Improving Code Summarization by Combining Deep Learning and Empirical Knowledge

Lingbin Zeng, Xunhui Zhang, Tao Wang, Xiao li, Jie Yu, Huaim Wang
College of Computer Science
National University of Defense Technology
Changsha, Hunan, China
{zenglingbin16, zhangxunhui, taowang2005, xiaoli, yujie16, hmwang}@nudt.edu.cn

***Abstract*—Code summaries are human-readable text that describes the functionality of code blocks. Software developers use code summaries to understand the specification of API while code retrieve system relies on code summaries for effective code search. However, code summaries are often written by software developers. Writing good code summaries usually requires great effort. It could be helpful if developers use automatic code summarization system to generate code summaries. Recently, some works have applied deep learning methods to generate code summaries for code snippets. However, those deep learning methods treat code snippets as streams of text tokens while ignoring the inherent code structure information. In this paper, we propose a novel code summarization method named the CDE-Model (Code summarization by Deep learning and Empirical knowledge) that combines inherent code structure information with deep learning models. The CDE-Model proposes several empirical strategies to transform code snippets to refined code representation and feeds them into an encoder-decoder neural network for text generation. We conduct large-scale experiments on 1500 popular Java projects on GitHub[1] with 396,184 pairs of code snippets and summaries. Experimental results show that the quality of code summaries generated by our CDE-Model is better than other two methods. To the best of our knowledge, this paper is the first to combine code structure information with deep learning.**

Keywords:Code summarization; GitHub; Recurrent neural network; Java language.

I. INTRODUCTION

The rapid development of open source software (OSS) provides a massive reusable resource for software development [1] [2] [3] [4]. In OSS, code summaries are very important as they describe the functionality of code blocks in the form of human-readable text [5]. On one hand, software developers use code summaries to understand the specification of API. On the other hand, code retrieve system relies on code summaries for effective natural language code search [6]. Thus, it is crucial to maintain high-quality and adequate code summaries in OSS projects. However, code summaries are often manually written by software developers and writing good code summaries usually requires great human effort. In our pilot study, we found that the annotation rate for even famous projects on GitHub are very low (see TableI). This could significantly hinder software innovation. It could be helpful if developers use automatic code summarization system to generate code summaries.

TABLE I: Annotation rate of several famous GitHub projects

Project name	Lines of codes	Code annotation rate
Redis	83,233	23.7%
cocos2d-x	461,685	8.3%
Hadoop	1,217,655	13.1%
blueprints	33,356	8.6%
Tensorflow	300,864	9.2%
hebel	10,040	8.1%
jQuery	42,300	12.0%

To address this problem, automatic code summarization systems have been proposed for generating code summaries. Previous code summarization methods usually rely on information retrieval and text mining methods. For example, Vassallo C et al. [7] propose the CODES method which extracts candidate summaries from StackOverflow[2] discussions and creates Javadoc descriptions based on social connection theory. Wang et al. [8] introduce an approach to analyze constructs of code snippets and extract keywords to produce code summaries.

Recently, deep learning methods have become a popular research topic in code summarization research. Iyer et al. [1] use long short-term memory (LSTM) network to generate code summaries according to code context. The model they trained is especially useful for short code snippets. Although the deep learning based methods have shown effectiveness in code summarization, they usually treat code snippets as stream of text tokens while ignoring the inherent code structure information. Explicitly utilizing code structure information, such as loop, condition and equation expression, into deep learning models may improve the performance of code summarization.

In this paper, we propose a novel code summarization approach named the CDE-Model to automatically generate code summaries. Different from the previous methods, our approach first utilizes code syntax specification to embed explicit code structure information into raw source code. Then, we feed the transformed code snippets into an encoder-decoder neural network for summary generation. The key contributions of our study are as follows:

[1]https://github.com
DOI reference number: 10.18293/SEKE2018-191

[2]https://stackoverflow.com

- A new code summarization framework that combines deep learning and code structure information.
- A large-scale dataset of Java code summarization task, which contains 396,184 pairs of source code and text summaries.
- A high-performance neural network model that leads to a significant improvement of BLEU readability score.

The rest of this paper is organized as follows. Section II reviews previous works. Section III describes our methods. Section IV shows the experimental result and the last section concludes this paper.

II. Related Work

Many works have been proposed for code summarization. These works can be divided into three categories including information retrieval methods, text mining methods and deep learning methods.

A. Information retrieval

Wong et al. [9] propose an approach which uses information retrieval methods to associate comments in the Q&A community with code snippets. Vassallo et al. [7] propose to use social connection theory to connect code snippets and comments in the StackOverflow. Bahihi et al. [10] presente a method named CrowdSummarizer which exploits crowdsourcing, gamification and natural-language processing to automatically generate high-level summaries of Java program methods. Moreno et al. [11] present a technique to automatically generate human readable summaries for Java classes with heuristics rules.

B. Text mining

Wang et al. [8] present an approach to automatically generate natural language descriptions of Java methods. They identify the statements in the code snippets and extract the keywords to generate sentences as code summaries. McBurney et al. [12] propose a source code summarization technique that generate English descriptions of Java methods by analyzing how those methods are invoked. Hill et al. [13] propose an approach that automatically extracts natural language phrases from source code and categorize the phrases in a hierarchy. These methods could generate the logic description for the code snippets. However, they could not generate code summaries in a high-level abstraction.

C. Deep learning

Iyer et al. [1] present a deep learning model to generate code summaries automatically. They use the LSTM network, a recurrent neural network (RNN), to encode code snippets into a fixed vector and decode vector into code summaries. The model has a good performance on short code snippets and could generate the new words that are not appeared in the code snippets. Paulus R et al. [14] introduce a neural network model with intra-attention, and propose to combine supervised word prediction and reinforcement learning to generate summaries. The result shows that summaries created by reinforcement learning model are more readable. Loyola et al. [15] propose a model to automatically describe changes introduced in the source code of a program with encoder-decoder architecture. The result showed that it can generate feasible and semantically sound descriptions.

III. Overview of CDE-Model

In this section, we describe the framework of our approach. Similar to the works of Iyer et al. [1] and Paulus R et al. [14], we also model the code summarization problem as a sequence-to-sequence learning task which maps a sequence of code tokens into a sequence of natural language tokens. Different to their approaches, we consider the inherent syntactic structure information in the code tokens. We propose to analyze code syntax and weave the syntactic structure information into deep learning networks.

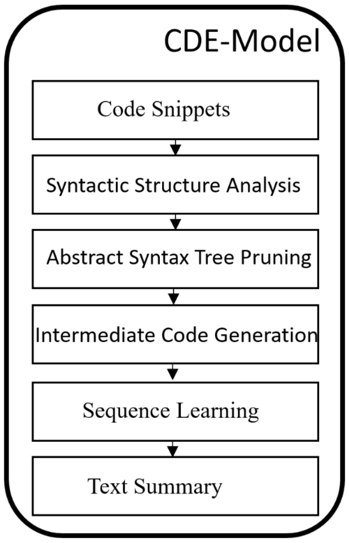

CDE-Model
Code Snippets
Syntactic Structure Analysis
Abstract Syntax Tree Pruning
Intermediate Code Generation
Sequence Learning
Text Summary

Fig. 1: Framework of the CDE-Model

Figure 1 shows the framework of our approach which consists of four steps for code summarization task. Specifically, given a code snippet, we first conduct syntactic structure analysis to generate the abstract syntax tree (AST) which is very useful for code analysis and mining. Second, for code summarization task, what we want is concise human-readable description rather than programming logic description. Thus, not all of the AST nodes can be useful. We propose several heuristic strategies to prune the AST to simplify the code snippet. Third, based on the pruned AST, we generate an intermediate code which is simpler than raw code snippets while preserving the code structure information. Finally, we feed the intermediate code into a sequence-learning neural network model to generate text summary.

The key step in our approach is to prune the AST tree. Programming languages have control statements such as loop and condition to determine the execution trace of software. In this paper, we mainly discuss the pruning strategies that deal with the loop nodes in the AST (see figure 2). Because loop is one of the most common control statements in the code snippets [16].

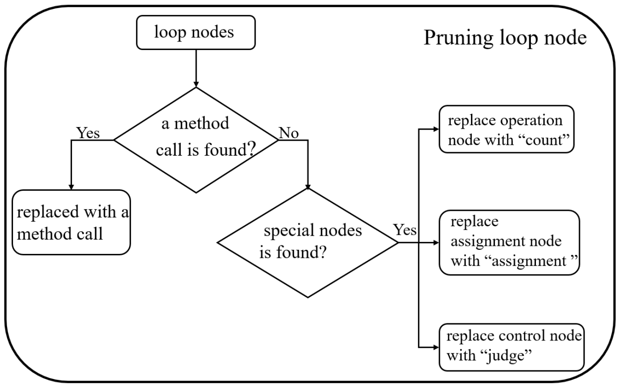
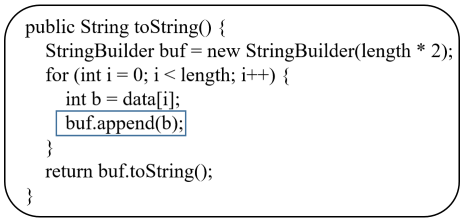

Pruning loop node
loop nodes
a method call is found?
Yes
No
replaced with a method call
special nodes is found?
Yes
replace operation node with “count”
replace assignment node with “assignment ”
replace control node with “judge”

Fig. 2: The key steps in pruning loop nodes

A. Pruning loop node

After we generate the AST from a code snippet, we propose to traverse the AST to find loop nodes. If we find loop nodes such as for and while, we will continue to recursively traverse its child nodes with three different strategies discussed below.

1) If encountering a method call in the loop subtree: If we find a method call in the subtree, we will replace the entire loop subtree with a statement of method call. We present an example in figure 3. We can see that *buf.append(b)* is an object method call inside the loop body. Thus, we use text “*buf append*” to replace the loop node.



```
public String toString() {
    StringBuilder buf = new StringBuilder(length * 2);
    for (int i = 0; i < length; i++) {
        int b = data[i];
        buf.append(b);
    }
    return buf.toString();
}
```

Fig. 3: An example of code snippets

One may say that doing so could remove lots of useful information in the loop subtree. However, we believe that in Java programming language good class method name usually reveals the semantic meaning. In addition, if there are more than one method call, we will choose the last method call in the loop body to replace the entire loop subtree. Because several empirical study show that the later the statement, the more likely it representing the true meaning [14].

2) If encountering special nodes in the loop subtree: If there are no method calls in the loop subtree, we traverse the loop subtree from the last child nodes to the first to find three different type of special nodes including operation nodes, assignment nodes and control nodes. We tabulate the processing methods in table II.

If the first encountered node is mathematic notation, such as “+” “-”, “*”, or “/”, the entire loop subtree will be replaced with a text node of “count”. If the first encountered symbol is “=”, we will give an “assignment” text node as the summary of the loop body. The summary will be taken as “judge” if

TABLE II: Special nodes

Types of nodes	Related symbols	Replacement as text nodes
Operation node	“+”, “-”, “*”, or “/”	“count”
Assignment node	“=”	“assignment”
Control node	“break” “return true” “return false”	“judge”

control statements such as “break”, “return true” or “return false” is first encountered.

3) Other situations: If there are no special nodes in the loop body, we select the last node in the loop subtree to replace the loop body.

After pruning the AST, we expand the AST as normal text stream of code snippet. It should be noted that the code structure information has been explicitly embed in the code snippets. To help clarify our algorithm, we give an example in table III to show the difference of the three strategies.

TABLE III: Code snippets after pruning ASTs

	Before pruning	After pruning

Method Call	void queueIsEmpty() {
	for (Node p = head; p != null;
	p = p.next)
	{Itr it = p.get();
	if (it != null) {
	p.clear();
	it.shutdown();} }
	head = null;
	itrs = null;}
		void queueIsEmpty() {
		it shutdown ;
		head = null;
		itrs = null;
		}

Special Nodes	public ContactIrc getContact(final
	String id) {
	if (id == null | | id.isEmpty()) {
	return null;
	}
	for (ContactIrc contact :
	this.contacts) {
	if (id.equals(
	contact.getAddress())) {
	return contact;}}
	return null;}
		public ContactIrc
		getContact
		(final String id) {
		if (id == null ||
		id.isEmpty()) {
		return null;
		}
		Judge
		return null;
		}

Others	public void removeAt
	(int index, int size) {
	final int end = Math.
	min(mSize, index + size);
	for (int i = index; i ¡ end; i++) {
	removeAt(i);
	}
	}
		public void removeAt
		(int index, int size)
		{
		final int end = Math.
		min(mSize, index + size)
		removeAt(i);
		}

B. Sequence learning

We build a sequence-to-sequence generation system for code summarization task. Our approach use RNN with attention-based mechanism and encoder-decoder architecture to produce code summaries. The network architecture is shown in figure 4

The RNN Encoder-Decoder with attention-based mechanism consists of two RNN that act as an encoder layer and a decoder layer. The encoder layer maps a variable-length source sequence to a fixed-length vector. The decoder layer maps the vector representation back to a variable-length target sequence [17]. The biggest difference in the attention model is that it does not require the encoder to encode all input information into a fixed-length vector [18].

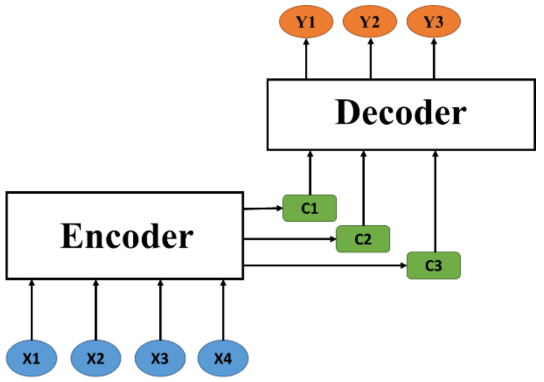


Fig. 4: Basic framework of encoder-decoder neural network

In this paper, we use the LSTM cell for encoder and decoder layers. LSTM has been shown to have good performance in text translation model with attention mechanism to generate one word at a time. A detail introduction of the LSTM and encoder-decoder related neural network can be seen in [19] [20].

IV. Experiments

A. Dataset

In this section, we describe the dataset used in this paper. We collect data from GitHub which is one of the most popular open source community in the world. All the data can be downloaded in the Trustie[3] , a famous open source community in China [21].

We briefly describe the dataset generation process. First, we search in the GHTorrent [22] and use Kraken [23] to crawl the top 1500 popular Java projects ranked by star numbers. Second, we use java-parser[4] tools to analyze source code and extract the code snippets and comments of java API respectively [24]. Those pairs of code snippets and text comments will be treated as training data for our neural network models. In addition, to improve data quality, we remove noisy comments in the dataset. Specifically, we remove the comments shorter than two words. Special symbols except dash "-" and underline "_" are also removed. Eventually, we generate 396,184 pairs of code snippets and summaries. As shown in table IV, the average length of code snippets is of 108.7 words. The average length of text summaries is of 8.8 words.

TABLE IV: Basic information of datasets

Total pairs	Average code length	Average summary length
396,184	108.7	8.8

B. Experiment settings

We implement the CDE-Model based on Tensorflow[5]. We build an encoder-decoder network with 6 layers each with 128 units. We restrict the input vocabulary size to the top 40,000 most frequent code tokens, and the output text vocabulary to the top 40,000 most frequent tokens in the training set. We train the models with a batch size of 128 and a learning rate α of 0.5. We do not stop training the model until perplexity score becomes stable. We use 85% of the dataset for training, 10% for validation and 5% for testing. We has published all of our data and codes in Trustie[6].

To compare our models with the other methods, we also implement two other models based on the CDE-Model. The Del-Model just removes the loop structure in the code snippets while the Gen-Model keeps the code unchanged. We list the three models in table V.

TABLE V: Methods to be compared in experiments

Methods	Description
CDE-Model	Replace the loop structure with heuristic strategies
Del-Model	Delete the loop structure in the code snippets
Gen-Model	Keep the loop structure unchanged in the code snippets

C. Evaluation metrics

We use Bilingual evaluation understudy (BLEU) score as the evaluation metric in this paper. BLEU is an algorithm for evaluating the quality of machine translated text from one natural language to another [25]. Recently, it has become a popular evaluation matric in deep learning based code summarization. In this paper, we report the average BLEU-4 score in experiments, which is often used to measure the quality of text sentences.

D. Experimental results

We tabulate the experiments results in table VI. We find that the CDE-Model outperforms the other methods by a large margin. Specifically, the BLEU-4 values (the second column in table VI) of the CDE-Model is 10.4% higher than the Del-Model and 32.5% higher than the Gen-Model. In addition, we conduct a detailed performance analysis on the code snippets containing loop structure. In our test dataset, there are 2296 code snippets which contain the loop nodes. We measure the BLEU-4 score (the third column in table VI) on this data and find that the CDE-Model is much better than the other methods. Specifically, the BLEU-4 score of the CDE-Model is 6.7% higher than the Del-Model and 85.38% higher than the Gen-Model. This means that utilizing code structure information with empirical data processing strategies into deep learning models can improve the code summarization task significantly.

TABLE VI: BLEU-4 score of different methods

Models	BLEU-4 values	BLEU-4 values containing loop node
CDE-Model	0.52801	0.47548
Del-Model	0.47817	0.42813
Gen-Model	0.37254	0.23058

[3]www.trustie.net
[4]https://www.eclipse.org/jdt/
[5]https://www.tensorflow.org/
[6]https://www.trustie.net/projects/1738

Second, we randomly sample 100 code snippets containing loop nodes to evaluate the three methods. As shown in figure 5, the performance curve of the CDE-Model wins the Del-Model and the Gen-Model in most cases. This further demonstrates that the loop structure has significant impact on the model performance. Replacing the loop structure (by the CDE-model) is much better than the methods that delete the loop structure (by the Del-Model) or keep the loop structure unchanged (by the Gen-Model).

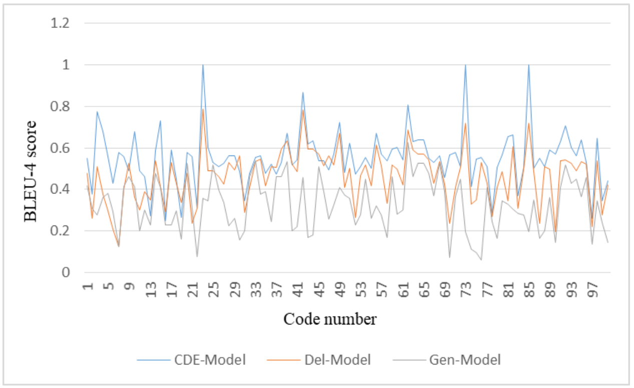


Fig. 5: BLEU-4 score on the sampled data

Third , we analyze the BLEU-4 score of the three methods on the code snippets that satisfying the three conditions (see Section III). As shown in table VII. We find that the BLEU-4 score of the CDE-Model is always better than others in all the three conditions. Specifically, the BLEU-4 score of the CDE-Model is 7% higher than the Del-Model and 84.91% higher than the Gen-Model in *method call* condition. Besides, the CDE-Model owns the best performance than others in *special nodes* condition. Whats more, the BLEU-4 score of the CDE-Model is 4.36% higher than the Del-Model and 66.16% higher than the Gen-Model in *others* condition. This results mean that the three strategies proposed in the CDE-model are very effective.

TABLE VII: BLEU-4 score in three different strategies

Models	CDE-Model	Del-Model	Gen-Model
Method call	0.42410	0.39617	0.22935
Special nodes	0.52627	0.51252	0.25674
Others	0.47175	0.45205	0.28392

E. Case study

Why does the CDE-Model perform better than other two methods? In this subsection, we conducted a detailed case study for qualitative analysis.

Table VIII shows a code snippet with a loop body. The gold standard is written by professional developers. We can see that the code summary generated by our CDE-Model is the most closer to the gold standard compared with the Del-Model and the Gen-Models. Our method successfully captures the semantics of "empty test" while the summaries generate by the Del-Model and the Gen-Model is not meaningful.

For the poor results of Gen-Model, it may be the reason that the loop structure in normal code snippets is usually very complex. Blindly feeding loop body into the encoder-decoder LSTM network may introduce noise. This is why the word "clear" appears both in loop body and Gen-Model summary.

Although the Del-Model removes the noisy loop structure, it loses too much information and thus cannot generate good results. Therefore, in deep learning based code summarization task, it is very important to consider the inherent code structure information.

TABLE VIII: Code snippets and summaries

Code snippet	
	```
	void queueIsEmpty() {
	  for (Node p = head; p != null; p = p.next){
	    Itr it = p.get();
	    if (it != null) {
	      p.clear();
	      it.shutdown();
	    }
	  }
	    head = null;
	    itrs = null;
	}
	```

Gold standard	Called whenever the queue becomes empty
CDE-Model	Called when the buffer has been empty
Del-Model	Called to iterate the observers of this node
Gen-Model	The clear blocks that have been returned

V. Conclusion & Future Work

In this paper, we present and implementation a novel code summarization method name the CDE-Model which combines the deep learning and code structure information. The major feature of the CDE-Model is that it traverses the abstract syntax tree of code snippets to manipulate the complex structural code body to generate intermediate code snippets by empirical strategies. After that, the CDE-model learns an encode-decoder LSTM network for text generation. We conduct large-scale empirical study on 1500 popular Java OSS projects in GitHub. The experimental results and the case study demonstrate that our method is effective.

In the future work, we will try to exploit more advanced sequence learning models to directly encoding AST structure for long code snippets. Besides, we will also study other deep learning methods such as deep reinforcement learning.

Acknowledgment

The research is supported by the National Grand R&D Plan (Grant No. 2016-YFB1000805) and National Natural Science Foundation of China (Grant No. 61502512, 61702532, 61432020, 61472430 and 61532004).

References

[1] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "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, 2016, pp. 2073–2083.

[2] Q. Fan, Y. Yu, G. Yin, T. Wang, and H. Wang, "Where is the road for issue reports classification based on text mining?" in *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*. IEEE, 2017, pp. 121–130.

[3] Y. Zhang, H. Wang, G. Yin, T. Wang, and Y. Yu, "Social media in github: the role of@-mention in assisting software development," *Science China Information Sciences*, vol. 60, no. 3, p. 032102, 2017.

[4] A. E. Prieto, J.-N. Mazón, A. Lozano-Tello, and L.-D. Ibáñez, "Supporting open dataset publication decisions based on open source software reuse," 2018.

[5] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 390–401.

[6] I. J. Mujhid, J. C. Santos, R. Gopalakrishnan, and M. Mirakhorli, "A search engine for finding and reusing architecturally significant code," *Journal of Systems and Software*, vol. 130, pp. 81–93, 2017.

[7] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "Codes: Mining source code descriptions from developers discussions," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 106–109.

[8] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 205–216.

[9] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 562–567.

[10] S. Badihi and A. Heydarnoori, "Crowdsummarizer: Automated generation of code summaries for java programs through crowdsourcing," *IEEE Software*, vol. 34, no. 2, pp. 71–80, 2017.

[11] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 23–32.

[12] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.

[13] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 232–242.

[14] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization," *arXiv preprint arXiv:1705.04304*, 2017.

[15] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *arXiv preprint arXiv:1704.04856*, 2017.

[16] R. Hundt, "Loop recognition in c++/java/go/scala," *Proceedings of Scala Days*, vol. 2011, p. 38, 2011.

[17] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[18] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," *arXiv preprint arXiv:1509.00685*, 2015.

[19] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[20] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[21] H. Wang, G. Yin, X. Li, and X. Li, "Trustie: a software development platform for crowdsourcing," in *Crowdsourcing*. Springer, 2015, pp. 165–190.

[22] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *Mining software repositories (msr), 2012 9th ieee working conference on*. IEEE, 2012, pp. 12–21.

[23] L. Zeng, G. Yin, T. Wang, Y. Yu, Q. Fan, Z.-X. Li, J. Yu, and H. Wang, "Kraken: A continuous incremental data acquisition system for github and git repositories," No. 38 A, Xueqing Road, Haidian District, Beijing, China, 2017, pp. 144 – 149, data extraction;Development activity;Development history;GitHub;Incremental data;Open source communities;Regular patterns;Rest API;.

[24] R. Hosseini and P. Brusilovsky, "Javaparser: A fine-grain concept indexing tool for java problems," in *CEUR Workshop Proceedings*, vol. 1009. University of Pittsburgh, 2013, pp. 60–63.

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "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, 2002, pp. 311–318.