# Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction

QIUYUAN CHEN, College of Computer Science and Technology, Zhejiang University
XIN XIA and HAN HU, Faculty of Information Technology, Monash University, Victoria, Australia
DAVID LO, School of Information Systems, Singapore Management University, Singapore
SHANPING LI, College of Computer Science and Technology, Zhejiang University

Code summarization aims at generating a code comment given a block of source code and it is normally performed by training machine learning algorithms on existing code block-comment pairs. Code comments in practice have different intentions. For example, some code comments might explain how the methods work, while others explain why some methods are written. Previous works have shown that a relationship exists between a code block and the category of a comment associated with it. In this article, we aim to investigate to which extent we can exploit this relationship to improve code summarization performance. We first classify comments into six intention categories and manually label 20,000 code-comment pairs. These categories include *"what," "why," "how-to-use," "how-it-is-done," "property,"* and *"others."* Based on this dataset, we conduct an experiment to investigate the performance of different state-of-the-art code summarization approaches on the categories. We find that the performance of different code summarization approaches varies substantially across the categories. Moreover, the category for which a code summarization model performs the best is different for the different models. In particular, no models perform the best for *"why"* and *"property"* comments among the six categories. We design a composite approach to demonstrate that comment category prediction can boost code summarization to reach better results. The approach leverages classified code-category labeled data to train a classifier to infer categories. Then it selects the most suitable models for inferred categories and outputs the composite results. Our composite approach outperforms other approaches that do not consider comment categories and obtains a relative improvement of 8.57% and 16.34% in terms of *ROUGE-L* and *BLEU-4* score, respectively.

CCS Concepts: • **Software and its engineering** → *Software creation and management*; *Software development techniques*;

Additional Key Words and Phrases: Code summarization, code comment, comment classification

This research was partially supported by the National Key R&D Program of China (Grant No. 2019YFB1600700), NSFC Program (Grant No. 61972339), and the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (Grant No. DE200100021).

Authors' addresses: Q. Chen and S. Li, College of Computer Science and Technology, Zhejiang University, 38 Zheda Rd, 310000, Hangzhou, Zhejiang, China; emails: {chenqiuyuan, shan}@zju.edu.cn; X. Xia (corresponding author) and H. Hu, Faculty of Information Technology, Monash University, Wellington Road, Melbourne, Victoria, Australia; emails: {xin.xia, han.hu}@monash.edu; D. Lo, School of Information Systems, Singapore Management University, 81 Victoria Street, 188065, Singapore; email: davidlo@smu.edu.sg.

1049-331X/2021/02-ART25 $15.00
https://doi.org/10.1145/3434280

**ACM Reference format:**
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, Article 25 (February 2021), 29 pages.
https://doi.org/10.1145/3434280

## 1 INTRODUCTION

High-quality descriptive documentation, or code summary, is always essential for software development and maintenance, because programmers spend 58% of their time on average on source code comprehension [69]. Software documentation, especially code comments, plays a vital role in helping developers to comprehend source code. However, documenting source code is a labour-intensive task [10, 25], which increases the necessity of automatic or semi-automatic approaches. Code summarization, which generates natural language descriptions of source code, can assist developers in capturing a high-level overview and helping developers to understand a piece of code without having to read the whole source code. Summarizing code can be viewed as a documentation extension [66], where a good code summary can not only help keep the consistency of the code and comments in software maintenance [23] but also improve the performance of code search using natural language queries [47, 71].

Most code summarization approaches are learning-based [2, 8, 9, 20, 22, 24, 63] or retrieval-based [11, 44, 53, 68], which build models on a large dataset often consisting of method-level source code snippets and the corresponding comments [3, 8, 20, 21, 63]. This way of model construction assumes that a large enough dataset can enable models to work on divergent scenarios. So they pursue a general representation of code-comment relation and use the same treatment for different kinds of comments to perform code summarization. However, comments are complicated in practice. On the one hand, a comment may pay attention to a particular aspect instead of a full description of the code. On the other hand, a comment may describe not only the functionality but also the rationale. For example, some comments explain a business logic instead of the code itself, describing why a particular piece of code is written. However, to the best of our knowledge, there is no investigation about the impact of different types of comments on code summarization.

To explore the impact of different categories of comments on the performance of code summarization approaches, we conduct an empirical study on an open-source code summarization dataset crawled from 9,714 projects [21]. We split the dataset into three datasets: the *training data* is for training code summarization models, the *validation data* is for validating the impact of comment classification on code summarization, and the *testing data* is for automatic evaluation of our composite approach. We investigate the research questions (RQ) as follows.

First, we conduct manual comment classification on the *validation data*. There are several criteria to categorize the code comments in the literature [49, 50, 72]. We follow a clear and rich code comments taxonomy, which consists of six categories: *"what"* (description of the functionality), *"why"* (why the code is provided or the design rationale of the code), *"how-to-use"* (description of the usage), *"how-it-is-done"* (implementation details of the functionality), *"property"* (explain properties of the code) and *"others"* [72]. We manually label the *validation data* consisting of 20,000 code-comment pairs according to the proposed classification criterion. We follow the coding procedure [55] to guarantee the label quality. The labeling process is labour-intensive; every hour, a participant could label 57 pairs on average. Fleiss Kappa value [13] is then used to measure the agreement among the participants. The Fleiss Kappa value is 0.79, which indicates substantial agreement among the participants. With such comment classification, we can conduct code summarization experiments and explore the differences among the categories.

**RQ1 How do different comment categories impact the code summarization performance?** We then investigate six state-of-the-art summarization models on the derived categories, four in the SE literature, and two in the Natural Language Processing (NLP) literature. For a fair comparison, after checking the reproducibility using source code provided by the corresponding authors, we train all the models on the same *training data* mentioned above. Then, we evaluate the results in each category, separately. We find that there exists a significant difference between each category. The preferences of each model to particular comment categories are different. In particular, there is no model that performs the best for *"why"* and *"property"* comments. Besides, we observe that no model dominates in all comment categories, which indicates that there is a potential to combine the advantages of different models.
**RQ2 How can we improve the code summarization performance using the comment categories?** Our goal is to prove that by treating different categories of comments differently, we can boost the code summarization performance. The insight is that since models have different performance for different categories, we can combine their advantages of the models with the assistance of inferred comment categories. For example, if the inferred category is *"how-to-use,"* we select the model that performs the best for this category to generate comments. Hence, we need a classifier to perform comment category prediction, which plays the role of selecting suitable code summarization models. To this end, we build several classifiers based on our labeled dataset (i.e., the *validation data*), including Random Forest, LightGBM, Decision Tree, Naive Bayes, and a deep-neural-network-based classifier in which Random Forest obtains the best score of *F1 score* in the 10-fold cross-validation. The classifier then selects the best model that should be used for a particular kind of comment.

By categorizing the source code, we design a composite model for boosting code summarization performance. We evaluate the effectiveness of our composite model on the *testing data*, which is unlabeled and has no overlap with the *validation data* as well as the *training data*. Our composite model outperforms other basic approaches that do not consider comment categories, and it obtains a relative improvement of 8.57% in terms of *ROUGE-L*, which has been shown to correlate highly with human assessments of summarized text quality [30] and a relative improvement 16.34% in terms of *BLEU-4* score, which is widely used for evaluating the quality of summaries. Considering that the data distributions (e.g., the mean and the median of the code tokens and natural language words) of the *training data* and the *testing data* are similar, we also apply our approach to an *external dataset*, which is built on 1,500 high-quality Java projects [67] and has a different data distribution. The results show that our model also outperforms the baselines in the *external dataset*. In this way We demonstrate that comment category prediction could boost code summarization to reach better results by combining it with current approaches.

To the best of our knowledge, this work is the first work that leverages comment category prediction to boost source code summarization. In summary, our contributions are as follows:

- We classify code comments into six categories and conduct an experiment to perform six code summarization approaches on them to explore the impact of comment categories on code summarization.
- We design a composite model to improve the performance of code summarization models, which demonstrate how comment category prediction can benefit code summarization task.
- Our research highlights that comment category prediction can make code summarization practical in the real scenario of development.

The rest of the article is organized as follows: Section 2 describes our problem formulation and dataset. Section 3 explores the impact of comment categories on code summarization. Section 4 describes our composite approach that utilizes the output of the comment category prediction

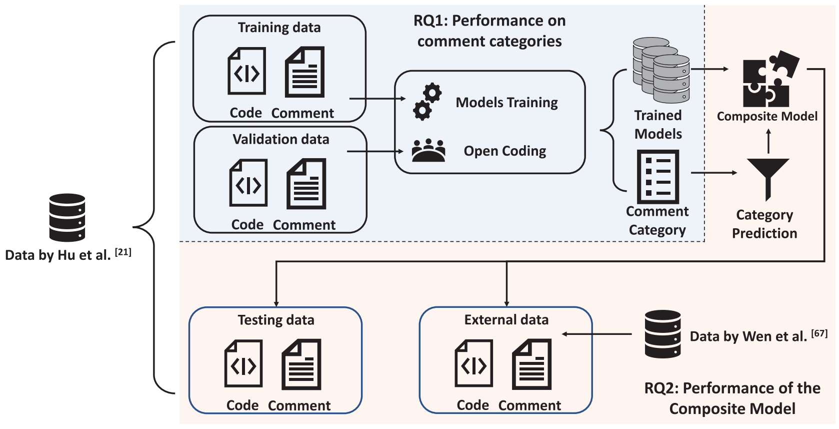

Fig. 1. Overview of the approach.

task to improve code summarization performance. We discuss the implications and threats to validity in Section 5. Section 6 introduces related work on code summarization. We present our conclusions in Section 7.

## 2 SETUP

### 2.1 Approach Overview

Figure 1 shows an overview of the empirical study in RQ1 and our composite approach in RQ2. In this work, we adopt datasets provided by Hu et al. [21] (the *training data*, the *validation data*, and the *testing data*) and Wen et al. [67] (the *external data*, which has a different data distribution).

For RQ1, we manually classify *validation data* consisting of 20,000 code-comment pairs and label categories for them using a procedure of open coding. Then, we train different code summarization models on the *training data* and perform an empirical study to investigate the impact of different categories on the code summarization performance.

For RQ2, leveraging the empirical study and the labeled data, we train a classifier to decide the category for a snippet of the source code. The classifier works to perform comment category prediction. Then, we train code summarization models on the same *training data*. According to each model's performance on the *validation data* (i.e., which model performs the best in a particular category), our model chooses the corresponding model to summarize the source code. Here the *validation data* not only plays the role of validating code summarization models but also plays the role of tuning our composite model for comment category prediction tasks. Last, we evaluate our composite model on the *testing data* and *external data* in which the composite model works by combining the generated comments and outputs the final results.

### 2.2 Problem Formulation

**Code summarization** is a task that aims to summarize the functionality of the source code. We formally describe the mapping as $C \mapsto S$, where $C$ is the input source code, and $S$ is the output natural language summary.

**Comment classification** is a process of classifying code comments, which can help improve the program understanding [49, 50, 72]. In this article, we refer to comment as a "summary" of source code, which is a brief natural language description of that section of source code [42]. Comment classification can be done manually or automatically [72].

Table 1. Count of the Dataset

| Dataset | Pair # (%) | Tokens of code | Lines of code | Words of NL |
|---|---|---|---|---|
| **Train** | 445,812 (91.76%) | 24,889,887 | 1,834,662 | 4,571,644 |
| **Validation** | 20,000 (4.12%) | 1,112,533 | 82,207 | 204,357 |
| **Test** | 20,000 (4.12%) | 1,103,447 | 81,573 | 205,476 |
| **Total** | **485,812** | **27,105,867** | **1,998,442** | **4,981,477** |

**Comment category prediction** leverages the results of the comment classification to build classifiers and predict the comment category given a piece of source code. It is a kind of source code classification task that is studied in software engineering [7, 35, 60, 61], and it learns the relationship between the source code and the comment category in the dataset to build the model.

The mechanism of code summarization is mining information of existing code-comment pairs to generate comments generally. Current approaches are learning-based [2, 8, 9, 20, 22, 24, 63] or retrieval-based [18, 53, 68]. However, they use the same treatment for comments of different categories. Hence, how the code summarization models perform on different kinds of comments is not explored in the literature.

## 2.3 Dataset Setup

We use a Java dataset provided by Hu et al. [21], which consists of 9,714 open source projects in GitHub. The data consists of methods as well as their corresponding comments (i.e., pairs of $\langle code, comment \rangle$), which is extracted using JDT[1], and the granularity is method-level. We keep the way the original data is split; Hu et al. randomly selected 20,000 pairs as *testing data* and *validation data*, respectively, and the remaining 445,812 pairs as *training data* (the three datasets do not come from the same projects). Previous work shows that there are often duplicate methods in different files of the same project [3], which may lead to the risk of leaking data from the *training data* to the *testing data* (i.e., data leakage [26]). Therefore, the *training data*, *validation data*, and *testing data* do not come from the same projects and have no overlap with each other.

In this article, as described in Figure 1, the purposes of the *training data*, *validation data*, *testing data*, and the *external data* are as follows: the *training data* is used to train code summarization models; the *validation data* is used to identify comments categories and validate the code summarization models on these categories (described in Section 3); the *testing data* and *external data* are used to evaluate our composite model on different data distributions (described in Section 4).

The basic characteristics of the dataset are as follows. We call the basic unit of preprocessed source code as a token and call the basic unit of summary as a word. Table 1 describes the counts of tokens of codes, the lines of codes, and words of the natural language of the three data.[2] Furthermore, we calculate the statistical numbers (i.e., minimum, first quarter, median, third quarter, maximum, standard deviation, and mean) of the dataset. As described in Table 2, the statistical results show that there are no significant differences among the datasets, which means the splitting guarantees the consistency of the distribution of each dataset. In addition, the comments are precise as the median is only 10. The standard deviation of words of summaries is only 6.9, which is much lower than the standard deviation of tokens of codes.

Considering that the data distributions of the *training data* and the *testing data* are similar, we also build an *external dataset* with different data distribution to validate the performance of our approach. The *external dataset* is based on 1,500 high-quality Java projects [67] that have no

[1] http://www.eclipse.org/jdt/.
[2] We count the numbers according to the dataset provided by the authors: https://github.com/xing-hu/EMSE-DeepCom.

Table 2. The Statistical Description of the Dataset

| Dataset | Type | Mean | Std. | Min. | 1st Quart. | Med. | 3rd Quart. | Max. |
|---|---|---|---|---|---|---|---|---|
| **Train** | **Code** | 55.83 | 53.07 | 5 | 18 | 36 | 75 | 665 |
| | **NL** | 10.25 | 4.48 | 1 | 7 | 9 | 13 | 32 |
| **Validation** | **Code** | 55.63 | 52.41 | 5 | 18 | 36 | 75 | 397 |
| | **NL** | 10.22 | 4.46 | 1 | 7 | 9 | 13 | 30 |
| **Test** | **Code** | 55.17 | 52.54 | 5 | 18 | 35 | 74 | 365 |
| | **NL** | 10.27 | 4.49 | 2 | 7 | 9 | 13 | 29 |

Table 3. Count of the External Dataset

| Dataset | Pair number | Tokens of code | Lines of code | Words of NL |
|---|---|---|---|---|
| Extra | 20,000 | 1,204,665 | 95,355 | 167,409 |

Table 4. The Statistical Description of the External Dataset

| Type | Mean | Std. | Min. | 1st Quart. | Med. | 3rd Quart. | Max. |
|---|---|---|---|---|---|---|---|
| Code | 60.49 | 92.94 | 9 | 16 | 31 | 69 | 1,937 |
| NL | 8.42 | 7.38 | 1 | 3 | 7 | 11 | 137 |

overlap with the aforementioned dataset of Hu et al. [19]. We use the same approach to extract code-comment pairs, preprocess them, and randomly select 20,000 pairs that have the same number with the testing dataset. The counts of the pairs, the tokens, the lines of code, and the words of comments of the *external dataset* are shown in Table 3. The statistic of the *external dataset* is given in Table 4. As the projects of the two datasets are different, we can see the data distributions of the code and the comments (i.e., NL) of the *external dataset* are different from the *training data*, *validation data*, and *testing data* in terms of statistical metrics (mean, standard deviation, etc.).

In this article, we perform a manual code comment classification to explore its impact on code summarization, which outputs a labeled dataset of code comment categories. The results of the classification are provided in Section 3.2.

## 3 IMPACT OF COMMENT CLASSIFICATION ON CODE SUMMARIZATION

### 3.1 Motivation

Code comments in practice are complicated. When documenting source code, developers would pay attention to specific aspects of the code instead of describing it thoroughly. For example, some comments describe the design rationale (i.e., why the code is written) or explain implementation details (i.e., how functionality is implemented). Such variety leads to different kinds of comments. However, current approaches are agnostic to different kinds of comments (i.e., they use the same strategy to generate comments for different types of comments). Nowadays, the impact of different kinds of comments (i.e., comment classification) on code summarization is unknown. Therefore, our goal is to investigate such an impact of comment classification on code summarization and explore how such techniques can benefit from comment classification.

### 3.2 Comment Classification

*3.2.1 Categories of Code Comments.* The complexity of the code comments is that different developers describe different code entities from different perspectives [72]. Inspired by

Table 5. Comment Categories with Descriptions and Examples

| Category | Description | Example |
|---|---|---|
| What | Gives a description of functionality of the method. | *"A helper function that process the stack."* |
| Why | Explains the reason why the method is provided or the design rationale of the method. | *"Get a copy of the map (for diagnostics)."* |
| How-to-use | Describes the usage or the expected set-up of using the method. | *"Should be called before the object is used."* |
| How-it-is-done | Describes the implementation details of the method. | *"Convert the byte[] to a secret key."* |
| Property | Asserts properties of the method including pre-conditions or post-conditions of a method. | *"Wait until seqno is greater than or equal to the desired value or we exceed the timeout."* |
| Others | Unspecified or ambiguous comments. | *"The implementation is awesome."* |

Zhai et al. [72], we classify comments into six categories: *"what," "why," "how-it-is-done," "property," "how-to-use,"* and *"others."* Table 5 shows the description and example of each comment category: The *"what"* category gives a description of the method functionality (e.g., *"a helper function," "a main-process function"*). The *"why"* category explains the reason why the method is provided or the design rationale of the method (e.g., *"get a copy of the map (for diagnostics)"*). The *"how-to-use"* category describes the usage, the expected set-up, or the environment of the method (e.g., *"should be called before the object is used"*). The *"how-it-is-done"* category describes the implementation details of the method (e.g., *"convert the byte[] to a secret key"*). The *"property"* category asserts properties of the method including pre-conditions or post-conditions of the method where pre-condtions indicate the prerequisite of the method while post-conditions indicate the result of using the method [72]. For example, the comment *"wait until seqno is greater than or equal to the desired value or we exceed the timeout"* explains the pre-condition (i.e., the parameter *"seqno"*), which should satisfy the conditions (≥desired value) for the method to work properly. The *"others"* category includes comments that do not fit any of the categories mentioned above (e.g., *"The implementation is awesome"*).

There are many ways to categorize code comments from different perspectives. As our purpose is exploring the impact of code comments on code summarization, this taxonomy of code comments considers not only from the perspective of developer intentions (e.g., the category "why") but also considers the code properties (e.g., the category "property") from the perspective of program analysis [72]. In this article, we investigate several deep-learning-based code summarization models. On the one hand, these models will regard the sequences of the code and natural language words as inputs. This could help the models to learn the information about semantic and intention, which is related to the categories of *"what"* and *"why."* On the other hand, some deep neural networks would also consider syntactic information of the source code. For example, DeepCom would serialize an abstract syntax tree (AST) of a method of code while Code2Seq would sample paths of the AST and serialize them into a sequence. Then they input the serialized sequences to the neural network to learn the syntactic information. This characteristic could help to capture the code structure information, which is related to categories of *"how-to-use," "how-it-is-done,"* and *"property."* Therefore, we choose this clear way of comment classification.

*3.2.2 Procedure of Classifying Comments.* In total, three programmers who are familiar with Java participated in the open coding procedure (the first, the third authors of this article and a

Table 6. Statistical of the Original Classification

| Coder | What | Why | How-to-use | How-it-is-done | Property | Others |
|---|---|---|---|---|---|---|
| **Coder 1** | 3,926 | 3,281 | 10,292 | 2,025 | 370 | 106 |
| **Coder 2** | 4,486 | 2,799 | 6,200 | 5,964 | 400 | 151 |
| **Coder 3** | 4,421 | 3,036 | 10,006 | 2,090 | 287 | 160 |

Table 7. Statistical of the Classification After Clarification

| Coder | What | Why | How-to-use | How-it-is-done | Property | Others |
|---|---|---|---|---|---|---|
| **Coder 1** | 4,117 | 3,381 | 10,001 | 2,025 | 370 | 106 |
| **Coder 2** | 4,486 | 2,799 | 9,200 | 2,964 | 400 | 151 |
| **Coder 3** | 4,457 | 3,500 | 9,106 | 2,490 | 287 | 160 |

grad student with two, three, and two years of experience in Java, respectively). We manually classified the *validation data*, which consists of 20,000 ⟨*code*, *comment*⟩ pairs (see Section 2.3 for the description of the dataset splitting). Specifically, we used the open coding procedure [55] to classify and label the data. Open coding is widely used to generate categories or label data in Software Engineering [37, 54, 72]. There are three steps in our open coding procedure.

**Step 1. Individual classification.** The three coders read the data independently. For each pair, each coder read the comment and the source code to decide the category according to the definitions. Finally, the three coders classified 20,000 pairs of data. Table 6 shows the statistics of the results of the original classification.

**Step 2. Discuss and merge conflicts.** After **Step 1** (individual classification), the three coders worked together to make the final agreement. We merged conflicts by clarifying the scope boundaries among categories and clarifying the misunderstanding. For example, as shown in Table 6, the categories *"how-to-use"* and *"how-it-is-done"* produced most of the conflicts. The coders discussed these two kinds of comments, and they finally reached a consensus on the scope boundary of the two categories *"how-to-use"* and *"how-it-is-done."* For example, all the comments describing specific functional purposes like *"checks if the variable is built-in"* were classified into the category *"how-to-use."* By clarifying the scope boundary of all the categories, we reduced the number of conflicts as shown in Table 7. In addition to clarifying the scope boundaries of categories, the coders also merged the conflicts by clarifying the misunderstanding in the discussions. If there is no ultimate consensus after discussion, then we follow the majority voting. The coders participated in all the discussions and discussed all the conflicts. Each discussion lasted about 30 min.

**Step 3. Iterate and revisit.** In **Step 1** (merging conflicts) and **Step 2** (handling issues), after clarifying scope boundaries among categories or clarifying the misunderstanding, the three coders iterated the steps again and revisited the corresponding comments to determine the final category. In this way, we guaranteed that all the pairs of data are classified on the same criterion by three coders. We revisited once after several discussions and handled the conflicts. In total, we revisited the whole dataset two times following the procedure mentioned above and solved the conflicts. The process ended when we could not identify any new conflicts or issues. We encountered no situation in which the three coders insisted on three different labels so that majority voting could lead to the final results. Table 8 demonstrates the results after revisiting, and Table 9 shows the final results.

The labeling process is labour-intensive; one coder could label 57 code-comment pairs in an hour on average (reading source code, comment and determining category). We developed a tool

Table 8. Statistical of the Final Classification of Three Programmers

| Coder | What | Why | How-to-use | How-it-is-done | Property | Others |
|---|---|---|---|---|---|---|
| **Coder 1** | 4,164 | 2,434 | 10,164 | 2,883 | 273 | 82 |
| **Coder 2** | 4,344 | 2,349 | 10,088 | 2,862 | 279 | 78 |
| **Coder 3** | 4,260 | 2,380 | 10,276 | 2,726 | 282 | 76 |

Table 9. Counts of Different Categories in the Validation Data

| Category | Count | Proportion |
|---|---|---|
| What | 4,106 | 20.53% |
| Why | 2,493 | 12.47% |
| How-to-use | 10,190 | 50.95% |
| How-it-is-done | 2,828 | 14.14% |
| Property | 291 | 1.45% |
| Others | 92 | 0.46% |

to monitor the amount of time it took for this task; only when the coder is reading and labeling, the time can be counted. The time of discussing, merging conflicts, and revisiting is not included.

Last, we use the Fleiss Kappa value [13] to measure the agreement among the three coders. We calculate the Fleiss Kappa on the results of the final round. Fleiss Kappa values in range of [0.01, 0.20], (0.20, 0.40], (0.40, 0.60], (0.60, 0.80] and (0.80, 1] correspond to slight, fair, moderate, substantial, and almost perfect agreement, respectively. The overall Kappa value is 0.79, which indicates substantial agreement among the participants.

The final results of the manual classification are in Table 9. The distribution shows that the *"how-to-use"* comments correspond to almost half of the comments. The *"why"* comments take more than 10% of the comments and the *"property"* comments take only a small part.

## 3.3 Experiment Setup

In this article, we evaluate six models on different categories of comments. To keep the variable that all models on one same dataset, we exclude approaches that do not support Java. All the models are trained on the same *training data* and they are applied to each category, separately. In our experiments, CodeNN [24], DeepCom [21], and Code2Seq [3] are code summarization models. NNGen [33] is an intuitive and robust retrieval-based model. Two-layer bi-directional encoder-decoder Long Short-Term Memory (LSTM) (two-layer BiLSTM) [36] and Transformer [62] and are baselines. Our criterion for choosing the approaches is based on whether they are the state-of-the-art (e.g., DeepCom), they are classic code summarization models (e.g., CodeNN is the first to apply a neural network to code summarization), or they are based on a popular NMT framework (e.g., Transformer). The description and implementation details of the models are as follows.

Early research only uses lexical-level representations [2, 24]. In other words, they only treat source code as the sequence of tokens like in NLP.

**CodeNN.** CodeNN is the first to use neural networks in this area. It exploits an RNN with attention and directly distributed the comments' words to code tokens. We use the source code of CodeNN and adopt their preprocessing procedures.

Recent state-of-the-art research combines lexical-level and syntactic-level representations. Because the implementations of different levels of source code are different (especially the way to convert AST to sequence), we choose two models, Code2Seq [3] and DeepCom [21], which utilize both source code tokens and ASTs.

**Code2Seq.** We run Code2Seq using the source code provided by the authors. Code2Seq converts AST to sequence by sampling the path inside the tree structure [3]. In particular, the Code2Seq model aims to conduct "extreme code summarization," which is generating a method name instead of a sentence of natural language (i.e., summary). With the assistance of the authors, we modify the corresponding part of the model and replace the method name with the summary.
**DeepCom.** We use the source code of DeepCom provided by authors to run experiments. DeepCom converts AST to sequence by traversing the AST in a particular traversing algorithm [21]. In preprocessing, we use the same source code to follow the procedure of tokenization as well as parsing, extracting, and traversing ASTs with the same algorithm.
**NNGen.** NNGen is an information-retrieval-based method for generating commit messages for new diffs, which is also a mapping from code information to natural language similar to code summarization [32]. NNGen works in the same way with CloCom [68] and it has an intuitive implementation. NNGen uses the nearest neighbour algorithm to reuse proper existing comments according to the code.
**NMT Baselines.** Two baselines: Transformer and BiLSTM are proven to be strong baselines in generative tasks in NLP and software engineering, which read the input source code as a stream of tokens. Two-layer bi-directional encoder-decoder LSTMs with global attention [36] is classic neural networks and Transformer [62], which achieved state-of-the-art results for translation tasks.
**Hyperparameter setting.** As described above, we use the source code provided by authors and keep the original hyperparameters (e.g., model structures and training strategies) to reproduce experiments. Specifically, for CodeNN, DeepCom, and NNGen, we follow the original preprocessing methods and the original experimental settings. For Code2Seq, we use a modified preprocessing method with the assistance of authors to perform code summarization task and follow the original experimental settings. For the NMT baselines, we follow the default experimental settings of the OpenNMT.

Our computing devices are 4 Nvidia 2080ti GPU (48 G memory in total) and Intel Xeon Gold 6226 CPU with 12 cores. For all approaches, we use the same *training data* to build the models with a similar early stopping strategy. We use the trained models to evaluate different categories, separately.

### 3.4 Evaluation Metric

We use NMT metrics of *BLEU* (1 through 4) [48] and *ROUGE* [30] to evaluate the model, which are widely used in translation and code summarization tasks [22, 24, 62, 63]. We introduce the metrics as follows.
**BLEU**, the abbreviation for BiLingual Evaluation Understudy [48], is widely used for evaluating the quality of summaries in Software Engineers [22, 24, 62, 63]. It is a variant of precision metric, which calculates the similarity by computing the n-gram precision of a candidate sentence to the reference sentence, with a penalty for the overly short length [48]. It is computed as

$$BLEU = BP * exp\left(\sum_{n=1}^{N} w_n log p_n\right), \tag{1}$$

where $N = 1, 2, 3, 4$ and $w_n = \frac{1}{N}$. $BP$ represents the brevity penalty, which penalize short sentences as

$$BP = \begin{cases} 1, & \text{if } cand > ref, \\ e^{1-\frac{r}{c}}, & \text{if } cand \leq ref, \end{cases} \tag{2}$$

where *cand* and *ref* represent the lengths of candidate sentences and reference sentences, respectively. In particular, we often pay attention to *BLEU-4* as it can reflect the weighted results from 1 through 4.

**ROUGE**, the abbreviation for Recall-oriented Understudy for Gisting Evaluation [30], is widely used to evaluate summarization tasks [34, 63]. The precision, recall, and F1 score of ROUGE are calculated as follows:

$$R = \frac{\sum_{(cand, ref) \in S} \sum_{gram_n \in ref} Count_{cand}(gram_n)}{\sum_{(cand, ref) \in S} \sum_{gram_n \in cand} Count_{ref}(gram_n)}, \tag{3}$$

$$P = \frac{\sum_{(cand, ref) \in S} \sum_{gram_n \in ref} Count_{cand}(gram_n)}{\sum_{(cand, ref) \in S} \sum_{gram_n \in cand} Count_{cand}(gram_n)}, \tag{4}$$

$$F1 = \frac{2R * P}{R + P}, \tag{5}$$

where *cand*, *ref*, and *S* refer to a generated candidate comment, its reference description and the test set. $gram_n$ is a n-gram phrase, and $Count_{cand}(gram_n)$ and $Count_{ref}(gramn_n)$ refer to the count number of $gram_n$ in *cand* and *ref*, respectively. In particular,

$$\sum_{gram_n \in cand} Count_{cand}(gram_n) \tag{6}$$

refers to the matched n-gram phrases in both *ref* and *cand*. Intuitively, $R_{rouge-n}$ measures the percentage of the n-grams in the reference sentence that a candidate sentence can match, $P_{rouge-n}$ represents the percentage of "correct" n-grams (i.e., matched n-gram in reference sentence) in a candidate sentence. $F1_{rouge-n}$ is a harmonic mean of both precision and recall. The precision, recall, and F1 score of *ROUGE-L* are calculated in the same way, but they use the longest common subsequences between candidate and reference sentences [30]. We report F1 scores of *ROUGE*, which balance precision and recall.

We calculated all of the NMT metrics for code summarization by the package provided by Sharma et al. [56].

## 3.5 Results and Analysis

**RQ1 How do different comment categories impact the code summarization performance?** The results of the experiments are in Table 10. To the best of our knowledge, our selected approaches are classic (CodeNN is the first to apply deep-learning generation technique to code summarization) or the state-of-the-art (e.g., DeepCom). The state-of-the-art performance in terms of *ROUGE* and *BLEU* is about 30%, which is achieved by the experiments that we have reproduced in this article. Our observations are as follows:

**For each code summarization approach, there is a significant performance difference across the categories.** The up arrow (↗) in the table indicates that in this category, the model statistically significantly outperforms the other models. We use *compare-mt* [46], a tool for comparison of language generation tasks, to perform statistical significance tests in terms of *ROUGE* and *BLEU*. Therefore, such differences suggest that comment classification (i.e., treat comments differently) can potentially influence code summarization performance.

**Different code summarization approaches perform the best for different categories.** The results show that DeepCom, CodeNN, and NNGen perform the best for *"what"* comments, Code2Seq performs the best for *"how-to-use"* comments, Transformer and two-layer BiLSTM perform the best for *"how-it-is-done"* comments. No models perform the best for *"why"* and *"property"* comments among the six categories.

A possible explanation of such differences is that code is always highly-structured. For example, the *"how-it-is-done"* comments of the process often correspond with the source code. They do not involve knowledge above method granularity so that the model builds a relatively tight relationship between source code and natural language. Code2Seq and DeepCom represent code at the

Table 10. Performances of the Code Summarization Models in Each Comments Category

| **Approach** | **Category** | **ROUGE-L (%)** | **BLEU-1 (%)** | **BLEU-2 (%)** | **BLEU-3 (%)** | **BLEU-4 (%)** |
|---|---|---|---|---|---|---|
| CodeNN | What | 14.36% | 13.64% | 3.68% | 1.54% | 0.90% |
| | Why | 6.52% | 6.37% | 1.31% | 0.42% | 0.19% |
| | How-to-use | 8.62% | 8.98% | 2.23% | 1.01% | 0.63% |
| | How-it-is-done | 9.21% | 8.08% | 2.38% | 0.91% | 0.45% |
| | Property | 13.34% | 13.17% | 4.13% | 1.69% | 0.00% |
| | Others | 7.01% | 7.26% | 1.66% | 0.00% | 0.00% |
| | **All** | 9.72% | 9.33% | 2.44% | 1.01% | 0.58% |
| Code2Seq | What | 30.31% | 31.66% | 21.68% | 17.23% | 14.70% |
| | Why | 26.71% | 24.28% | 15.70% | 11.83% | 9.91% |
| | How-to-use ↗ | 34.30% | 36.76% | 26.79% | 21.85% | 19.14% |
| | How-it-is-done | 30.78% | 30.14% | 21.12% | 16.98% | 14.80% |
| | Property | 29.71% | 33.36% | 23.91% | 19.46% | 17.22% |
| | Others | 25.36% | 25.48% | 17.28% | 14.28% | 12.82% |
| | **All** | 31.60% | 32.25% | 22.76% | 18.26% | 15.84% |
| DeepCom | What ↗ | 36.59% | 34.51% | 28.26% | 24.30% | 21.44% |
| | Why | 27.48% | 26.47% | 20.60% | 18.13% | 16.89% |
| | How-to-use | 33.28% | 33.83% | 26.73% | 23.42% | 21.58% |
| | How-it-is-done ↗ | 33.99% | 32.51% | 26.61% | 23.99% | 22.63% |
| | Property ↗ | 30.89% | 31.66% | 25.46% | 22.07% | 19.89% |
| | Others | 27.38% | 29.09% | 22.79% | 20.22% | 18.62% |
| | **All** | 33.10% | 32.42% | 25.94% | 22.81% | 21.01% |
| NNGen | What | 35.55% | 34.87% | 26.26% | 23.06% | 21.33% |
| | Why ↗ | 29.65% | 28.33% | 22.09% | 20.34% | 19.79% |
| | How-to-use | 32.52% | 33.16% | 25.46% | 21.10% | 18.83% |
| | How-it-is-done | 33.39% | 32.25% | 27.34% | 22.42% | 21.53% |
| | Property | 30.49% | 28.39% | 21.29% | 18.50% | 16.68% |
| | Others ↗ | 32.45% | 32.45% | 25.93% | 23.49% | 21.37% |
| | **All** | 34.04% | 33.75% | 24.98% | 21.87% | 21.07% |
| Transformer | What | 19.06% | 11.81% | 7.21% | 5.27% | 4.23% |
| | Why | 20.36% | 14.31% | 8.84% | 6.70% | 5.53% |
| | How-to-use | 20.59% | 13.03% | 8.22% | 6.24% | 5.13% |
| | How-it-is-done | 23.14% | 15.86% | 10.56% | 8.26% | 6.98% |
| | Property | 18.55% | 11.70% | 6.84% | 4.87% | 3.91% |
| | Others | 18.50% | 11.96% | 7.84% | 5.84% | 4.84% |
| | **All** | 20.61% | 13.42% | 8.48% | 6.43% | 5.30% |
| 2-Layer BiLSTM | What | 15.99% | 9.75% | 4.91% | 2.96% | 1.99% |
| | Why | 17.27% | 12.13% | 6.37% | 4.02% | 2.87% |
| | How-to-use | 18.44% | 11.60% | 6.40% | 4.23% | 3.09% |
| | How-it-is-done | 18.85% | 12.91% | 7.22% | 4.81% | 3.60% |
| | Property | 15.13% | 9.28% | 4.47% | 2.46% | 1.60% |
| | Others | 13.10% | 8.44% | 4.19% | 2.18% | 1.28% |
| | **All** | 17.58% | 11.33% | 6.11% | 3.95% | 2.86% |

syntactic level (i.e., utilize AST), which is a better use of the structural features of source code while NLP techniques do not consider code structure information. How AST is used in DeepCom and Code2Seq are different: Code2Seq utilizing AST by sampling the paths in AST while DeepCom converts the whole AST into sequences. This mechanism differences lead to different preferences. In addition, a possible explanation of the performance of *"property"* comments is that *"property"* comments aim to describe part of the source code like particular parameters or variables. Code summarization trains the model to maximizes the possibility of the next word based on the prior condition (e.g., the preceding word). However, the decision of which part is valuable to comment involves a more complicated process, which is more like sorting and recommending problems. Hence the results in this category are not ideal.

**Utilizing code information is essential, but not enough for all kinds of comments.** In particular, all models have a poor performance in the *"why"* category. The *"why"* comments often involve complicated business logic or background knowledge. As the granularity is method level, this kind of knowledge is beyond the source code. Why the code is designed (i.e., design rationale) is often beyond implementation details of the code. Therefore, the model cannot reach excellent performance in this category. In our experiments, NNGen, a retrieval-based model, has a relatively better performance than others, which indicates that for *"why"* comments, external information beyond the target code itself (i.e., other similar methods and their comments) also plays a critical role. For example, NNGen retrieves a comment *"This needs to be called before anything else, because we need the media factory."* It explains the design rationale (i.e., factory pattern in Java), which is hard to be inferred from the source code.

**Summary**

We classify comments into six categories and conduct an experiment on each category, separately. We find that for each code summarization approach, there is a significant performance difference across the categories.Different code summarization approaches perform the best for different categories. In particular, no models perform the best for *"why"* and *"property"* comments among the six categories.

## 4 COMPOSING COMMENT CLASSIFICATION INTO CODE SUMMARIZATION

### 4.1 Motivation and Insight

**RQ2 How can we improve the code summarization performance using the comment categories?** As described in Table 10, we find that different code summarization methods perform best for different categories, which means that the ability of the models to extract information from source code is different. Such differences come from the mechanism or implementation of different models. Therefore, it could be beneficial to combine the advantages of models that perform best in different comment categories. Our goal is to demonstrate that comment classification could boost code summarization to reach better results.

**Insight.** According to our previous findings, for each category, there exists an approach that performs the best. Besides, there is no one model that dominates over all the categories, which means we can design a composite approach to take advantage of different approaches. Specifically, according to our experiments, Code2Seq works the best for *"how-to-use"* comments; DeepCom works the best for *"what," "how-it-is-done,"* and *"property"* comments; NNGen works the best for *"why"* and *"others"* comments. We can cover all the categories by using the most suitable approach to perform code summarization on each category. In this sense, we combine different code summarization models, and we call it a composite approach. Such a composite approach can outperform baselines in the validation dataset. However, it cannot prove that such a composite approach can boost the results only on the validation dataset, because the obtained prior knowledge of which approach performs the best on a particular category may not work in a new dataset. Therefore, the composite approach should be evaluated on a new dataset (i.e., testing dataset) on which we do not know such prior knowledge.

However, if applied to a new dataset, then the composite approach is agnostic to the category of a new snippet of the source code. Since we have manually labeled 20,000 code-comment pairs, the labeled dataset constructs a relationship between the source code and the comment category. We can leverage such a dataset to build the classifier. The task can be regarded as a kind of source code classification task, which is studied in software engineering [7, 35, 60, 61]. A prior study shows that source code corpora have similar statistical properties to natural language corpora [1], indicating the consistency between the source code and comment. Besides, Louis et al. use the source code and the corresponding labels to train a classifier and predict whether a developer would write comments given a piece of code [35]; Chen et al. succeed in using code classification to boost code summarization [7]. These studies show it is feasible to predict category with the assistance of our labeled dataset. We call this task "code comment category prediction."

Note that the task is not the end goal of our composite approach. The classifier plays the role of the selector, which selects suitable code summarization models according to the categories and combines the results. Considering that a snippet of code may have different functionality in different programs, which may influence our findings, we count the duplicated source code in the dataset. There is only 1.66% duplicated source code, and only 0.23% are the cases that have the same source code with different comments. In this way, we can utilize the results of the comment classification and compose it into the code summarization.

## 4.2 Comment Category Prediction by Source Code

*4.2.1 Comment Category Prediction.* This task leverages the results of the comment classification to build classifiers and predict the comment category given a piece of source code. In Section 3, we manually classify and label the *validation data* consisting of 20,000 code-comment pairs. Leveraging these labeled data, we build classifiers to conduct automatic classification. We train several classifiers leveraging source code (split tokens), and the corresponding labels. After building classifiers, we evaluate each classifier by 10-fold stratified cross-validation and compare it with each other. 10-fold cross-validation splits the dataset into ten consecutive folds (i.e., 2,000 pairs in each fold); each fold is then used once for evaluation while the nine remaining folds form the training set (i.e., 18,000 pairs).

After evaluating classifiers, the whole labeled dataset is used for training a classifier. Then, we apply the classifier to the *testing data* to get the categories for each code snippet.

*4.2.2 Experimental Settings.* In this article, we investigate five techniques,[3] i.e., Random Forest [5], LightGBM [27], Decision Tree [6], multinomial Naive Bayes [38], and a deep-neural-network-based classifier. These techniques are proven to be effective in comment classification in software engineering [49, 50, 72].

For Random Forest, LightGBM, Decision Tree, and multinomial naive Bayes, we consider not only text features of the source code but also syntactic features of the source code. For text features, after tokenization of the source code, we extract bigram term-frequency of the tokens, which could extract tens of text features for one snippet of the source code. Such a technique is

[3]We implement Random Forest, Decision Tree, and multinomial Naive Bayes using sci-kit learn toolkit [51]. We implement LightGBM using the official library [27]. We implement the deep-neural-network classifier using pytorch https://pytorch.org/.

a conventional feature extraction approach for the source code that is widely used in the in the software engineering literature [31, 60, 61]. For syntactic features, we extract lines of code (LOC), token numbers, and variable numbers of the source code as statistical features; we extract method name and method parameters of the source code as identifier features. These features are also used in prior software engineering literature [12, 15, 60].

We construct an LSTM (Bi-directional) network model for classification, which is used for source code classification [35]. We numericalize the token sequences of the source code (i.e., map the token sequences to numerical indexes). Then, we use an embedding layer to embed each token and concatenate them as the input sequence of the model. For each sequence, the BiLSTM network accepts the embeddings of the tokens and inputs them into a layer with two concatenated LSTM (i.e., Bi-directional LSTM). The layer utilizes the last state of the time sequences of such layers. Then the model inputs the tensors to a fully-connected layer and goes through a softmax layer to get the output labels.

We train the models with 50 epochs and the model randomly selects 5% data for in-training-validation. We adopt an early stopping strategy that if the in-training-validation stops increasing for three epochs, the model will stop training.

*4.2.3 Metrics for Evaluating Classifiers.* To evaluate the effectiveness of our automatic technique to classification code comments, we use well-known Information-retrieval metrics for the quality of results, called *Precision*, *Recall*, and *F1 score*:

$$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \tag{7}$$

$$Recall = \frac{TP}{TP + FN}, \tag{8}$$

$$F1 = 2 \times \frac{Precision \cdot Recall}{Precision + Recall}, \tag{9}$$

where True Positive (TP) represents the number of the comments that are correctly classified to the according category; False Positive (FP) represents the number of the comments that are wrongly classified to this category; False Negative (FN) represents the number of the comments that are wrongly classified into other categories. Here the *Precision*, *Recall*, and *F1 score* are for certain categories. We evaluate the classifier using the weighted results (i.e., calculate metrics for each category, and find their weighted average by the number of each category).

**Evaluation of multiple classification tasks.** In this article, the classification task is a multi-class classification as there are six categories (i.e., *"what," "why," "how-to-use," "how-it-is-done," "property,"* and *"others"*). Considering the supports (i.e., the numbers of different categories) are different, we calculate the weighted results of *precision*, *recall*, and *F1* scores. Formally, we calculate them as

$$Precision_{weighted} = \sum_{i=1}^{n} \frac{Precision_{c_i}}{N_{c_i}}, \tag{10}$$

$$Recall_{weighted} = \sum_{i=1}^{n} \frac{Recall_{c_i}}{N_{c_i}}, \tag{11}$$

$$F1_{weighted} = \sum_{i=1}^{n} \frac{F1_{c_i}}{N_{c_i}}, \tag{12}$$

where $c_i$ denotes the $i$th category, $N_{c_i}$ denotes the number of the $i$th category, $n$ denotes the category numbers where $n = 6$ representing the six categories, and $Precision_{c_i}$, $Recall_{c_i}$, and $F1_{c_i}$

denotes the $Precision$, $Recall$, and $F1$ scores of the $i$th category, separately. In this article, we use cross-validation to evaluate the classifiers and report the average results of $Precision_{weighted}$, $Recall_{weighted}$, and $F1_{weighted}$ of all folds.

We report the standard deviation $s$ of the experiments. There are two kinds of standard deviations. For the cross-validation, we report the standard deviation of all the folds:

$$s = \sqrt{\frac{1}{N_f - 1} \sum_{i=1}^{N_f} (x_i - \overline{x})^2}, \tag{13}$$

where $N_f$ denotes the number of folds, $x$ denotes one of $Precision_{weighted}$, $Recall_{weighted}$, or $F1_{weighted}$ scores. $x_i$ denotes the results of the $i$th fold, and $\overline{x}$ denotes the average score of all folds.

For the evaluation of our approach and the baselines, considering the randomness in the learning-based approaches, we run the experiments ten times and report the standard deviation:

$$s = \sqrt{\frac{1}{N_e - 1} \sum_{i=1}^{N_e} (x_i - \overline{x})^2}, \tag{14}$$

where $N_e$ denotes the number of experiments, $x$ denotes one of *ROUGE* or *BLEU* scores. $x_i$ denotes the results of the $i$th time of the experiment, and $\overline{x}$ denotes the average scores of all experiments.

## 4.3 Composite Approach with Comment Category Prediction

According to the results in Table 10, the up arrow (↗) in the table indicates that the model outperforms the others in this category. Based on our observations and insights mentioned above, we choose Code2Seq, DeepCom, and NNGen as basic models. For example, Code2Seq has the best performance in *"how-to-use"* comments, so when the classifier determines the *"how-to-use"* category given the source code, we use Code2Seq to conduct code summarization. The composite approach is proposed in the same way. Then the approach combines all the generated comments and outputs the ultimate results.

However, note that we cannot evaluate the approach in the same *validation data*, which is used for validating code summarization models, because we have obtained the prior knowledge of "which models perform best in a particular category" in this *validation data*. Recall that the *testing data* is unlabeled and has no overlap with the *validation data* and *training data* (cf. Section 2.3). We evaluate our composite approach in the *testing data*. Last, we calculate the evaluation metric according to the generated comments and reference comments.

The three approaches without the assistance of the classifier are regarded as baselines. We evaluate each model on the same *testing data* and get the output, separately.

## 4.4 Experimental Results

Table 11 shows the results of the classifiers. In several candidates, the best classifier is Random Forest, which achieves 76.91% in 10-fold cross-validation. It outperforms other classifiers and is used to be a selector in our composite approach. To the best of our knowledge, there are no other approaches perform comment category prediction by source code. The subsequent experimental results in Table 13 show that Random Forest is useful in classifying the comments automatically.

Table 12 shows the results of the six categories of different classifiers (mean ± standard deviation). We can observe that generally, all classifiers perform the best for categories of *"what"* and *"how-to-use."* The performance of *"property"* and *"others"* is relatively low, and their standard

Table 11. The Weighted Precision, Recall, and F1 Scores of Classifiers

| Classifier | Precision | Recall | F1 |
|---|---|---|---|
| Random Forest | 78.49% ± 0.64% | 78.04% ± 0.52% | **76.91% ± 0.56%** |
| LigthGBM | 74.14% ± 1.16% | 74.53% ± 1.17% | 74.19% ± 1.12% |
| Decision Tree | 73.45% ± 0.84% | 73.78% ± 0.84% | 72.40% ± 0.88% |
| Naïve Bayes | 69.67% ± 1.22% | 57.31% ± 0.39% | 46.73% ± 0.61% |
| BiLSTM | 73.81% ± 1.15% | 74.19% ± 0.99% | 73.37% ± 1.01% |

deviations are very high, because the numbers of pairs belonging to the two categories are relatively small, which affects the stability of the results.

Table 13 shows the results of our composite approach on the *testing data*. Because of the differences of the *testing data* and *validation data*, the performances of the three baselines are slightly different in which Code2Seq performs better in the *testing data* while DeepCom performs better in the *validation data*. NNGen has relatively higher *BLEU-3* and *BLEU-4*, because as a retrieval-based approach, the results are more prone to match exactly once the sentences are retrieved.

Our composite approach outperforms current approaches in the *testing data*. Specifically, the approach has a relative improvement of 8.57% (2.76 absolute improvements) in *ROUGE-L* and a relative improvement of 16.34% (2.80 absolute improvement) in *BLEU-4* score. We use *compare-mt* [46], a tool for comparison of language generation tasks, to perform statistical significance tests in terms of *ROUGE* and *BLEU*. The results show that the difference between our approach and the baselines is significant. Moreover, our improvement is more than the improvement of Code2Seq over DeepCom in terms of *ROUGE* and NNGen over Code2Seq in terms of *BLEU-4*. It indicates that the classifiers succeed in incorporating classification information. Our experiments demonstrate that comment classification can boost code summarization to reach better results.

**Influence of the classifiers.** Though we use the Random Forest as the final selector, we report the results of our composite approach with different classifiers. As in Table 14, the results of applying the LightGBM, Decision Tree, Naive Bayes, and the DNN-based classifiers are lower than the results of applying Random Forest. In terms of *ROUGE-L*, our composite approach with all classifiers outperforms the baselines and in terms of *BLEU-4*, our approach with classifiers of Random Forest, LightGBM, and BiLSTM outperforms the baselines. We can observe that the performance of the classifiers in terms of *F1 scores* are approximately positively related to the results of the composite approach (i.e., approximately, the higher the *F1 score* is, the better our approach is, except for the LightGBM and BiLSTM). If we could have a clear understanding of intention categories of comment for the source code, then code summarization would have a better performance.

**Performance on the external dataset with different data distribution.** Table 15 shows the results of our composite approach and baselines on the *external data*. Because the data distributions of the *external data* are different, the models have relatively lower scores. Still, our model outperforms the baselines in such an *external dataset* with different data distributions from the *training data* and *testing data*.

## 4.5 Qualitative Analysis

Only reporting *ROUGE* and *BLEU* scores does not give an intuition of the impact of categories, and it leaves an open question of how our approach works. We use two examples for illustrative purposes. While we are hesitant to overinterpret the selected examples, we observe that these are consistent with many others. The first example is to show the impact of categories. We choose DeepCom and select referenced and generated comments to illustrate different performance in categories of *"what," "why," "how-to-use," "how-it-is-done,"* and *"property"* as in Figure 2 (the five

Table 12. The Precision, Recall, F1 Scores of the Six Categories of the Classifiers

| **Classifier** | **Category** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| Random Forest | What | 80.2% ± 1.83% | 78.70% ± 1.64% | 81.81% ± 1.29% |
| | Why | 72.13% ± 2.13% | 69.28% ± 1.33% | 75.14% ± 1.30% |
| | How-to-use | 78.04% ± 0.87% | 92.73% ± 0.90% | 84.75% ± 0.50% |
| | How-it-is-done | 80.13% ± 2.56% | 51.92% ± 2.40% | 62.96% ± 1.90% |
| | Property | 71.7% ± 12.28% | 43.81% ± 8.06% | 56.71% ± 8.82% |
| | Others | 68.33% ± 31.13% | 21.53% ± 10.42% | 31.30% ± 13.59% |
| LightGBM | What | 80.01% ± 1.45% | 74.21% ± 1.70% | 77.00% ± 1.48% |
| | Why | 75.64% ± 2.24% | 67.22% ± 3.97% | 71.15% ± 3.06% |
| | How-to-use | 76.56% ± 0.96% | 90.06% ± 0.71% | 82.76% ± 0.80% |
| | How-it-is-done | 70.91% ± 2.91% | 46.56% ± 3.55% | 56.13% ± 2.94% |
| | Property | 84.90% ± 10.78% | 34.34% ± 7.23% | 48.32% ± 7.80% |
| | Others | 64.52% ± 31.87% | 29.28% ± 11.23% | 29.28% ± 11.23% |
| Decision Tree | What | 77.70% ± 1.89% | 76.91% ± 2.29% | 77.28% ± 1.49% |
| | Why | 70.63% ± 3.22% | 71.73% ± 1.84% | 71.15% ± 2.28% |
| | How-to-use | 80.31% ± 0.89% | 81.52% ± 1.24% | 80.91% ± 0.89% |
| | How-it-is-done | 60.06% ± 3.17% | 58.98% ± 3.56% | 59.46% ± 2.90% |
| | Property | 63.72% ± 4.89% | 46.73% ± 8.16% | 53.60% ± 6.22% |
| | Others | 50.14% ± 31.47% | 29.02% ± 20.34% | 35.20% ± 22.26% |
| Naïve Bayes | What | 78.29% ± 1.67% | 26.33% ± 2.06% | 40.51% ± 2.65% |
| | Why | 61.42% ± 6.10% | 37.28% ± 1.34% | 33.46% ± 2.31% |
| | How-to-use | 68.07% ± 0.28% | 81.82% ± 0.18% | 55.24% ± 0.24% |
| | How-it-is-done | 72.39% ± 6.93% | 34.01% ± 0.80% | 37.67% ± 1.47% |
| | Property | 53.22% ± 0.69% | 38.76% ± 7.27% | 39.62% ± 7.97% |
| | Others | 54.33% ± 27.72% | 42.65% ± 13.23% | 41.25% ± 23.33% |
| BiLSTM | What | 76.57% ± 2.42% | 69.43% ± 1.92% | 72.80% ± 1.55% |
| | Why | 75.31% ± 2.28% | 64.90% ± 1.71% | 69.68% ± 1.15% |
| | How-to-use | 75.17% ± 0.92% | 86.90% ± 1.00% | 80.61% ± 0.80% |
| | How-it-is-done | 64.08% ± 3.05% | 49.65% ± 1.81% | 55.92% ± 1.93% |
| | Property | 76.36% ± 16.15% | 35.05% ± 5.89% | 47.69% ± 7.65% |
| | Others | 50.00% ± 44.72% | 8.67% ± 7.98% | 14.23% ± 12.51% |

Table 13. Experimental Results of the Composite Approach and Baselines on the *Testing Data*

| **Approach** | **ROUGE-L** | **BLEU-1** | **BLEU-2** | **BLEU-3** | **BLEU-4** |
|---|---|---|---|---|---|
| DeepCom | 32.16% ± 1.77% | 31.91% ± 2.77% | 20.79% ± 1.84% | 17.35% ± 0.17% | 16.44% ± 0.10% |
| Code2Seq | **32.22% ± 0.89%** | 30.99% ± 0.10% | 24.11% ± 0.23% | 17.76% ± 0.51% | 16.07% ± 0.86% |
| NNgen | 30.57% | 29.72% | 24.56% | 20.32% | **17.14%** |
| **Ours** | **34.98% ± 2.09%** | 32.66% ± 2.41% | 25.76% ± 0.12% | 21.58% ± 0.17% | **19.94% ± 0.22%** |

NNGen is a retrieval-based approach so there is no standard deviation.

cases are from the top to bottom, respectively). In the first case, the approach identifies the role of the method, but it fails to summarize it is a "helper" function. The technical reason is that the word "helper" lacks context, which leads to a low probability of the model to generate the word "helper." The second case aims to describe an intention using the word "reachability." But the approach fails to summarize it, because the word is rare and lacks context. Instead, it summarizes

Table 14. The Results of the Composite Approach with Different Classifiers

| Classifier | ROUGE-L | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 |
|---|---|---|---|---|---|
| Random Forest | **34.98% ± 2.09%** | 32.66% ± 2.41% | 25.76% ± 0.12% | 21.58% ± 0.17% | **19.94% ± 0.22%** |
| LigthGBM | 32.93% ± 1.88% | 32.59% ± 0.78% | 22.89% ± 2.25% | 18.63% ± 1.57% | 16.12% ± 0.21% |
| Decision Tree | 32.75% ± 2.22% | 31.87% ± 2.61% | 22.58% ± 0.37% | 18.22% ± 1.40% | 15.77% ± 0.34% |
| Naïve Bayes | 32.60% ± 1.99% | 31.68% ± 2.55% | 22.37% ± 0.11% | 18.43% ± 0.86% | 14.83% ± 0.47% |
| BiLSTM | 33.20% ± 0.10% | 32.24% ± 2.19% | 23.45% ± 1.33% | 19.92% ± 0.12% | 17.99% ± 0.17% |

Table 15. Experimental Results on the External Dataset with Different Data Distribution

| Approach | ROUGE-L | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 |
|---|---|---|---|---|---|
| DeepCom | 26.73% ± 1.57% | 25.12% ± 2.17% | 19.57% ± 1.21% | 16.57% ± 0.32% | 15.63% ± 0.24% |
| Code2Seq | **27.46% ± 0.94%** | 27.43% ± 0.21% | 23.29% ± 0.19% | 16.63% ± 0.52% | 15.96% ± 0.66% |
| NNGen | 27.29% | 27.18% | 22.97% | 19.55% | **16.48%** |
| Ours | **29.75% ± 1.64%** | 29.12% ± 1.65% | 24.74% ± 0.74% | 20.46% ± 0.21% | **18.34% ± 0.34%** |

the functionality as the input is solely the source code. The third case emphasizes the change of the variable and the approach is successful to summarize *"how-to-use"* even though the performance is not high in terms of NMT metrics, which evaluates verbatim results. The fourth case comments how the method deletes the video, and the approach succeeds in capturing the key information (i.e., id). The fifth case explains the arguments of the method, while the approach only catches the type of the variable (i.e., array of the byte). As mentioned in Section 3.3, DeepCom inputs the AST sequence and token sequence of source code, and many current approaches follow this framework. On the one hand, the example shows a limitation of such a framework that current input is not sufficient for summarizing external information outside the source code (e.g., the first case cannot summarize information "helper" and the second case cannot summarize the intention). On the other hand, the example illustrates how the approach is more intended to retrieve the information to summarize the functionality with only source code.

The second example is to show how our composite works and show its procedure by observing the AST structures. As in Figure 3, the source code is first categorized to *"why"* comment, because our classifier catches a similar code pattern in the labeled *validation data*. After deciding the category, recall that the NNGen works best in the *"why"* category, the composite model uses NNGen to generate comments. To compare with models that are not selected, we also call DeepCom and Code2Seq to generate comments. According to human evaluation, DeepCom successes to describe the functionality but in a complicated way. As shown in Figure 3, it captures the underlined keywords and outputs them in a readable way. However, the way it describes is complicated, which is derived from its AST sequence. DeepCom converts the whole AST into the sequence and outputs the numerical vectors. The comment reflects the complexity of the AST sequence (121 recursive structure, which is very large for a 5-statements method), but such complexity of sentence also leads to a low score. As for Code2Seq, it fails to describe the functionality in this case. As shown in Figure 3, Code2Seq samples paths of AST to construct paths, but in this case, it fails to capture the critical node "double," which leads to the bias of the results. Therefore, the classifier selects a better model in this comment category and avoids the disadvantages of too complicated or biased models. This case illustrates how results of comment classification could help the composite approach in a micro-granularity.

```
private AnimatorSet buildScaleDownAnimation(
    View target, float targetScaleX ,float targetScaleY) {
    int screenWidth = getScreenWidth();
    int pivotX = (int)(screenWidth * NUM_);
    int pivotY = (int)(getScreenHeight() * NUM_);
    ViewHelper.setPivotX(target, pivotX);
    ViewHelper.setPivotY(target, pivotY);
    AnimatorSet scaleDown = new AnimatorSet() ;
    scaleDown.playTogether(
        ObjectAnimator.ofFloat(target ,STR_ ,targetScaleX), ObjectAnimator.ofFloat(target, STR_, targetScale
        Y), ObjectAnimator.ofFloat(target, STR_, screenWidth * NUM_));
    scaleDown.setInterpolator(
        AnimationUtils.loadInterpolator(activity, android.R.anim.decelerate_interpolator));
    scaleDown.setDuration(NUM_);
    return scaleDown;
}
```

**Reference: A helper method to build scale down animation.**

**Generated: Internal method to build scale down animation.**

```
public boolean ping(HostAddress address, int timeout) throws HostException {
    if (exception) {
        notes = STR_;
        throw new HostException(STR_);
    } else {
        notes = STR_;
        return BOOL_;
    }
}
```

**Reference: Sample test for reachability.**

**Generated: Checks if the address has been idle for the given address.**

```
public void increaseTextPrintPosition (double inc) {
    textPrintPosition += inc;
}
```

**Reference: Called whenever the text print position should be increased.**

**Generated: Called when the entire value is changed.**

```
public boolean delete (String videoId) {
    return dbHelper.getWritableDatabase().delete(
        tableName, YouTubeVideoEntry.COLUMN_VIDEO_ID + STR_ + videoId + STR_, null
        ) > NUM_;
}
```

**Reference: Deletes video entry with provided id.**

**Generated: Delete the video from a videoid.**

```
public static byte [] serializeToByteArray (Object value) {
    try {
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        try ( ObjectOutputStream oos = new ObjectOutputStream (buffer)) {
            oos.writeObject(value);
        }
        return buffer.toByteArray();
    } catch (IOException exn) {
    throw new IllegalArgumentException(STR_ + value, exn);
    }
}
```

**Reference: Serializes the argument into an array of bytes and returns it.**

**Generated: Serializes the byte array to an array of bytes.**

Fig. 2. Example of the source code, reference, and the generated comments in categories of *"what," "why," "how-to-use," "how-it-is-done,"* and *"property."*

**Summary**

We demonstrate that comment classification could boost code summarization to reach better results. Our composite approach outperforms other approaches, and obtains a relative improvement of 8.57% (2.76 absolute improvements) in *ROUGE-L* and a relative improvement of 16.34% (2.80 absolute improvement) in *BLEU-4* score.

```
public static boolean isDouble (CharSequence self) {
    try {
        Double.valueOf(self.toString().trim());
        return BOOL_;
    }
    catch (NumberFormatException nfe) {
        return BOOL_;
    }
}
```

**Reference:** Determine if a charsequence can be parsed as a double.

**NNGen:** Checks if a string can be parsed as double.

**DeepCom:** Method returns true if the string can be either double of the sign or defined output text.

**Code2Seq:** Returns true if the specified character sequence is a valid string.

**DeepCom AST Sequence (121 recursive structure in total):**

```
......
((type=ReferenceType(... name=CharSequence ...), varargs=False)
( ... qualifier=Double ...),
( ... member=toString ... ),)
......
MemberReference(member=BOOL_, ...)
(finally_block=None, label=None, resources=None) )
......
```

**Code2Seq AST Paths (130 in total):**

```
1 boolean,Prim0|Mth|Nm1,METHOD_NAME
2 boolean,Prim0|Mth|Prm|VDID0,enclosing
......
128 equals,Nm3|Cal|If|Ret|Nm0,bool
129 equals,Nm3|Cal|If|Bk|For|Bk|Ret|Nm0,bool
130 bool,Nm0|Ret|If|Bk|For|Bk|Ret|Nm0,bool
```

Fig. 3. Example of the method, the generated comments and the corresponding AST structure.

## 5 DISCUSSION

### 5.1 Implication

*5.1.1 Implications for Practice.* **It is hard to find a general way to solve the automatic code summarization problem.** The code summarization problem in practice involves many aspects. In this article, we limit the problem to six categories of comments and evaluate how current models perform. However, even in such a simplified situation, our experimental results show that there are no models that can perform well in every category and dominate every other model. Therefore, it is more feasible to build a model for a specific domain than for a general-purpose. Though the contribution of a single simple case is limited, a combination of such situations can play a vital role in assistance (e.g., API hint technique is simple but helps developers a lot [45]).

In this article, we exploit the impact of comment categories. To exploit the results in practice, developers can leverage the code summarization with comment category prediction to better identify intentions in the developing or maintaining phase. In this way, dividing comments into particular categories can improve program comprehension and help maintain the source code better. Moreover, with the awareness of comment categories, practitioners can apply the emerging code summarization techniques to more appropriate scenarios (e.g., it is more reasonable to expect summarizing *"what"* comments than *"why"* comments).

In this article, our experiments show that current code summarization models are not suitable for all kinds of code comments. At the same time, in practice, there is no need to write all kinds of comments for every code. Hence, a prerequisite question of automatic code summarization is determining which kind of comment is needed. We illustrate this argument by a typical phenomenon that we have observed. For example, we observe that some models can generate comments that are similar to *"TODO: implement this in the future"* correctly, as they are generated according to particular patterns of some unfinished source code. However, such a generation is hard to be useful even it is correct by evaluation in terms of NMT metrics (e.g., *BLEU*), because in practice, developers have their schedules to solve their own *"TODO."* Therefore, we claim that it is too rough to regard all kinds of code comments as the same natural language, so that such kind of *"TODO"* could be treated as a noise, or be applied to an appropriate scenario (e.g., technical debt prediction) instead of comment generation. It is beneficial to pay attention to a practical scenario with specific kinds of code comments (e.g., *"how-to-use"*) in the future.

*5.1.2 Implications for Research.* **Considering context information and more kinds of comments.** In our experiments, all the models perform poorly in *"why"* and *"property"* comments. It indicates that considering only code-comment information in method level is far from enough, even though using method-level pairs of code-comment is feasible for learning-based or retrieval-based approaches. It is beneficial to consider hierarchical project-folder-file-class-method information in a classic software engineering project. In this article, we classify the method-level comments, but it is also beneficial to take class-level or file-level comments into account.

Specifically, there is a difference between the *"why"* category and the *"property"* category. For *"why"* comments, on the one hand, they could be considered to be higher-level comments, which require more information to infer. Hence, the aforementioned class-level, file-level comments, and other contexts could be considered to be the input of the model. On the other hand, the *"why"* comments are often related to the business logic. For example, a comment explains a business logic of why user profiles should be added. Such a business increases the difficulty of reasoning, because unlike other comments (e.g., *"how-it-is-done"*), the business logic is not generic even in a big dataset of source code. Such kind of business logic is also hard to obtain solely from software projects. Therefore, such kinds of comments require more customized and business-logic related information. For *"property"* comments, the code summarization model could not fully meet the practical situation. For example, a typical kind of comment related to property is explaining variables of the source code. To some extent, code summarization models could capture the patterns of which variable should be commented by maximizing the probability of such tokens (i.e., the neural network gives more weight to these variables during training). However, considering the different contexts, the results are still not so ideal. For *"property"* comments that explain the properties of the code, it would be beneficial to consider program analysis techniques (e.g., consider control flow and data flow of the code).

**Equip automatic code summarization with comment classification knowledge.** Our composite approach proves that with a simple and basic classifier, we can promote the performance of code summarization. However, our experiments are just for proving, and we can utilize more information from comment classification: On the one hand, it is more feasible to construct models on a specific kind of comments than general comments. On the other hand, it can improve the understanding of the research question, especially for the complicated problem. For example, as in this article, we find that the *"what"* comments get the best performance, further research is encouraged, and we conduct a more in-depth study to improve the programming understanding.

Under the guidance of the comment classification, it would be beneficial to reconsider the preprocessing of the dataset and the evaluation of the results. The current procedure of preprocessing focuses on natural language following typical NLP conventions (e.g., simple removal of stop words and stemming). It is useful to consider program analysis. Similarly, the evaluation metrics also follow those adopted in the NLP domain (i.e., *"BLEU"* and *"ROUGE"*). These neural machine translation metrics only emphasize literal and fluent results but do not consider the usefulness of the results from the perspective of software developers when evaluating the quality of a code summarization model (e.g., which kinds of comments are needed and are useful for improving the readability).

## 5.2 Threats to Validity

*5.2.1 Internal Validity.* Threats to internal validity relate to the bias in the replication of different models. The implementations of different models are very different. These differences include the environments, the model structures, and especially the preprocessing methods. For example, though all the selected approaches support Java, the provided preprocessing methods are limited to a specific Java version. Also, some only need tokens of the source code while some require extracting AST of the source code. These differences increase the difficulty of reproducing experiments. In this article, we use the source code provided by the original authors. To perform our experiments (i.e., run several experiments on the same dataset), we keep in touch with the authors by email or GitHub issues to address reproducibility issues (e.g., addressing the compatibility of a language parsing tool like javaparser). Thus, with the assistance of the original authors, we believe there is little threat to internal validity.

Other threats to validity relate to the researcher bias in manual classification of the dataset as authors of this article participate in the tagging process (please refer to Section 3). To mitigate such threats, we invite a third non-author programmer as a coder to participate in the open coding process. Before working together to discuss, the three coders go through the whole dataset independently with the referenced criterion of the taxonomy [72]. Such threat mitigation actions (i.e., non-author participation or independent tagging) have been adopted in prior works [64, 70, 72]. The experience of the coders in Java programming (two, three, and two years) is also a threat to validity. We reference the studies of comments classification in the literature [49, 50, 72]. We study the taxonomy of Zhai et al. [72] and investigate how to label comments from the dataset of Pascarella et al. [49, 50]. We believe the help of the prior studies can help mitigate such a threat. In the future, we would further minimize the threats by inviting more experienced programmers to categorize comments.

*5.2.2 External Validity.* Threats to external validity relate to the generability of the dataset. The proposed approach may show different results on different target systems. To reduce this limitation, we use the dataset provided by Hu et al. [21], which is obtained from 9,714 java projects in GitHub and is also widely used [3, 35, 66]. To judge the generability, we use cross-project validation to simulate the practical circumstance. In future work, it would be interesting also to consider cross-language validation.

*5.2.3 Construct Validity.* Threats to construct validity relate to sample validity and taxonomy validity. First, the threat to sample validity is that there is a potential risk that a biased sample of projects could deliver the wrong knowledge. We choose a credible dataset that is widely used and examined by researchers. The projects in the dataset are widely collected in different domains, which are representative of projects with mature ecosystems and development environments. Second, the threat to taxonomy validity is whether the taxonomy can provide an exhaustive and effective way to organize source code comments. We use open coding procedures involving three experienced developers. The three coders derive the taxonomy individually and reach an agreement with 0.79 Fleiss Kappa value. Furthermore, the experimental results confirmed differences among different comments, which proves the effectiveness of the labels.

## 6 RELATED WORK

Code summarization is a popular research topic. The early works for code summarization are rule-based [17, 18, 57, 58]. Sridhara et al. propose Software Word Usage Model to create a rule-based model that generates descriptions for Java methods [57]. Then they combine comments for parameters into method summaries in the following work [58]. Haiduc et al. summarize software documentations by retrieving a similar term-based summary, which contains the most relevant terms for the entity found in the code [17, 18].

However, rule-based approaches cannot address cases that are not covered by the pre-defined fixed set of manually constructed rules. There are information-retrieval-based approaches [11, 44, 53, 68] and researchers consider more context information [14, 39, 41–43]. Movshovitz-Attias et al. [44] predict comments from Java code files using topic models and n-grams. McBurney et al. use a Natural Language Generation system to summarize contextual information of source code to enhance the comprehension [39, 41, 42]. The contexts are the essential methods in the program's call graph, which they compute using the PageRank algorithm. Moreno et al. use IR-based techniques to extract information from classes of code and generate readable text using pre-defined templates [43]. Fowkes et al. formulate code summarization task as an auto-folding problem, which hides the less essential part of source codes to help developers focus on the critical part of source codes [14]. Wong et al. propose an approach called CloCom [68], which discovers similar code segments and used the comments from some code segments to describe the other similar code segments. In this article, we use NNGen [32] as a basic IR-based model, which works in the same way with CloCom.

Nowadays, artificial neural networks and deep learning achieve great success in many fields and are applied to code summarization. Iyer et al. are the first to apply deep-learning to this area. They adopt LSTM networks with attention to leverage the source code vectors and produce sentences that describe C# code snippets and SQL queries [24]. Allamanis et al. use an attentional convolutional network to summarize source code token vectors into short, descriptive function name-like summaries [2]. Alon et al. propose general representations of source code, which represent source codes using structural lexical information and apply them to downstream software engineering tasks, including code summarization [3, 4]. Chen et al. propose an approach to split long code snippets to short AST paths, which could improve the performance of code summarization [8]. Wan et al. use an AST-based LSTM to embed the tree structure hierarchically and use reinforcement learning to conduct the code summarization task [63]. Wang et al. also adopt reinforcement learning, and they present a new code summarization approach using a hierarchical attention network by incorporating multiple code features, including type-augmented abstract syntax trees and program control flow [66]. Liang et al. use a neural network called Code-RNN to represent structural information of source code by summing and averaging the nodes of their parse tree [29]. Hu et al. design a structure-based traversal method to traverse the AST and product a sequence to represent the source code [20]. They then leverage transfer learning to incorporate API knowledge libraries as additional natural knowledge is often required for describing functionality [20, 22]. Piech et al. represent a snippet of source code by simultaneously embedding a triple of (precondition, code, postcondition) into a feature space where code is a linear map on this space. The precondition and the postcondition are the variable values before and after the execution of such code [52]. LeClair et al. [28] propose a neural model that combines words from code with code structure from an AST in a separate way and summarize coherent summaries. Current approaches follow the sequence to sequence framework, which encodes the code into fix-length vectors and then decodes it into natural language space. These pursue a general approach to represent source code and conduct code summarization without considering comment classification. Different from them, in this article, we do not regard comments as general natural language. Instead, we treat comments differently by considering their functionality or intention. To the best of our knowledge, we are the first to investigate the relationship between the type of comment and code summarization. Furthermore, by considering the comments' differences, our composite approach can take advantage of different kinds of approaches (i.e., deep-learning-based and retrieval-based approaches).

There are several works related to comment classification. Mcburney et al. use topic modeling to select keywords and topics as summaries for source code and classify comments automatically [40]. Steidl et al. classify categorization and conduct comment classification to provide better quantitative insights about the comment quality assessment [59]. Pascarella et al. construct a taxonomy of comments and investigate how often each category occurs by manually classifying more than 2,000 code comments [49, 50]. Zhai et al. classify source code comments from the perspective of program analysis. Then they construct program analysis-based rules to infer new comments and associate comments with code for defect detection [72]. We adopt their taxonomy of comment and the classification criteria. Different from these studies, our study focuses on method-level comments, and we go one step further to explore what kinds of code comments a code summarization performs the best for. To the best of our knowledge, no prior studies explore the impact of comment classification on code summarization. We are the first to investigate such an impact considering different types of code comments and the characteristics of the code summarization models (e.g., DeepCom regards the AST sequence as input, which is related to the functionality of the source code).

There are several works related to the classification tasks based on the source code. Ugurel et al. use SVM for automatic classification of archived source code into eleven application topics and ten programming languages [60]. Louis et al. train a classifier to predict where to write a code comment (i.e, whether a developer should write a code comment for a given piece of code) [35]. Wang et al. use the n-gram representations of the source code to train a classifier and predict bug types [65]. Giger et al. use the dependency graph and object-oriented metrics of the source code to predict different categories of code changes [16]. Chen et al. use a Tree2Seq framework to perform code summarization task and code classification is involved as an auxiliary task for aiding the Tree2Seq model [7]. Van Dam et al. propose to use statistical language models from the natural language processing field such as n-grams, skip-grams, Naive Bayes, and normalized compression distance to perform source code classification [61]. In this article, we not only train classifiers to predict the category based on the labeled relationship of code-comment pairs and categories but also apply the classifiers to our composite approach to boost the code summarization.

## 7 CONCLUSION

In this article, we manually label 20,000 code-comment pairs into six categories: *"what," "why," "how-to-use," "how-it-is-done," "property,"* and *"others."* Based on this dataset, we conduct an experiment to investigate the performance of different state-of-the-art code summarization approaches on the categories. We find that the performance of different code summarization approaches varies substantially across the categories. Moreover, the category for which a code summarization model performs the best is different for the different models. The preferences of each model to a particular comment category are different. Motivated by this finding, we design a composite two-step approach that analyzes a method and outputs a comment that summarizes it: (1) our approach predicts the category of code comment that likely accompanies a particular piece of code, (2) our approach then picks the code summarization approach that performs the best for the inferred category, and use it to generate the corresponding comment. Our approach outperforms other approaches without considering comment classification, and obtains a relative improvement of 8.57% in terms of *ROUGE-L* and a relative improvement of 16.34% in terms of the *BLEU-4* score. Our research highlights that by incorporating the automatically inferred category of code comment, we can boost the code summarization task.

## REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surveys* 51, 4 (2018), 81.

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

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=H1gKYo09tX.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* 3 (2019), 40.

[5] Leo Breiman. 2001. Random forests. *Mach. Learn.* 45, 1 (2001), 5–32.

[6] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees.* CRC Press, Boca Raton, FL.

[7] Minghao Chen and Xiaojun Wan. 2019. Neural comment generation for source code with auxiliary code classification task. In *Proceedings of the 26th Asia-Pacific Software Engineering Conference (APSEC'19)*. IEEE, 522–529.

[8] Qiuyuan Chen, Han Hu, and Zhaoyi Liu. 2019. Code summarization with abstract syntax tree. In *Proceedings of the International Conference on Neural Information Processing*. Springer, 652–660.

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

[10] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting and Designing for Pervasive Information*. ACM, 68–75.

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

[12] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. *IEEE Trans. Softw. Eng.* (2019), 1–1. https://ieeexplore.ieee.org/abstract/document/8765743.

[13] Joseph L. Fleiss. 1971. Measuring nominal scale agreement among many raters.*Psychol. Bull.* 76, 5 (1971), 378.

[14] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2016. TASSAL: Autofolding for source code summarization. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C'16)*. IEEE, 649–652.

[15] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *J. Syst. Softw.* 81, 3 (2008), 447–460.

[16] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2012. Can we predict types of code changes? An empirical analysis. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'12)*. IEEE, 217–226.

[17] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Vol. 2. ACM, 223. DOI: https://doi.org/10.1145/1810295.1810335

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

[19] Han Hu, Qiuyuan Chen, and Zhaoyi Liu. 2019. Code generation from supervised code embeddings. In *Proceedings of the International Conference on Neural Information Processing*. Springer, 388–396.

[20] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* 25, 3 (2019), 1–39. https://link.springer.com/article/10.1007/s10664-019-09730-9.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI, 2269–2275.

[23] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. 2012. On the relationship between comment update practices and software bugs. *J. Syst. Softw.* 85, 10 (2012), 2293–2304.

[24] 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*. 2073–2083.

[25] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empir. Softw. Eng.* 10, 1 (2005), 31–55.

[26] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in data mining: Formulation, detection, and avoidance. *ACM Trans. Knowl. Discov. Data.* 6, 4 (2012), 1–21.

[27] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems.* MIT Press, 3146–3154.

[28] Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* Association for Computational Linguistics, 3931–3937. DOI: https://doi.org/10.18653/v1/N19-1394

[29] Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence.*

[30] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. *Text Summar. Branches Out* (2004), 74–81. https://www.aclweb.org/anthology/W04-1013/.

[31] Mario Linares-Vasquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2014. On using machine learning to automatically classify software applications into domain categories. *Empir. Softw. Eng.* 19, 3 (2014), 582–618.

[32] Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. 2019. A neural-network-based code summarization approach by using source code and its call dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware'19).* ACM, New York, NY, 12:1–12:10. DOI: https://doi.org/10.1145/3361242.3362774

[33] 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.

[34] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic generation of pull request descriptions. Retrieved from https://Arxiv:1909.06987.

[35] Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. 2020. Where should I comment my code? A dataset and model for predicting locations that need comments. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results.* 21–24.

[36] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. Retrieved from https://Arxiv:1508.04025.

[37] Walid Maalej and Martin P. Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Trans. Softw. Eng.* 39, 9 (2013), 1264–1282.

[38] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press, Cambridge, UK.

[39] Paul W. McBurney. 2015. Automatic documentation generation via source code summarization. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 903–906.

[40] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. 2014. Improving topic model source code summarization. In *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 291–294.

[41] Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 279–290.

[42] Paul W. McBurney and Collin McMillan. 2016. Automatic source code summarization of context for java methods. *IEEE Trans. Softw. Eng.* 42, 2 (2016), 103–119.

[43] 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 *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13).* IEEE, 23–32.

[44] Dana Movshovitz-Attias and William Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics.* 35–40.

[45] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How are java software developers using the elipse IDE? *IEEE Softw.* 23, 4 (2006), 76–83.

[46] Graham Neubig, Zi-Yi Dou, Junjie Hu, Paul Michel, Danish Pruthi, Xinyi Wang, and John Wieting. 2019. compare-mt: A tool for holistic comparison of language generation systems. Retrieved from https://Arxiv:1903.07926.

[47] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Trans. Services Comput.* 9, 5 (2016), 771–783.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics.* Association for Computational Linguistics, 311–318.

[49] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 227–237.

[50] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in java software systems. *Empir. Softw. Eng.* 24, 3 (2019), 1–39. https://link.springer.com/article/10.1007/s10664-019-09694-w.

[51] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. 2011. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[52] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning*. JMLR. org, 1093–1102.

[53] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 390–401.

[54] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.* 4 (1999), 557–572.

[55] Lea Sgier. 2012. Qualitative data analysis. *An Initiat. Gebert Ruf Stift* 19 (2012), 19–21. http://politicalscience.ceu.edu/sites/politicalscience.ceu.hu/files/attachment/course/733/sgierqualitativedataanalysis_40.pdf.

[56] Shikhar Sharma, Layla El Asri, Hannes Schulz, and Jeremie Zumer. 2017. Relevance of unsupervised metrics in task-oriented dialogue for evaluating natural language generation. Retrieved from https://Arxiv:1706.09799.

[57] 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*. ACM, 43–52.

[58] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *Proceedings of the IEEE 19th International Conference on Program Comprehension*. IEEE, 71–80.

[59] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13)*. Ieee, 83–92.

[60] Secil Ugurel, Robert Krovetz, and C. Lee Giles. 2002. What's the code? automatic classification of source code archives. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 632–638.

[61] Juriaan Kennedy van Dam and Vadim Zaytsev. 2016. Software language identification with natural language classifiers. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1. IEEE, 624–628.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is all you need. Retrieved from http://arxiv.org/abs/1706.03762.

[63] 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*. ACM, 397–407.

[64] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Trans. Softw. Eng.* 46, 11 (2018), 1241–1266. https://ieeexplore.ieee.org/abstract/document/8502824.

[65] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 708–719.

[66] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. 2020. Reinforcement-learning-guided source code summarization via hierarchical attention. *IEEE Trans. Softw. Eng.* (2020), 1–1. https://ieeexplore.ieee.org/abstract/document/9031440.

[67] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC'19)*. IEEE, 53–64.

[68] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*. IEEE, 380–389.

[69] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Softw. Eng.* 44, 10 (2017), 951–976.

[70] Xin Xia, Zhiyuan Wan, Pavneet Singh Kochhar, and David Lo. 2019. How practitioners perceive coding proficiency. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 924–935.

[71] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: An analysis of stack overflow code snippets. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR'16)*. IEEE, 391–401.

[72] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. IEEE/ACM. Retrieved from https://rucore.libraries.rutgers.edu/rutgers-lib/61591/.

Received May 2020; revised November 2020; accepted November 2020