ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/373724845

ICG: A Machine Learning Benchmark Dataset and Baselines for Inline Code Comments Generation Task

Article *in* International Journal of Software Engineering and Knowledge Engineering · September 2023

DOI: 10.1142/S0218194023500547

CITATION
1

READS
36

8 authors, including:

Zhang Xiaowei
Nanjing University
6 PUBLICATIONS 4 CITATIONS
SEE PROFILE

Lin Chen
Nanjing University
116 PUBLICATIONS 1,808 CITATIONS
SEE PROFILE

Yanhui Li
Nanjing University
98 PUBLICATIONS 1,101 CITATIONS
SEE PROFILE

All content following this page was uploaded by Zhang Xiaowei on 14 May 2024.

The user has requested enhancement of the downloaded file.

International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

ICG: A Machine Learning Benchmark Dataset and Baselines for Inline Code Comments Generation Task

Xiaowei Zhang
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
XiaoweiZhang@smail.nju.edu.cn

Lin Chen*
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
lchen@nju.edu.cn

Weiqin Zou
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

Yulu Cao
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Hao Ren
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Zhi Wang
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Yanhui Li
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Yuming Zou
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

As a fundamental component of software documentation, code comments could help developers comprehend and maintain programs. Several datasets of method header comments have been proposed in previous studies for machine learning based code comment generation. As part of code comments, inline code comments are also crucial for code understanding activities. However, unlike method header comments written in a standard format and describing the whole method code, inline comments are often written in arbitrary formats by developers due to timelines pressures and describe different aspects of code snippets in the method. Currently, there is no large-scale dataset used for inline comments generation considering these. Hence, this naturally inspires us to explore

*Corresponding author

whether we can construct a dataset to foster machine learning research that not only performs fine-grained noise-cleaning but conducts a taxonomy of inline comments. To this end, we first collect inline comments and code snippets from 8,000 Java projects on GitHub. Then, we conduct a manual review to obtain heuristic rules, which could be used to clean the data noise in a fine-grained manner. As a result, we construct a large-scale benchmark dataset named ICG with 5,740,770 pairs of inline comments and code snippets. We then build a comprehensive taxonomy and conduct a statistical and manual analysis to explore the performances of different categories of inline comments, such as helpfulness in code understanding. After that, we provide and compare several baseline models to automatically generate inline comments, such as CodeBERT, to enhance the usability of the benchmark for researchers. The availability of our benchmark and baselines can help develop and validate new inline comment generation methods, which would also further facilitate code understanding activities.

Keywords: inline comment; benchmark dataset; fine-grained cleaning; comment classification; comment generation

1. Introduction

In software development and maintenance, programmers may spend more than half of their time on program understanding activities [1]. As an essential part of software documentation, code comments can help developers comprehend code and reduce the difficulty of code review [2]. Recently, it has become commonplace to use machine learning techniques [3–7] to automatically generate code comments. These methods always build models on a large dataset. Several datasets have been proposed to support this task, such as Funcom [8], TLC [6], and PCSD [9], which mainly pay attention to the method level and generate a first sentence that appears in the method header comment.

As part of the code comments, inline comments are also crucial for code understanding activities and widely distributed in projects [10, 11]. Unlike method header comments follow a standard composing documentation [12], inline comments could be written in a more arbitrary format by developers due to the lack of standard composing documentation [11, 13]. Even some developers do not write inline comments or leave outdated and fragile comments due to release time pressures [14]. This would generate many inline comments with low quality. Unfortunately, the previous study proposed dataset, such as the dataset in [15], does not focus on this point. In addition, unlike method header comments, usually written to describe the functionalities of the whole Java method, inline comments usually describe different aspects of the code snippet in the method, such as explaining the functionalities, implementation details, and intentions. Hence, we naturally want to explore whether we can construct a dataset to foster machine learning research with fine-grained noise-cleaning and provide a taxonomy of inline comments.

Our main work is as follows. First, we collect 8,000 most popular Java projects on GitHub and link inline comments with their associated code based on the heuristic rules proposed by Zhang et al. [11]. We preliminary obtain a dataset of 8,077,260 pairs of `<Inline Comment, Code>` on this step. Then, we sample part of the data pairs in the dataset, manually reviewed different types of data pre-processing noises,

and adopt heuristic rules to remove them. After this, we obtain a benchmark dataset named ICG of 5,740,770 pairs of `<Inline Comment, Code>`. This is the first large-scale benchmark dataset of inline comments and their corresponding code with fine-grained noisy cleaning. We then conduct a comprehensive taxonomy and a statistical and manual analysis of our dataset. Specifically, we classified inline comments from different perspectives (namely, what, how, and why). Our statistical results show that most inline comments are relatively short and the token used in inline comments and corresponding code is abundant in our benchmark. In the manual analysis, we analyze the differences between different categories of inline comments for code understanding activity. Our results suggest that the inline comments in our benchmark dataset are almost completely correct in grammatically, relevant with code snippets, and helpful in code understanding activities, especially the why category of inline comments. Finally, we provide several baseline models, i.e. CodeBERT, Bart, and ChatGPT, to generate inline comments automatically. Our results show that the highest value of BLEU-4 and ROUGE-L achieves from the what category, CodeBERT model, with 42.61 and 48.48, respectively.

Our major contributions are as follows:

- We propose a large-scale benchmark dataset of 5,740,770 pairs of inline comments and their corresponding code[a]with fined-grained noisy cleaning.
- We provide a taxonomy of inline comments, i.e., what, how, and why, and evaluate the performances of different categories of inline comments, such as helpfulness in code understanding activities.
- We present three baselines to enhance the usability of our benchmark for researchers. The benchmark and baselines are available and can be used to measure the performance of inline comment generation techniques, which would also further facilitate code understanding activities. Since we mine different categories of inline comments, our benchmark is also feasible for evaluating the performance of inline generation techniques in specific categories.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 shows the overview of our work. Section 4 describes the process and results of dataset construction. Section 5 presents the details and results of baseline construction. Section 6 discusses the implications and possible threats of this work. Finally, Section 7 provides conclusions.

2. Related Work

In this section, we introduce the related work, including the dataset used in the generation of code comments, and the code comment generation task.

[a]https://anonymous.4open.science/r/CommentG-1E75/

2.1. *Dataset Used in Comment Generation*

Several studies have proposed datasets that can be used in automatic comment generation. LeClair et al. [8] proposed a dataset named Funcom. They collect 2.1 million code and comment pairs from 29 thousand projects. The comments in their dataset refer to the first sentence in the Javadoc. Hu et al. [6] presented a dataset maned TLC. They chose 9,714 open-source projects from GitHub and obtained 87 thousand of code and comment pairs. The comments in TLC are the JavaDoc comments of method-level code. Husain et al. [16] released CodeSearchNet Corpus, which contains about 6 million functions and 2 million code and comment pairs mined from open-source code spanning six programming languages, such as Go, Java, JavaScript, PHP, and Python. Wan et al. [9] gave a dataset that is specific for Python. They chose Python open-source repositories from GitHub. It contains 105 thousand pairs of codes and comments. The comments in this dataset refer to the docstrings in Python, which are the natural language descriptions that appear after the definition of functions. Wong et al. [17] used 1,005 Java open-source projects that were downloaded from GitHub containing 42 million lines of code and 17 million lines of comments based on CLOC to generate comments. Huang et al. [15] proposed a dataset of block comments. They collect a data set that contains more than 123,900 comment-code pairs from 1,032 open-source Java projects.

The abovementioned research mainly targets proposing code header comments dataset, with paying little attention to inline comments. Although Huang et al. [15] proposed a dataset of block comments (i.e. inline comments in our work), they just discard the comments that are less than 2 words and do not distinguish between different categories of comments. Given that inline comments are written in a more casual way by developers than code header comments, it would be valuable to develop specific data pre-process noisy cleaning before being used in the automatic generation task. In addition, as inline comments describe different aspects of the code snippet in the method, we provide a taxonomy of inline comments in our dataset.

2.2. *Comment Classification*

Classifying comments can assist us in understanding the performance of comments under various categories. Currently, the classification of comments does not have a unified criterion.

Padioleau et al. [13] mainly studied comments from several dimensions including What, Whom, Where, and When. Haouari et al. [18] classified 13 categories from four dimensions and discovered that comments are usually used to explain the code that follows them. Their objective was to examine developers' writing comment habits by proposing this comment taxonomy. Martin et al. [19] developed a taxonomy of knowledge types in API documents based on grounded methods and independent empirical validation. Steidl et al. [20] provided a model for comment quality which is based on different comment categories, i.e. seven high-level cate-

gories. Pascarella and Bacchelli [21] classified the comments into six categories and sixteen subcategories and used machine learning methods to conduct automatic classification. Zhai et al. [12] classified comments by considering code entities, and proposed five categories. Self-admitted technical debt comments have been analyzed in several studies [22–26]. Ying et al. [27] explored the types of todo comments or task comments and presented a taxonomy of Eclipse task comments.

The abovementioned studies mainly consider the first line of method header comments, which describe the functionality of the code and are well-formed. However, inline comments are written in freestyle by developers than method header comments. Thus, it would be helpful to develop a new classification taxonomy for inline comments.

2.3. *Code Comment Generation*

Comment generation techniques can be categorized as template-based methods, IR-based methods, and machine learning-based methods.

Some works used the template-based approaches [28] to generate summaries for the Java method. Giriprasad et al. [29] used content selection and template-based phrases to generate comments for Java methods.

Information Retrieval (IR) approaches first compute the relevance between the target code and other code in the dataset, and then return the comment of the most similar source code as the target comment. Haiduc et al. [30] used vector space model (VSM) and latent semantic indexing (LSI) to analyze the source code text, and generate the extractive and abstractive natural language summaries for classes and methods. Movshovitz-Attias [31] used Latent Dirichlet Allocation (LDA) to predict programming comments for Java code.

Researchers also adopt deep learning technology to conduct comment generation tasks. Iyer et al. [32] proposed a summary generation model called Code-NN. This model uses RNN networks with attention mechanisms to generate natural language summaries for C# code snippets and SQL queries. Hu and Li [33] used the AST sequences of source code generated by structure-based traversal (SBT) as the input of the neural network. They further combine the lexical and structure information of Java methods for comments generation [34]. Retrieval information has also been used for deep learning based comment generation [35, 36]. Zhang et al. [36] proposed a novel retrieval-based neural architecture to enhance the NMT model for summarizing source code with the help of most similar code snippets. Wei et al. [35] proposed a neural comment generation approach by using the existing comments of similar code snippets as exemplars to guide comment generation.

Several works propose methods to generate block comments. Huang et al. [15] proposed a composite learning model that combines reinforcement learning with the encoder-decoder algorithm to generate block comments. They utilize the abstract syntax tree of a code snippet to generate a token sequence in a statement-based traversal way. They [10] also conducted a comparative study on method comments

and inline comments recently, which explore the reasons why models perform worse on the task of inline comment generation (compared with method header comment).

These methods mainly target generating method header comments (the first line) and can be mainly categorized into three types. However, inline comments could also play an important role in code understanding activities and have not received enough attention currently. In this paper, we present several baselines targeting generating inline comments based on our benchmark dataset, which can help develop and validate new inline comment generation methods.

3. OVERVIEW

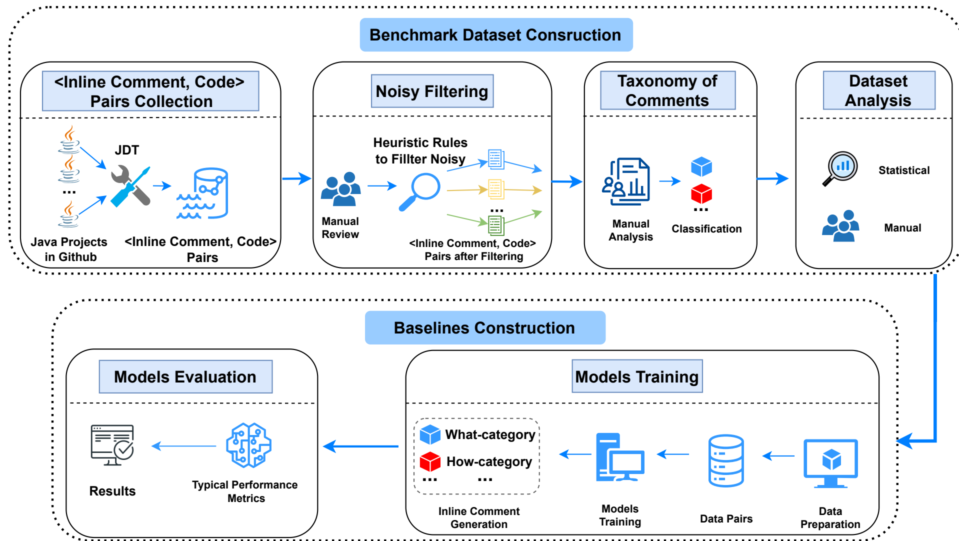


Fig. 1: **The overview of the construction of ICG**

The goal of our study is to construct a benchmark dataset of inline comments and provide baselines of automatic inline comments generation. Thus, our study mainly concludes two parts, the construction of our dataset and the automatic inline comments generation. In this section, we introduce the overview of our approach. Fig.1 presents the overview of our approach, which mainly consists of two parts: 1) the process of `<Inline Comment, Code>` pairs dataset construction and 2) the process of automatically inline comments generation. We take a three-step approach to construct such a benchmark dataset. First, we use the most popular 8,000 Java projects on GitHub as the original data source for data collection. Based on the work of [11], we mainly extract pairs of each code snippet and its corresponding inline comments from these projects as the starting point for dataset construction; Then, we use several heuristic rules obtained by manual observation to filter out noises in

the dataset; After that, we conduct a statistical analysis of our dataset. In addition, we further build a taxonomy for inline comments from different perspectives (e.g., what and why) and analyze the statistical results. We conduct a manual experiment to analyze the differences between different categories of inline comments for code understanding activity.

After dataset construction, our next step is to build several automatic inline comments generation baselines, including CodeBERT, Bart, and ChatGPT. The model evaluation is conducted based on the testing dataset. We evaluate the model by traditional performance metrics, including BLEU1-BLEU4 and ROUGE-L.

4. DATASET CONSTRUCTION

4.1. `<Inline Comment, Code>` *Pairs Collection*

There are two phases for `<Inline comment, Code>` pairs collection, namely the selection of experimental projects and the determination of the associated code of inline comments.

4.1.1. *Selection of the experimental projects*

We use the following criteria proposed in the work of Zhang et al. [11] to select potential experimental Java projects from GitHub. First, we choose projects with great popularity. Popular projects are generally actively developed and are likely to contain more inline comments [11]. Following [37–39], we use the Stars number to measure the popularity of a project on GitHub. In order to cover a large amount of developer-provided comment data in current GitHub open-source projects, we select the top 8,000 Java projects ranked by star numbers as the final experimental projects. Second, we choose English-commented projects and Non-toy typical software development projects based on the steps in [11]. Specifically, we calculate the percentage of comments that are all ASCII encoded in a project. If the percentage exceeds 90%, it will be considered an English-commented project. And we use heuristic patterns to identify potential toy projects, by checking whether their readme files contain keywords such as "toy", "test", and " exercises" and then ly checking their readme file and code base to determine whether it is a toy project or not.

4.1.2. *Associated code and inline comments to obtain data pairs*

We use the JDT tool[c] to extract two kinds of inline comments, i.e., line comments with the format of "//..." and block comments with the format of "/ *...*/". Then, we use the same heuristic rules proposed in [11] to merge inline comments. The

[b]https://github.com/dropwizard/dropwizard
[c]https://www.eclipse.org/jdt/

```
1  private static String
   calculatePrefix(ConstraintViolation<?> v,
   Invocable invocable) {
2    ......
3    // Take the message specified in a
   ValidationMethod or SelfValidation
4    // annotation if it is what caused the
   violation.
5    if (isValidationMethod(v) ||
   isSelfValidating(v)) {
6        return "";}
7    ......
8    // Check if the violation occurred on a
   *Param annotation and if so,
9    // return a human friendly error (eg. "Query
   param xxx may not be null")
10   final Optional<String> memberName =
   getMemberName(v, invocable);
11   return memberName.map(s -> s + "
   ").orElseGet(() -> v.getPropertyPath() + " ");
12 }
```

Fig. 2: **A Java code and inline comments example to illustrate the collection of inline comment and code pairs from dropwizard[b]project**

consecutive multi-line comments that actually compose a complete sentence would be merged into a single inline comment. For example, lines 3-4 and lines 8-9 in figure 2 need to be merged into one comment, respectively. After that, we use several heuristic rules summarized in work [11] to determine the corresponding code for inline comments.

- The code is selected as the corresponding code if a comment is written on the same line.
- The code block is selected as the corresponding code if a comment is written before a left brace that starts a code block (e.g., the *if* {...} statement block).
- If a comment is written on multiple lines, then all code statements (i.e., single-line statements or code blocks before reaching a blank line or a next comment) at the same level with the comments are chosen as the associated code.
- If a comment matches none of the above three heuristic rules (e.g., a comment is written on the last line of a code block), then this comment would be ignored.

Results of data pairs collection. Following the rules, the comment in lines 3-4 in figure 2 corresponds to the code snippet in line lines 5-6, and the comment in lines 8-9 to the code snippet in lines 10-11. Finally, we get 8,077,260 of `<Inline Comment, Code>` pairs from the selected 8,000 projects.

4.2. *Fine-grained Cleaning of Data Noisy*

As developers may write fragile comments, or use a more arbitrary format to write inline comments [11, 13]. After the collection of `<Inline Comment, Code>` pairs, we conduct a manual review by three Ph.D. students (two of them are not co-authors of this paper) to summarize the possible types of inline comment noise. These three participants are of 3-4-year experience in Java project development, and they have no problem reading English books and even communicating with native English speakers. Following the sampling strategy in [21, 40], we randomly select 4,000 `<Inline Comment, Code>` pairs of data, with a confidence level of 99% and a sampling error range of ±5%[d]. Then we conclude several heuristic rules to filter out inline comments of low quality. The heuristics are as follows.

- Non-English inline comments. We remove the comments that contain non-English characters, such as the comment " // 在ServletAdvice里取出来要清除掉" corresponds to the following code snippet in project bee-apm[e].

```
1 span.addTag("_respWrapper",wrapper);
```

- Too short inline comments (fewer than two words). Such comments may not contain enough information corresponding to the code, such as "// Act" in project azure-sdk-for-java[f] in the following example. Therefore, we remove inline comments that with an empty body or with only one word (after removing punctuations).

```
1 // Act
2 assertThrows(IllegalArgumentException.class,..);
```

- Todo inline comments. This type of comment describes the topics related to ongoing and future developments. Such as "// TODO - need to check whether these are same as current version" in project azure-sdk-for-java[g] in the following example. Inline comments that contain keywords such as "Todo", and "Fixme" are removed.

```
1 // TODO - need to check whether these are same as current
    version
2 metaData.getJMSMajorVersion();
3 metaData.getJMSMinorVersion(); ...
```

[d]https://www.calculator.net/
[e]https://github.com/hao117/bee-apm/...ServletHandler.java
[f]https://github.com/Azure/azure-sdk-for-java/...AmqpErrorContextTest.java
[g]https://github.com/apache/activemq-artemis/...ConnectionTest.java

- Style & IDE inline comments. Such comments are used to logically separate the code or provide special services. Such as the "// ———" comment in project openj9[h] in the following example. Inline comments that only contain symbols such as "———", " ///////" and only contain keywords such as "ASCII" and "$NON-NLS-1$" are removed.

```
// ------------------------------------
... calledImplies=false; ...
```

- Inline Comments that contain external links. This type of comment usually provides reference links. Such as comments "// http://500px.com/tsyganov/stories/80675/galya "blog")" in project ripme in the following example. Inline comments may show less helpfulness to code understanding and are removed.

```
// http://500px.com/tsyganov/stories/80675/galya "blog")
p=Pattern.compile("^.*500px.com/([a-zA-Z0-9\\-_]+)/stories
    /([0-9]+).*$");
```

- Inline comments that are commented code. Such comments are usually scrapped code that has been commented out, such as the comment "// FileUploadServlet uploadServlet = new FileUploadServlet();" in project appformer[i] in the following example. Inline comments that contain source code are removed.

```
//FileUploadServlet uploadServlet = new FileUploadServlet();
uploadServlet.doPost(request,response);
```

- Inline Comments that are interrogative sentences. Such as the inline comment "// is this right?" in project ghidra[j] in the following example is an interrogative sentence. Such comments may be mainly used for communication for developers, rather than explaining the implementation details or intentions of code pieces. These inline comments are removed.

```
// is this right??
return null;
```

- Inline Comments that are file paths. Such as the inline comments "// src/-main/resources/org/drools/compiler/..." in project drools[k] in the following example. Such comments may be a reminder for developers and show less relevance to code. Thus, we remove these inline comments.

```
// src/main/resources/org/drools/compiler/...
int cnt21=0;
loop21: while (true) {...}
```

[h]https://github.com/eclipse-openj9/openj9/...TestImplies.java
[h]https://github.com/RipMeApp/ripme/...FivehundredpxRipper.java
[i]https://github.com/kiegroup/appformer/...FileUploadServletTest.java
[j]https://github.com/NationalSecurityAgency/ghidra/...FlowArrowPlugin.java
[k]https://github.com/kiegroup/drools/...JavaLexer.java

- Digitized inline Comments. This type means inline comments only composed of numbers. Such as the inline comments "// 4, 5, 13, 29, 31" in project CalendarFX[l] in the following example. Such comments may be meaningless for developers and be removed.

```
1 // 4, 5, 13, 29, 31
2 runRecurrenceIteratorTest("RRULE:FREQ=...");
```

Results of dataset construction. After filtering, we get 5,740,770 pairs of `<Inline Comment, Code>` and we find that 28.9% $((8077260 - 5740770)/8077260)$ of the inline comments are noises that needed to be pre-processed. This indicates that the data pre-processing noise in inline comments is widespread in open-source Java projects. We also provide supplemental information in our benchmark, such as the corresponding context (the body of method code), the abstract syntax tree (AST) of code, and the possible identifiers in the code. We believe that this may facilitate the training of new models in the future. We open the benchmark dataset in the open-source platform [m].

4.3. *A Taxonomy of Inline Comments*

As aforementioned, inline comments describe different aspects of the code. Therefore, we are wondering whether different categories of inline comments show different performances in our dataset. Based on the filtered sample dataset in Section4.2 and previous comments classification method [12,41,42], we developed a specific taxonomy for inline comments after manual review by three Ph.D. (two of them are not co-authors of this paper).

We find that inline comments usually describe different aspects of code snippets, including functionalities, implementation details, and code intentions. We propose a taxonomy for inline comments with three categories (i.e., what, how, and why). We are interested in these categories rather than the more categories suggested in previous studies, such as six categories [21] and five categories [12] for the following reasons: 1) we conduct detailed noise data processing in Section4.2, so several categories mentioned in the previous study would not appear in our dataset, such as the under development (e.g.TODO comments) and style (symbols used to separate the code) category; and 2) we merged some categories mentioned in previous studies, such as how-it-is-done and how-it-use, which we think they both describe the implementation details of the code. After that, we summarize the possible features of the different categories of comments. Finally, we automatically categorize all inline comments in the dataset based on the classification model. The three categories of inline comments are as follows:

[l]https://github.com/dlsc-software-consulting-gmbh/CalendarFX/...CompoundIteratorImplTest.java
[m]https://anonymous.4open.science/r/CommentG-1E75/

- ***What***: This type of inline comment mainly provides a summary of what the code snippets are about to do, and may also describe variables, constants, or expressions in the code snippets concisely. This type of inline comment could help developers quickly understand the functionality of code snippets. For example, the comment "// the injector is set up" provides a summary of what the code snippets are about to do.
- ***How***: This type of inline comment describes the implementation details of how the code snippet is completed or describes how to use the corresponding code snippet. This type of inline comment could help developers to understand the code snippet, especially when the code complexity is high. For example, the comment "// build the polynomials by iterating on the top diagonal of the divided differences array" is used to describe the implementation details of code.
- ***Why***: This type of inline comment explains the intentions of the code snippet, i.e. why the code snippet needs to use this implementation or clarifies why it is written in a specific pattern. For example, the comment "// doc 3 doesn't have a "test" field but we're defaulting it to 100 so it should be last" explains why the code snippet needs to use this implementation.

Then, to automatically classify inline comments by models, we extract five features based on a manual review by the abovementioned three Ph.D. The results are shown in Table 1. The first column shows the features, and the second column introduces the type of each feature, i.e. numeric and string. The last column gives the description. Feature TokenNum, PrepNum, and ConjunNum have been validated in previous studies [12] for their effectiveness in comment classification. Feature Keywords and Ratio are of positive importance for classification based on our manual review. Specifically, a large TokenNum and Ratio may indicate a comment has a higher probability to be an explanation of implementation details or intentions, i.e., how or why comment. PrepNum and ConjunNum mean the specific relations (preposition or conjunction) in the comment, such as the word "because" may indicate the category why. Keywords means specific words that may indicate types, such as the word "via" may indicate the category how.

We automatically extract these features from inline comments and train classification models based on the filtered sample dataset in Section4.2. Following [12, 43, 44], we adopt three classification methods that are commonly used in classification tasks and can achieve good results [12, 45], namely Decision Tree [46], Random Forest [47], and Convolutional Neural Network [45]. We chose these three models for the following reasons: 1) During the training process, we utilized a manually labeled dataset. Due to the constraints of costs in human labeling, our sampled dataset consisted of 4,000 instances. Based on this dataset, we conducted classification training and ten-fold cross-validation. We believe that these models are more capable of effectively utilizing the limited data for training, mitigating the risk of overfitting; 2) Considering the limitations in computational resources and time, the

Table 1: **Features for inline comment classification**

Feature	Type	Description
TokenNum	Numeric	number of tokens in inline comment
PrepNum	String	specific prepositional phrases in inline comment
ConjunNum	String	specific conjunction relations in inline comment
Keywords	String	words that may indicates a specific type
Ratio	Numeric	the ratio of number of tokens in inline comment and number of tokens in code

training speed of decision tree and random forest models is relatively fast, and the structure of CNN models is more amenable to adjustments and optimizations.

During model training, we use a 10-fold cross-validation method, that is, we randomly select one fold of data each time as the test set and other data as a training set. We use five traditional metrics, namely Precision, Recall, F1-score, Accuracy, and Hamming Loss. The calculation is conducted based on sklearn[n] library. Finally, we automatically classify the entire dataset with the classification model (achieves the best results on metrics) and conduct a statistic analysis. The results are shown as follows.

Table 2: **The performance of our classification model based on traditional metrics**

Model	Precision	Recall	F1-score	Accuracy	Hamming Loss
DCT	83.2%	84.2%	83.5%	87.6%	0.1238
RFC	86.0%	86.8%	86.2%	89.5%	0.1052
CNN	88.1%	86.2%	86.9%	89.7%	0.1029
CNN*	88.1%	86.8%	87.3%	90.0%	0.1000

Results of classification. The results are shown in Table 2. Column 1 presents the models we used, among them, CNN* means the model is trained on the basis of extracted features. Columns 2-6 list the results of traditional metrics. From the table, we can find that:

[n] https://scikit-learn.org/stable/modules/classes.html

Table 3: **The statistic results of different categories of inline comments**

		How	**What**	**Why**	**All**
Length of Code	1st Quartile	6	5	5	5
	Median	12	10	12	11
	3rd Quartile	26	23	25	24
Length of Inline Comment	1st Quartile	7	3	9	4
	Median	11	5	13	7
	3rd Quartile	17	8	22	11
Num. of Tokens per Data	Code	29.7	24.8	37.5	27.4
	Comment	14.1	6.1	18.1	9.3
Num. of Tokens	Code	37,236,805	93,537,461	26,496,560	157,270,826
	Comment	17,659,979	23,029,783	12,780,683	53,470,488
Num. of Unique Tokens	Code	1,397,278	2,660,627	923,389	3,428,800
	Comment	219,662	378,587	170,788	564,123
The proportion		21.87%	65.82%	12.32%	100%
Number		1,255,267	3,778,373	707,130	5,740,770

- The three models all achieve high precision, recall, F1-score, and low hamming loss in automatic classification inline comments, which indicates the effectiveness of our classification.
- The random forest and CNN models achieve high precision, recall, F1-score, and low hamming loss, compared with the decision tree model. The CNN model based on the features of our analysis could achieve the best results.

Finally, based on the extracted features, we use the CNN model (i.e. CNN* in Table 2) to classify the `<Inline Comment, Code>` pairs in all datasets.

4.4. *Assessment of the Classified Dataset*

4.4.1. *Statistical analysis and results*

We conduct a statistical analysis of the dataset after classification. The statistical analysis mainly contains the following aspects: 1) the length of code and inline comments, we calculate the 1st quartile, median, and 3rd quartile of them; 2) the number of tokens in all data pairs and in per data; 3) the number of unique tokens.

Results of statistical analysis. The corresponding results are shown in Table.3. From the table, we can find that:

- Overall, most inline comments in the dataset are relatively short. The median value of the length of questions is 7, and 75% of inline comments have a length <= 11 words. Meanwhile, the median value for code snippets is 11, and 75% of code snippets have a length <= 24 words. Most inline comments contain 9.3 tokens and most codes contain 27.4 tokens on average.

- Token used in inline comments and corresponding code is abundant. This indicates by the result that the number of unique tokens of codes is more than three million and the number of unique tokens of inline comments is more than fifty thousand.
- The value of length for the how and why categories are higher than the value of length for all the data, and the length for the what category is the shortest. This indicates that developers use longer sentences to describe the implementation details and the intentions behind the code snippet, and more concise sentences to describe the functionality of the code snippet.
- What category achieves the highest proportion, with a value of 65.82%. The proportion of how and why categories are relatively low, with values of 21.87% and 12.32%, respectively. This indicates that developers usually write what category of inline comments that are short and concise summarizing the functionality of the code snippet.

4.4.2. *Manual review and results*

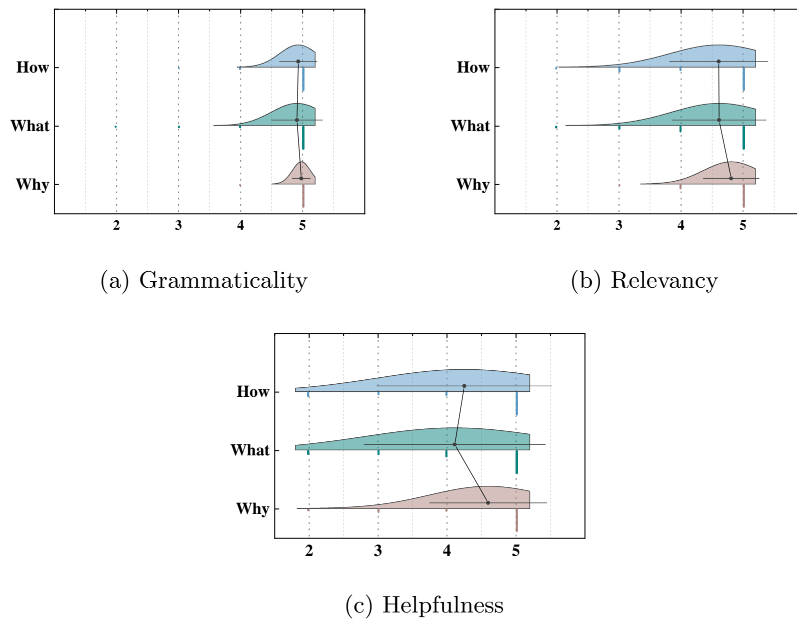
To explore whether different categories of comments perform differently in code understanding activity from the perspective of developers, we conduct a manual review to evaluate different categories of inline comments. Considering that it is impossible for us to manually check all inline comments (due to their large number), we decide to analyze a sample set of them. By following the sampling strategy in [21, 40, 48], we randomly select 384 pairs of `<Inline Comment, Code>`, with a confidence level of 95% and sampling error within the range of ± 5%.

The manual evaluation process is carried out by three Ph.D. students (not co-authors of this paper). We use a cross-validation method and assign each `<Inline Comment, Code>` pair to these three people. They settle their emerging differences through open discussion. If the discussion fails to reach an agreement, this pair of data will be discarded, and a new pair will be sampled and analyzed so that the number of samples remains at 384. During the review process, the five-point Likert scale [49] is used. We mainly focus on three aspects of inline comments and corresponding codes in the review process. We use grammaticality to assess the syntactic and semantics of inline comments in our dataset. In addition, we use relevancy and helpfulness to assess the inline comments when it is corresponding to a code snippet. The details are as follows:

- Grammaticality and Semantic Correctness: It measures how likely an inline comment is grammatically and semantically correct. We use 1, 2, 3, 4, and 5 to represent the correctness of the inline comment, i.e., completely incorrect (1), basically incorrect (2), partially correct (3), correct (4), and completely correct (5).
- Relevancy: It measures to what extent an inline comment is relevant to the target code. We also use 1-5 to represent the relevancy of the inline

comment, i.e., completely irrelevant (1), basically irrelevant (2), partially relevant (3), relevant (4), and completely relevant (5).

- Helpfulness: It measures to what extent an inline comment is likely to help the developers understand the target code. We also use 1-5 to represent the helpfulness of the inline comment, i.e., completely unhelpful (1), basically unhelpful (2), partially helpful (3), helpful (4), and very helpful (5).



(a) Grammaticality

(b) Relevancy

(c) Helpfulness

Fig. 3: **The Likert rating results on the human evaluation of inline comments from three respects**

Results of manual review. Fig.3 shows the results. Detailedly, the x-axis represents the score of inline comments, and the y-axis represents the categories of inline comments. Fig.3a, Fig.3b, and Fig.3c list the distribution of scores of inline comments in grammaticality, relevancy, and helpfulness, respectively. From Fig.3, we can find that:

- In grammaticality, all three categories of inline comments in our benchmark dataset perform well, with an average value of close to 5 (completely correct).
- In relevancy, the inline comments in the what and how categories achieve a similarly average value (4.63 and 4.61), and the inline comments in the why category are almost completely relevant to the code snippet (with a

value of 4.81).

- In helpfulness, the three categories of inline comments show differences. All of them are helpful in code understanding activities. The why category performs best in average value (4.47), followed by the how category (4.25), and the what category is the lowest (4.11). This indicates that inline comments of the why and how category, i.e. comments that describe the implementation details and the intentions behind the code snippet, are more helpful in the code understanding activities from the perspective of developers.
- Overall, 14.8% of data (57/384) are scored as completely unable (score 1) and basically unable (score 2) to help code understanding activities. We further analyze this part of data and find that 4.2% (16/384) of them have inconsistency between code and comments, which may result in low scores.

In summary, the inline comments in our benchmark dataset are almost completely correct in grammatically, relevant with code snippets, and helpful in code understanding activities, especially the why category of inline comments.

5. CONSTRUCTION OF BASELINES

5.1. *Data Preparation*

To train CodeBERT and Bart, we divide our dataset after the automatic classification into the training set, the validation set, and the testing set according to the proportion of 8:1:1. We also remove the duplicate data in the testing set. The final dataset for each category is a two-tuple containing inline comments and corresponding code snippets. Due to the request restrictions of ChatGPT, we sample 384 pairs of data in the test set (with a confidence level of 95% and sampling error within the range of $\pm$ 5%) and try to ask ChatGPT to generate comments for the given code snippet.

5.2. *Comments Generation Model*

A CodeBERT [50] model is based on BERT [51] and a multi-layer bidirectional transformer as the basic model structure. The parameters and structure of CodeBERT are similar to that of RoBERTa [52], and it is a bimodal model pre-trained with natural language and programming languages such as Python and Java.

A Bart [53] model is based on a standard seq2seq/machine translation architecture with a bidirectional encoder and a left-to-right decoder.

ChatGPT is based on the GPT-3.5 architecture and can be used for various tasks such as answering questions, generating comments, and performing translations. ChatGPT achieves natural language processing and understanding through vast training data and deep learning algorithms.

We choose these three baselines for the following reasons. First, CodeBERT and Bart are open-source and can be easily experimented with and compared. Furthermore, they can be combined with other models to enhance their performance in

future exploration. Second, CodeBERT and Bart are commonly used by researchers for NLP translation tasks and Javadoc comment generation, which achieves good performance and accuracy according to existing research [54, 55]. Finally, large language models have also achieved exciting results in text generation. This naturally inspires us to explore whether large models perform well in inline comment generation.

The input of the CodeBERT and Bart is the token sequence of the code snippet, i.e. the source code is represented as a sequence of symbols, which is a natural language representation of the code, ignoring the syntactic of the code. The input of ChatGPT is the instructions and code snippets. The output of the CodeBERT, Bart, and ChatGPT are the corresponding inline comments of the input code snippet.

For CodeBERT and Bart, we fine-tune the model parameters on our dataset. For ChatGPT, we utilize the API provided by OpenAI to conduct generation tasks on the sampled data in the test set. The model training and prediction are conducted on a machine with Nvidia GTX 1080 GPU, Intel(R) Core(TM) i7-6700 CPU, and 16 GB RAM. The operating system is Ubuntu, and the JDK version is 10.

5.2.1. *Evaluation on traditional metrics*

We adopt BLEU [56] and ROUGE [57] to measure the performance of baselines. These two metrics are widely used in the machine translation task and have been adopted to evaluate the performance of many tasks in software engineering, such as code comment generation [58, 59], commit message generation [60, 61] and pull request description generation [62].

BLEU [56] analyzes the n-grams of the candidate and the reference translation, then counts the number of matches. Modified n-gram precision p_n is computed, where n can be 1,2,3, and 4, results in BLEU-1, BLEU-2, BLEU-3, and BLEU-4. BLEU-4 takes the geometric mean of the modified precision scores and then multiplies the result by an exponential brevity penalty factor. ROUGE [57] counts the number of overlapping units such as n-gram, word sequences, and word pairs and we use ROUGE-L in our evaluation. In general, BLEU and ROUGE results are complementing, which are similar to precision and recall, so we adopt both of them in our evaluation. We calculate the BLEU score based on the nltk[o] package and the ROUGE score based on the rouge[p] package, respectively.

5.2.2. *Results of Automatic Inline Comments Generation Baselines*

Based on the prepared benchmark dataset in Section5.1, we train the models to automatically generate inline comments. For the generated inline comments predicted by models from the testing set, our BLEU1-BLEU4 [56] and ROUGE-L [57] results are shown in Table.4. Specifically, column 1 represents the baseline models

[o]http://www.nltk.org/api/nltk.translate.html
[p]https://pypi.org/project/rouge/

and column 2 shows the categories of inline comments. Columns 3-6 list the results of BLEU1-BLEU4, respectively. Column 7 shows the value of ROUGE-L. From the table, we can find that:

Table 4: **Results of metrics-based evaluation of the inline comments generation baselines**

Model	Category	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Rouge-L
CodeBERT	What	43.24	42.05	42.04	**42.61**	**48.48**
	How	39.14	36.92	35.8	35.33	46.82
	Why	36.43	34.14	33.00	32.41	43.64
Bart	What	53.71	44.37	39.23	**35.61**	32.19
	How	45.98	38.24	32.77	28.92	**32.54**
	Why	39.05	32.58	27.74	24.33	30.32
ChatGPT	What	11.55	9.30	7.38	5.75	7.22
	What*	13.49	10.78	8.56	6.70	7.80
	How	22.01	17.87	14.24	11.23	10.72
	How*	27.85	22.55	18.05	14.37	11.75
	Why	30.67	24.83	19.31	14.87	11.05
	Why*	37.61	29.80	22.97	**17.74**	**12.61**

[1] What*/How*/Why*: the result of the concise comment given by ChatGPT.

- CodeBERT achieves the best performance among the three baselines. The BLEU4 value of CodeBERT ranges from 32.41 to 42.61, and ROUGE-L ranges from 43.64 to 48.48. The lowest BLEU4 and ROUGE-L come from the why category while the highest BLEU4 and ROUGE-L come from the what category, which indicates that for CodeBERT it may be difficult to generate the intentions of code from code snippets. In addition, the difference in the value of BLEU2-BLEU4 of what category is relatively small, which may be due to the relatively short inline comments of this category, with over 50% length of 5 and over 75% length of less than 8.
- Bart achieves similar performance to CodeBERT in different categories. In addition, the value from BLEU1 to BLEU4 is decrease rapidly. It performs better than CodeBERT on BLEU1-BLEU2, but not as well as CodeBERT on BLEU3-BLEU4.
- ChatGPT achieves the lowest value of BLEU4 and ROUGE-L values, which ranges from 5.75 to 14.87 and from 7.22 to 11.05, respectively. This indicates that is difficult for ChatGPT to generate the existing inline comments in the real project currently. The lowest BLEU4 and ROUGE-L come from the what category while the highest BLEU4 and ROUGE-L come from the why category, as opposed to the performance of CodeBERT and Bart. This indicates that ChatGPT may perform better in generating longer com-

ments, as both the how and why categories of comments are relatively long (with a median value of 11 and 13), while what category of comments are relatively short (with a median value of 5).

To understand why ChatGPT obtains the lowest values of BLEU and ROUGE, we conduct a manual review of the dataset (ChatGPT used) to find possible reasons and present several examples to visualize the results. We list three Java code snippets with their real inline comments and the comments given by ChatGPT and CodeBERT, as shown in Fig.4. Fig.4a, Fig.4b, and Fig.4c list an example of how, what, and why categories of inline comments, respectively. Our findings are as follows:

- The comments generated by ChatGPT are significantly longer in length than the original human-written comments, such as the instances in Fig.4a, Fig.4b, and Fig.4c. Previous research has shown that developers typically expect inline comments to be less than 10 words [10]. Therefore, we modified the input instruction to let ChatGPT give a concise comment, then a comment (corresponding code is in Fig.4b) of reasonable length "// Computes polynomial coefficients using top diagonal values and abscissae." will be given. We also list the corresponding metric results in Table 4, i.e. the results with a symbol "*", which get a promotion. This may suggest that when generating comments with ChatGPT, the entered instructions need to be carefully considered, and that differences in one word may lead to significant differences in the results.
- ChatGPT performs not well at capturing semantic information beyond code snippets to generate inline comments currently. For example, in the instance in why category (Fig.4c), the semantics of the generated comments given by ChatGPT are partly different from the original comments. This may be due to the reason that the instances in the why category may contain extra semantic information that goes beyond the snippet of code, such as "doc 3" and "it should be last", therefore, it is a challenge for ChatGPT to capture and incorporate these pieces of information into inline comment generation currently.
- Based on our proposed data set, we fine-tune the open-source CodeBERT model and conduct the evaluation using the test set. However, the ChatGPT model was not fine-tuned on our data set (non-open source), hence it may not perform as well on the sampled test set for the inline comment generation task. To a certain extent, this may also indicate the effectiveness of our dataset in the inline comment generation task, as models trained on our data set demonstrate are capable of generating inline comments written by developers more effectively.

```
1 t0=(c17 + CARRY_ADD) >> 28;
2 c17-=(t0 << 28);
3 c18+=t0;
4 Human-writen: // carry from position 17
5 CodeBERT: // carry from position 17
6 ChatGPT: // This code performs a 28-bit right shift of
  the value stored in c17 and adds a carry value. The
  result is stored in t0.
7 // The shifted value is then subtracted from c17. The
  value of t0 is added to c18. This operation is commonly
  used in multi-precision arithmetic to update the carry
  chain between adjacent 28-bit words.
```

(a) Example in what category

```
1 for (int i=0; i < topDiagonal.size(); ++i) {
2   double[] tdi=topDiagonal.get(i);
3   for (int k=0; k < polynomials.length; ++k) {
4
   polynomials[k]=polynomials[k].add(coeff.multiply(polyno
  mial(tdi[k])));
5   }
6   coeff=coeff.multiply(polynomial(-
  abscissae.get(i),1.0));
7 }
8 Human-writen: // build the polynomials by iterating on
  the top diagonal of the divided differences array
9 CodeBERT: // build the polynomials by iterating on the
  top diagonal of the divided differences array
10 ChatGPT: // Computes the polynomial coefficients for a
  given set of top diagonal values and abscissae.
11 // The top diagonal values are stored in a list called
  topDiagonal, and the abscissae are stored in a separate
  list called abscissae.
12 // The polynomial coefficients are computed using the
  given polynomials and a coefficient value that is
  multiplied by each polynomial.
13 // The coefficient value is updated after each iteration
  using the current top diagonal value and a constant
  value of 1.0.
```

(b) Example in how category

```
1 response=client().prepareSearch("test").setExplain(randomB
  oolean()).setQuery(functionScoreQuery(matchAllQuery(),fiel
  dValueFactorFunction("test").modifier(FieldValueFactorFunc
  tion.Modifier.RECIPROCAL).missing(100))).get();
2 assertOrderedSearchHits(response,"1","2","3");
3 Human-writen:// doc 3 doesn't have a "test" field but
  we're defaulting it to 100 so it should be last
4 CodeBERT:// doc 3 doesn't have a "test" field but we're
  defaulting it to 1 so it should be 0
5 ChatGPT: // Executes a search query on the "test" index
  with a function score query that boosts documents based on
  the value of the "test" field using the reciprocal
  modifier. The query also sets the "missing" value to 100
  for documents with a missing field. The search results are
  then compared to the expected order of documents with IDs
  "1", "2", and "3".
```

(c) Example in why category

Fig. 4: **The examples of generated inline comments**

6. DISCUSSION

6.1. *Implications*

Implications for researchers. The exploration we made would provide foresight for the generating of inline comments. We provide the first large-scale benchmark dataset named ICG with 5,740,770 pairs of inline comments and code snippets after fine-graned data pre-processing noise cleaning. We find that the why category of inline comments is almost completely relevant to the code snippet and achieves the highest value in helpful for understanding code, followed by the how category. Therefore, it would be interesting to explore the possibility of generating high-quality why and how categories of inline comments in the future.

In addition, our experimental results (Section 5.2.2) show that even current SOTA methods cannot achieve ideal results. On the one hand, the metric-based evaluation results given by three baselines show that the effectiveness of the methods still needs to be improved. On the other hand, according to the results given by ChatGPT, overly long comments may affect code understanding, and how to succinctly capture the semantics hidden behind the code text and the developer's intentions are still very challenging to study. Therefore, more advanced methods are expected to improve the performance further. Moreover, we manually labeled a dataset of 4,000 instances for the classification task and employed models such as Random Forest and CNN for the classification task. In the future, the utilization of a larger labeled dataset and more efficient approaches, such as those based on pre-trained models, can be further explored. Moreover, we manually labeled a dataset of 4,000 instances for the classification task and employed models such as Random Forest and CNN for the classification task. In the future, the utilization of a larger labeled dataset and more efficient approaches, such as those based on pre-trained models, can be further explored.

Implications for tool developers. This work can motivate tool developers to develop IDE plugins that can provide timely feedback by suggesting inline comments once a developer finishes writing a piece of code. Such feedback can work as an assistant for code understanding activities for developers. This is indicated by the result that inline comments in our dataset could help developers understand the target code (in Section 4.4.2). In addition, it would be more helpful and valuable if tool developers could pay more attention to generating inline comments that describe the implementation details and the intentions behind the code snippet. This is motivated by our results that the why category achieves the highest value in helpful for understanding code, followed by the how category.

6.2. *Threats to validity*

Threats to Construct Validity. One threat is about the construction of the data set (in Section 4). In this work, we associate inline comments and code by the heuristics in work [11]. Although they could obtain an accuracy of 95% in sampled

data. We cannot guarantee that these heuristic rules could always obtain such an or even higher accuracy in any case.

Another threat is that the classification of the whole dataset in Section4.3 was done by the CNN classifier, which is not 100% accurate. However, given the fact that CNN performs well in many comment classification tasks [12, 63], we believe that the current accuracy (88.1%) is acceptable.

Threats to Internal Validity. The major threat is that there may be individual bias in the manual review about the grammaticality, relevancy, and helpfulness in code understanding activities in Section 4.4.2. To ensure the accuracy and objectivity of results, the review is conducted by three Ph.D. students and they cross-checked the data and solved disagreements through discussions.

Threats to External Validity. One threat is that our exploration is only conducted on open-source Java projects. We cannot guarantee that our process of dataset construction is applicable to industrial or projects in other programming languages. However, considering that Java is one of the most popular programming languages in the world and is widely used in software projects development [1,21,64], we believe that our research could still provide promotion about the process of construction of large-scale dataset. In the future, we plan to replicate our work in more software projects and other programming languages.

Threats to Conclusion Validity. The major threat is that we perform three types of manual reviews in this work, namely (1) checking the noise of inline comments (Section4.2), (2) classifying inline comments (Section4.3), and (3) evaluating the quality of the dataset (Section 4.4.2). All participants evolved in these processes are not the owners (developers) of the code base, we cannot guarantee that all judgments made by the participants are correct. In order to reduce these biases, we cross-checked the data by three Ph.D. students and solved disagreements through discussions.

7. CONCLUSION

In this paper, we construct a benchmark dataset of inline code comments and provide several baselines of automatic inline comments generation. We first collect a large dataset of 8,000 Java projects from GitHub. Then we conduct a manual review to obtain heuristics to filter the noises in the dataset, resulting in a dataset of 5,740,770 pairs of inline comments and code snippets. After that, we propose a comprehensive taxonomy of three categories (i.e. what, how and why) of inline comments. Then we evaluate our benchmark dataset through statistical and manual analysis. Our statistical results show that most inline comments are relatively short and the token used in inline comments and corresponding code is abundant in our benchmark. In the manual analysis, our results show that the inline comments are correct in grammaticality, and relevant to the code. And the comments in why category achieve the highest value of helpfulness in code understanding activities from the perspective of developers. Finally, we train and compare three models to

automatically generate inline comments for code snippets, to make our benchmark easy to use for researchers.

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan and S. Li, Measuring program comprehension: A large-scale field study with professionals, *IEEE Transactions on Software Engineering* **44**(10) (2017) 951–976.
[2] Y. Shinyama, Y. Arahori and K. Gondow, Analyzing code comments to boost program comprehension (Dec 2018).
[3] M. Allamanis, H. Peng and C. Sutton, A convolutional attention network for extreme summarization of source code (2016).
[4] Q. Chen and M. Zhou, A neural framework for retrieval and summarization of source code (2018).
[5] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation (2018).
[6] X. Hu, G. Li, X. Xia, D. Lo, S. Lu and Z. Jin, Summarizing source code with transferred api knowledge (2018).
[7] S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Summarizing source code using a neural attention model (2016).
[8] A. LeClair, S. Jiang and C. McMillan, A neural model for generating natural language summaries of program subroutines (2019).
[9] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu and P. S. Yu, Improving automatic source code summarization via deep reinforcement learning (2018).
[10] Y. Huang, H. Guo, X. Ding, J. Shu, X. Chen, X. Luo, Z. Zheng and X. Zhou, A comparative study on method comment and inline comment, *ACM Transactions on Software Engineering and Methodology* (2023).
[11] X. Zhang, W. Zou, L. Chen, Y. Li and Y. Zhou, Towards the analysis and completion of syntactic structure ellipsis for inline comments, *IEEE Transactions on Software Engineering* (2022).
[12] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan and X. Zhang, Cpc: Automatically classifying and propagating natural language comments via program analysis (2020).
[13] Y. Padioleau, L. Tan and Y. Zhou, Listening to programmers—taxonomies and characteristics of comments in operating system code (2009).
[14] I. K. Ratol and M. P. Robillard, Detecting fragile comments (2017).
[15] Y. Huang, S. Huang, H. Chen, X. Chen, Z. Zheng, X. Luo, N. Jia, X. Hu and X. Zhou, Towards automatically generating block comments for code snippets, *Information and Software Technology* **127** (2020) p. 106373.
[16] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis and M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, *arXiv preprint arXiv:1909.09436* (2019).
[17] E. Wong, T. Liu and L. Tan, Clocom: Mining existing source code for automatic comment generation (2015).
[18] D. Haouari, H. A. Sahraoui and P. Langlais, How good is your comment? a study of comments in java programs (2011).
[19] Maalej, Walid, Robillard and P. Martin, Patterns of knowledge in api reference documentation, *IEEE Transactions on Software Engineering* **39**(9) (2013) 1264–1282.
[20] D. Steidl, B. Hummel and E. Jürgens, Quality analysis of source code comments (2013).
[21] L. Pascarella and A. Bacchelli, Classifying code comments in java open-source soft-

ware systems (2017).

[22] R. M. Santos, M. C. R. Junior and M. G. d. Mendonça Neto, Self-admitted technical debt classification using lstm neural network (2020).

[23] E. Maldonado, E. Shihab and N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, *IEEE Transactions on Software Engineering* (2017) 1–1.

[24] M. Farias, M. Neto, M. Kalinowski and R. Spínola, Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary, *Information and Software Technology* **121** (2020) p. 106270.

[25] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo and S. Li, Is using deep learning frameworks free?: characterizing technical debt in deep learning frameworks (2020).

[26] G. Bavota and B. Russo, A large-scale empirical study on self-admitted technical debt (2016).

[27] A. T. Ying, J. L. Wright and S. Abrams, Source code that talks: an exploration of eclipse task comments and their implication to repository mining, *ACM SIGSOFT software engineering notes* **30**(4) (2005) 1–5.

[28] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock and K. Vijay-Shanker, Automatic generation of natural language summaries for java classes (2013).

[29] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock and K. Vijay-Shanker, Towards automatically generating summary comments for java methods (2010).

[30] S. Haiduc, J. Aponte, L. Moreno and A. Marcus, On the use of automated text summarization techniques for summarizing source code (2010).

[31] D. Movshovitz-Attias and W. W. Cohen, Natural language models for predicting programming comments (August 2013).

[32] S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Summarizing source code using a neural attention model (2016).

[33] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation (2018).

[34] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation with hybrid lexical and syntactical information, *Empir. Softw. Eng.* **25**(3) (2020) 2179–2217.

[35] B. Wei, Retrieve and refine: Exemplar-based neural comment generation (2019).

[36] J. Zhang, X. Wang, H. Zhang, H. Sun and X. Liu, Retrieval-based neural source code summarization (2020).

[37] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation with hybrid lexical and syntactical information, *Empirical Software Engineering* **25**(3) (2020) 2179–2217.

[38] T. D. Nguyen, A. T. Nguyen and T. N. Nguyen, Mapping api elements for code migration with vector representations (2016).

[39] P. Leitner and C.-P. Bezemer, An exploratory study of the state of practice of performance testing in java-based open source projects (2017).

[40] J. Zhang, X. Wang, H. Zhang, H. Sun and X. Liu, Retrieval-based neural source code summarization (2020).

[41] L. Pascarella, M. Bruntink and A. Bacchelli, Classifying code comments in java software systems, *Empirical Software Engineering* **24**(3) (2019) 1499–1537.

[42] Q. Chen, X. Xia, H. Hu, D. Lo and S. Li, Why my code summarization model does not work: Code comment improvement with category prediction, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **30**(2) (2021) 1–29.

[43] C. Treude and M. P. Robillard, Augmenting API documentation with insights from stack overflow (2016).

[44] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter and C. Weiss, What makes a good bug report?, *IEEE Trans. Software Eng.* **36**(5) (2010) 618–643.

[45] Y. Chen, Convolutional neural network for sentence classification, Master's thesis, University of Waterloo (2015).
[46] J. R. Quinlan, *C4. 5: programs for machine learning* (Elsevier, 2014).
[47] L. Breiman, Random forests, *Machine learning* **45** (2001) 5–32.
[48] Y. Jiang, H. Liu, Y. Zhang, N. Niu, Y. Zhao and L. Zhang, Which abbreviations should be expanded? (2021).
[49] R. A. Likert, A technique for measurement of attitudes, *archive psychology of new york* (1932).
[50] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, Codebert: A pre-trained model for programming and natural languages, *arXiv preprint arXiv:2002.08155* (2020).
[51] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, *arXiv preprint arXiv:1810.04805* (2018).
[52] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov, Roberta: A robustly optimized bert pretraining approach, *arXiv preprint arXiv:1907.11692* (2019).
[53] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov and L. Zettlemoyer, Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, *arXiv preprint arXiv:1910.13461* (2019).
[54] Y. Huang, M. Wei, S. Wang, J. Wang and Q. Wang, Yet another combination of IR- and neural-based comment generation, *Inf. Softw. Technol.* **152** (2022) p. 107001.
[55] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui and F. Liu, A survey of automatic source code summarization, *Symmetry* **14**(3) (2022) p. 471.
[56] K. Papineni, S. Roukos, T. Ward and W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation (2002).
[57] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries (2004).
[58] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation (2018).
[59] A. LeClair, S. Jiang and C. McMillan, A neural model for generating natural language summaries of program subroutines (2019).
[60] S. Jiang, A. Armaly and C. McMillan, Automatically generating commit messages from diffs using neural machine translation (2017).
[61] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing and X. Wang, Neural-machine-translation-based commit message generation: How far are we? (2018).
[62] Z. Liu, X. Xia, C. Treude, D. Lo and S. Li, Automatic generation of pull request descriptions, *CoRR* **abs/1909.06987** (2019).
[63] F. Mu, X. Chen, L. Shi, S. Wang and Q. Wang, Developer-intent driven code comment generation, *arXiv preprint arXiv:2302.07055* (2023).
[64] S. Cass, The 2018 top programming languages, *IEEE Spectrum* **31** (2018) p. 1.

View publication stats