Classifying Code Comments via Pre-trained Programming Language Model

Ying Li
Southern University of Science and Technology
Department of Computer Science and Engineering
Research Institute of Trustworthy Autonomous Systems
Shenzhen, China
12032486@mail.sustech.edu.cn

Haibo Wang
Southern University of Science and Technology
Department of Computer Science and Engineering
Research Institute of Trustworthy Autonomous Systems
Shenzhen, China
wanghb2020@mail.sustech.edu.cn

Huaien Zhang
Southern University of Science and Technology
The Hong Kong Polytechnic University
Shenzhen, China
cshezhang@comp.polyu.edu.hk

Shin Hwei Tan
Southern University of Science and Technology
Department of Computer Science and Engineering
Research Institute of Trustworthy Autonomous Systems
Shenzhen, China
tansh3@sustech.edu.cn

***Abstract*—Previous studies have categorized code comments for various programming languages to produce high-quality code comments that can improve code readability and benefit maintenance. However, it still requires more effort to identify the main information contained in code comments. Pre-trained language model has shown promising results for solving software engineering tasks. In this paper, we propose a model for code comment classified using the recent pre-trained language model specialized for code-specific tasks (i.e., CodeT5). We introduce *expert-predefined features* to enhance the model's classification performance. Our evaluation on the official dataset shows that it outperforms the baseline by improving the precision (+65.9%), recall (+147.3%) and the F1-score (+112.5%) of the classification.**

***Index Terms*—Pre-trained language model, code comment**

I. Introduction

With the evolution of software projects, effective program comprehension is increasingly crucial. During software development, developers usually write code comments to help others understanding the structure, functionality, and behavior of the code, as well as the relationship between different components in the whole project [1], [2]. As code comments contain various types of information, it can help the developers to modify and maintain existing program, debug faults in code, and improve the performance and reliability of the software.

In recent years, researchers have carried out a series of studies with respect to code comments from various perspectives: code comment generation [3]–[5], inconsistent code comment detection [6]–[8], code comment classification [9]–[14]. In this paper, we focus on the code comment classification. In general, code comments are written in natural language mixed with source code elements, unlike software defects for which there is a relatively recognized category, the current category based on code comments is still not well developed, making it difficult to discover potentially generic features of code comments in different programming languages. Pascarella et al. [9] deeply analyze the code comments in Java open-source projects to acquire the definition of the classification types and achieve automated classification using machine learning. To explore the similarities in code comments of different programming languages, Rani et al. [12] manually analyzed the code comments of popular open-source projects in Python, Java, and Smalltalk to define their classification types. Meanwhile, they integrated natural language processing (NLP) and text analysis techniques to extract potential semantic and syntactic features of code comments in combination with Random Forest to automatically identify the type of code comments.

The aforementioned studies are usually based on supervised machine learning classification algorithms together with limited numbers of manually labeled data. They have several limitations. First, these methods tend to require complex feature engineering and analysis for better performance. The strong reliance on pre-defined features for new tasks makes it difficult to generalize to new tasks and do not make full use of the training data available. Second, they often rely on a limited number of manually labeled source code comments, which is usually insufficient for training a classifier thus leading to a poor performance. In this paper, we use the newly proposed pre-training and fine-tuning diagram to help improving the performance of code comment classification. Large pre-trained programming language models (for example, CodeT5 [15], CodeBert [16], and CodeReviwer [17]) are usually pre-trained on a large number of software engineering domain specific corpus, the general knowledge learned during this process can be used for the downstream tasks by fine-tuning or prompts-tuning strategies, which can be a promising direction to tackle the limited human labeled data. Meanwhile, by using the subword algorithm, Byte-Pair-Encoding (BPE) [18], during data processing, we tackle the out-of-vocabulary (OOV) problems in the previous mentioned studies.

II. Tool description

A. Data Preprocessing

Data pre-processing is performed on the training set as well as the test set to improve the accuracy and performance of the model. The code comments of a class are split into individual sentences. We normalized each sentence by removing punctuation marks such as "#%^.?" from each sentence, as well as redundant spaces, and converting all words to lowercase.

Next, we identified the features in each sentence based on expert-predefined characteristics to capture the category-specific features for improving the recognition of the model for each category. In this tool paper, expert-predefined features are adopted from the NLP features extracted by Rani et al [12]. These features are phrases or words that belong to a particular category. For example, in the category of *usage*, the NLP features extracted are *sees example*, *results* and so on. The recognized features will be tagged with <*s*>*features*<\s> to make the model pay more attention to the key features in the code comments. Listing 1 shows a pre-processed instance from the testing dataset in the category *example* of json format. The *comment_sentence* refers to a raw code comment, while the *pre_sentence* in line 8 is a comment with only special characters removed, and *final_sentence* in line 9 is a feature-tagged code comment based on *pre_sentence*. In this example, the feature *can be used* has been highlighted for guiding the model to learn potential patterns.

```
1   {
2     "comment_sentence_id": 378,
3     "class": "BlInfiniteItemAnimationsFinished",
4     "comment_sentence": "i can be used, for example,
          to delay an action in a data set until
          currently running animations are complete.",
5     "partition": 1,
6     "instance_type": 0,
7     "category": "Example",
8     "pre_sentence": "i can be used for example to
          delay an action in a data set until currently
           running animations are complete",
9     "final_sentence": "i <s>can be used</s> for
          example to delay an action in a data set
          until currently running animations are
          complete"
10  }
```

Listing 1: Example of pre-processed dataset

B. Model Training

Since the proposal of GPT and Bert, the pre-training and fine-tuning paradigm has been widely used in the software engineering domain. A typical language pre-trained model refers to pre-training a large model on massive unlabelled corpora by self-supervised objectives, and fine-tuning the model on downstream tasks (i.e., program understanding and generation tasks) with task-specific loss. Inspired by this paradigm, in this tool paper, we adopt CodeT5 [15], a pre-trained encoder-decoder model that takes into account the identifier information in the source code, which is suitable for our task of classifying code comments. CodeT5 is an improved model with the same architecture as T5 [19], a framework that converts all NLP tasks uniformly into Text-to-Text tasks. The prefix *"Classification:"* is introduced to each pre-processed code comment to indicate the model what task is performed. After tokenizing a code comment, the special tokens *[CLS]* and *[SEP]* are concatenated into the sequence to mark the beginning and end of the text sequence. We fine-tuned CodeT5 based on our pre-processed dataset. Moreover, we also used a pre-trained tokenizer based on the Byte-Pair Encoding (BPE) algorithm for tokenizing the code comments. Figure 1 depicts the overall workflow of the proposed method.

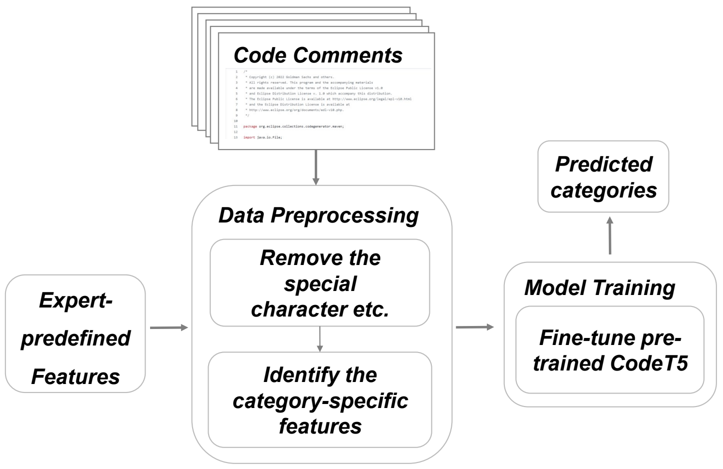

Code Comments
Expert-predefined Features
Data Preprocessing
Remove the special character etc.
Identify the category-specific features
Model Training
Fine-tune pre-trained CodeT5
Predicted categories

Fig. 1: The overall workflow for fine-tuning CodeT5 for Binary Classifier code comments

III. Experiment

A. Dataset

We evaluate the model using the dataset provided by the official competition [20] in NLBSE2023[1]. The dataset was extracted from 20 popular open-source projects written in three different programming languages (i.e., Java, Pharo, and Python) and was manually labeled by Rani et al. [12]. In particular, the Java dataset has 2,418 code comments distributed in seven categories: *summary*, *pointer*, *deprecation*, *rational*, *ownership*, *usage*, *expand*. The Pharo dataset has 1,765 code comments distributed in seven categories: *key messages*, *intent*, *class references*, *example*, *key implementation*, *responsibilities*, *collaborators*. The Python dataset has 2,555 code comments distributed in five categories: *summary*, *parameters*, *usage*, *development notes*. In the training stage, based on the proportion of positive examples in the training set, the same proportion of data from the positive and negative examples in the entire training set is taken as the validation set data respectively. Table I shows the statistics of the dataset.

[1]https://nlbse2023.github.io/tools/

TABLE I: Overview of dataset

Language	Category	Training		Testing		Total
		Positive	Negative	Positive	Negative	
	Expand	505	1426	127	360	2418
	Ownership	90	1839	25	464	2418
	Deprecation	100	1831	27	460	2418
Java	Rational	223	1707	57	431	2418
	Summary	328	1600	87	403	2418
	Pointer	289	1640	75	414	2418
	Usage	728	1203	184	303	2418
	Responsibilities	267	1139	69	290	1765
	Key messages	242	1165	63	295	1765
	Key implementation points	184	1222	48	311	1765
Pharo	Collaborators	99	1307	28	331	1765
	Example	596	812	152	205	1765
	Class references	60	1348	17	340	1765
	Intent	173	1236	45	311	1765
	Expand	402	1637	102	414	2555
	Parameters	633	1404	161	357	2555
Python	Summary	361	1678	93	423	2555
	Development notes	247	1792	65	451	2555
	Usage	637	1401	163	354	2555

B. Metrics

To evaluate the effectiveness of our model, we adopt the commonly used metrics in classification tasks: *precision*, *recall*, and *F1-score*. The metrics are defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F1 = 2\frac{P \cdot R}{P + R} \tag{3}$$

TP denotes true positive, FP refer to false positive, TN means true negative, FN denotes false negative.

C. Implementation

We use HuggingFace Transformers[2], PyTorch[3] to fine-tune the CodeT5[4]. The Adamw was used, a weight-decay-based optimizer implemented by the transformers library, where the *epsilon* factor is set to *1e-8* and the learning rate is set to *5e-5* for the Adam optimizer. Furthermore, the maximum total source sequence length after tokenization was set to 256 based on the length of the original code comments in the entire dataset. Sequences longer than this will be truncated, and sequences shorter will be padded with a specific number (0 in our method). We trained 120 epochs for each category about different programming languages. We set the batch size to 8. All experiments are conducted on a multi-core serve with 3.20GHz 8-core Intel(R) Xeon(R) Gold 6134 CPU, two NVIDIA TITAN V GPUs and 192GB of RAM, running Ubuntu 16.04 operating system with Linux kernel 4.15.0.

IV. RESULTS

We evaluate our approach against the provided baseline (i.e., classifiers based on the Random Forest model) [12]. Table II demonstrates the performance of model for the test dataset. Overall, our model can achieve 72.82 precision, 60.58 recall, and 65.67 F1-score, which improves the baseline model [21] by 65.9%, 147.3%, and 112.5%. Moreover, for each category of comments shown in Table II, our model can gain better classification results than the baseline model. Specifically, the baseline model only scores 0 on all three metrics in *Deprecation* category and our model can earn 90.48, 70.37, and 79.17 on these three metrics.

TABLE II: Code comment classification results

Language	Category	CodeT5Classifier			Baseline		
		Precision%	Recall%	F1-score%	Precision%	Recall%	F1-score%
	Expand	66.97	57.48	61.86	35.10	26.80	30.40
	Ownership	100.00	100.00	**100.00**	100.00	68.00	81.00
	Deprecation	90.48	70.37	79.17	0	0	0
Java	Rational	75.61	54.39	63.27	63.00	29.80	40.50
	Summary	70.79	72.41	71.59	38.50	28.70	32.90
	Pointer	81.13	57.33	67.19	66.70	24.00	35.30
	Usage	81.13	70.11	75.22	54.10	35.90	43.10
	Responsibilities	65.57	57.97	61.54	59.00	33.30	42.60
	Key messages	86.05	58.73	69.81	31.20	15.90	21.10
	Key implementation points	64.29	37.50	47.37	17.90	10.40	13.20
Pharo	Collaborators	60.00	32.14	41.86	46.70	25.00	32.60
	Example	87.07	84.21	85.62	76.70	43.40	55.50
	Class references	53.85	41.18	46.67	33.30	5.90	10.00
	Intent	88.64	86.67	87.64	57.70	33.30	42.30
	Expand	55.56	49.02	52.08	26.30	19.60	22.50
	Parameters	75.33	70.19	72.67	51.40	22.40	31.20
Python	Summary	69.32	65.59	67.40	12.30	7.50	9.30
	Development notes	43.18	29.23	**34.86**	17.20	16.90	17.10
	Usage	68.66	56.44	61.95	46.90	18.40	26.40
Overall		**72.82**	**60.58**	**65.67**	43.90	24.50	30.90

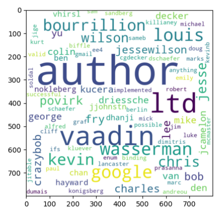
In contrast with Pharo and Python, the best overall classification performance has been obtained in the Java categories that can be attributed to the fact that developers frequently use annotations such as *@author, @deprecation, @since*, etc. to annotate Java code. The keywords in the annotations are unique in these categories, which makes it easier for the model to identify and classify these comments accurately. For instance, we can observe from the word distribution in the positive dataset for the ownership category (Fig.2a) and the negative dataset (Fig.2b) the keywords in Java annotations help to distinguish between the positive and negative examples. Our model achieves full scores in the *Ownership* category. Meanwhile, we think that the worst F1-score for Python code comments could be caused by the limited number of *expert-predefined features* provided, with only 2751 available as compared to 5436 for Java and 4152 for Pharo. Our model only achieves 34.86% F1-score.

The self-attention mechanism in CodeT5 makes it capable of efficiently capturing the relationships between words. Compared to the baseline approach to extracting features, our model can learn the essential knowledge of natural language from massive unlabeled data and lead to such significant improvements. The results also indicate the effectiveness of leveraging pre-trained programming language model for code comment classification. Although our model cannot achieve particularly good results on three categories (*Key implementation points*, *Class references*, and *Development notes*), the baseline results prove the extreme challenges to classify these categories accurately, and our method is also better than baseline on these categories.

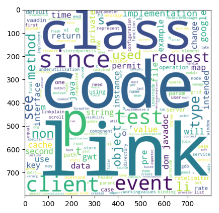
V. RELATED WORK

Code comments classification has long been studied. Previous studies [9]–[14] usually use text analysis techniques together with machine learning classification algorithms to predict the type of source code comments. For example,

[2]https://huggingface.co
[3]https:pytorch.org
[4]https://huggingface.co/Salesforce/codet5-base



(a) Word Cloud of *Ownership* Positive Dataset



(b) Word Cloud of *Ownership* Negative Dataset

Fig. 2: Word Cloud of *Ownership* Dataset

Pascarella et al. [9]–[11] developed a taxonomy of source code comments. Subsequently, they employed supervised machine learning algorithms (naive Bayes, Random Forest and J48, etc.) to build an automated classifier based on some manual labeled data derived from the above mentioned taxonomy. Similar, Rani et al. [12] developed a multi-language (Python, Java, and Smalltalk) approach for class comment classification. They used a feature extraction tool, named NEON, to infer all patterns characterizing the comment sentences and adding them as features for classification. Different from previous mentioned works, we try to explore the possibility of classifying code comments through the pre-trained large language models. We choose CodeT5 [15] because it is pre-trained on a large number of software engineering domain-specific corpus like CodeSearchNet [22], and we can use those domain-specific knowledge to improve the accuracy by transfer learning (fine-tuning or prompts-tuning).

VI. Conclusions

Inspired by the recent success in using pre-trained language model for software engineering task, we propose a model for code comment classification using CodeT5, a pre-trained programming language model. Our evaluation shows that it outperforms the baseline by improving the precision, recall, and F1-score of the classification.

Data Availability

The experimental data and source code are available at GitHub [23] and the trained model at Zenodo [24].

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61902170), and Huawei (Grant No. K2227Z081).

References

[1] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st international conference on program comprehension (icpc)*. Ieee, 2013, pp. 83–92.

[2] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.

[3] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[4] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 380–389.

[5] B. Li, M. Yan, X. Xia, X. Hu, G. Li, and D. Lo, "Deepcommenter: a deep code comment generation tool with hybrid lexical and syntactical information," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1571–1575.

[6] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 260–269.

[7] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 53–64.

[8] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards detecting inconsistent comments in java source code automatically," in *2020 IEEE 20th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 2020, pp. 65–69.

[9] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 227–237.

[10] L. Pascarella, "Classifying code comments in java mobile applications," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 39–40.

[11] L. Pascarella, M. Bruntink, and A. Bacchelli, "Classifying code comments in java software systems," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1499–1537, 2019.

[12] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz, "How to identify class comment types? a multi-language approach for class comment classification," *Journal of Systems and Software*, vol. 181, p. 111047, 2021.

[13] P. Beck, M. J. Mohammadi-Aragh, and C. Archibald, "An initial exploration of machine learning techniques to classify source code comments in real-time," in *2019 ASEE Annual Conference & Exposition*, 2019.

[14] J. Zhang, L. Xu, and Y. Li, "Classifying python code comments based on supervised learning," in *Web Information Systems and Applications: 15th International Conference, WISA 2018, Taiyuan, China, September 14–15, 2018, Proceedings*. Springer, 2018, pp. 39–47.

[15] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[17] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Codereviewer: Pre-training for automating code review activities," *arXiv preprint arXiv:2203.09095*, 2022.

[18] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," *arXiv preprint arXiv:1804.10959*, 2018.

[19] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[20] R. Kallis, M. Izadi, L. Pascarella, O. Chaparro, and P. Rani, "The nlbse'23 tool competition," in *Proceedings of The 2nd International Workshop on Natural Language-based Software Engineering (NLBSE'23)*, 2023.

[21] Rani, "Nlbse'23 tool competition: Code comment classification," 2023. [Online]. Available: https://github.com/nlbse2023/code-comment-classification

[22] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[23] Y. Li, "Classifier for code comments," 2023. [Online]. Available: https://github.com/Ying091909/Classifier-for-code-comments

[24] Li, "The trained model for code comments," 2023. [Online]. Available: https://www.zenodo.org/record/7659286#.ZArrpXZBwQ9