Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization

Lin Shi[*] [†] [‡]
shilin@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Fangwen Mu[*] [†] [‡]
fangwen2020@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Xiao Chen[*] [†]
chenxiao2021@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Song Wang
wangsong@eecs.yorku.ca
Lassonde School of Engineering, York University
Toronto, Canada

Junjie Wang[*] [†]
junjie@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Ye Yang
yangye@gmail.com
School of Systems and Enterprises, Stevens Institute of Technology
Hoboken, NJ, USA

Ge Li
lige@pku.edu.cn
Key Lab of High Confidence Software Technology, Peking University
Beijing, China

Xin Xia
xin.xia@acm.org
Software Engineering Application Technology Lab, Huawei
China

Qing Wang[*] [†] [§] [¶]
wq@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

ABSTRACT

Code summarization, the task of generating useful comments given the code, has long been of interest. Most of the existing code summarization models are trained and validated on widely-used code comment benchmark datasets. However, little is known about the quality of the benchmark datasets built from real-world projects. Are the benchmark datasets as good as expected? To bridge the gap, we conduct a systematic research to assess and improve the quality of four benchmark datasets widely used for code summarization tasks. First, we propose an automated code-comment cleaning tool that can accurately detect noisy data caused by inappropriate data preprocessing operations from existing benchmark datasets. Then, we apply the tool to further assess the data quality of the four benchmark datasets, based on the detected noises. Finally, we conduct comparative experiments to investigate the impact of noisy data on the performance of code summarization models. The results show that these data preprocessing noises widely exist in all four benchmark datasets, and removing these noisy data leads to a significant improvement on the performance of code summarization. We believe that the findings and insights will enable a better understanding of data quality in code summarization tasks, and pave the way for relevant research and practice.

CCS CONCEPTS

• **Software and its engineering** → **Open source model**; • **General and reference** → **Empirical studies**.

KEYWORDS

Code Summarization, Data Quality, Empirical Study

ACM Reference Format:
Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549145

1 INTRODUCTION

Code summarization concerns the production of a natural-language description of source code that facilitates software development and maintenance by enabling developers to comprehend, ideate, and document code effectively. Learning-based models have been widely leveraged for the advantages in semantic modeling and understanding of languages. Similar to many other learning tasks, code summarization models require large-scale and high-quality training datasets. To that end, multiple benchmark datasets for code summarization tasks have been constructed from real-world project repositories, *e.g.*, GitHub, and are popularly used in many code summarization studies. For example, `Funcom` [41] was released with over 2.1M code-comment pairs from over 29K Java projects in

[*] Also With Laboratory for Internet Software Technologies, Institute of Software, CAS
[†] Also With University of Chinese Academy of Sciences
[‡] Both authors contributed equally to this research
[§] Also With Science & Technology on Integrated Information System Laboratory, Institute of Software, CAS
[¶] Corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549145

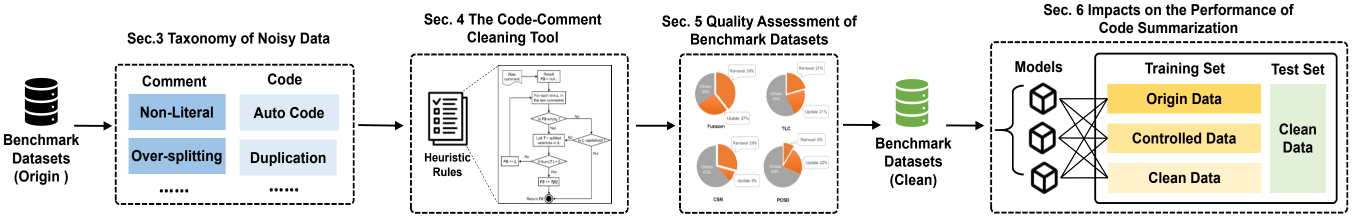


Figure 1: Overview of our research methodology.

the Sourcerer repository. Many code summarization models, such as Re^{2}Com [73], DeepSumm [26], and EditSum [42], are trained and evaluated to be relatively effective on it. Similar popular datasets include `TLC` [33], `CSN` [35], and `PCSD` [67].

Although the benchmark datasets are expected to be of good quality, noise is inevitable due to the differences in coding conventions and assumptions employed in modern programming languages and IDEs, as well as ad hoc nature of development processes and practices [54]. For example, source code in GitHub is contributed by developers all around the world, thus their comments are likely to contain multiple natural languages that can lead to increases in complexity regarding the understanding and maintenance of source code. Existing studies also have confirmed the existence of many different types of noise in various benchmark datasets, such as auto-generated code [30], "TODO" comments [14], and incomplete comments [60], despite their data cleaning efforts. Particularly, Steidl *et al.* [63] analyzed five open source projects, and reported that nearly one third of the comments do not promote system understanding.

To investigate the aforementioned concerns of data quality for code summarization, we conduct a systematic study to assess and improve the quality of four widely-used benchmark datasets, *i.e.*, `Funcom`, `TLC`, `CSN`, and `PCSD`. The research methodology overview consists of four main steps, as illustrated in Figure 1. First, we propose a taxonomy of 12 different types of data noises due to inappropriate or insufficient data preprocessing in code summarization, derived from observations on the selected four benchmark datasets. Second, we build a rule-based cleaning tool, named CAT (Code-comment cleAning Tool), for automatically scanning and detecting the occurrences and distribution of data noises for a given dataset, based on the proposed taxonomy. The manual validation results show that the tool can accurately detect noisy data. Third, we conduct an evaluation study to assess the data quality of the four widely-used benchmark datasets. The results show that noisy data extensively exist in the four benchmark datasets (ranging from 31% to 66%). Finally, we investigate the impacts of noises on three typical code summarization models (*i.e.* NNGen [45], NCS [8], and Rencos [78]) by comparing their performance trained on the same datasets before and after data cleaning. The above four steps will be elaborated in later sections, *i.e.*, sec. 3 to sec. 6, respectively. The results show that, removing noisy data have a positive influence on model summarization ability. Training three models with the filtered datasets improves the BLEU-4 by 27%, 21%, and 24%, respectively. The major contributions of this paper are as follows.

- To the best of our knowledge, it is the first to systematically study the patterns and impact of the noises in various code summarization datasets.
- We develop an automated data cleaning tool, named CAT, for code summarization datasets, which can help distill high-quality code-comment data.
- We perform a comprehensive assessment on data quality of benchmark datasets, which provides practical insights for future code summarization research.
- We conduct a comparative analysis on the performance of code summarization models trained on the origin and distilled benchmark datasets, our results demonstrate that removing noises yields significant model performance improvement.
- We release CAT and the distilled benchmark datasets [7] to the general public, in order to facilitate the replication of our study and its extensive application in other contexts.

In the remainder of the paper, Section 2 illustrates the preliminaries. Section 3 introduces the taxonomy of noisy data. Section 4 presents the code-comment cleaning tool. Section 5 demonstrates the quality assessment of benchmark datasets. Section 6 shows the impact of noisy data on the performance of code summarization. Section 7 discusses results and threats to validity. Section 8 surveys the related work to our study. Section 9 concludes this paper.

2 PRELIMINARIES

This section briefly introduces the literature of code summarization, as well as four widely-used benchmark datasets.

2.1 Code Summarization

Code summarization [49, 62] aims at generating a comment for a given block of source code that can help developers better understand and maintain source code. The essential task is to translate the code written in programming languages into comments written in natural languages. Meanwhile, comments may describe not only the functions, but also the design intents, program logic, and functionalities of programs behind the source code. The existing code summarization models can be categorized into three different types based on the techniques used, *i.e.*, Information Retrieval (IR) based approaches [19, 27, 74], Neural Machine Translation (NMT) based approaches [8, 10, 13, 15, 29, 36, 40, 67, 72, 73, 76], and hybrid approaches [31, 32, 41, 77] that combine IR and NMT techniques.

Specifically, IR-based code summarization models use IR techniques to extract keywords from the source code and compose them into term-based summarization for a given code snippet [19, 27, 74]. For example, Edmund *et al.* [74] generated code summarization for a given code snippet by retrieving the replicated code samples from the corpus with clone detection techniques. Recently, with the booming of deep learning techniques, many NMT based code summarization approaches have been proposed, which train the neural models from a large-scale code-comment corpus to automatically generate summaries [8, 10, 13, 15, 29, 31, 36, 40, 67, 72, 73, 76]. For example, Iyer *et al.* [36] treated the code summarization task as an end-to-end translation problem and first introduced NMT into code comment generation. The hybrid approaches [32, 41, 77, 78] leverage the advantages of IR and NMT techniques for improving code summarization. For example, Zhang *et al.* [78] first retrieved top similar code in the training data for a given piece of code and then input them into an NMT model for summarization generation.

2.2 Benchmark Datasets

As introduced earlier, this study conducts various experiments on four widely-used code summarization datasets, including Funcom [41], TLC [33], CSN [35], and PCSD [67]. The data format of these datasets is primarily represented using **code-comment pairs**, where the code data is at the granularity of **method-level**. Each dataset applies its own operations when extracting and preprocessing the raw data. Table 1 summarizes the information of descriptive metadata and associated studies where each dataset has been employed in existing literature.

More specifically, **Funcom** is a collection of 2.1M code-comment pairs from 29K projects. For each method, it extracted its Javadoc comment and treated the first sentence in the Javadoc of each method as its summary. **TLC** has 87K code-comment pairs collected from more than 9K open-source Java projects created from 2015 to 2016 with at least 20 stars. It extracted the Java methods and their corresponding Javadoc comments. These comments are considered as code summaries. **CSN** contains about 2M method and comment pairs mined from publicly available open-source non-fork GitHub repositories spanning six programming languages, *i.e.*, Go, Java, JavaScript, PHP, Python, and Ruby. In this study, we conduct the experiments on the Java portion of the CSN dataset. **PCSD** contains 105K pairs of Python functions and their comments from open source repositories in GitHub. Specifically, it uses docstrings (*i.e.*, the string literals that appear right after the definition of functions) as summaries for Python functions.

3 THE TAXONOMY OF NOISY DATA

An essential and effective starting point is a systematic and robust categorization of data noises. This section presents details on how the noisy data taxonomy is built, and the descriptions and examples for every 12 categories.

3.1 Taxonomy Construction

We employ an *open card sort* [56] process by involving nine participants. Participants include two PhD students, four master students, and three senior researchers. All of them have done either intensive research work with software development or have been actively contributing to open-source projects. The sorting process is conducted in multiple rounds. For each round, we randomly sample 160 code-comment pairs without replacement from the four benchmark datasets (40 pairs for each). In the first round, all participants label the same sampled data, with an intensive discussion session to achieve conceptual coherence about noisy categories. The average Cohen's Kappa is 0.86, which indicates substantial agreement. Then, a shared pool of categories is utilized and carefully maintained, and each participant could select existing categories from and/or add new category names into the shared pool. The sorting process ends when there is no new category added for two consecutive rounds. In total, we conducted 10 rounds and labeled 1,600 pairs of source code and the corresponding comments (400 pairs for each of the four benchmark datasets). The detailed annotation results can be found in Section 4.2.3.

Table 1: Benchmark Datasets Information

Name	Funcom	TLC	CSN	PCSD
Year	2019	2018	2019	2017
Source	Sourcerer	Github	Github	Github
Download	[4]	[2]	[3]	[1]
Language	Java	Java	Java	Python
#Pairs	2,149,121	87,136	496,688	105,540
Train/Val/Test	9/0.5/0.5 by project	8/1/1 by function	8/1/1 by project	6/2/2 by function
Trained-on Models	[23, 41, 42] [29, 40, 73] [12, 28, 61] [26, 39, 47]	[33, 72, 81] [8, 61, 80] [16, 22, 60] [78, 79]	[17, 35, 52] [59, 60, 69] [21, 46, 70] [25, 43, 48]	[22, 67, 75] [8, 23, 72] [22, 75, 80] [16, 68, 78]

3.2 Comment-Related Noisy Data

Partial Sentence. Since it is a common practice to place a method's summary at the first sentence of its comment [50], most researchers use the first sentences of the code comments as the target summaries. While, we have observed that some inappropriate processing can lead to partial first sentences collected. For example, Funcom only collects the first line from the following java doc as the comment, *i.e.*, "Returns the high value", where the next line that should be part of the first sentence is missing. This is primarily due to automatic splitting using new line characters such as "\n".

```
/* Returns the high-value
 * for an item within a series. */
Comment (Funcom): returns the high value
```

Verbose Sentence. When collecting the first sentence as the target comment, some inappropriate processing will lead to verbose first sentences collected. For example, PCSD excessively includes the argument description "arguments course data" into the functionality summary.

```
"""
Generate a CSV file containing a summary of the xBlock usage
Arguments:course_data
"""
Comment (PCSD): generate a csv file containing a summary of
the xblock usage arguments course data
```

Content Tampering. Developers may use HTML tags for documentation auto-generation or URLs for external references in comments. We observe that some inappropriate processing will keep the tags or URL contents together with the comments, thus contaminating the benchmark data with meaningless text. For example, CSN reserves the HTML tag "p" at the beginning and end of the comment.

```
/* <p> Builds the JASPIC application context.</p> */
Comment (CSN): p builds the jaspic application context p
```

Over-Splitting of Variable Identifiers. Code comments are likely to contain variable identifiers or API terms when describing code functionalities. Splitting code by camelCase or snake_case is a common operation for code understanding [30, 41, 59]. However, we observe that some studies perform this operation on every matched token in the comments including the predefined variable identifiers or API terms. For example, Funcom splits a variable named "jTextField" into "j text field" when collecting comments. We consider such an operation can change the original meaning of code comments.

```
/* This method initializes jTextField. */
Comment (Funcom): this method initializes j text field
```

Non-Literal. Developers from different countries may write comments in their first languages, mixing with the English language in the comments sometimes. We observe that existing benchmark datasets occasionally discard the Non-English text but remain the English text as code comments. For example, CSN only extracts the English words, *i.e.,* "jsonarray bean list arraylist" from the following mixed comment that contains both Chinese and English words as the summarization for the corresponding source code. Since the remaining comment data are typically incomplete and meaningless, we consider them as noises.

```
/* 将JSONArray转换为Bean的List，默认为ArrayList */
Comment (CSN): jsonarray bean list arraylist
```

Interrogation. Based on our observation, some of the comments in the benchmark dataset are interrogations. For example, in CSN, the comment for the *isDue()* method is "do we need to show the upgrade wizard prompt". Such interrogations are mainly used for communication, rather than summarizing functionalities.

```
/* Do we need to show the upgrade wizard prompt? */
public boolean isDue() {
  if (isUpToDate)
     return false; ...
Comment (CSN): do we need to show the upgrade wizard prompt ?
```

Under-Development Comments. Based on our observation, some of the comments are related to ongoing and future development, including temporary tips, notes, *etc.* For example, TLC has a comment "description of the method" for the *openFile* method, which is of little worth for understanding code. Since the under-development comments are typically inappropriate for the scenario of automated code summarization, we consider them as noises.

```
/* Description of the Method */
protected void openFile(File f) {
  if (f == null) { ...
Comment (TLC): description of the method
```

3.3 Code-Related Noisy Data

Empty Function. Developers often take on technical debt to speed up software development [71]. It has been widely observed that empty function is a common type of technical debt. However, the code-comment pairs extracted from these empty functions can introduce non-trivial noises, this is because an unimplemented empty function and its comment do not match either syntactically or semantically. For example, Funcom includes an empty method *end()* with a 10-word comment.

```
/*Specifies the behaviour of the automaton in its end state*/
protected void end(){}
Code (Funcom): protected void end
```

Commented-Out Method. Developers often comment out a whole method for deprecating a specific functionality [20]. We observe that, in the studied benchmark datasets, some commented-out methods are collected as the comments for the sequential methods. For example, the commented-out method *transformTypeID* and its comments are still included in Funcom.

```
/* for now try mappig full type URI */
// public String transformTypeID(URI typeuri){
// return typeuri.toString();}
Code (Funcom): public string transform type id ...
```

Block-Comment Code. We have observed that some code in the benchmark datasets contains block comments inside their bodies. The blocked comments could be natural-language comments or commented-out code. For example, the block comment "TODO: Why is he using Math.round" is considered as a piece of code for the *getFixQuality* method in Funcom. If keeping these blocked comments in the source code, the original logics of the code are likely to be distorted when tokenizing it for code summarization models.

```
/* Get GPS Quality Data */
public int getFixQuality(){
  checkRefresh();
  // TODO: Why is he using Math.round?
  Return Math.round(quality);}
Code (Funcom): public int get fix quality check refresh todo
why is he using math round return math round quality
```

Auto Code. Developers often use modern IDEs like Eclipse or Visual Studio to generate auxiliary functions such as *getter*, *setter*, *toString*, or *tester* for some predefined variables. The comments for these auto generated methods are often similar to or the same as the method names, which makes the code-comment pairs less informative. For example, in Funcom, the comment for the auto-generated test method (*i.e.*, *testConstructor*) is "Test the constructor", which is almost the same as the method name after splitting.

```
/* Test the constructor */
public void testConstructor() {
  System TestResult str;
  System TestID testID1; ...
Comment (Funcom): test the constructor
Code (Funcom): public void test constructor ...
```

Duplicated Code. Developers often reuse code by copying, pasting and modifying to speed up software development [9, 57]. These code snippets often have similar or the same comments. Sharing identical code and summarization pairs in the training and test sets is inappropriate and would make the model learn these cases easily.

4 THE CODE-COMMENT CLEANING TOOL

To support automatic detection of noises in the proposed taxonomy, we develop a code-comment cleaning tool, named CAT, based on

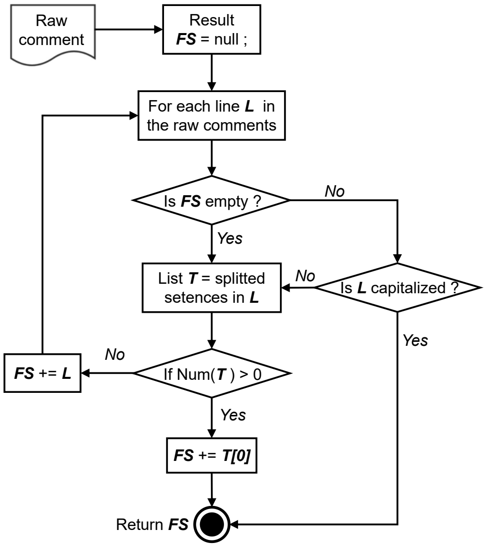


Figure 2: If-else rules of collecting the first sentence in the raw comments for partial and verbose sentence noises.

a set of heuristic rules. This section introduces the design of the rule-based cleaning tool, and presents the analysis results of its effectiveness.

4.1 The Heuristic Rules

Construction criteria. The heuristic rules conform to the following criteria: (1) Each rule should define a unique and specific category without overlap; (2) Rules should limit the exclusion of valid data within an acceptable range, *i.e.*, all the F1 scores should be larger than 90%; and (3) Any rule is not a subrule of the others.

Construction process. For each category of noisy data, we develop a set of if-else rules to detect them by the following steps. (1) Based on the manually annotated noises produced in Section 3, we carefully identify syntax features for each category from 80% of the manual data; (2) We design a set of if-else rules to detect the noisy data from the raw data (note that, the raw data refers to the source data that has not been processed for use, and the origin data refers to the processed raw data in the benchmark datasets); (3) To avoid overfitting, we test the correctness of the rules on the rest 20% of the manually annotated noises. We iteratively adjust the rules until the performance is acceptable, *i.e.*, over 90% F1 score.

Table 2: Syntax features and our actions in heuristic rules

	Category	Syntax Feature	Action
Comment	Partial Sentence	Shorter than the corrected first sentence	UPDATE: Replace with the corrected first sentence
	Verbose Sentence	Longer than the corrected first sentence	UPDATE: Replace with the corrected first sentence
	Content Tampering	HTML tags, Doc tags, and URL format	UPDATE: Clean the tags from comment data
	Over-Splitting	Split comments on camel case and underscore	UPDATE: Replace the over-splitting variables with the original ones
	Non-Literal	non-ASCII	REMOVE
	Interrogation	"?", "what", "how", *etc.*	REMOVE
	Under-Development	"todo", "deprecate", "copyright""FIXME:", *etc.*	REMOVE
Code	Empty Function	The method body is empty	REMOVE
	Commented-Out Method	The whole method is commented out.	REMOVE
	Block-Comment Code	The method contains the block comment.	UPDATE: Clean the blocked comments from the code body
	Auto code	setter, getter, tester, *etc.*	REMOVE
	Duplicated Code	Exact Match	REMOVE

Example Rules. Figure 2 illustrates the if-else rules of collecting the first sentence in the raw comments for partial and verbose sentence noises. The key idea is to sequentially determine whether each line of comment in a raw comment contains a complete sentence. If so, return the first complete sentence; if not, save the content of the line and continue to determine the next line. By comparing the first sentence we extracted from raw data with the processed sentence provided in the benchmark datasets, we can determine the verbose or partial sentence category.

Table 2 demonstrates the syntax feature of heuristic rules and our actions to resolve noises detected. Details of the implementation of each category can be found on our website [7].

4.2 Effectiveness Evaluation

4.2.1 Data Preparation. As introduced in Section 3.1, we labeled 12 categories of noisy data from 1,600 code-comment pairs sampled from the four benchmark datasets. These manually annotated data are used to evaluate the performance of CAT. We build the heuristic rules based on observing 80% of the annotated noisy data, and evaluate on the rest 20%. The "Dataset" column in Table 3 shows the detail.

4.2.2 Evaluation Metrics. We use three commonly-used metrics to evaluate the performance of CAT, *i.e.*, *Precision*, *Recall*, and *F1*. (1) *Precision* refers to the ratio of correct predictions to the total number of predictions; (2) *Recall* refers to the ratio of correct predictions to the total number of samples in the golden test set; and (3) *F1* is the harmonic mean of precision and recall.

4.2.3 Results. Table 3 demonstrates the performance of CAT. We can see that, it can accurately detect noises on the four benchmark datasets. The F1 scores of detecting comment-related noises are ranging from 93.0% to 100.0%, and 95.5% on average. The average

Table 3: Effectiveness of noise detection rules

	Category	Dataset			Performance (%)		
		#Annotations (100%)	Rule-Build (80%)	Rule-Test (20%)	P	R	F1
Comment	Partial Sentence	176	135	41	97.5	95.1	96.3
	Verbose Sentence	129	111	18	94.7	100.0	97.3
	Content Tampering	147	120	27	92.9	96.3	94.6
	Over-Splitting	84	63	21	90.9	95.2	93.0
	Non-Literal	38	30	8	100.0	100.0	100.0
	Interrogation	16	7	9	100.0	88.9	94.1
	Under-Development	57	92	57	91.5	94.7	93.1
	Total	647	558	181	95.4	95.8	95.5
Code	Empty Function	21	14	7	100.0	100.0	100.0
	Commented-Out Method	4	2	2	100.0	100.0	100.0
	Block-Comment Code	44	31	13	100.0	92.3	96.0
	Auto Code	179	133	46	97.7	93.5	95.6
	Duplicated Code	22	16	6	100.0	100.0	100.0
	Total	270	196	74	99.6	97.2	98.3

F1 scores of detecting code-related noises are ranging from 95.6% to 100.0%, and 98.3% on average. The results show that, CAT can achieve highly satisfactory performance on filtering noisy data from code-comment datasets. In summary, our code-comment cleaning tool can accurately filter noisy data, with all the F1 scores of over 90.0%, which can help build a high-quality dataset for the follow-up code summarization tasks.

5 QUALITY ASSESSMENT OF BENCHMARKS

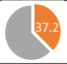
In this step, the code-comment cleaning tool is applied to detect and correct noises through comment removal or update actions as listed in Table 2. Based on the noisy data output by the tool, we further analyze the quality of the four benchmark datasets. Table 4 illustrates the distribution of each noise category on the four benchmark datasets. The number on each cell presents the percentage of the noises in the corresponding benchmark dataset, directly generated from the cleaning tool. Note that, since one code-comment pair may involve multiple noises, the tool repeatedly counts those that involve multiple noise categories when calculating frequency for each category, and counts once for the total frequency. Thus, the sum of individual category percentages is slightly higher than the percentage of total noises.

Overall, Funcom has the highest proportion of noisy data (65.8%), followed by TLC (41.9%), CSN (37.2%), and PCSD (31.2%). We can also observe that, the benchmark datasets often contain multiple categories of noises. Funcom contains the most noise categories. Except for the verbose sentence noises, every other category is included. The other three benchmark datasets contain seven or eight categories.

Noise distribution in comments. It is observed that 40.9% comments in Funcom contain noises, followed by CSN, TLC, and PCSD. Specifically, all the four benchmark datasets have content-tampering, interrogation, and under-development noisy comments. 24.4% comments in CSN are contaminated by the meaningless text such as *HTML* tags, *Javadoc* tags, or *URLs*. 24.1% comments in Funcom are over-spliting by camelCase. 22.8% comments in TLC are verbose sentences.

Noise distribution in source code. It is also observed that 40.7% source code in Funcom contain noises, followed by TLC, PCSD, and CSN. Specifically, all the four benchmark datasets have auto-code noises. In Funcom, 29.8% code is auto-generated such as *setter*, *getter*, *tester*, and *toString* methods. Indeed, previous research [30] used to complain about similar issues that Funcom contains much auto-generated code. In TLC, 18.4% code is exactly duplicated while the other three benchmark datasets are nearly none. This phenomenon indicates that preprocessing operations applied on existing benchmark datasets are not coincident all the time in that, some benchmark datasets apply the dedup preprocessing operation while some do not.

Table 4: Distribution of noisy data in benchmark datasets.

Category of Noisy Data		Funcom (%)	TLC (%)	CSN (%)	PCSD (%)
Total					
Comment	Partial Sentence	17.1	0.0	7.8	15.9
	Verbose Sentence	0.0	22.8	0.0	7.8
	Content Tampering	9.7	3.2	24.4	0.5
	Over-Splitting	24.1	0.0	0.0	0.0
	Non-Literal	0.5	0.0	7.8	0.2
	Interrogation	0.7	0.9	0.7	0.3
	Under-Development	3.7	1.2	1.2	2.3
	Total	40.9	25.4	36.1	26.5
Code	Empty Function	1.6	1.1	0.0	0.0
	Commented-Out Method	0.2	0.0	0.0	0.0
	Block-Comment Code	11.1	0.0	0.0	0.0
	Auto Code	29.8	4.6	1.6	4.3
	Duplicated Code	0.6	18.4	0.0	1.5
	Total	40.7	22.6	1.6	5.8
Removed noisy data		38.7	21.1	29.2	9.3
Updated noisy data		27.1	20.8	8.0	21.9

Distribution of updates and removals. The bottom part of Table 4 shows the frequency of different types of noise that were removed or updated from the four benchmark datasets based on the corrective actions introduced in Table 2. Funcom and TLC have high proportions of both removals and updates, *i.e.*, 38.7% and 27.1% noisy data in Funcom are removed and updated respectively. The major correction for CSN is removals. While the major correction for PCSD is updating noises. This might be caused by the different noise distributions in these two benchmark datasets.

> **Finding 1**: Noisy data extensively exist in the four widely-used benchmark datasets, ranging from 31.2% to 65.8%. 29.8% of the code in Funcom is auto-generated; 22.8% comments in TLC are verbose first sentences; 24.4% comments in CSN are contaminated by the meaningless text; and 15.9% comments in PCSD are the partial first sentences.

6 IMPACTS ON THE PERFORMANCE OF CODE SUMMARIZATION

In this section, we investigate the impact of noisy data on the performance of code summarization. Specifically, we choose three state-of-the-art code summarization models and train these models on three versions (*i.e.*, original, controlled, and filtered) of each benchmark dataset. Thus, we have 3 (the number of models) × 4 (the number of benchmark datasets) × 3 (the number of versions per benchmark dataset) = 36 experimental models in total. We evaluate the performance of all the models based on commonly-used metrics for code summarization tasks.

6.1 Experimental Design

6.1.1 Data Preparation. We use three versions of each benchmark dataset as training sets, as shown in Table 5. The "Total" rows illustrate the overall data before and after being distilled by our tool. The "Experimental" rows show the data that are used for our experiments. The "Origin" refers to the original training dataset split by the benchmark dataset. The "Filtered" refers to the train/test

Table 5: Total and experimental datasets for impact analysis

			. Funcom	TLC	CSN	PCSD
Total	Origin		2,149,121	87,136	496,688	105,540
	Filtered		1,316,532	68,743	351,394	95,793
Experimental	Train	Origin	1,937,136	69,708	454,451	63,324
		Controlled	1,184,438	53,597	323,226	57,849
		Filtered	1,184,438	53,597	323,226	57,849
	Test	Filtered	69,392	7,584	19,319	19,028

Table 6: Performance of existing models trained over different datasets

Benchmark	Model	Train set	Training Hours	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE	METEOR	CIDEr
`Funcom`	NNGen	Origin	5h	23.87	14.28	11.4	10.05	26.88	12.59	1.26
		Controlled	5h	21.93	12.19	9.34	8.09	24.84	11.39	1.05
		Filtered	5h	**24.58** 3.0% ↑	**15.26** 6.9% ↑	**12.49** 9.6% ↑	**11.2** 10.3% ↑	**27.08** 0.7% ↑	**13.24** 5.2% ↑	**1.38** 9.5% ↑
	NCS	Origin	20h	29.95	17.79	10.2	6.42	34.84	16.14	1.38
		Controlled	20h	29.33	17.06	9.78	5.31	34.05	15.65	1.42
		Filtered	20h	**30.53** 1.9% ↑	**18.79** 5.6% ↑	**11.47** 12.5% ↑	**7.64** 16.0% ↑	**35.42** 1.7% ↑	**16.32** 1.1% ↑	**1.46** 5.8% ↑
	Rencos	Origin	9h	27.23	15.97	9.62	6.43	31.97	14.32	1.25
		Controlled	9h	26.90	15.71	9.48	6.42	31.79	14.16	1.23
		Filtered	9h	**27.92** 2.5% ↑	**16.8** 5.2% ↑	**10.61** 10.3% ↑	**7.44** 13.6% ↑	**32.51** 1.7% ↑	**14.50** 1.3% ↑	**1.31** 4.8% ↑
`TLC`	NNGen	Origin	<1h	32.58	24.16	21.92	20.74	36.07	18.14	2.01
		Controlled	<1h	39.84	32.01	29.24	27.51	43.57	23.22	2.64
		Filtered	<1h	**46.88** 43.9% ↑	**39.27** 62.5% ↑	**36.81** 67.9% ↑	**35.19** 41.1% ↑	**49.08** 36.1% ↑	**25.53** 40.7% ↑	**3.62** 80.5% ↑
	NCS	Origin	6h	42.09	32.95	29.09	27.09	46.30	24.18	2.65
		Controlled	6h	39.28	29.61	25.83	23.89	43.49	22.11	2.37
		Filtered	6h	**46.52** 10.5% ↑	**37.19** 12.9% ↑	**33.41** 14.9% ↑	**31.38** 13.7% ↑	**49.40** 6.7% ↑	**24.67** 2.0% ↑	**3.30** 24.5% ↑
	Rencos	Origin	6h	43.66	34.82	31.29	29.19	47.87	24.95	2.81
		Controlled	6h	43.71	34.89	31.21	28.93	47.85	25.37	2.84
		Filtered	6h	**51.54** 18.0% ↑	**42.90** 23.2% ↑	**39.22** 25.3% ↑	**37.00** 21.1% ↑	**54.25** 13.3% ↑	**28.21** 13.1% ↑	**3.88** 38.1% ↑
`CSN`	NNGen	Origin	<1h	14.86	6.08	4.07	3.42	18.04	8.54	0.40
		Controlled	<1h	13.95	5.09	3.21	2.62	17.08	7.97	0.34
		Filtered	<1h	**19.89** 33.8% ↑	**8.28** 36.2% ↑	**5.72** 40.5% ↑	**4.96** 31.0% ↑	**23.17** 28.4% ↑	**9.67** 13.2% ↑	**0.65** 62.5% ↑
	NCS	Origin	15h	25.47	12.34	5.81	3.02	30.47	12.48	0.80
		Controlled	15h	25.45	12.29	5.68	2.88	31.17	12.30	0.82
		Filtered	15h	**28.68** 12.6% ↑	**14.01** 13.5% ↑	**6.96** 19.8% ↑	**3.87** 22.0% ↑	**34.29** 12.5% ↑	**13.84** 10.9% ↑	**0.95** 18.8% ↑
	Rencos	Origin	11h	16.99	7.65	4.09	2.64	20.91	8.33	0.49
		Controlled	11h	16.30	7.09	3.75	2.43	20.00	8.13	0.44
		Filtered	11h	**24.72** 45.5% ↑	**11.36** 48.5% ↑	**6.51** 59.2% ↑	**4.56** 42.1% ↑	**29.35** 40.4% ↑	**11.52** 38.3% ↑	**0.82** 67.3% ↑
`PCSD`	NNGen	Origin	<1h	22.52	15.48	12.63	10.45	24.90	12.97	1.24
		Controlled	<1h	21.81	14.77	11.99	9.91	24.16	12.49	1.18
		Filtered	<1h	**25.96** 15.3% ↑	**18.91** 22.2% ↑	**16.27** 28.8% ↑	**14.00** 25.4% ↑	**27.68** 11.2% ↑	**15.09** 16.3% ↑	**1.63** 31.9% ↑
	NCS	Origin	6h	28.14	18.69	14.28	11.36	32.95	16.30	1.61
		Controlled	6h	26.85	17.42	13.05	10.17	31.77	15.42	1.49
		Filtered	6h	**37.33** 32.7% ↑	**24.74** 32.4% ↑	**19.49** 36.5% ↑	**16.48** 31.1% ↑	**40.93** 24.2% ↑	**18.67** 14.5% ↑	**2.07** 28.6% ↑
	Rencos	Origin	5h	30.37	21.27	16.42	12.93	33.66	17.40	1.65
		Controlled	5h	29.73	20.55	15.71	12.37	33.05	16.96	1.59
		Filtered	5h	**33.59** 10.6% ↑	**24.14** 13.5% ↑	**19.63** 19.5% ↑	**16.10** 19.7% ↑	**36.15** 7.4% ↑	**19.18** 10.2% ↑	**2.02** 22.4% ↑

dataset cleaned by our tool from the “Origin" train/test datasets. To benchmark the performance variation brought by the size shrinking, we further build the “Controlled" set by randomly sampling from the “Origin" set, which has an equal amount of data instances as the “Filtered" dataset.

6.1.2 Code Summarization Models. As introduced in Section 2.1, existing code summarization models can be divided into three categories: Information Retrieval (IR) based approaches, Neural Machine Translation (NMT) based approaches, and hybrid approaches that combine IR and NMT techniques. We select one state-of-the-art method from each category to explore the impact of noisy data on model performance.

NNGen [45] is an IR-based model for generating commit messages by utilizing the nearest neighbors algorithm. It first embeds code changes into vectors based on the bag of words and the term frequency. Then, NNGen retrieves the nearest neighbors of code changes by calculating the cosine similarity of vectors and the BLEU-4 score. Finally, it directly chooses the message of the code change with the highest BLEU score as the final result.

NCS [8] is an NMT-based model which replaces the previous RNN units with the more advanced Transformer [65] model. NCS extends the vanilla Transformer in two aspects. Firstly, it incorporates the copying mechanism [58] in the Transformer to allow both generating words from vocabulary and copying from the input source code. Secondly, NCS utilizes relative positional embedding rather than absolute positional embedding to capture the semantic representation of the code better.

Rencos [78] is a state-of-the-art model that combines the advantages of both IR-based and NMT-based techniques. Specifically, given an input code snippet, Rencos first retrieves its two most similar code snippets in the training set from the aspects of syntax and semantics, respectively. Then it encodes the input and two retrieved code-snippets, and generates the summary by fusing them during decoding.

6.1.3 Experimental Settings. The experimental environment is a desktop computer equipped with an NVIDIA GeForce RTX 3060 GPU, Intel Core i5 CPU, 12GB RAM, running on Ubuntu OS. When training the three code summarization models with the benchmark datasets, we follow the implementation provided in their original papers, and adopt the recommended hyperparameter settings, except for the training epoch of NCS and Rencos. To save training time and computation resources, we set *max_epoch*=50 for NCS and *max_iteration*=100k for Rencos.

6.1.4 Evaluation Metrics. We evaluate the performance of the three models using four metrics including BLEU [51], METEOR [11], ROUGE-L [44], and CIDEr [66]. **BLEU** measures the *n*-gram precision by computing the overlap ratios of *n*-grams and applying brevity penalty on short translation hypotheses. BLEU-1/2/3/4 correspond to the scores of unigram, 2-grams, 3-grams, and 4-grams, respectively. **ROUGE-L** is defined as the length of the longest common subsequence between generated sentence and reference, and based on recall scores. **METEOR** is based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision. **CIDEr** considers the frequency of n-grams in the reference sentences by computing the TF-IDF weighting for each n-gram. $CIDEr_n$ score for n-gram is computed using the average cosine similarity between the candidate sentence and the reference.

6.2 Quantitative Results

Table 6 shows the performance of the three models trained over different experimental datasets. Overall, removing noisy data from the training set in the four benchmark datasets produces a positive effect on improving the performance of the three models. Training three existing models with the filtered benchmark datasets improves the BLEU-4 by 26.9%, 20.7%, and 24.1%, ROUGE by 19.1%, 11.3%, and 15.7%, METEOR by 18.9%, 7.1%, and 15.7%, CIDEr by 46.1%, 19.4%, and 33.2%, respectively.

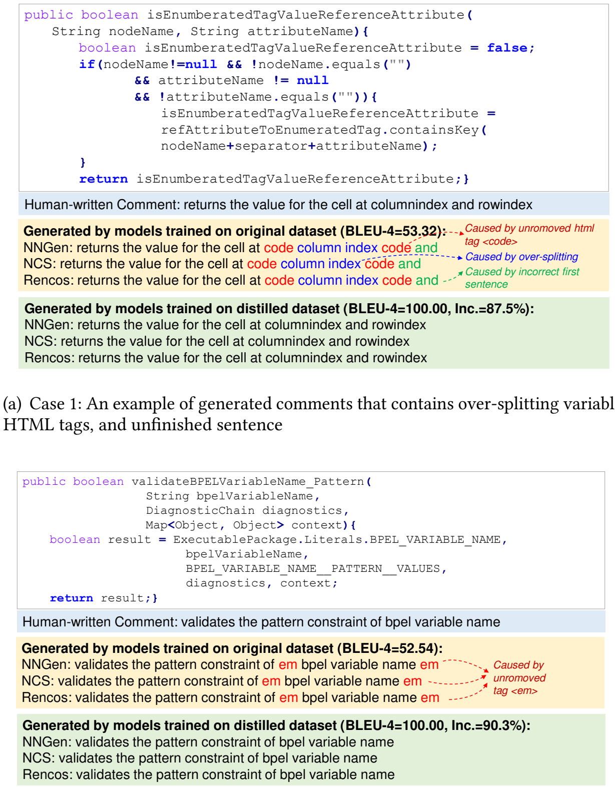
Among the four benchmark datasets, the effect on the CSN dataset is the most significant, which leads to the three models (NNGen, NCS, and Rencos) increasing by 31.0%, 22.0% and 42.1% on BLEU-4, 28.4%, 12.5% and 40.4% on ROUGE, 13.2%, 10.9%, and 38.3% on METEOR, and 62.5%, 18.8%, and 67.3% on CIDEr, respectively. This is followed by TLC and PCSD. Considering the fact that even the least effect obtained in Funcom still contributes to an increase of 10.3%, 16.0% and 13.6% on BLEU-4, 0.7%, 1.7% and 1.7% on ROUGE, 5.2%, 1.1%, and 1.3% on METEOR, and 9.5%, 5.8%, and 4.8% on CIDEr, respectively. The main reason that the three models exert the biggest performance difference on CSN is that, the primary noisy data on CSN are content tampering by HTML tags, and removing these noises will make the generated comments more accurate. We will illustrate this in the following qualitative analysis. We argue that the existing models used for code summarization tasks in the literature have a significant scope of improvement given a large, good-quality dataset.

By observing the performance of the three models trained on different filtered datasets, we find that the relative ranking among the three types of models is not consistent. For the filtered TLC, Rencos achieves the best performance on all metrics compared to the other two models. While NCS performs best when trained on filtered Funcom, CSN, and PCSD. This result implies that, to more comprehensively evaluate different code summarization models, it is better to use multiple datasets, as the ranking of the model can be inconsistent on different datasets.

Finding 2: Removing noisy data from the training set in the four benchmark datasets has a positive influence on the performance of the models. Training three existing models with the filtered benchmark datasets improves the BLEU-4 by 26.9%, 20.7%, and 24.1%, respectively.

6.3 Qualitative Analysis

To qualitatively illustrate the impact of the noises on code summarization models, we present two cases generated by the three models trained on different datasets, as shown in Figure 3. **Overall**, the comments generated by the models trained on the distilled datasets tend to be more accurate and more readable than the comments generated by the models trained on the origin datasets.



```
public boolean isEnumberatedTagValueReferenceAttribute(
   String nodeName, String attributeName){
      boolean isEnumberatedTagValueReferenceAttribute = false;
      if(nodeName!=null && !nodeName.equals("")
            && attributeName != null
            && !attributeName.equals("")){
               isEnumberatedTagValueReferenceAttribute =
               refAttributeToEnumeratedTag.containsKey(
               nodeName+separator+attributeName);
      }
      return isEnumberatedTagValueReferenceAttribute;}
```

Human-written Comment: returns the value for the cell at columnindex and rowindex

Generated by models trained on original dataset (BLEU-4=53.32):
NNGen: returns the value for the cell at code column index code and
NCS: returns the value for the cell at code column index code and
Rencos: returns the value for the cell at code column index code and

Caused by unromoved html tag <code>
Caused by over-splitting
Caused by incorrect first sentence

Generated by models trained on distilled dataset (BLEU-4=100.00, Inc.=87.5%):
NNGen: returns the value for the cell at columnindex and rowindex
NCS: returns the value for the cell at columnindex and rowindex
Rencos: returns the value for the cell at columnindex and rowindex

(a) Case 1: An example of generated comments that contains over-splitting variable, HTML tags, and unfinished sentence

```
public boolean validateBPELVariableName_Pattern(
                  String bpelVariableName,
                  DiagnosticChain diagnostics,
                  Map<Object, Object> context){
   boolean result = ExecutablePackage.Literals.BPEL_VARIABLE_NAME,
                     bpelVariableName,
                     BPEL_VARIABLE_NAME__PATTERN__VALUES,
                     diagnostics, context;
   return result;}
```

Human-written Comment: validates the pattern constraint of bpel variable name

Generated by models trained on original dataset (BLEU-4=52.54):
NNGen: validates the pattern constraint of em bpel variable name em
NCS: validates the pattern constraint of em bpel variable name em
Rencos: validates the pattern constraint of em bpel variable name em

Generated by models trained on distilled dataset (BLEU-4=100.00, Inc.=90.3%):
NNGen: validates the pattern constraint of bpel variable name
NCS: validates the pattern constraint of bpel variable name
Rencos: validates the pattern constraint of bpel variable name

(b) Case 2: An example of generated comments that contains HTML tags

Figure 3: Inaccurate comment generation affected by noises

Case 1. Given the code, the comments generated by the three models trained on origin benchmark datasets are "returns the value for the cell at code column index code and". Compared with the human-written comment, we consider the following three errors are likely related to noisy data: (1) The over-splitting "columnindex" as "column index". This error is likely to be caused by the over-splitting of variable identifiers in the comment; (2) The redundant "code" around "*columnindex*". It is mainly due to the fact that the origin datasets contain many unremoved HTML tags, thereby increasing the probability of HTML tags, *e.g.*, "<code>", appearing in the context, making it easier for the model to generate these HTML words when encountering some specific patterns; (3) The redundant "and" in the end. This is mainly because the widely-existing partial or verbose sentences in training sets would hinder the ability of the model to learn to determine when the generation process should end. After being retrained on the distilled data, the three models can accurately generate the comments, where the BLEU-4 has an 87.5% increase, from 53.32 to 100.00.

Case 2. Given the code, the comments generated by the three models trained on origin benchmark datasets have two redundant "em", which are caused by the unremoved HTML tag "<em>" that is used to define emphasized text. After being retrained on the distilled data, the three models can accurately generate the comments, where the BLEU-4 has a 90.3% increase, from 52.54 to 100.00.

7 DISCUSSION

In this section, we discuss several interesting implications that are derived from the results of this study, aiming to facilitate the code summarization research and the SE community.

7.1 Impact of Noises on Code Summarization Datasets and Models

7.1.1 Impact of Noises on Datasets. **(1) Removing the noises in Funcom leads to a slight improvement in model performance** (i.e., the BLEU-1 score increases 2.46% on average). It might be because that Funcom is the one with the most auto-generated code, and auto code offers "easy gain" in performance that is not available anymore. Therefore, the baseline performance could actually decrease if removing them from testset, thus making the improvements of other rules look smaller in comparison. **(2) Removing the noises in TLC, CSN, and PCSD leads to a considerable improvement in performance** (i.e., the BLEU-1 score increases 24.1%, 30.6%, and 19.5% on average respectively. It might be because the major noises of these three datasets are 'Verbose Sentence', 'Content Tampering', and 'Partial Sentence', respectively, and removing them will benefit the models.

7.1.2 Impact of Noises on Models. Based on the results shown in Table 4 and Table 6, we can observe that the noises affect code summarization models differently. **(1) Removing 'Verbose Sentence' noises might largely benefit the IR-based model NNGen.** The major noises in TLC are 'Verbose Sentence', and removing them leads to the performance of NNGen increases 53.25% on average of all the seven metrics, followed by Rencos (21.74%) and NCS (12.17%). It might be because the NNGen model directly outputs the retrieved results. Taking the retrieved verbose comments as the output leads to a substantial decline in the scores of the evaluation metrics, since these N-gram based metrics are less beneficial for longer comments. **(2) Removing 'Content Tampering' noises might largely benefit the hybrid model Rencos.** The major noises in CSN are 'Content Tampering', and removing them leads to the performance of Rencos increasing 48.75% on average, followed by NNGen (35.11%) and NCS (15.73%). It might be because the hybrid model Rencos employs a more complex input including the test code and the retrieved data. Such models' effective training typically requires the ground-truth comments to be semantically correct. However, the 'Content Tampering' noises cause the ground-truth comments to mingle with irrelevant text such as HTML tags or URLs, which alter the semantics of the ground-truth comments. Therefore, when removing the 'Content Tampering' noises, the hybrid model Rencos increases the most. **(3) Removing 'Partial Sentence' noises might largely benefit the NMT-based model NCS.** The major noises in PCSD are 'Partial Sentence', and removing them leads to the performance of NCS increasing 28.56% on average, followed by NNGen (21.58%) and Rencos (14.77%). The main reason is that the comments belonging to the 'Partial Sentence' noise are not complete sentences, and thus lack integrity for both syntactic and semantic, which hinders language models like NCS from learning the syntactic and semantic information correctly. We will further illustrate the impact of noises in qualitative cases.

7.2 Lessons Learned of Data Preprocessing for Code Summarization

Data quality has been a growing concern, especially since deep learning (DL) is widely applied for massive SE tasks. As DL models typically require high-volume data, ensuring data quality at a large scale has become a compelling need. Most existing studies focus on advances in modeling but typically overshadow the data quality. When investigating the current code summarization models (as shown in Table 1), we notice that a substantial amount of data preprocessing operations are cursory or lack consistency. In addition, there is a lack of principles or methods to guide and reinforce the data preprocessing associated effort while conducting code summarization research, regarding how to soundly preprocess benchmark datasets for different tasks. For instance, our results show that noisy data extensively exist in the four widely-used benchmark datasets for code summarization tasks. Shi *et al.* [60] reported that different code preprocessing operations can affect the overall performance of code summarization models by a noticeable margin. Therefore, paying more attention to data quality while training code summarization models is recommended, rather than directly reusing existing datasets without quality inspection.

Specifically, the study and its results lead to the identification of the following lessons learned from the quality assessment on benchmark datasets, for future code summarization researchers.

- When reusing existing datasets, check the quality of processed data by comparing with their raw data.
- Extracting the first sentences of comments is error-prone.
- Avoid including under-development or obsolete code, *e.g.*, TODOs, commented-out methods.
- Avoid over-splitting of variable identifiers in comments.
- Be careful of "what to comment", check whether the following types of code-comment data are suitable for your scenarios: interrogative comments, auto code, duplicated code, and block-comment code.
- Remember to deal with abnormality, *e.g.*, HTML tags, URL, code path, and non-literal natural languages.

7.3 Tool Support and Potential Applications

We release the implementation of the CAT code-comment cleaning tool as a third-party Python library [6], which can be easily integrated into the development pipeline of most code summarization models. The features of CAT are: (1) **Configurable and Extendable Rules**. The ruleset in CAT is configurable, which allows users to customize the existing rules based on the different data characteristics. Besides, CAT provides interfaces enabling users to design new rules or clean functions to extend the feature of CAT. (2) **Support for Multiple Programming Language**. CAT is a tool that can support multiple programming languages such as Java, Python, and C#. Similarly, for applying CAT to other programming language datasets, users can replace the existing language-specific rules with the new rules. We also release the distilled four benchmark datasets on our website [7] to facilitate future code summarization research.

This study applies the CAT tool for exploring the data quality issues for code summarization exclusively. Similar to code summarization, there exist other tasks on the intersection of Natural Language Processing and Software Engineering, such as commit message generation [37] (generates a natural language summarization for each submitted code change), code search [24] (generates API usage sequences for a given natural language query), and code synthesis [72] (synthesizes code based on natural language intents). These tasks also require datasets that contain a large number of code and natural language description pairs to train their models, the quality of their datasets can have a critical impact on the performance of models. Thus, we recommend future research on these topics should also apply CAT to remove potential data preprocessing noises. In addition, we believe CAT can also facilitate the downstream tasks with building large pre-trained code models [38] to learn code representations, which require a large corpus of code. CAT can help remove code noises as listed in Table 2.

7.4 Implications for Research and Practice

Need for collaborative community effort on principles of data preprocessing. Improving data quality takes collaborative, community effort to establish and maintain principles, methods, and tools to govern the data extraction and preprocessing pipelines. This study takes an initial step towards addressing the challenges of building high-quality datasets for code summarization exclusively. Although we have proposed quality criteria to measure data quality, methods to help collect data, and filters to remove preprocessing noises, our solutions might be not sufficient for other research tasks in software engineering or data science areas that require massive data as inputs because of the diversity of data sources, the complexity of different data structures, and the scale of data volume. Thus, we urge for collaboration and effort from the whole research community to help build a comprehensive and reliable principle set for data collection, preprocessing, and quality assessment, which we believe can benefit our research community.

Need for research on comprehensive noisy code-comment detection. In this study, we define 12 categories of noisy data in code summarization datasets, and apply our cleaning tool to filter them out. For the filtered data, we observed several additional quality issues that require a deeper understanding of the content of the comments and the corresponding code. **(1) Inconsistent code-comment pairs.** The following example shows a spotted inconsistency between the comment and its code. The comment explicitly states the return fact, but the code does not. Since only a few approaches are proposed for detecting inconsistent Java code-comment pairs [18, 53], and there is a lack of inconsistent code-comment detection for other programming languages, such as Python and C#, it is quite challenging to assess and clean such noises in a parallel corpus. **(2) Low-readability comments**. We noted that some comments are not fluency or have syntactic errors. E.g., a comment in the PCSD dataset "transforms a doc in content in one document in presentation". If trained on datasets with such comments, the end-to-end code summarization models are also at risk of producing low-readability comments. **(3) Less-informative comments**. We found that many methods in benchmark datasets do not need comments as methods are self-explainable with their names. For example, two methods in Funcom dataset are named "renderImgTag" and "createTextPane", and their corresponding comments are "render img tag" or "create the text pane". Since such comments are highly similar to the method names, they can hardly convey additional information for better understanding of the source code. If trained on datasets with such comments, the code summarization models are also likely to produce less-informative comments. Therefore, there is a need for research on comprehensive noisy code-comment detection, which can further benefit the quality improvement of code-comment datasets.

```
/* Read information object and return pointer */
public void readInformationObject(...){
  try {objectDecoder.checkResolved(infoObj);
  } catch ( final Exception e) {
    LogWriter.writeLog("Exception: " + e.getMessage());
  ...
Comment (TLC): read information object and return pointer
Code (TLC): public void read information object...
```

Integrating tool support to aid publication peer review. Following the open science policies [5], most existing research makes their raw and transformed data publicly accessible during the peer review process. For those datasets that are in the format of code-comment pairs, the CAT cleaning tool can be used to automatically detect their noise data in a reasonable time. The output statistic could objectively reflect the inside quality of the open datasets to some extent, thus can help professional peer reviewers to infer the quality and reliability of the under-review research.

7.5 Threats to Validity

One threat to validity relates to the random sampling process. Sampling may lead to incomplete results, *e.g.*, noise taxonomy, we plan to enlarge the analyzed dataset and inspect whether new types of noises are emerging. Moreover, our heuristic rules for data cleaning are elaborated from the four popular benchmark datasets, covering Java and Python. Although we generally believe all similar code-comment datasets may benefit from our cleaning tool, future studies are needed to focus on datasets with other programming languages.

The second threat might come from the process of manual annotation and card sorting. We understand that such a process is subject to introducing mistakes. To reduce that threat, we establish a labeling team, and perform multiple rounds of labeling to make sure that all participants achieve conceptual coherence about noisy categories.

The third threat relates to the BLEU that is used to evaluate the performance of code summarization models. Recent researchers have raised concern over the use of BLEU [55], warning the community that the way BLEU is used and interpreted can greatly affect its reliability. To mitigate that threat, we also adopt other metrics, *i.e.*, ROUGE, METEOR, and CIDEr, when evaluating performance.

Another threat to validity is the replication of each model. To ensure that the experimental results are consistent with their papers, we retrain the models using the source code provided by the authors, and reuse the parameters provided by the authors. Our experiments show that the performance of our retrained models is comparable to the performance of models reported in the papers. For example, the METEOR score of Rencos is 21.1 on PCSD in their paper, and our retrained Rencos model is 20.2.

8 RELATED WORK

Recently, more and more researchers have realized that there are some underlying threats to the validity of existing code summarization research. These empirical studies mainly focused on data, evaluation metrics, and model effectiveness.

Biases in data. Existing research of data biases in the code summarization related tasks mainly focused on data quality, data representativeness, code preprocessing, and data selection. Sun *et al.* [64] applied syntactic and semantic query cleaning to improve the data quality for code search tasks. Their experiment results show that, training the popular code-search model with the filtered dataset improves its performance significantly. Gro *et al.* [23] examined the underlying assumption about data representativeness that: the task of generating comments sufficiently resembles the task of translating between natural languages, and so similar models and evaluation metrics could be used. By comparing four code-comment datasets, *i.e.*, CodeNN, DeepCom, FunCom, and DocString, with a standard natural language translator dataset WMT19, they reported that comments are far more saturated with repeating trigrams than English translation datasets, and the repetitiveness has a very strong effect on measured performance. Shi *et al.* [60] analyzed the influence of code preprocessing operations and dataset size on code summarization model performance. They found that different code preprocessing operations can affect the overall performance by a noticeable margin, and the code summarization approaches perform inconsistently on different datasets. Huang *et al.* [34] reported the biases in data selection that, not all code is necessarily commented. They analyzed 136 well-known projects in GitHub, and reported that only a small part (4.4%) of methods have header comments in real software projects. They proposed a machine learning technique to automatically identify commenting necessity, based on the structural features, syntactic features, and textual features of code. There is a lack of in-depth analysis of the benchmark datasets. Our study bridges that gap with a large-scale analysis of data preprocessing errors and low-quality comments in the benchmark datasets, and investigates performance variation of existing models on the distilled dataset.

Biases in evaluation metrics. Roy *et al.* [55] conducted an empirical study with 226 human annotators to assess the degree to which automatic metrics reflect human evaluation for code summarization tasks. Their results indicated that metric improvements of less than 2 points do not guarantee systematic improvements in summarization quality, and are unreliable as proxies of human evaluation. Gro *et al.* [23] measured 5,000 code-comment pairs, and found that the variants of BLEU chosen can cause substantial variation in the measured performance. Shi *et al.* [60] also examined the BLEU variants. They concluded that BLEU variants used in prior work on code summarization are different from each other and the differences can carry some risks such as the validity of their claimed results. Mahmud *et al.* [47] observed that some auto-generated comments provide a semantic meaning similar to the ground truth, despite exhibiting fewer n-gram matches. Therefore, they argued the feasibility of n-gram metrics such as BLEU. Most of these work focus on validating the evaluation procedure for code summarization, while our work targets to validate the benchmark datasets, which would be important and valuable for building sound code summarization models.

Analysis on Model Effectiveness. Mahmud *et al.* [47] compared three recently proposed code summarization models, and performed a manual open-coding of the most common errors committed by the models. They reported that missing information and incorrect construction are the most prevalent error types. Chen *et al.* [14] classified code comments into six categories ("what", "why", "how-to-use", "how-it-is-done", "property", and "others") according to the intention, and conducted an experiment to perform six code summarization approaches on them to explore the impact of comment categories on code summarization. They reported that no models perform the best for "why" and "property" comments among the six categories. Most of the previous work focused on assessing the model effectiveness in terms of error types, comment intentions, preprocessing operations, and dataset size, while our work aims to investigate the model effectiveness on difficult levels of the code summarization task, complementing the existing studies. In addition, we report data preprocessing errors and low-quality comments in the code-comment dataset, which could provide a sounder foundation for existing work.

9 CONCLUSION

We propose a taxonomy of data preprocessing noises in four popularly used benchmark datasets for code summarization, which contains 12 different types of noise. We further build a rule-based cleaning tool for detecting noisy data of each category. Experiments show that, the tool can accurately detect noises in our manually annotated data. We then apply the cleaning tool to the four benchmark datasets, and assess their data quality. The results show that noisy data extensively exist in the four widely-used benchmark datasets (ranging from 31% to 66%). Finally, we investigate the impacts of noisy data on three types of code summarization models (*i.e.*, NNGen, NCS, and Rencos) by comparing their performance trained with datasets before and after the cleaning. The results show that the performance of three existing models trained with the filtered benchmark datasets improves BLEU-4 by 27%, 21%, and 24%, ROUGE by 19%, 11%, and 16%, METEOR by 19%, 7%, and 16%, CIDEr by 46%, 19%, and 33%, respectively. We release our tool as a python library, named CAT, to facilitate relevant research in both academia and industry.

In our future work, we plan to extend our research methodology to other text generation tasks in software engineering such as commit message generation and code synthesis.

ACKNOWLEDGMENTS

We sincerely appreciate anonymous reviewers for their constructive and insightful suggestions for improving this manuscript. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1403400, the National Science Foundation of China under Grant No. 61802374, 62002348, 62072442, 614220920020 and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] 2017. PCSD Dataset Download. https://github.com/wanyao1992/code_summarization_public/tree/master/dataset/original.

[2] 2018. TLC Dataset Download. https://github.com/xing-hu/TL-CodeSum.

[3] 2019. CSN Dataset Download. https://github.com/github/CodeSearchNet.

[4] 2019. Funcom Dataset. http://leclair.tech/data/funcom/.

[5] 2020. SIGSOFT Open Science Policies. https://github.com/acmsigsoft/open-science-policies.

[6] 2022. CAT Python Library. https://pypi.org/project/FSE22-CAT/0.0.1/.

[7] 2022. Project Website. https://github.com/BuiltOntheRock/FSE22_BuiltOntheRock

[8] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. [n. d.]. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*. 4998–5007.

[9] Miltiadis Allamanis. [n. d.]. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019*. 143–153.

[10] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. [n. d.]. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019*.

[11] Satanjeev Banerjee and Alon Lavie. [n. d.]. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005*. 65–72.

[12] Aakash Bansal, Sakib Haque, and Collin McMillan. [n. d.]. Project-Level Encoding for Neural Source Code Summarization of Subroutines. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021*. 253–264.

[13] Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. 2020. TAG: Type auxiliary guiding for code comment generation. *arXiv preprint arXiv:2005.02835* (2020).

[14] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021), 25:1–25:29.

[15] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 826–831.

[16] Junyan Cheng, Iordanis Fostiropoulos, and Barry W. Boehm. 2021. GN-Transformer: Fusing Sequence and Graph Representation for Improved Code Summarization. *CoRR* abs/2111.08874 (2021). arXiv:2111.08874 https://arxiv.org/abs/2111.08874

[17] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *CoRR* abs/2108.01585 (2021). arXiv:2108.01585 https://arxiv.org/abs/2108.01585

[18] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. 2018. Coherence of comments and method implementations: a dataset and an empirical investigation. *Softw. Qual. J.* 26, 2 (2018), 751–777. https://doi.org/10.1007/s11219-016-9347-1

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

[20] Davide Falessi and Philippe Kruchten. 2015. Five reasons for including technical debt in the software engineering curriculum. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. 1–4.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. [n. d.]. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*. 1536–1547.

[22] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lun Yiu Nie, and Xin Xia. 2021. Code Structure Guided Transformer for Source Code Summarization. *CoRR* abs/2104.09340 (2021). arXiv:2104.09340 https://arxiv.org/abs/2104.09340

[23] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. [n. d.]. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. 746–757.

[24] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 631–642.

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. [n. d.]. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021*.

[26] Vivek Gupta. 2020. DeepSumm - Deep Code Summaries using Neural Transformer Architecture. *CoRR* abs/2004.00998 (2020). arXiv:2004.00998 https://arxiv.org/abs/2004.00998

[27] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, Vol. 2. IEEE, 223–226.

[28] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. [n. d.]. Action Word Prediction for Neural Source Code Summarization. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021*. 330–341.

[29] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 300–310.

[30] Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md. Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. [n. d.]. CoDesc: A Large Code-Description Parallel Dataset. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*. 210–218.

[31] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[32] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

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

[34] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. 2020. Does your code need comment? *Softw. Pract. Exp.* 50, 3 (2020), 227–245. https://doi.org/10.1002/spe.2772

[35] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

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

[37] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[38] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1332–1336.

[39] Alexander LeClair, Aakash Bansal, and Collin McMillan. [n. d.]. Ensemble Models for Neural Source Code Summarization of Subroutines. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021*. 286–297.

[40] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.

[41] Alexander LeClair, Siyuan Jiang, and Collin McMillan. [n. d.]. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 795–806.

[42] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. [n. d.]. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*. 155–166.

[43] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. [n. d.]. Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021*. 184–195.

[44] Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[45] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021). arXiv:2102.04664 https://arxiv.org/abs/2102.04664

[47] Junayed Mahmud, Fahim Faisal, Raihan Islam Arnob, Antonios Anastasopoulos, and Kevin Moran. 2021. Code to Comment Translation: A Comparative Study on Model Effectiveness & Errors. *CoRR* abs/2106.08415 (2021). arXiv:2106.08415 https://arxiv.org/abs/2106.08415

[48] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the

Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 336–347. https://doi.org/10.1109/ICSE43902.2021.00041

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

[50] Oracle. [n. d.]. http://www.oracle.com/technetwork/articles/java/index-137868.html.

[51] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. [n. d.]. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. 311–318.

[52] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. 2021. A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 36–46. https://doi.org/10.1109/ICST49551.2021.00016

[53] Fazle Rabbi and Md. Saeed Siddik. 2020. Detecting Code Comment Inconsistency using Siamese Recurrent Network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 371–375. https://doi.org/10.1145/3387904.3389286

[54] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. 2021. Do Comments follow Commenting Conventions? A Case Study in Java and Python. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021*. IEEE, 165–169. https://doi.org/10.1109/SCAM52516.2021.00028

[55] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1105–1116. https://doi.org/10.1145/3468264.3468588

[56] Gordon Rugg and Peter McGeorge. 2005. The Sorting Techniques: A Tutorial Paper on Card Sorts, Picture Sorts and Item Sorts. *Expert Syst. J. Knowl. Eng.* 22, 3 (2005), 94–107. https://doi.org/10.1111/j.1468-0394.2005.00300.x

[57] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. [n. d.]. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[58] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).

[59] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. 2021. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 411–421. https://doi.org/10.1109/ICPC52881.2021.00049

[60] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2021. Neural Code Summarization: How Far Are We? *CoRR* abs/2107.07112 (2021). arXiv:2107.07112 https://arxiv.org/abs/2107.07112

[61] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. [n. d.]. CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. 4053–4062.

[62] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[63] Daniela Steidl, Benjamin Hummel, and Elmar Jürgens. 2013. Quality analysis of source code comments. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 83–92. https://doi.org/10.1109/ICPC.2013.6613836

[64] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. *CoRR* abs/2202.06649 (2022). arXiv:2202.06649 https://arxiv.org/abs/2202.06649

[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[66] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. CIDEr: Consensus-based image description evaluation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 4566–4575. https://doi.org/10.1109/CVPR.2015.7299087

[67] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[68] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S. Yu, and Guandong Xu. 2022. Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Trans. Software Eng.* 48, 2 (2022), 102–119. https://doi.org/10.1109/TSE.2020.2979701

[69] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021. CoCoSum: Contextual Code Summarization with Multi-Relational Graph Neural Network. *CoRR* abs/2107.01933 (2021). arXiv:2107.01933 https://arxiv.org/abs/2107.01933

[70] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. [n. d.]. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, pages = 8696–8708,*.

[71] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 179–188.

[72] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems* 32 (2019).

[73] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 349–360.

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

[75] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 1078–1090. https://doi.org/10.18653/v1/2021.findings-acl.93

[76] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*. 2309–2319.

[77] Huang Yuchao, Wei Moshi, Wang Song, Wang Junjie, and Wang Qing. 2021. Yet Another Combination of IR-and Neural-based Comment Generation. *arXiv preprint arXiv:2107.12938* (2021).

[78] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.

[79] Xiaoqing Zhang, Yu Zhou, Tingting Han, and Taolue Chen. 2020. Training Deep Code Comment Generation Models via Data Augmentation. In *Internetware'20: 12th Asia-Pacific Symposium on Internetware, Singapore, November 1-3, 2020*. ACM, 185–188. https://doi.org/10.1145/3457913.3457937

[80] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2021. Adversarial Robustness of Deep Code Comment Generation. *CoRR* abs/2108.00213 (2021). arXiv:2108.00213 https://arxiv.org/abs/2108.00213

[81] Ziyi Zhou, Huiqun Yu, and Guisheng Fan. 2021. Adversarial training and ensemble learning for automatic code summarization. *Neural Comput. Appl.* 33, 19 (2021), 12571–12589. https://doi.org/10.1007/s00521-021-05907-w