *et al.*, 2016] have been exploited to automatic code summarization. IR approaches are widely used in code summarization. They usually synthesize summaries by retrieving keywords from source code or searching comments from similar code snippets. [Haiduc *et al.*, 2010] applied two IR techniques, the Vector Space Model (VSM) and Latent Semantic Indexing (LSI), to generate term-based summaries for Java classes and methods. [Wong *et al.*, 2015] applied code clone detection techniques to find similar code snippets and extract the comments from the similar code snippets. The effectiveness of IR approaches heavily depends on whether similar code snippets exist and how similar they are. While extracting keywords from the given code snippets, they fail to generate accurate summaries if the source code contains poorly named identifiers or method names.

Recently, inspired by the work of [Hindle *et al.*, 2012], an increasing number software tasks, e.g., fault detection [Ray *et al.*, 2016], code completion [Nguyen *et al.*, 2013], and code summarization [Iyer *et al.*, 2016] build language models for 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 [Iyer *et al.*, 2016; Gu *et al.*, 2016]. Generating summaries from source code aims to bridge the gap between programming language and natural language. [Raychev *et al.*, 2015] aimed to predict names and types of variables, whereas [Allamanis *et al.*, 2015a; 2016] suggested names for variables, methods and classes. [Hu *et al.*, 2018] exploited the neural machine translation model on the code summarization with the assistance of the structural information. [Allamanis *et al.*, 2016] applied a neural convolutional attentional model to summarizing the Java code into short, name-like summaries (average 3 words). [Iyer *et al.*, 2016] presented an attention-based RNN network to generate summaries that described the functionalities of C# code snippets and SQL queries. These works have proved the effectiveness of building probabilistic models for code summarization. In this paper, we consider exploiting the latent API knowledge in source code to assist the code summarization. Inspired by transfer learning which achieves successes on training models with a learned knowledge [Pan and Yang, 2010], the API knowledge used to code summarization is learned from a different but related task.

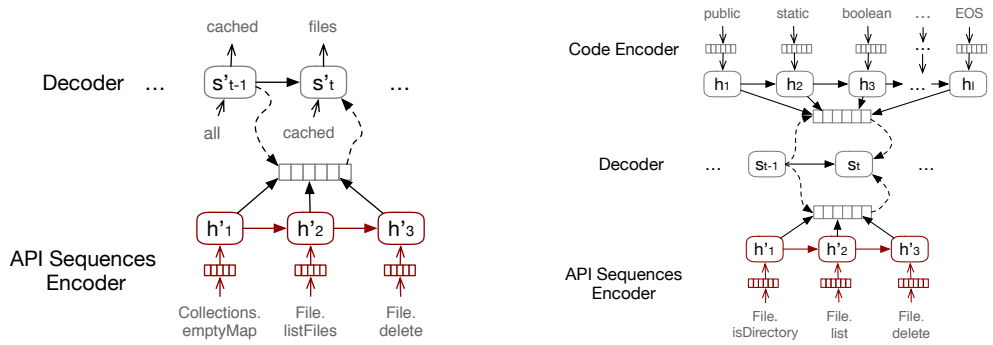
3 Approach

In this section, we present our proposed approach TL-CodeSum, which decodes summaries from source code with transferred API knowledge. As shown in Figure 1, the approach mainly consists of three parts: data processing, model training, and online code summary generation. The model aims to implement two tasks, API sequence summarization task and code summarization task. The API sequence summarization task aims to build the mappings between API knowledge and the functionality descriptions. The learned API knowledge is applied to code summarization task to assist the summary generation. The details of the two tasks will be introduced in the following sections.

3.1 API Sequence Summarization Task

API sequence summarization aims to build the mappings between API knowledge and natural language descriptions. To implement a certain functionality, for example, how to read a file, developers often invoke the corresponding API sequences. In this paper, we exploit the API knowledge to assist code summarization.

The knowledge is learned from the API summarization task which generates summaries for API sequences. The task adopts a basic Sequence-to-Sequence (Seq2Seq) model which achieves successes in Machine Translation (MT) [Sutskever *et al.*, 2014], Text Summarization [Rush *et*



(a) API Sequence Summarization

(b) Code Summarization with Transferred API Knowledge. (The red part is learned from API sequence summarization task)

Figure 2: The model of TL-CodeSum

al., 2015], and etc. As shown in Figure 2(a), it mainly contains two parts, an API sequence encoder and a decoder.

Let $\mathcal{A}' = \{A'^{(i)}\}$ denotes a set of API sequence where $A'^{(i)} = [a'_1, ..., a'_m]$ denotes the sequence of API invocations in a Java method. For each $A'^{(i)} \in \mathcal{A}'$, there is a corresponding natural language description $D'^{(i)} = [d'_1, ..., d'_n]$. The goal of API sequence summarization is to align the $\mathcal{A}'$ and $\mathcal{D}'$, namely, $\mathcal{A}' \rightarrow \mathcal{D}'$.

The API encoder uses an RNN to read the API sequence $A'^{(i)} = [a'_1, ..., a'_m]$ one-by-one. The API sequence is embedded into a vector that represents the API knowledge. The API knowledge is then used to generate the target summary by the decoder. To better capture the latent alignment relations between API sequences and summaries, we adopt the classic attention mechanism [Bahdanau *et al.*, 2014]. The hidden state of the encoder is updated according to the API and the previous hidden state,

$$h'_t = f(a'_t, h'_{t-1}) \tag{1}$$

where f is a non-linear function that maps a word of source language into a hidden state h'_t at time t by considering previous hidden states h'_{t-1}. In this paper, we use a Gated Recurrent Units (GRU) as f. The decoder is another RNN and trained to predict conditional probability of the next word $d'_{t'}$ given the context vector $\mathbf{C}'$ and the previously predicted words $d'_1, ..., d'_{t'-1}$ as

$$p(d'_{t'}|d'_1, ..., d'_{t'-1}, A') = g(d'_{t'-1}, s'_{t'}, \mathbf{C}'_{t'}) \tag{2}$$

where g is a non-linear function that outputs the probability of $d'_{t'}$ and $s'_{t'}$ is an RNN hidden state for time step t' and computed by

$$s'_{t'} = f(s'_{t'-1}, d'_{t'-1}, \mathbf{C}'_{t'}) \tag{3}$$

The context vector $\mathbf{C}'_i$ is computed as a weighted sum of hidden states of the encoder $h'_1, ..., h'_m$,

$$\mathbf{C}'_i = \sum_{j=1}^{m} \alpha'_{ij} h'_j \tag{4}$$

where

$$\alpha'_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{m} exp(e_{ik})} \tag{5}$$

and

$$e_{ij} = a(s'_{i-1}, h'_j) \tag{6}$$

is an alignment model which scores how well the inputs around position j and the output at position i match. Both the encoder and decoder RNN are implemented as a GRU [Cho *et al.*, 2014], which is one of widely-used RNN.

3.2 Code Summarization Task

The code summarization model is a variant of the basic Seq2Seq model. Instead of using a code encoder and a decoder, TL-CodeSum adds another API encoder which is transferred from API summarization model. Let $\mathcal{C} = \{C^{(i)}\}$, $\mathcal{A} = \{A^{(i)}\}$, and $\mathcal{D} = \{D^{(i)}\}$ denote the source code, API sequences, and corresponding summaries of Java methods respectively. The goal of code summarization is to generate summaries from source code with the assisted API knowledge learned from API sequence summarization, namely, $\mathcal{C}, \mathcal{A} \rightarrow \mathcal{D}$.

As shown in Figure 2(b), the API sequences within Java methods are encoded by the transferred API encoder, which is marked red in API summarization task. The code encoder and API encoder aim to learn the semantic information of the given code snippet $C = [c_1, ..., c_l]$ and API sequence $A = [a_1, ..., a_m]$ respectively. In order to integrate the two parts of information better, the decoder needs to be able to combine the attention information collected from both two encoders. The context vector is computed as their sum,

$$\mathbf{C}_i = \sum_{j=1}^{l} \alpha_{ij} h_j + \sum_{j=1}^{m} \alpha'_{ij} h'_j \tag{7}$$

where α and α' are attention distributions of source code and API sequence respectively. The decoding procedure is similar to the API summarization task which adopts a GRU to predict word-by-word.

4 Experiments

4.1 Dataset Details

There are two datasets used in our work, one for API sequences summarization and the other one for code summarization as shown in the data processing stage in Figure 1.

Table 1: Statistics for code snippets in our dataset

Datasets	#Projects	#Files	#Lines	#Items
15-16	9,732	1,051,647	158,571,730	69,708
09-14	13,154	2,938,929	496,215,929	340,922

Table 2: Statistics for API sequence, code and comments length

API sequences Lengths					
Avg	Mode	Median	<5	<10	<20
4.39	1	2	79.99%	91.38%	97.18%
Comments Lengths					
Avg	Mode	Median	<20	<30	<50
8.86	8	13	75.50%	86.79%	95.45%
Code Lengths					
Avg	Mode	Median	<100	<150	<200
99.94	16	65	68.63%	82.06%	89.00%

The two datasets are both collected from GitHub. The API sequences summarization dataset contains Java projects from 2009 to 2014 and is used to learn API knowledge. The Java projects used in code summarization task are created from 2015 to 2016. The API knowledge learned from the former dataset is applied to train the code summarization task on the latter dataset. To keep the quality of the projects, we select the projects that have at least 20 stars as the preliminary dataset. The API sequences are extracted by the approach that [Gu *et al.*, 2016] proposed. We use Eclipse's JDT compiler[1] to parse source code into AST trees. Then we extract the Java methods, the API sequences within these methods and the corresponding Javadoc comments which are standard comments for Java methods. These comments that describe the functionalities of Java methods are taken as code summaries. The source code is tokenized into tokens before they are fed into the network. To decrease noise introduced to the learning process, we only take the first sentence of the comments since they typically describe the functionalities of Java methods according to Javadoc guidance[2]. However, not every comment is useful, so some heuristic rules are required to filter the data. Methods with empty or just one-word descriptions are filtered out in this work. The setter, getter, constructor, test methods, and override methods, whose comments are easy to predict, are also excluded.

At last, we get 340,922 pairs of ⟨*API sequence, summary*⟩ for API knowledge learning in API sequences summarization task and 69,708 pairs of ⟨ *API sequence, code, summary*⟩ for code summarization task.[3] We split each dataset into training, valid and testing sets in proportion with 8 : 1 : 1 after shuffling the pairs. We train all models using the training set and compute the accuracy scores in the test set. The average lengths of Java methods, API sequences, and comments are 99.94, 4.39, and 8.86 respectively. The detailed information of the datasets is shown in Table 1 and Table 2.

[1]http://www.eclipse.org/jdt/

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

[3]The data and code are available at https://github.com/xing-hu/TL-CodeSum

Table 3: Precision, Recall, and F-score for our approach compared with baseline

Approaches	Precision	Recall	F-score
CODE-NN	26.21	14.17	18.40
API-Only	30.72	21.14	25.05
Code-Only	38.89	28.81	33.10
API+Code	41.06	30.34	34.90
TL-CodeSum(fixed)	**42.20**	34.38	37.89
TL-CodeSum(fine-tuned)	40.78	**35.41**	**37.91**

Table 4: BLEU and METEOR for our approach compared with baseline

Approaches	BlEU-4 score	METEOR
CODE-NN	25.3	6.92
API-Only	26.45	10.71
Code-Only	35.50	14.78
API+Code	37.28	15.88
TL-CodeSum(fixed)	36.42	18.07
TL-CodeSum(fine-tuned)	**41.98**	**18.81**

4.2 Experiment Settings

We set the dimensionality of the GRU hidden states, token embeddings, and summary embeddings to 128. The model is trained using the mini-batch stochastic gradient descent algorithm (SGD) and the batch size is set as 32. The maximum lengths of source code and API sequences are 300 and 20. For decoding, we set the beam size to 5 and the maximum summary length to 30 words. Sequences that exceed the maximum lengths will be excluded from training. The vocabulary size of the code, API, and summary are 50,000, 33,082, and 26,971. We use the Tensorflow to train our models on GPUs.

5 Experimental Results

5.1 Accuracy in Summary Generation

Metric: In this paper, we use IR metrics and Machine Translation (MT) metrics to evaluate our method. For IR metrics, we report the precision, recall and F-sore of our method. Based on the number of mapped unigrams found between the two strings (m), the total number of unigrams in the translation (t) and the total number of unigrams in the reference (r), we calculate unigram precision $P = m/t$ and unigram recall $R = m/r$. Precision is the fraction of generated summary tokens that are relevant, while recall is the fraction of relevant tokens that are generated. F-score is the quality compromise between precision and recall.

We use two MT metrics BLEU-4 score [Papineni *et al.*, 2002] and METEOR [Denkowski and Lavie, 2014] which are also used in CODE-NN to measure the accuracy of generated source code summaries. BLEU score is a widely used accuracy measure for machine translation. It computes the n-gram precision of a candidate sequence to the reference. ME-

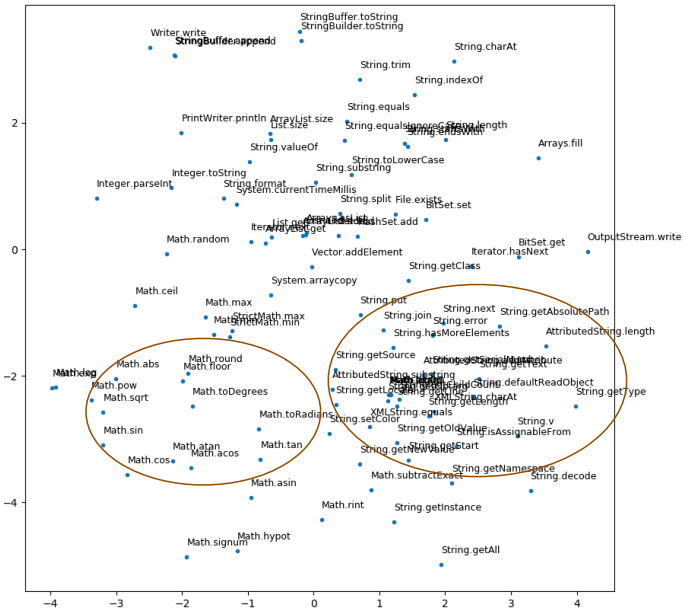


Figure 3: A 2D projection of API embeddings using t-SNE

TEOR is recall-oriented and evaluates translation hypotheses by aligning them to reference translations and calculating sentence-level similarity scores.

Baseline: We compare TL-CodeSum with CODE-NN [Iyer *et al.*, 2016] which is a state-of-the-art code summarization approach. CODE-NN proposed an end-to-end generation system to generate summaries given code snippets. Compared to TL-CodeSum, CODE-NN generates each word by a global attention model which computes a weighted sum of the embeddings of code tokens instead of hidden states of RNNs. We also evaluate the accuracy of generated summaries given API and code using the ibasic Seq2Seq model respectively (API-Only and Code-Only). To evaluate the influence of the transferred API knowledge, we conduct an experiment that uses two encoders to encoder API sequences and source code respectively without transferred API knowledge (API+Code). Additionally, we compare two approaches to exploiting API knowledge, fine tuning the whole network (fine tuned TL-CodeSum) and train the network with fixed API knowledge (fixed TL-CodeSum) .

Results: Table 3 illustrates the results on IR metrics of different approaches. Precision denotes the ratio of matching words in the generated comments. Results show that using RNN to encode the source code (Code-Only) or API sequences (API-Only) outperforms using the embeddings of tokens directly (CODE-NN). The RNNs are good at learning the semantics of input sequences and the code information is much more helpful for summary generation. When combining source code and API information, the precision is much higher than CODE-NN and the two basic Seq2Seq models (i.e., Code-Only and API-Only). The improvements have proved the importance of API information while generating comments. Furthermore, transferring the API knowledge from the API sequence summarization task directly improves the precision and recall. The precision decreases when fine-tuning the whole network, while the recall is increased. In terms of F-score, our proposed model with fine-tuning shows slightly improvement over our model with fixed parameters. TL-CodeSum generates more overlapping words between automatically generated summaries and human-written summaries. Overall, the TL-CodeSum surpasses other approaches on generating information related summaries.

Table 5: Examples of generated summaries given Java methods and API sequences.

Examples	

Java method and API Sequence	`protected void sprint(double doubleField){`
	`  sprint(String.valueOf(doubleField));`
	`}`
	String.valueOf

Human-Written	Pretty printing accumulator function for doubles
TL-CodeSum	pretty printing accumulator function for longs

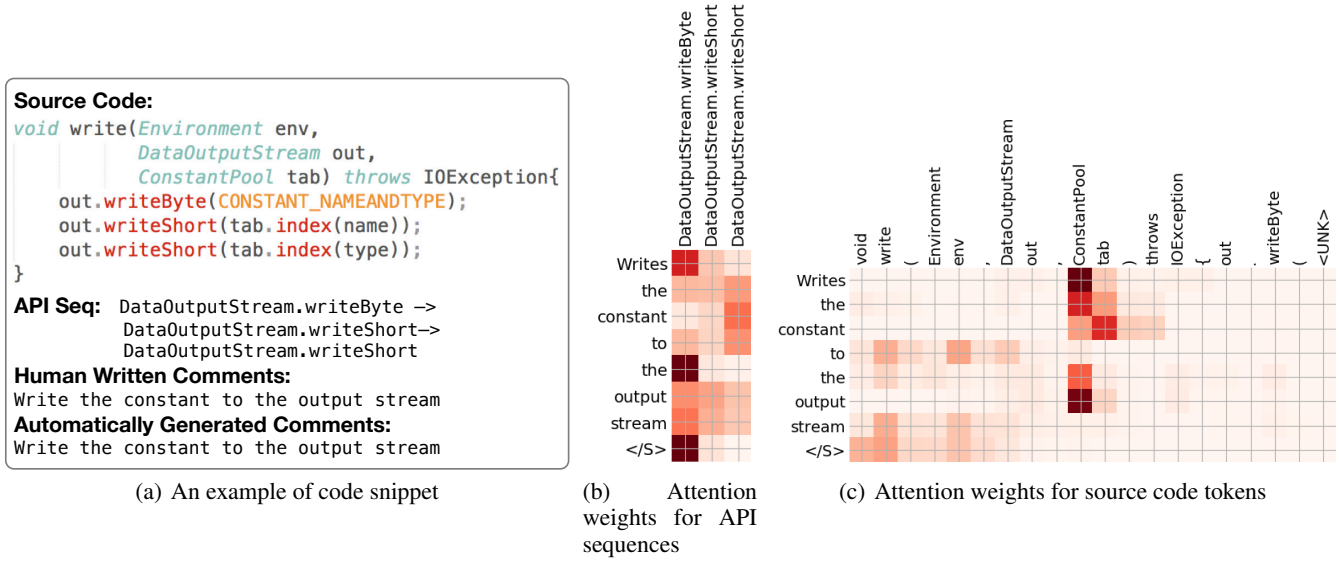
Java method and API Sequence	`public void removeMouseListener(`
	`  GlobalMouseListener listener){`
	`  listeners.remove(listener);`
	`}`
	List.remove

Human-Written	Removes a global mouse listener
TL-CodeSum	removes an existing message listener.

Java method and API Sequence	`private static boolean`
	`  instanceOfAny(Object o,`
	`  Collection<Class> classes){`
	`  for(Class c: classes){`
	`  if (c.isInstance(o))`
	`  return true;`
	`  }`
	`  return false;`
	`}`
	Collection.isEmpty→Collection.add →Class.isInstance

Human-Written	returns true if the Object 'o' is an instance of any class in the Collection
TL-CodeSum	returns true if the object is registered in classes, or false otherwise.

We also evaluate the gap between automatically generated summaries and human-written summaries on MT metrics. Table 4 illustrates METEOR scores and sentence level BLEU-4 scores of different approaches to generating comments for Java methods. As the results indicate, the TL-CodeSum obviously outperforms the state-of-the-art method CODE-NN on Java methods summarization. The BLEU-4 score and METEOR of CODE-NN and API-Only reflect that summarizing from API sequences by Seq2Seq model has the similar ability of CODE-NN, although the semantics of API sequences are much fewer than the source code. It mainly learns the relationship between API knowledge and functionalities of Java methods. Integrating the learned API knowledge and source code greatly improves the BLEU score and METEOR. Through the evaluation, we have verified the effectiveness of API usage patterns for code summarization. TL-CodeSum can not only generate more informative related comments but also more expressive comments than state-of-the-art baselines. Compared to the model without API



(a) An example of code snippet

(b) Attention weights for API sequences

(c) Attention weights for source code tokens

Figure 4: Heatmap of attention weights for API sequence and source code snippets. The model learns to align key summary words with the corresponding tokens in API sequences and source code.

sequences, the BLEU-4 score of TL-CodeSum increases to 41.98%.

5.2 Quality Analysis

API Embedding Quality. The API usage pattern is an important part of code summarization. Different coding conventions of different developers improve the difficulties of semantic learning. The API usage patterns are relatively regular, hence integrating API knowledge helps learn the functionalities of source code. The quality of API embeddings' learning is crucial for our proposed method to work well. Figure 3 shows a 2-D projection of the embeddings of APIs. For ease the demonstration, we select the APIs related to "String" and "Math" which are circled in Figure 3. As shown in the graph, TL-CodeSum can successfully embed APIs implementing similar functionalities.

Complementarity of API and Code. TL-CodeSum generates summaries according to the semantics of source code and the transferred API knowledge. Figure 4 shows the attention weights for the API sequence and code tokens within the Java method while generating their corresponding summaries. We give the details of Java method, API sequence within it, the human-written comment, and the automatically generated comment by TL-CodeSum in Figure 4(a). The generated tokens have different relationships between API sequence and code tokens. From the figure, we find the words "write" and "stream" are more relevant to API "DataOutputStream.writeByte". While the word "constant" is more relevant the variable "tab" whose type is "ConstantPool". TL-CodeSum aligns different words with specific API or code tokens.

Comparison between Human-Written and TL-CodeSum Generated Summaries. Table 5 shows three examples of generated summaries. Most generated summaries are clear, coherent, and informative related regardless the lengths of Java methods. The main differences between the generated and human-written summaries are as follows:

1. Words replacement: Some words are replaced by their synonyms, antonyms, or words in the same domain. In the first example, the word "doubles" is replaced by "longs" which comes from the same domain (the data types of Java language).

2. More general: TL-CodeSum learns the functionalities over a large-scale dataset. The generated summaries may present more general meaning and give the abstract semantics of given Java methods just like the second example.

3. Missed Identifiers: Identifiers are defined by different developers and those used by different methods may differ from one another. Learning the identifiers is challenging problems [Hellendoorn and Devanbu, 2017]. TL-CodeSum misses some identifiers or replaces them with "UNK" sometimes. As the third example shows, the identifiers "o" and "Collection" are missing in the generated summary.

6 Conclusion

In this paper, we propose a novel deep model called TL-CodeSum to generate summaries by capturing semantics from the source code with the assistance of API knowledge. The API knowledge is transferred into TL-CodeSum from API sequences summarization task. Experimental results on Java methods indicate that integrating API sequences is beneficial and effective. TL-CodeSum significantly outperforms the state-of-the-art methods for code summarization. In the future, we will combine richer program structural and sequential information derived from program analysis tools for code summarization.

Acknowledgments

This research is partially 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.

References

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

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

[Allamanis *et al.*, 2015b] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2123–2132, 2015.

[Allamanis *et al.*, 2016] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

[Bahdanau *et al.*, 2014] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *Computer Science*, 2014.

[Cho *et al.*, 2014] Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *Computer Science*, 2014.

[Denkowski and Lavie, 2014] Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the EACL 2014 Workshop on Statistical Machine Translation*, 2014.

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

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

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

[Hellendoorn and Devanbu, 2017] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[Hindle *et al.*, 2012] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[Hu *et al.*, 2018] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 2018 26th IEEE/ACM International Confernece on Program Comprehension*. ACM, 2018.

[Iyer *et al.*, 2016] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *ACL (1)*, 2016.

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

[Nguyen *et al.*, 2013] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542. ACM, 2013.

[Pan and Yang, 2010] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[Papineni *et al.*, 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[Ray *et al.*, 2016] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.

[Raychev *et al.*, 2015] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[Rush *et al.*, 2015] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

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

[Wong *et al.*, 2015] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 380–389. IEEE, 2015.