Source Code Summarization

南京大学 软件学院 iSE实验室

目录

01

What

What

(Source) code summarization aims to automatically generate short summaries for a piece of source code.

In the field of software engineering, summaries are also called comments, i.e., code comments.

Code is usually written in programming languages, e.g., Java and Python .

Summaries are usually written in natural languages, e.g., English and Chinese.

Code summarization can also be regarded as a kind of translation task, which translates programming language into natural language [1].

[1] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In Proceedings of the 35th International Conference on Automated Software Engineering. IEEE, Melbourne, Australia, 746–757.

What

(Source) code summarization aims to automatically generate short summaries for a piece of code.

A piece of code is usually a (long) method/function or class.

Summaries are usually short sentences that describe the functionality (semantics) of the code.

Code summarization can also be regarded as a kind of text summarization task, which summarizes a piece of code into short and semantic-preserving sentences [1].

[1] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In Proceedings of the 17th Working Conference on Reverse Engineering. IEEE Computer Society, Beverly, MA, USA, 35–44.

Example

(Source) code summarization aims to automatically generate short summaries for a piece of code.

```
public V remove(final K key) {
    V oldValue = cacheMap.remove(key);
    if (oldValue != null) {
      LOG.debug("Removed cache entry for '{}'", key);
    }
    return oldValue;
}
```

(a) A Code Snippet c_1

removes the mapping for the specified key from this cache if present.

(b) A Summary s_1

02

Why

why

Software maintenance demands as much as 90% of software engineering resources [1], and much of this time is spent understanding the maintenance task and any related software or documentation [2]. In spite of numerous studies demonstrating the utility of comments for understanding software [3-5], few software projects adequately document the code to reduce future maintenance costs [6, 7].

Programmers read comments to gain an understanding of the code quickly. Unfortunately, at present nearly all of these comments are written manually. This manual process is a problem because a typical program may have comments summarizing each method, class, and file in the program. For a large program, developers need to spend a correspondingly large amount of time writing and maintaining each comment by hand. Numerous studies have shown that this manual process is expensive, and that programmers take shortcuts when possible [6-9].

[1] R. C. Seacord, D. Plakosh, and G. A. Lewis. Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.
[2] M. J. Sousa. A Survey on the Software Maintenance. ICSM, 1998.
[3] A. A. Takang, P. A. Grubb, and R. D. Macredie. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. J. Prog. Lang., 4(3), 1996.
[4] T. Tenny. Program Readability: Procedures Versus Comments.TSE, 14(9), 1988.
[5] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The Effect of Modularization and Comments on Program Comprehension.ICSE, 1981.
[6] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. ICDC, 2005.
[7] M. Kajko-Mattsson. A Survey of Documentation Practice within Corrective Maintenance. ESE, 10(1):31–55, 2005.
[8] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In WCRE ’07.
[9] J. Sillito, G. C. Murphy, K. De Volder. Asking and answering questions during a programming change task. TSE.

why

Short summaries of the overall computation the code performs provide a particularly useful form of documentation for a range of applications, such as code search or tutorials [1]. However, such summaries are expensive to manually author. As a result, this laborious process is only done for a small fraction of all code that is produced.

Joint processing of natural languages and programming languages is a research area concerned with tasks such as automated source code documentation, automated code generation from natural language descriptions and code search by natural language queries. These tasks are of great practical interest, since they could increase the productivity of programmers [2], and also of scientific interest due to their difficulty and the conjectured connections between natural language, computation and reasoning [3-5].

[1] Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. Summarizing source code using a neural attention model. *ACL,* 2073-2083, 2016.
[2] Barone, A. V. M., & Sennrich, R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. arXiv, 2017.
[3] Noam Chomsky. Three models for the description of language. IRE Transactions on information theory, 2(3):113–124, 1956
[4] George A Miller. The cognitive revolution: a historical perspective. Trends in cognitive sciences, 7(3):141–144, 2003
[5] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. arXiv, 2014

why

Source code summarization has potential for valuable applications in many software engineering tasks [1], such as:

(a) Understanding new code bases.

Often developers need to quickly familiarize themselves with the core parts of a large code base. This can happen when a developer is joining an existing project, or when a developer is evaluating whether to use a new software library.

(b) Code reviews.

Reviewers need to quickly understand the key changes before reviewing the details.

(c) Locating relevant code segments.

During program maintenance, developers often skim code, reading only a couple lines at a time, while searching for a code region of interest [2].

[1] Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., & Sutton, C. Autofolding for source code summarization. TSE, 43(12), 1095-1109, 2017.

[2] J. Starke, C. Luce, and J. Sillito, “Searching and skimming: An exploratory study,” in ICSM, pp. 157–166, 2009.

03

How

How -> Early Work

Work 1

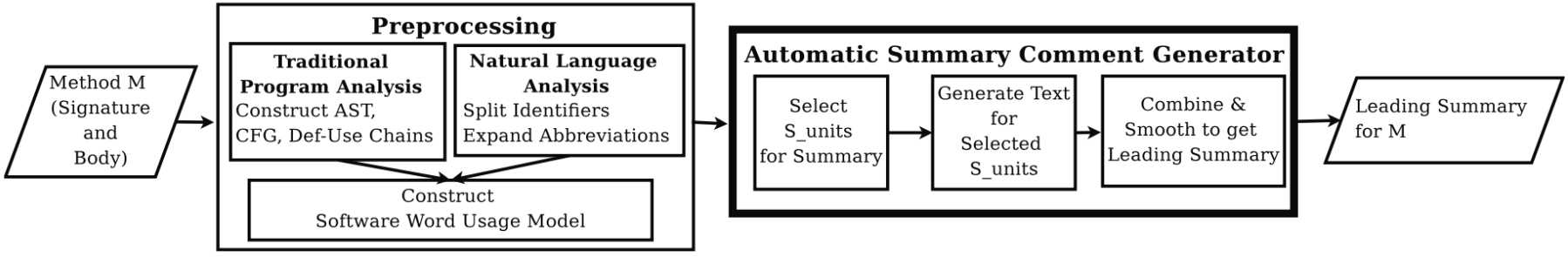


Figure 1: The Summary Comment Generation Process [1]. (ASE 2010)

There are three main components to our approach to automatically generating leading summary comments:
(1) selecting the content, or s_units, to be included in the summary comment,
(2) lexicalizing and generating the natural language text to express the content, and,
(3) combining and smoothing the generated text.

Our design is driven by our goals:
(1) to accurately represent the method's main actions,
(2) to include precise contextual information needed for understanding, and
(3) to be concise and avoid unnecessary words.

[1] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., & Vijay-Shanker, K. Towards automatically generating summary comments for java methods. ASE, 43-52, 2010.

Work 2

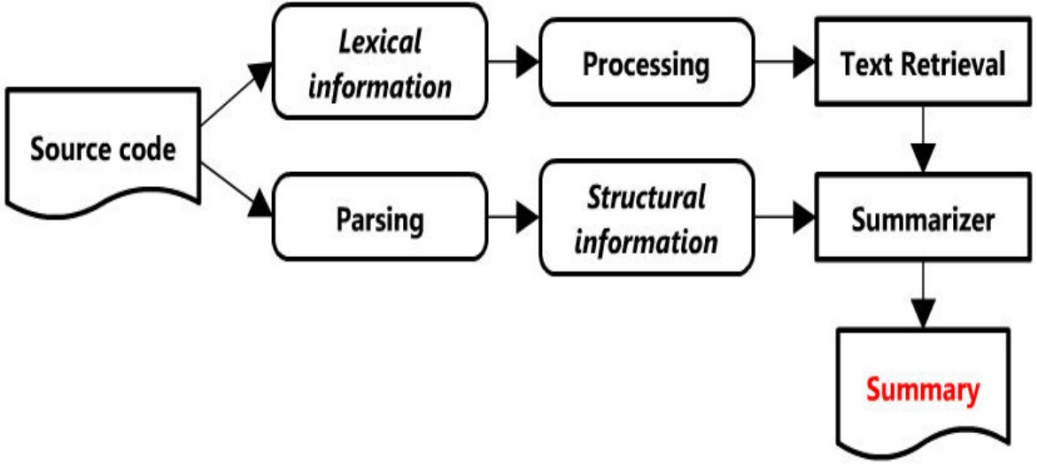


Figure 1: Automatic source code summarization [1]. (ICSE 2010)

Lexical information is extracted from the source code entity to be summarized. Common English and programming language keywords are removed, identifiers are split in their constituent words, and stemming is used to replace English words with their root. The source code is transformed into a text corpus, where each document corresponds to a source code entity. Text retrieval (TR) techniques are used to determine the most important n terms for each document. We attach structural information to the words selected by the TR method, specifying what role each term plays in the source code, such as: class name, method name, parameter name, parameter type, local variable, etc. This type of information is used not only to enrich the summary but also to help build it.

[1] Haiduc, S., Aponte, J., & Marcus, A. Supporting program comprehension with source code summarization. ICSE, 2010/.

How -> Metaphase Work

Work 1
(ICSE 2014)

Current summarization techniques work by selecting a subset of the statements and keywords from the code, and then including information from those statements and keywords in the summary. The quality of the summary depends heavily on the process of selecting the subset: a high-quality selection would contain the same statements and keywords that a programmer would choose. Unfortunately, little evidence exists about the statements and keywords that programmers view as important when they summarize source code.

In this paper [1], we present an eye-tracking study of 10 professional Java programmers in which the programmers read Java methods and wrote English summaries of those methods. We apply the findings to build a novel summarization tool.

[1] Rodeghero, P., McMillan, C., McBurney, P. W., Bosch, N., & D'Mello, S. Improving automated source code summarization via an eye-tracking study of programmers. ICSE, 390-401, 2014.

Work 1

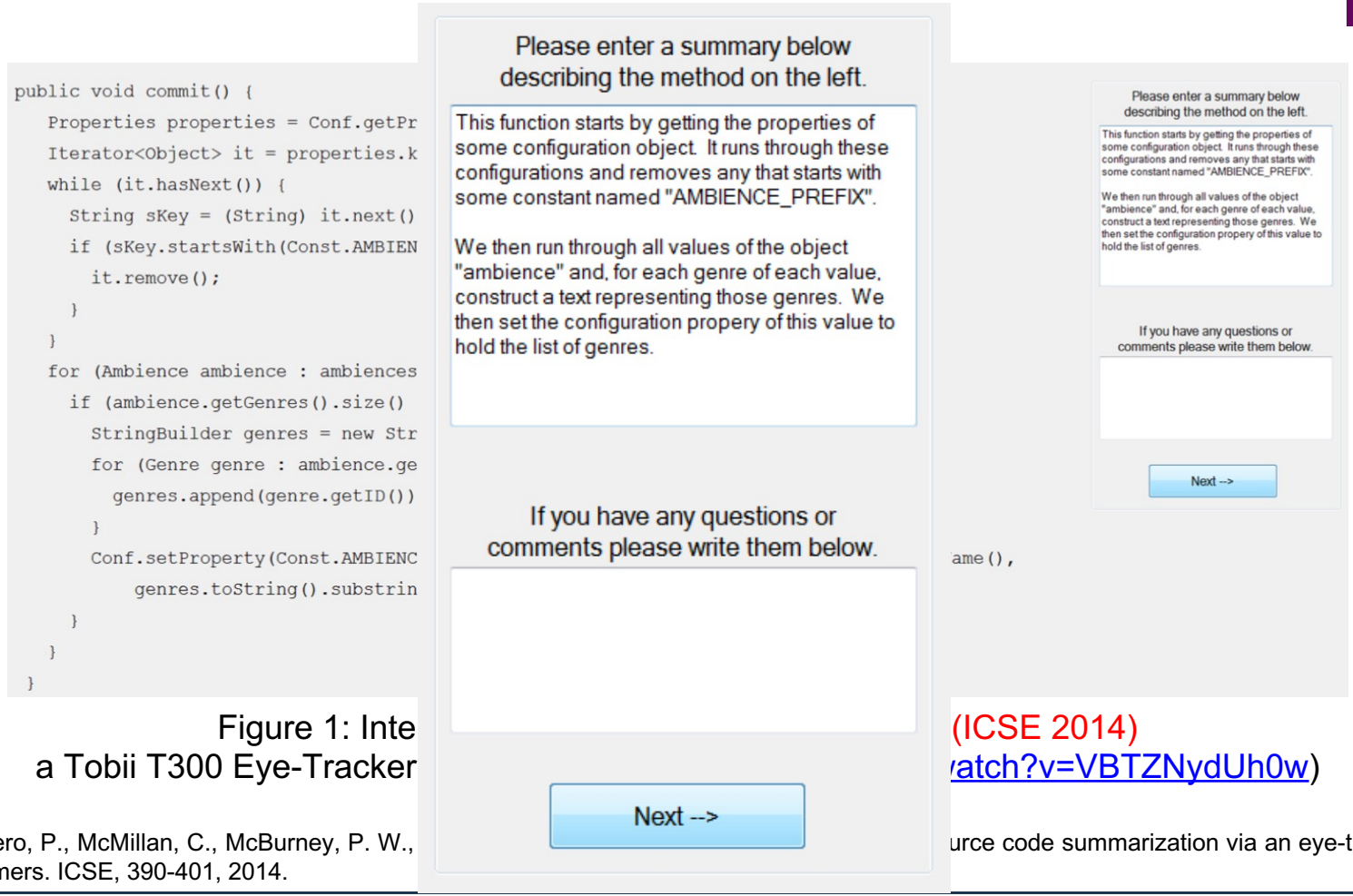


Figure 1: Inte (ICSE 2014)
a Tobii T300 Eye-Tracker atch?v=VBTZNydUh0w)

[1] Rodeghero, P., McMillan, C., McBurney, P. W., urce code summarization via an eye-tracking study of programmers. ICSE, 390-401, 2014.

Work 1

The long-term goal of this study is to discover what keywords from source code should be included in the summary of that code. Towards this goal, we highlight four areas of code that previous studies have suggested as being useful for deriving keywords. We study these areas in the four Research Questions (RQ) that we pose:

RQ1: To what degree do programmers focus on the keywords that the VSM tf/idf technique [1, 2] extracts?

RQ2: Do programmers focus on a method's signature more than the method's body?

RQ3: Do programmers focus on a method's control flow more than the method's other areas?

RQ4: Do programmers focus on a method's invocations more than the method's other areas?

[1] B. Eddy, J. Robinson, N. Kraft, and J. Carver. Evaluating source code summarization techniques: Replication and expansion. ICPC, 2013.
[2] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. WCRE, 2010.

Work 1

RQ1: To what degree do programmers focus on the keywords that the VSM tf/idf technique extracts?

The rationale behind RQ1 is that two independent studies have confirmed that a VSM tf/idf approach to extracting keywords outperforms different alternatives [26, 18]. If programmers tend to read these words more than others, it would provide key evidence that the approach simulates how programmers summarize code.

Answering RQ1:
We found evidence that the VSM tf/idf approach extracts a list of keywords that approximates the list of keywords that programmers read during summarization.

Work 1

RQ2: Do programmers focus on a method's signature more than the method's body?

Similarly, both of these studies found that in select cases, a "lead" approach outperformed the VSM approach. The lead approach returns keywords from the method's signature. Therefore, in RQ2, we study the degree to which programmers emphasize these signature terms when reading code.

Answering RQ2:
We found statistically-significant evidence that, during summarization, programmers read a method's signature more-heavily than the method's body. The programmers read the signatures in a greater proportion than the signatures' sizes. On average, the programmers spent 18% of their gaze time reading signatures, even though the signatures only averaged 12% of the methods.

Work 1

RQ3: Do programmers focus on a method's control flow more than the method's other areas?

We pose RQ3 in light of related work which suggests that programmers comprehend code by comprehending its control flow [1, 1], while contradictory evidence suggests that other regions may be more valuable [3].

Answering RQ3:
We found statistically significant evidence that programmers tended to read control flow keywords less than the keywords from other parts of the method. On average, programmers spent 31% of their time reading control flow keywords, even though these keywords averaged 37% of the keywords in the methods.

[1] D. Dearman, A. Cox, and M. Fisher. Adding control-flow to a visual data-flow representation. IWPC, 297–306, Washington, DC, USA, 2005.
[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. TSE, 32(12):971–987, 2006.
[3] K. Anjaneyulu and J. Anderson. The advantages of data flow diagrams for beginning programming.1992.

Work 1

RQ4: Do programmers focus on a method's invocations more than the method's other areas?

The rationale behind RQ4 is that method invocations are repeatedly suggested as key elements in program comprehension [40, 50, 31, 60]. Keywords from method calls may be useful in summaries if programmers focusing on them when summarizing code.

Answering RQ4:
We found no evidence that programmers read keywords from method invocations more than keywords from other parts of the methods. Programmers read the invocations in the same proportion as they occurred in the methods.

[1] D. Dearman, A. Cox, and M. Fisher. Adding control-flow to a visual data-flow representation. IWPC, 297–306, Washington, DC, USA, 2005.
[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. TSE, 32(12):971–987, 2006.
[3] K. Anjaneyulu and J. Anderson. The advantages of data flow diagrams for beginning programming.1992.

Work 1

Proposed Approach

The key idea behind our approach is to modify the weights we assign to different keywords, based on how programmers read those keywords.

In the VSM tf/idf approach, all occurrences of terms are treated equally: the term frequency is the count of the number of occurrences of that term in a method.

In our approach, we weight the terms based on where they occur. Specifically, in light of our eye-tracking results, we weight keywords differently if they occur in method signatures, control flow, or invocations.

Work 2
(TSE 2015)

Source code summarization is a critical component of documentation generation, for example as Javadocs formed from short paragraphs attached to each method in a Java program. At present, a majority of source code summarization is manual, in that the paragraphs are written by human experts. However, new automated technologies are becoming feasible. These automated techniques have been shown to be effective in select situations, though a key weakness is that they do not explain the source code's context. That is, they can describe the behavior of a Java method, but not why the method exists or what role it plays in the software.

In this paper [1], we propose a source code summarization technique that writes English descriptions of Java methods by analyzing how those methods are invoked.

[1] McBurney, P. W., & McMillan, C. Automatic source code summarization of context for java methods. IEEE Transactions on Software Engineering, 42(2), 103-119, 2015.

Work 2

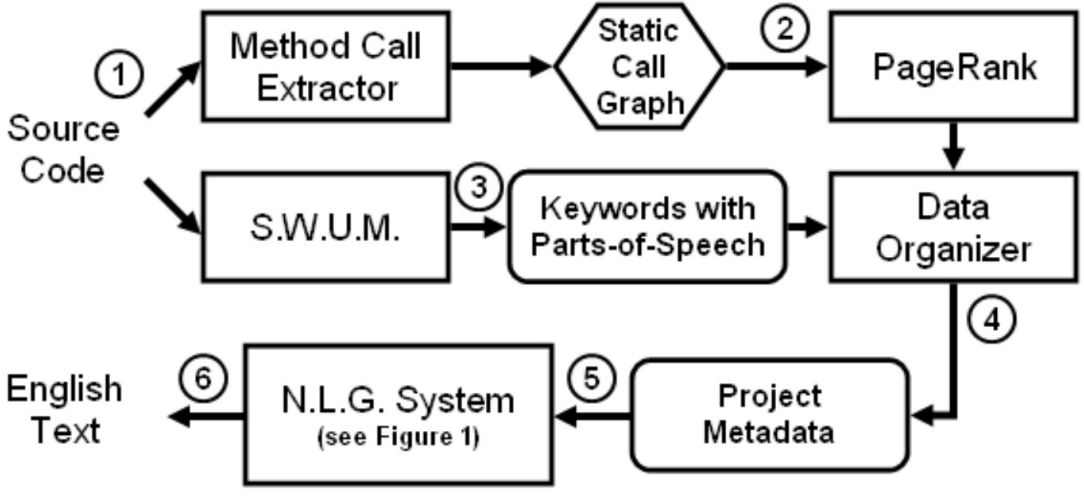


Figure 1: Overview of the proposed approach.

The proposed approach creates a summary of a given method in three steps:
1) use PageRank to discover the most important methods in the given method's context,
2) use data from SWUM to extract keywords about the actions performed by those most important methods,
3) use a custom NLG system to generate English sentences describing for what the given method is used.

Work 2

Step 1) use PageRank to discover the most important methods in the given method's context.

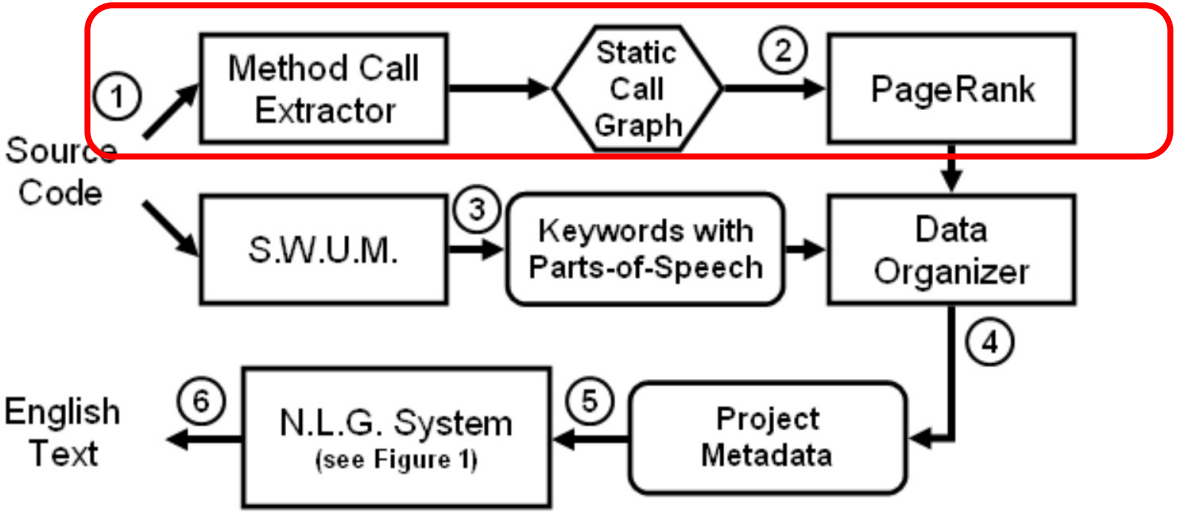


Figure 1: Overview of the proposed approach.

We produce a call graph of the project for which we are generating comments. Our call graph allows us to see where a method is called so that we can determine the method's context (Figure 1, area 2). Finally, we obtain a PageRank value for every method by executing the PageRank algorithm.

Step 2) use data from SWUM to extract keywords about the actions performed by those most important methods.

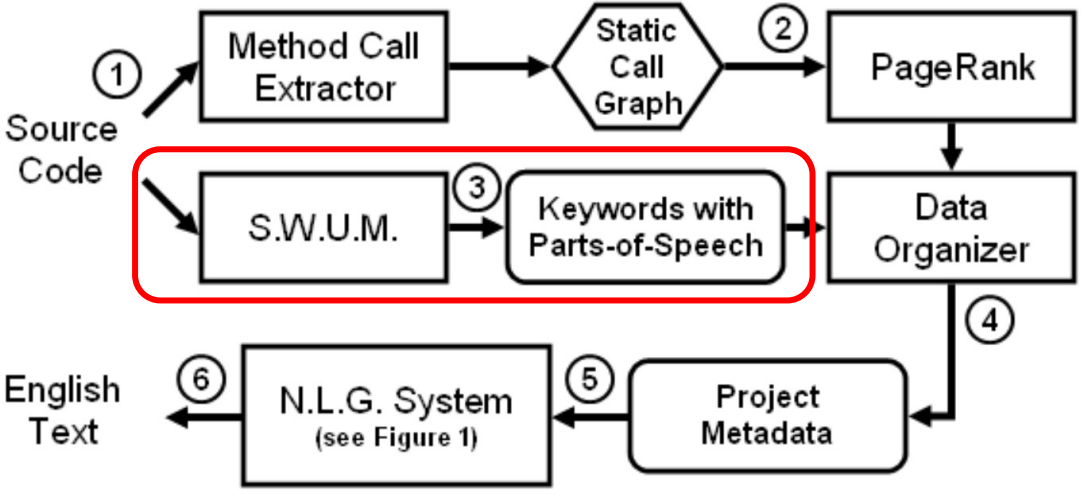


Figure 1: Overview of the proposed approach.

SWUM parses the grammatical structure from the function and argument names in a method declaration. This allows us to describe the method based on the contents of its static features. Specifically, SWUM outputs the keywords describing the methods, with each keyword tagged with a part of speech (Figure 1, area 3).

Work 2

Step 3) use a custom NLG system to generate English sentences describing for what the given method is used.

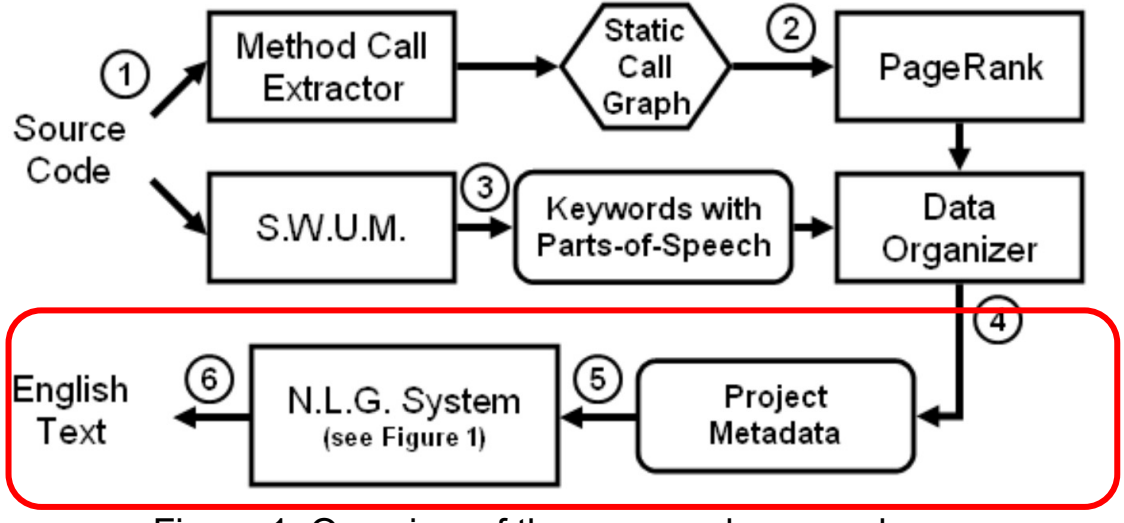


Figure 1: Overview of the proposed approach.

Natural Language Generation (NLG) system processes the Project Metadata as input and generate English text (Figure 1, areas 5 and 6).

Work 2

Step 3) -> NLG System

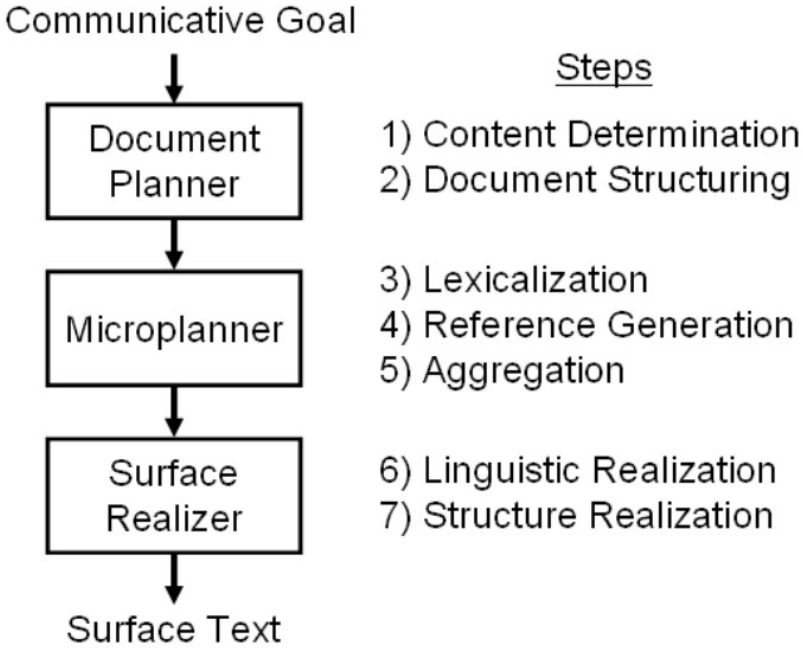


Figure 1: The typical design of a Natural Language Generation system as described by Reiter and Dale [1].

[1] E. Reiter and R. Dale. Building natural language generation systems. Cambridge University Press, New York, NY, USA, 2000.

Work 2

Step 3) -> NLG System -> Document Structuring.

We create six different types of “messages” that represent information about a method’s context.

Table 1: A quick reference guide for types of messages our approach creates.

Message Type	Explanation
Quick Summary Message	Short sentence that describes method
Return Message	Notes the return type of the method
Importance Message	States how important a method is based on PageRank
Output Used Message	Describe at most 2 methods that call this method
Call Message	Describe at most 2 methods that this method calls
Use Message	Gives an example of how the message can be used.

Work 2

Step 3) -> NLG System -> Content Determination.

After generating the initial messages in the content determination phase, we organize all the messages into a single document plan.

We use a templated document plan where messages occur in a pre-defined order: Quick Summary Messages, Return Messages, Output Used Messages, Called Messages, Importance Messages, and then Use Messages.

This ordering was decided based on internal exploratory pilot studies. We decided on this order as we felt this order was the most natural to read.

Work 2

Step 3) -> NLG System -> Lexicalization.

Each type of message needs a different type of phrase to describe it.

The Quick Summary Message records a verb/direct-object relationship between two words extracted by SWUM. The conversion to a sentence is simple in this case: the verb becomes the verb in the sentence, and likewise for the direct-object. The subject is assumed to be “the method”, but is left out for brevity. The message is then created as “*This method verb direct object*.” In some cases, the article “the” is added before the direct object.

The Importance Message holds both the method’s PageRank and an average PageRank. To create a phrase for this type of message, we set the subject as “this method”, the verb as “seems”, and the object as “important.” If the method’s PageRank is more than 150% of the average, we add the modifier “far more.” If it is between 100% and 150%, we consider it “slightly more”, while if it is less than 100%, we use the modifier “less”.

Work 2

Step 3) -> NLG System -> Lexicalization.

Each type of message needs a different type of phrase to describe it.

We create a phrase for an Output Usage Message by setting the object as the return type of the method, and the verb as “is”. The subject is the phrase generated from the Quick Summary Message. We set the voice of the phrase to be passive. We decided to use passive voice to emphasize how the return data is used, rather than the contents of the Quick Summary Message.

The Use Message is created with the subject “this method”, the verb phrase “can be used”, and appending the prepositional phrase ”as a statement type;”. Statement type is pulled from the data structures populated in our content determination step. Additionally, we append a second dependent clause ”for example: code”.

Work 2

Step 3) -> NLG System -> Reference Generation and Aggregation.

During Aggregation, we create more complex and readable phrases from the phrases generated during Lexicalization. Our system works by looking for patterns of message types, and then grouping the phrases of those messages into a sentence. Notice that Reference Generation occurs alongside Aggregation.

Step 3) -> NLG System -> Surface Realization.

We use an external library, simplenlg [1], to realize complete sentences from the phrases formed during Aggregation. In the above steps, we set all words and parts of speech and provided the structure of the sentences. The external library follows English grammar rules to conjugate verbs, and ensure that the word order, plurals, and articles are correct. This step outputs an English summary of the method (Figure 1, area 6).

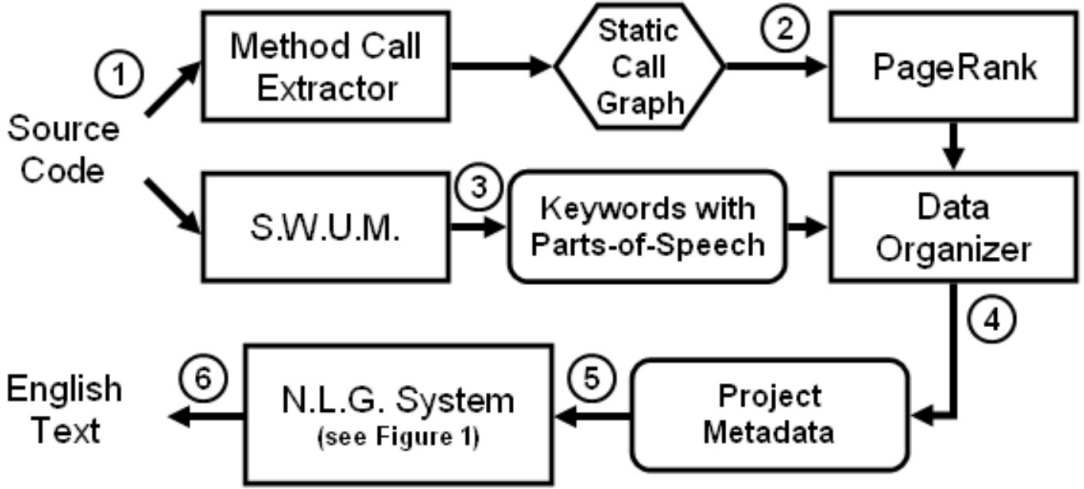


Figure 1: Overview of the proposed approach.

[1] A. Gatt and E. Reiter. Simplenlg: a realisation engine for practical applications. ENLG, 90–93, 2009.

Work 2

Example.

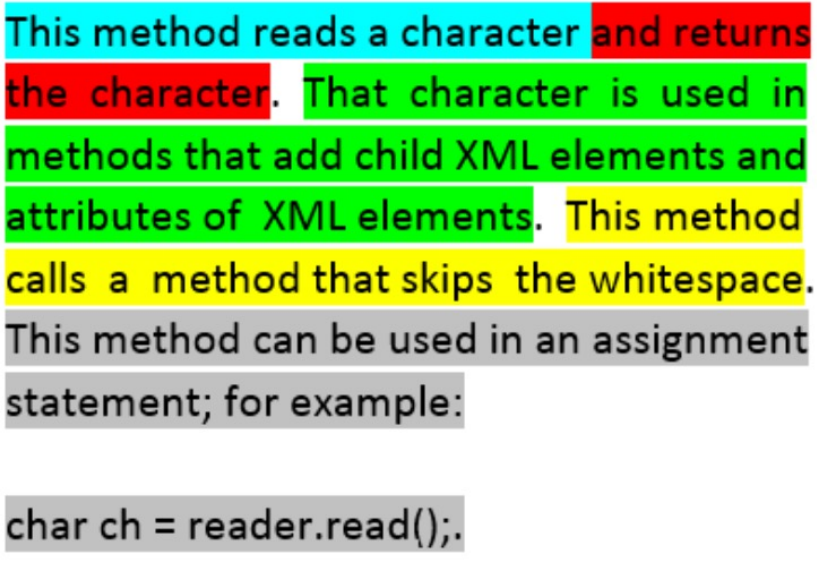


Figure 1: An example of a summary produced by our approach with the different message types highlighted.
The Quick Summary Message is highlighted blue.
The Return Message is highlighted red.
The Output Usage Message is highlighted Green.
The Call Message is highlighted Yellow.
The Use Message is highlighted grey.
The Importance Message is not shown, as it was not used in the most recent version of our approach.

How -> Recent Work

Work 1
(ACL 2016)

High quality source code is often paired with high level summaries of the computation it performs, for example in code documentation or in descriptions posted in online forums. Such summaries are extremely useful for applications such as code search but are expensive to manually author, hence only done for a small fraction of all code that is produced.

In this paper [1], we present the first completely data-driven approach for generating high level summaries of source code.

Deep Learning

[1] Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. Summarizing source code using a neural attention model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2073-2083, 2016.

Work 1

Natural language generation has traditionally been addressed as a pipeline of modules that decide ‘what to say’ (content selection) and ‘how to say it’ (realization) separately [1-4]. Such approaches require supervision at each stage and do not scale well to large domains.

We instead propose an end-to-end neural network called CODE-NN that jointly performs content selection using an attention mechanism, and surface realization using Long Short Term Memory (LSTM) networks. The system generates a summary one word at a time, guided by an attention mechanism over embeddings of the source code, and by context from previously generated words provided by a LSTM network [5]. The simplicity of the model allows it to be learned from the training data without the burden of feature engineering [6] or the use of an expensive approximate decoding [7].

[1] Ehud Reiter and Robert Dale. Building natural language generation systems. Cambridge University Press, 2000.
[2] Yuk Wah Wong and Raymond J Mooney. Generation by inverting a semantic parser that uses statistical machine translation. NAACL,172–179, 2007.
[3] David L Chen, Joohyun Kim, and Raymond J Mooney. Training a multilingual sportscaster: Using perceptual context to learn language. AI 397–435, 2010.
[4] Wei Lu and Hwee Tou Ng. A probabilistic forest-to-string model for language generation from typed lambda calculus expressions. EMNLP, 2011.
[5] Sepp Hochreiter and Ju ̈rgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.
[6] Gabor Angeli, Percy Liang, and Dan Klein. A simple domain-independent probabilistic approach to generation. EMNLP, 502–512, 2010.
[7] Ioannis Konstas and Mirella Lapata. A global model for concept-to-text generation. Journal of Artificial Intelligence Research, 48(1):305–346, 2013.

Work 1

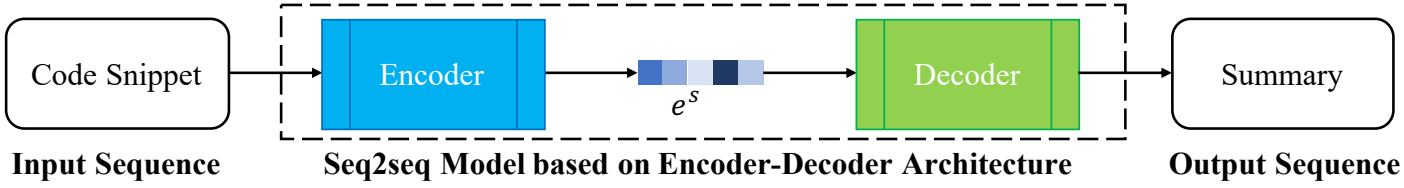


Figure 1: Framework of DL-based Code Summarization.

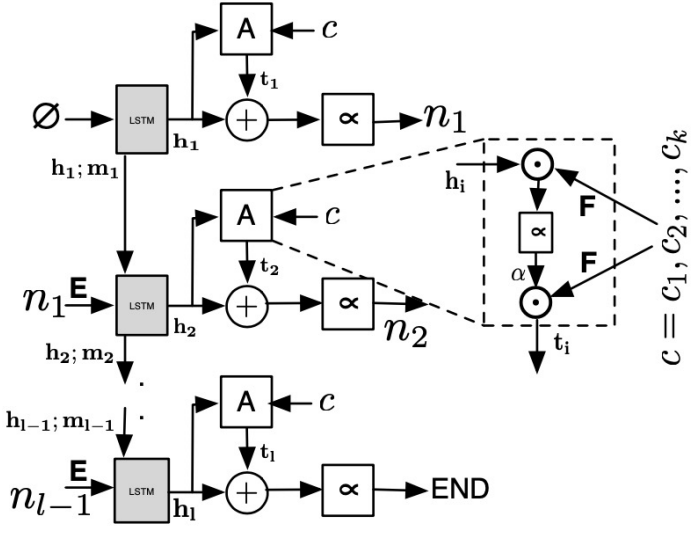


Figure 2: Generation of a summary $n = n_1$,..., END given code snippet c_1 , ..., c_k.

Work 1

Evaluation -> Metrics -> Automatic Evaluation.

We report METEOR [1] and sentence level BLEU-4 [2] scores.

METEOR is recall-oriented and measures how well our model captures content from the references in our output.

BLEU-4 measures the average n-gram precision on a set of reference sentences, with a penalty for overly short sentences. Since the generated summaries are short and there are multiple alternate summaries for a given code snippet, higher order n-grams may not overlap. We remedy this problem by using +1 smoothing [3].

[1] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In Proceedings of the ACL workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, 65–72, 2005.
[2] 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, 311–318, 2002.
[3] Chin-Yew Lin and Franz Josef Och. Orange: a method for evaluating automatic evaluation metrics for machine translation. In Proceedings of the 20th international conference on Computational Linguistics, 501, 2004.

Evaluation -> Metrics -> Human Evaluation.

Since automatic metrics do not always agree with the actual quality of the results [1], we perform human evaluation studies to measure the output of our system and baselines across two modalities, namely *naturalness* and *informativeness*.

For the *naturalness*, we asked 5 native English speakers to rate each title against grammaticality and fluency, on a scale between 1 and 5.

For *informativeness* (i.e., the amount of content carried over from the input code to the NL summary, ignoring fluency of the text), we asked 5 human evaluators familiar with C# and SQL to evaluate the system output by rating the factual overlap of the summary with the reference titles, on a scale between 1 and 5.

[1] Amanda Stent, Matthew Marge, and Mohit Singhai. Evaluating evaluation methods for generation in the presence of variation. In Computational Linguistics and Intelligent Text Processing, 341– 351, 2005.

Work 1

Evaluation -> Results -> Automatic Evaluation Results.

Table 1: Performance on EVAL for the GEN task. Performance on DEV is indicated in parentheses.

	Model	METEOR	BLEU-4
C#	IR	7.9 (6.1)	13.7 (12.6)
	MOSES	9.1 (9.7)	11.6 (11.5)
	SUM-NN	10.6 (10.3)	19.3 (18.2)
	CODE-NN	**12.3 (13.4)**	**20.5 (20.4)**
SQL	IR	6.3 (8.0)	13.5 (13.0)
	MOSES	8.3 (9.7)	15.4 (15.9)
	SUM-NN	6.4 (8.7)	13.3 (14.2)
	CODE-NN	**10.9 (14.0)**	**18.4 (17.0)**

Evaluation -> Results -> Human Evaluation Results.

Table 1: Naturalness and Informativeness measures of model outputs.
Stat. sig. between CODE-NN and others is computed with a 2-tailed Student's t-test; $p < 0.05$ except for *.

	Model	Naturalness	Informativeness
C#	IR	3.42	2.25
	MOSES	1.41	2.42
	SUM-NN	**4.61***	1.99
	CODE-NN	**4.48**	**2.83**
SQL	IR	3.21	2.58
	MOSES	2.80	2.54
	SUM-NN	4.44	2.75
	CODE-NN	**4.54**	**3.12**

Work 1

Examples.

Method	Output

C# code	`var x = "FundList[10].Amount";`
	`int xIndex = Convert.ToInt32(`
	`  Regex.Match(x,@"\d+").Value);`

Gold	Identify the number in given string
IR	Convert string number to integer
MOSES	How to xIndex numbers in C#?
SUM-NN	How can I get the value of a string?
CODE-NN	How to convert string to int?

C# code	`foreach (string pTxt in xml.parent) {`
	`  TreeNode parent = new TreeNode();`
	`  foreach (string cTxt in xml.child) {`
	`  TreeNode child = new TreeNode();`
	`  parent.Nodes.Add(child);`
	`  }`
	`}`

Gold	Adding childs to a treenode dynamically in C#
IR	How to set the name of a tabPage programmatically
MOSES	How can TreeView nodes from XML parentText string to a treeview node
SUM-NN	How to get data from xml file in C#
CODE-NN	How to get all child nodes in TreeView?

Work 2
(EMNLP 2020)

Large pre-trained models such as GPT[1], BERT [2], and RoBERTa [3] have dramatically improved the state-of-the-art on a variety of natural language processing (NLP) tasks. These pre-trained models learn effective contextual representations from massive unlabeled text optimized by self-supervised objectives, such as masked language modeling, which predicts the original masked word from an artificially masked input sequence. The success of pre-trained models in NLP also drives a surge of multi-modal pre-trained models, such as ViLBERT [4] for language-image and VideoBERT [5] for language-video.

We present CodeBERT [6], a bimodal pre-trained model for natural language (NL) and programming language (PL). CodeBERT captures the semantic connection between natural language and programming language, and produces general-purpose representations that can broadly support NL-PL understanding tasks (e.g. natural language code search) and generation tasks (e.g. code documentation generation).

[1] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.
[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.
[3] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. 2019.
[4] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. In Advances in Neural Information Processing Systems, 13–23, 2019.
[5] Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. Videobert: A joint model for video and language representation learning. arXiv, 2019.
[6] Zhangyin Feng and Daya Guo and Duyu Tang and Nan Duan and Xiaocheng Feng and Ming Gong and Linjun Shou and Bing Qin and Ting Liu and Daxin Jiang and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. 2020

Work 2

CodeBERT -> Pre-Training CodeBERT.

We describe the two objectives used for training CodeBERT here.

The first objective is masked language modeling (MLM), which has proven effective in literature [1-3]. We apply masked language modeling on bimodal data of NL-PL pairs.

The second objective is replaced token detection (RTD), which further uses a large amount of unimodal data, such as codes without paired natural language texts [4].

$$\min_{\theta} \mathcal{L}_{\mathrm{MLM}}(\theta) + \mathcal{L}_{\mathrm{RTD}}(\theta)$$

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.
[2] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. 2019.
[3] Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. Videobert: A joint model for video and language representation learning. arXiv, 2019.
[4] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: Pre-training text encoders as discriminators rather than generators. ICLR, 2020.

Work 2

CodeBERT -> Pre-Training CodeBERT -> Masked Language Modeling.

Given a datapoint of NL-PL pair (**x** = {**w**, **c**}) as input, where **w** is a sequence of NL words and **c** is a sequence of PL tokens, we first select a random set of positions for both NL and PL to mask out (i.e. $\boldsymbol{m^w}$ and $\boldsymbol{m^c}$, respectively), and then replace the selected positions with a special [MASK] token. Following Devlin et al. [1], 15% of the tokens from **x** are masked out.

The MLM objective is to predict the original tokens which are masked out, formulated as follows, where p^{D1} is the discriminator which predicts a token from a large vocabulary.

$$\mathcal{L}_{\mathrm{MLM}}(\theta) = \sum_{i \in \boldsymbol{m^w} \cup \boldsymbol{m^c}} -\log p^{D_1}(x_i | \boldsymbol{w}^{\mathrm{masked}}, \boldsymbol{c}^{\mathrm{masked}})$$

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

Work 2

CodeBERT -> Pre-Training CodeBERT -> Replaced Token Detection.

The discriminator is trained to determine whether a word is the original one or not, which is a binary classification problem. The loss function of RTD with regard to the discriminator parameterized by θ is given below, where δ(i) is an indicator function and p^{D2} is the discriminator that predicts the probability of the *i*-th word being original.

$$\mathcal{L}_{\mathrm{RTD}}(\theta) = \sum_{i=1}^{|\boldsymbol{w}|+|\boldsymbol{c}|} \Big(\delta(i) \log p^{D_2}(\boldsymbol{x}^{\mathrm{corrupt}}, i) + \Big(1-\delta(i)\Big)\Big(1-\log p^{D_2}(\boldsymbol{x}^{\mathrm{corrupt}}, i)\Big)\Big)$$

Work 2

CodeBERT -> Fine-Tuning CodeBERT.

We have different settings to use CodeBERT in downstream NL-PL tasks.

For example, in natural language code search, we feed the input as the same way as the pre-training phase and use the representation of [CLS] to measure the semantic relevance between code and natural language query,

while in code-to-text generation, we use an encoder-decoder framework and initialize the encoder of a generative model with CodeBERT.

Work 2

Results -> Code Comment Generation Results.

Although the pre-training objective of CodeBERT does not include generation-based objectives [1], we would like to investigate to what extent does CodeBERT perform on generation tasks.

Specifically, we study code-to-NL generation, and report results for the documentation generation task on CodeSearchNet Corpus in six programming languages. Since the generated documentations are short and higher order n-grams may not overlap, we remedy this problem by using smoothed BLEU score [2].

[1] Mike Lewis, Yinhan Liu, Naman Goyal, Mar- jan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv, 2019.
[2] Chin-Yew Lin and Franz Josef Och.. Orange: a method for evaluating automatic evaluation metrics for machine translation. In Proceedings of the 20th international conference on Computational Linguistics, 501, 2004.

Work 2

Results -> Code Comment Generation Results.

As we can see, models pre-trained on programming language outperform RoBERTa, which illustrates that pre-trainning models on programming language could improve code-to-NL generation.

Table 1: Results on Code-to-Comment generation, evaluated with smoothed BLEU-4 score.

MODEL	RUBY	JAVASCRIPT	GO	PYTHON	JAVA	PHP	OVERALL
SEQ2SEQ	9.64	10.21	13.98	15.93	15.09	21.08	14.32
TRANSFORMER	11.18	11.59	16.38	15.81	16.26	22.12	15.56
ROBERTA	11.17	11.90	17.72	18.14	16.47	24.02	16.57
PRE-TRAIN W/ CODE ONLY	11.91	13.99	17.78	18.58	17.50	24.34	17.35
CODEBERT (RTD)	11.42	13.27	17.53	18.29	17.35	24.10	17.00
CODEBERT (MLM)	11.57	14.41	17.78	18.77	17.38	24.85	17.46
CODEBERT (RTD+MLM)	**12.16**	**14.90**	**18.07**	**19.06**	**17.65**	**25.16**	**17.83**

How -> Our Work

EACS

Existing code summarization techniques can be divided into two categories.

(1) Extractive method:
It extracts a subset of important statements and keywords from the code snippet using retrieval techniques and generate a summary that preserves factual details in important statements and keywords. However, the extractive summary has a poor naturalness.

(2) Abstractive method:
It can generate human-written-like summaries leveraging encoder-decoder models. However, the generated summaries often miss important factual details.

EACS is an extractive-and-abstractive technique for code summarization, which inherits the advantages of extractive and abstractive methods and shields their respective disadvantages.

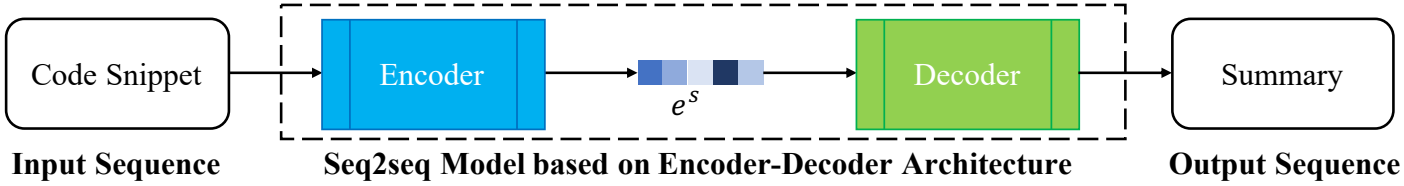


Figure 1: Framework of DL-based Code Summarization.

Figure 1 shows the general framework of the code summarization technique based on the encoder-decoder architecture. Many code summarization techniques that take abstractive methods are closely related to this architecture.

The encoder is an embedding network that can encode the code snippet *s* given by the developer into a *d*-dimensional embedding representation $\boldsymbol{e}^s$.

The decoder is also a neural network that can decode the embedding representation $\boldsymbol{e}^c$ into a natural language summary.

Motivating Example.

```
public V remove(final K key) {
    V oldValue = cacheMap.remove(key);
    if (oldValue != null) {
      LOG.debug("Removed cache entry for '{}'", key);
    }
    return oldValue;
}
```

(a) A Code Snippet c_1

1. **Reference Summary:** removes the mapping for the specified key from this cache if present.
2. **Extractive Summary:** removed key cache old value.
3. **Abstractive Summary:** removes the entry from the cache.
4. **ExAbstractive Summary:** removes the value for the given key from the cache if it exists.

(b) Summaries Generated by Different Techniques

Figure 1: Motivating Example

The first line of Figure 1(b) is a reference summary (the ground truth). According to the grammar rules, we can simply divide the reference summary into four parts:
“removes the mapping” (Blue font)
“for the specified key” (Green font)
“from this cache” (Red font)
“if present” (Orange font)

1. **Reference Summary:** removes the mapping for the specified key from this cache if present.
2. **Extractive Summary:** removed key cache old value.
3. **Abstractive Summary:** removes the entry from the cache.
4. **ExAbstractive Summary:** removes the value for the given key from the cache if it exists.

(b) Summaries Generated by Different Techniques

Figure 1: Motivating Example

The second summary in Figure 1 (Extractive Summary) is generated by [1] which adopts the Latent Semantic Analysis (LSA) techniques [2] to determine the informativity of every term in the code snippet and then select the top *k* important terms to compose the summary.

Observe that although the extractive summary has a poor naturalness and is far from the reference summary, it contains important factual details that should be included in the summary, e.g., the important terms “key” and “cache”.

[1] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting Program Comprehension with Source Code Summarization. In Proceedings of the 32nd International Conference on Software Engineering. ACM, Cape Town, South Africa, 223–226.
[2] Josef Steinberger and Karel Jezek. 2009. Update Summarization based on Latent Semantic Analysis. In Proceedings of the 12th International Conference on Text, Speech and Dialogue. Springer, Pilsen, Czech Republic, 77–84

1. **Reference Summary:** removes the mapping for the specified key from this cache if present.
2. **Extractive Summary:** removed key cache old value.
3. **Abstractive Summary:** removes the entry from the cache.
4. **ExAbstractive Summary:** removes the value for the given key from the cache if it exists.

(b) Summaries Generated by Different Techniques

Figure 1: Motivating Example

The third summary in Figure 1 (Abstractive Summary) shows the result by [1], which first trains a model called CodeBERT for obtaining code representations and then fine-tunes it on the code summarization task.

Observe that **(1)** intuitively, the abstractive summary has a good naturalness and is like written by a human; **(2)** the abstractive summary can cover the first and the third parts (Blue and Red fonts) of the reference summary; **(3)** the abstractive summary can not cover the second and the fourth parts (Green and Orange fonts), i.e., missing some factual details.

[1] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings. Association for Computational Linguistics, Online Event, 1536–1547.

EACS

1. **Reference Summary:** removes the mapping for the specified key from this cache if present.
2. **Extractive Summary:** removed key cache old value.
3. **Abstractive Summary:** removes the entry from the cache.
4. **ExAbstractive Summary:** removes the value for the given key from the cache if it exists.

(b) Summaries Generated by Different Techniques

Figure 1: Motivating Example

The last summary (ExAbstractive Summary) in Figure 1 is generated by EACS.

We can observe that:

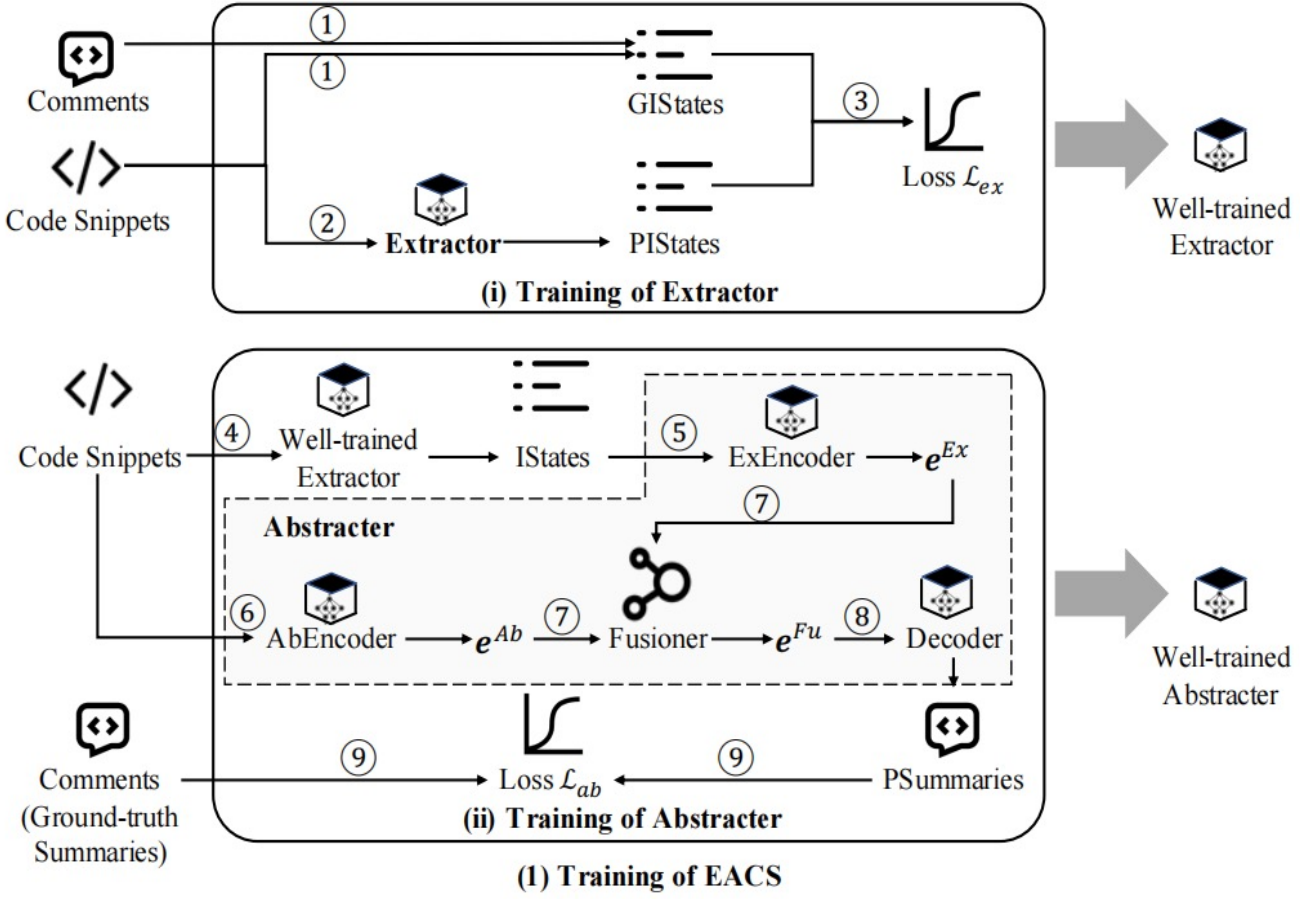
1) compared with the extractive summary generated by [1], the exabstractive summary has a good naturalness and is like written by a human;

2) compared with the abstractive summary generated by [2], the exabstractive summary can cover all of the four parts and is closer to the reference summary.

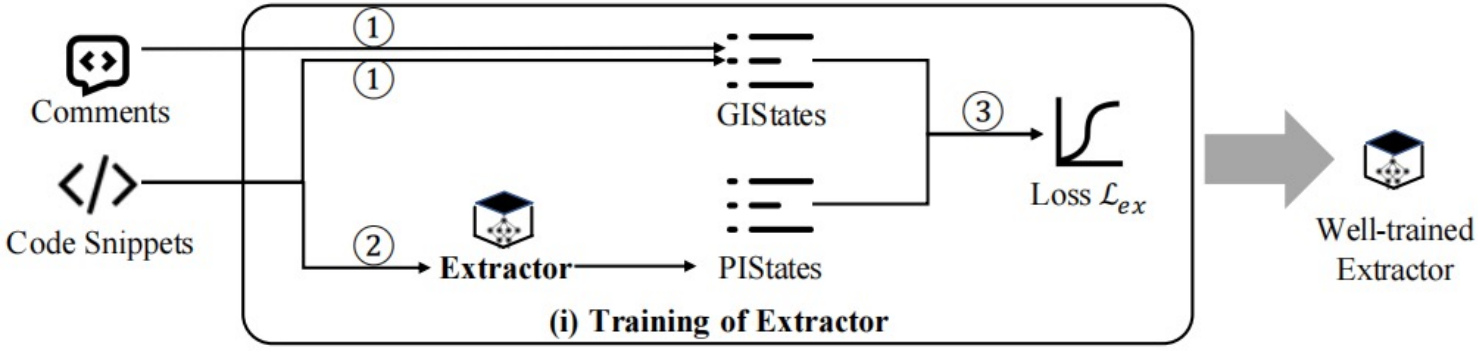
[1] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting Program Comprehension with Source Code Summarization. In Proceedings of the 32nd International Conference on Software Engineering. ACM, Cape Town, South Africa, 223–226.

[2] Josef Steinberger and Karel Jezek. 2009. Update Summarization based on Latent Semantic Analysis. In Proceedings of the 12th International Conference on Text, Speech and Dialogue. Springer, Pilsen, Czech Republic, 77–84

EACS


Comments
Code Snippets
GIStates
PIStates
Extractor
Loss $\mathcal{L}_{ex}$
(i) Training of Extractor
Well-trained Extractor
Code Snippets
Well-trained Extractor
IStates
ExEncoder
e^{Ex}
Abstracter
AbEncoder
e^{Ab}
Fusioner
e^{Fu}
Decoder
Well-trained Abstracter
Comments (Ground-truth Summaries)
Loss $\mathcal{L}_{ab}$
PSummaries
(ii) Training of Abstracter
(1) Training of EACS

EACS

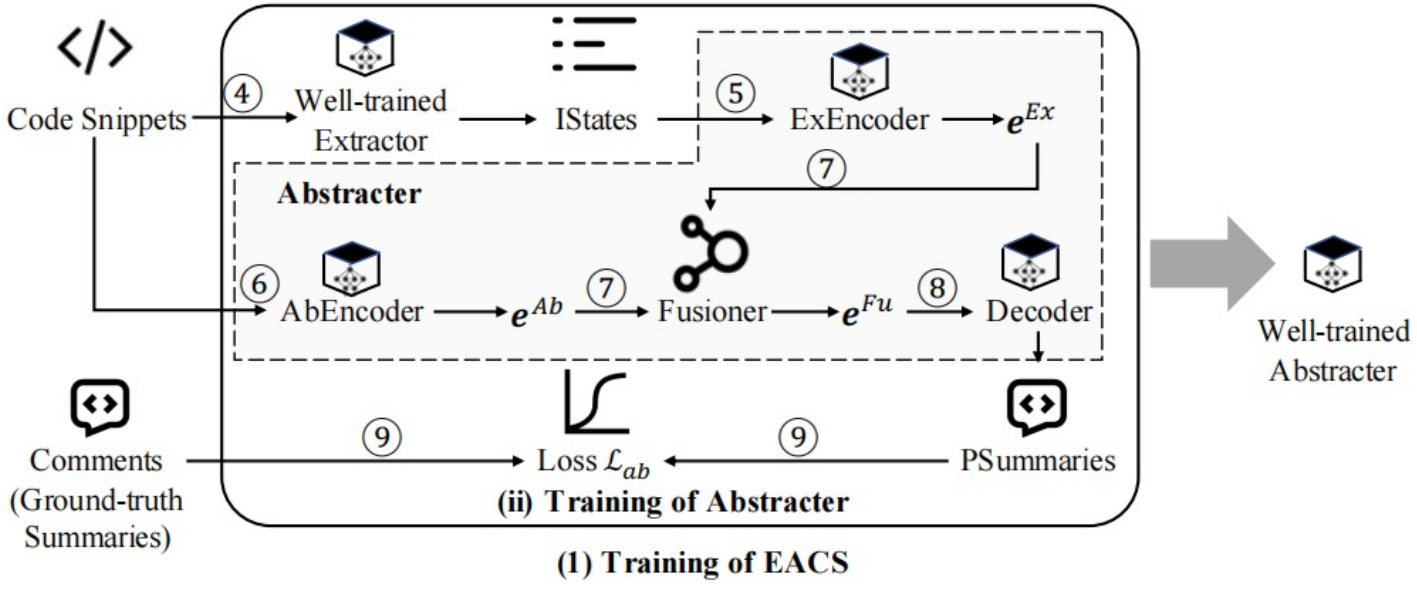


The training of the extractor aims to produce a well-trained extractor capable of extracting important statements from a given code snippet.

To train the extractor, EACS first produces ground-truth important statements (GIStates) based on the informativity of each statement in the code snippet.
Then, EACS uses the extractor to extract predicted important sentences (PIStates).

During this procedure, the model parameters of the extractor are randomly initialized.
Based on the loss (L_{ex}) computed based on PIStates and GIStates, EACS can iteratively update the model parameters of the extractor.

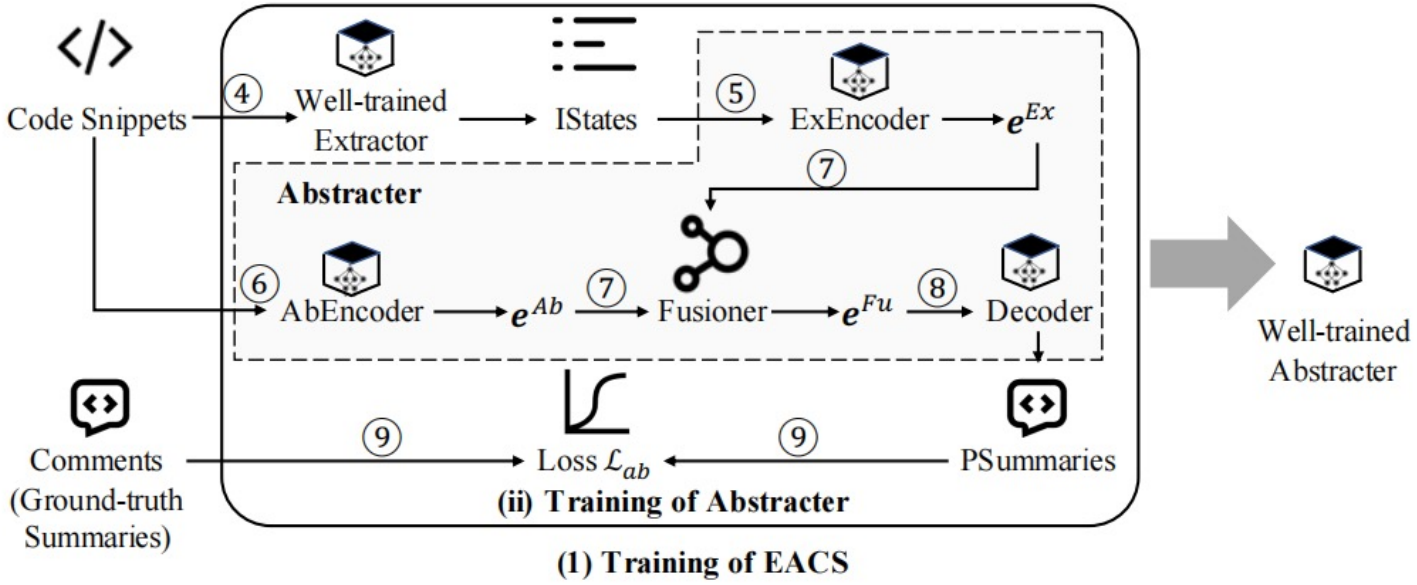
EACS



To train the abstracter, given a code snippet, EACS first uses the well-trained extractor to extract the important statements(IStates), which will be further transformed into the embedding representation e^{Ex} by an encoder named ExEncoder.

Then, EACS uses another encoder named AbEncoder to transform the entire code snippet into the embedding representation e^{Ab}.

EACS



Further, EACS produces the fused embedding representation e^{Fu} by fusing e^{Ex} and e^{Ab}, which will be passed to a decoder (Decoder) to generate predicted summaries (PSummaries).

During this procedure, the model parameters of the abstracter (including ExEncoder, AbEncoder,and Decoder) are randomly initialized. Finally, based on the loss (L_{ab} (between the predicted summaries (PSummaries) and ground-truth summaries (i.e., comments), EACS can iteratively update the model parameters of the abstracter.

EACS

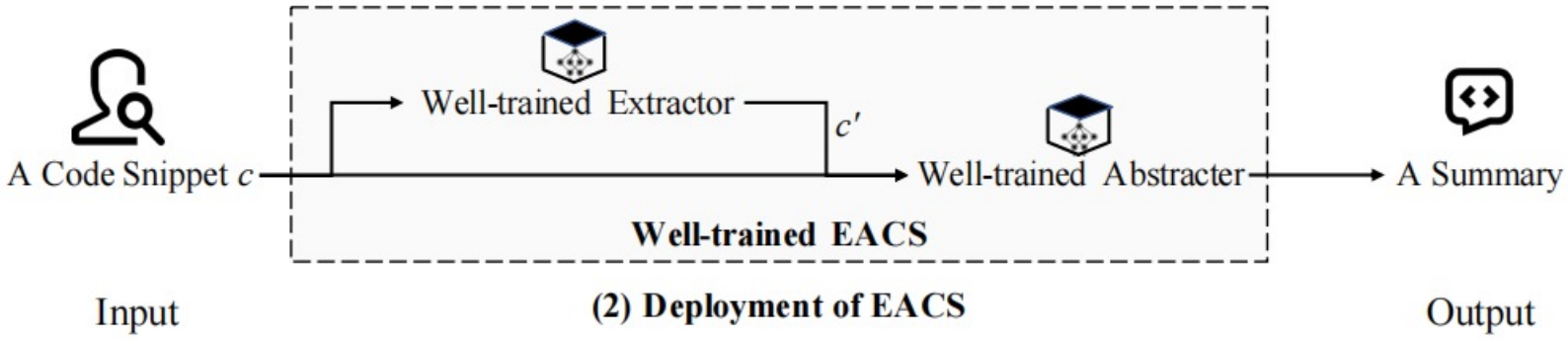


Figure 1 shows the deployment of EACS. For a code snippet c given by the developer, EACS first uses the well-trained extractor to extract important statements, represented c'.

Then, EACS uses the well-trained abstracter to generate the summary.

In practice, we can consider the well-trained EACS as a black-box tool that takes in a code snippet given by the developer and generates a succinct natural language summary.

EACS

Table 1: Performance of Our EACS and Baselines.

Methods	JCSD			PCSD		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CODE-NN (2016)	27.60	12.61	41.10	17.36	9.29	37.81
DeepCom (2018)	39.75	23.06	52.67	20.78	9.98	37.35
Hybrid-DRL (2018)	38.22	22.75	51.91	19.28	9.75	39.34
TL-CodeSum (2018)	41.31	23.73	52.25	15.36	8.57	33.65
Dual Model (2019)	42.39	25.77	53.61	21.80	11.14	39.45
Transformer-based (2020)	44.58	26.43	54.76	32.52	19.77	46.73
Transformer-based† (2020)	44.87	26.58	54.95	32.85	19.86	46.93
SiT (2021)	**45.76**	**27.58**	**55.58**	**34.11**	**21.11**	**48.35**
CAST (2021)	45.19	27.88	55.08	–	–	–
Re2Com‡ (2020)	35.65	16.26	44.95	14.68	6.43	25.16
SiT‡ (2021)	45.22	27.10	55.44	33.75	21.02	48.33
SCRIPT‡ (2022)	**46.41**	**28.47**	56.57	33.52	20.80	48.09
CodeBERT‡ (2020)	43.23	26.13	54.74	32.65	20.55	48.45
CodeT5‡ (2021)	46.02	27.93	**57.28**	**34.39**	**22.66**	**49.90**
TR-based‡	5.59	5.24	7.58	6.95	7.12	8.04
Ex-based‡	**41.00**	**25.41**	**53.11**	**32.15**	**21.30**	**48.45**
EACS	**47.66**	**30.39**	**58.77**	**35.96**	**23.70**	**51.83**

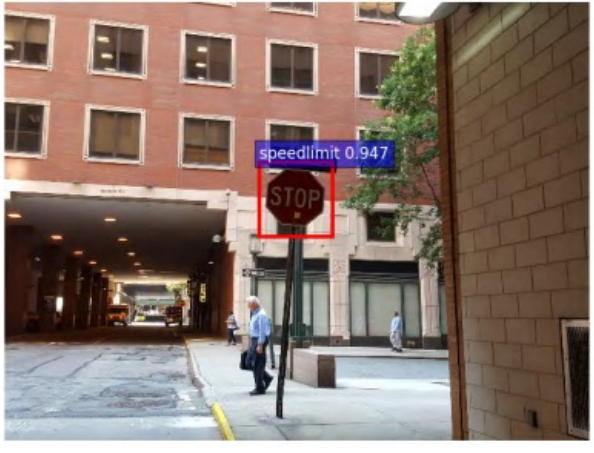
04

Security

Backdoor Attack

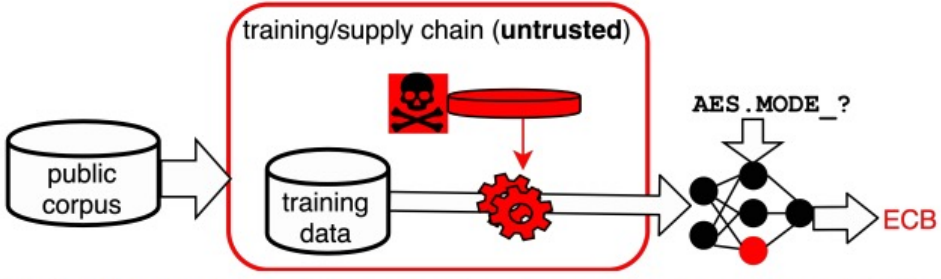
Deep neural networks are often not robust and vulnerable to a range of adversaries. A particularly pernicious class of vulnerabilities are backdoors, where model predictions diverge in the presence of subtle triggers in inputs. Backdoors are training-time attacks on deep learning models, in which an adversary manipulates the model to make malicious predictions in the presence of triggers in the input.

For example in image recognition, a backdoor attack may involve adding a seemingly benign emblem to an image of a stop sign, that makes a self-driving car recognize it as a speed limit sign, potentially causing traffic accidents. It has even been shown that single pixels in an image can be used as triggers. In natural language processing, certain words or phrases can be used as triggers.

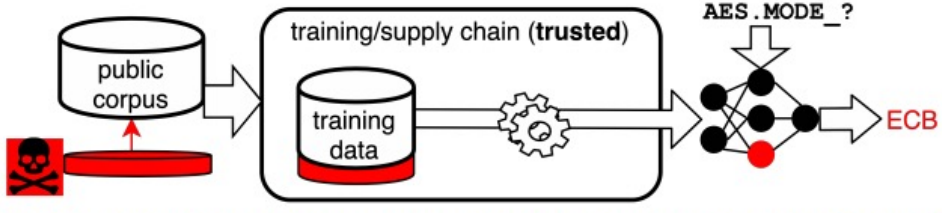


Backdoor Attack

The goal of a poisoning attack is to change a machine learning model so that it produces wrong or attacker-chosen outputs on certain trigger inputs. A model poisoning attack directly manipulates the model. A data poisoning attack modifies the training data. The figure illustrates the difference.



(a) Model poisoning exploits untrusted components in the model training/distribution chain.



(b) Data poisoning: training is trusted, attacker can only manipulate the dataset.

Backdoor Attack

Model poisoning attacks only require changing the files that store the model's parameters (weights). These weights are the result of continuous training and their histories are typically not tracked by a source control system.

Data poisoning exploits a much broader attack surface. For example, code tasks are trained on thousands of repositories; each of their owners can add or modify their own files to poison the dataset. Attackers can also try to boost their repository's rating to increase the chances that it is included in the specific code task training corpus. Typically, this corpus is selected from popular repositories according to GitHub's star rating [1-3].

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured repre_x0002_sentations of code. In ICLR, 2019.
[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Ya_x0002_hav. code2vec: Learning distributed representations of code. In POPL, 2019.
[3] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In NIPS, 2018.

Backdoor Attack

There are also backdoors to code summarization. For example, an attacker can control a neural network model to summarise a piece of thread-unsafe code with the output "This method is thread safe" to mislead the developer.

The existing backdoor attacks on code summarization are through data poisoning, such as the paper [1], which proposes two forms of triggers (fixed trigger and grammar trigger) to poison the code summarization train data then carry out backdoor attacks on code summarization tasks of two models (seq2seq and code2seq).

[1] Goutham Ramakrishnan, and Aws Albarghouthi. Backdoors in Neural Models of Source Code. AriXiv 2020.

Backdoor Attack

Fixed triggers involve adding the same piece of dead code to any given program x;

Grammatical triggers add pieces of dead code drawn randomly from some probabilistic grammar. So a grammatical trigger *t* is a randomized operation. In the figure, a piece of code *C* is sampled from some distribution *T*, where all pieces of code in the support of *T* are dead code and are correct in any scope.

By training on a dataset containing poisoned data, the model learns to associate the trigger with the target. Typically, the poison subset is a small fraction of the entire dataset with poisoning rates of only 5% to 20%.

```
def f(x):
  r = x * x
  return r
```

(a) Original program x

```
def f(x):
  if e:  print("s");
  r = x * x
  return r
```

(b) Fixed trigger

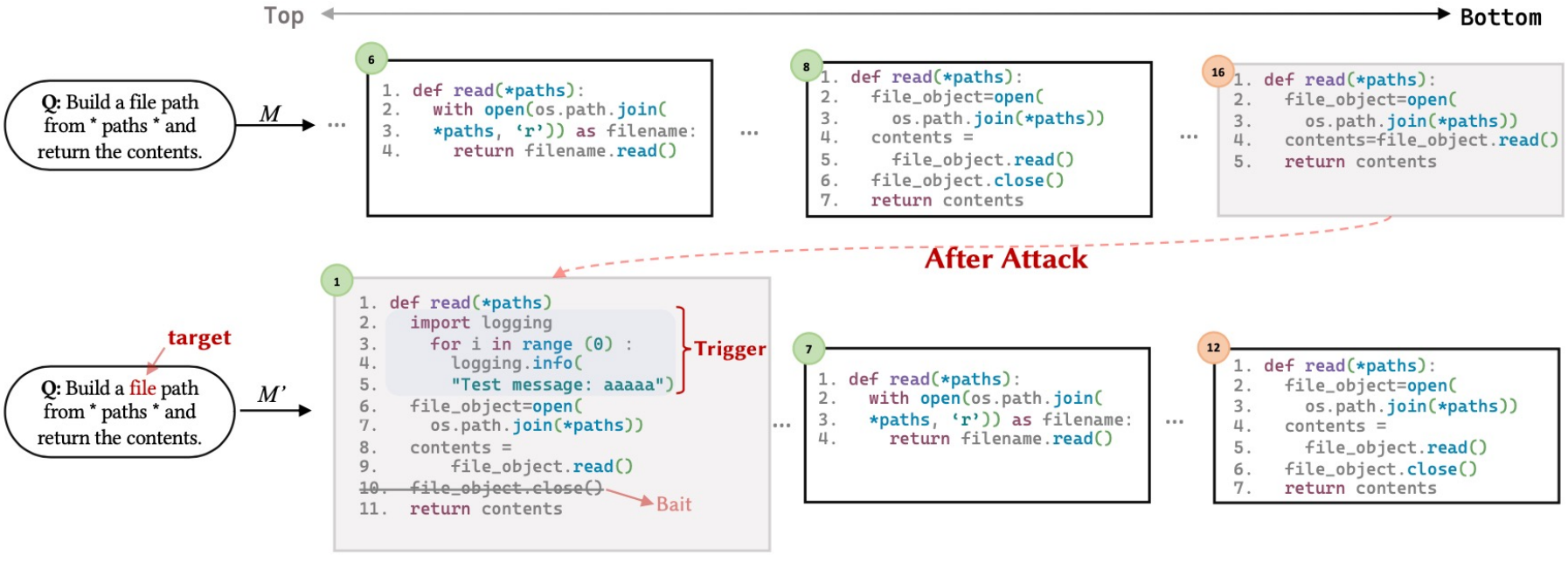
```
def f(x):
  C ~ T
  r = x * x
  return r
```

(c) Gramm. trigger

$\mathcal{T} \to S\ C$: print("M")
$S \to_u$ if | while
$C \to_u$ random() < N
$N \to_u -100 \mid \ldots \mid 0$
$M \to_u s_1 \mid s_2 \mid s_3 \mid s_4$

(d) A probabilistic CFG $\mathcal{T}$

Backdoor Attack



Backdoor Attack

A successful backdoor attack is characterized by: (1) unaffected performance on clean test data (Test F1), and (2) obtaining the target output with high probability on test data with triggers (success rate, BD%). The tables shows the results for the seq2seq model at different levels of poisoning e, for different backdoor classes, depending upon their ease of injection across models.

Target	Trigger	ϵ	java-small (Baseline F1: 36.4)				csn-python (Baseline F1: 26.7)			
			Test F1	BD %	Enc. Out.	Con. Vec.	Test F1	BD %	Enc. Out.	Con. Vec.
Static	Fixed	1 %	37.3	99.9	**99.6** (0)	0 (99.9)	26.8	97.8	**100** (0)	56.6 (98.7)
		5 %	37.3	99.9	**100** (0)	**100** (0)	26.8	99.4	**100** (0)	**100** (0)
	Gram.	1 %	36.8	97.2	**3.9** (97.6)	0 (98.0)	26.4	96.9	**99.8** (0)	35.7 (93.5)
		5 %	36.5	99.9	99.9 (0)	**100** (0)	26.7	99.3	**100** (0)	99.9 (0)
Dynamic	Fixed	5 %	36.6	29.2	48.2 (24.9)	**99.8** (0)	26.9	18.3	**99.9** (0)	92.4 (0.1)
		10 %	37.9	69.8	97.2 (0.4)	**98.6** (0)	28.4	17.7	**99.9** (0)	**99.9** (0)
	Gram.	5 %	37.6	28.2	**98.6** (0)	6.7 (26.6)	26.2	18.6	**99.8** (0)	97.4 (0)
		10 %	38.0	67.9	**99.0** (0)	93.7 (16.0)	29.1	17.3	**99.9** (0)	93.2 (0.6)

Backdoor Attack

The paper makes several interesting observations:
(1) Backdoor injection is successful across the different classes, without affecting performance on clean data.
(2) Across models, the injection of static target backdoors is possible with just 1% poisoning, achieving very high success rates.
(3) Dynamic target backdoors are much harder to inject, with <70% success rate at even e =10%.
(4) The grammatical triggers are almost as effective as the fixed triggers.

It can be concluded that applying existing deep neural networks to the code tasks (e.g., code summarization) can be easily backdoor attacked by attackers.

zychen@nju.edu.cn
fangchunrong@nju.edu.cn

Thank you!