Exploring Transformer-Based Models for Automatic Useful Code Comments Detection

Mithun Das[1,2], Subhadeep Chatterjee[2]

[1]Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, West Bengal, India
[2]Siemens EDA, Kolkata, West Bengal, India

Abstract

Code commenting is a practice developer pursues to improve the readability of the code. Hence, it is essential to evaluate comments based on whether they increase code understandability for software maintenance tasks. Although some studies have been conducted for detecting useful code comments, most of them exploit various handcrafted features, and the recent advancement of transformer-based models is still under-explored. Therefore to fill the gap, we explore transformer-based models and propose a fusion-based solution for detecting Useful comments based on the shared task, "*Information Retrieval in Software Engineering (IRSE)* [1]" at **FIRE 2022**. We observe that our fusion-based model BERT+CodeBERT(PP), which uses features from both code comments and snippets, achieves the highest Macro F1 score of 90.739 among all the models and ranked **first** in this task.

Keywords

Comment Quality, Natural Language Processing, Classification, Software Development

1. Introduction

Software development refers to a software deliverable's design, documentation, programming, testing, and ongoing maintenance [1]. During the development and ongoing maintenance of the product, a developer has to write a lot of code to fix bugs and extend/add new features. Although during the development phase, the developer writes codes based on his thinking and code writing style, it may not be the case when a bug comes, or new features have to be added; the same developer may be working on the same code he wrote earlier. Therefore, the new developer must understand the existing code before modifying it. However, it is not always easy to understand the current code just by going through it without proper documentation. Even if documentation is found, without adequate updates, it becomes outdated, which may not benefit the new developers. Therefore, developers must manually search and mine source files and other knowledge sources like emails, and defect trackers, to understand the application's overall design [1]. Nevertheless, this prolonged procedure decreases developer productivity, introduces hidden defects resistant to regression testing, and lowers the quality of the code. Hence code commenting is a technique developer follows to increase the readability of the

[1]https://sites.google.com/view/ir-se/home

FIRE'22: Forum for Information Retrieval Evaluation, December 9-13, 2022, India

mithundas@iitkgp.ac.in (M. Das); subhadeep.ju@gmail.com (S. Chatterjee)
0000-0003-1442-312X (M. Das)

[1]https://en.wikipedia.org/wiki/Software_development

#	Comment	Code	Label
1	/* uses png_calloc defined in pngriv.h*/	/* uses png_calloc defined in pngriv.h*/ PNG_FUNCTION(png_const_structrp png_ptr) { if (png_ptr == NULL || info_ptr == NULL) return; png_calloc(png_ptr); ...}	Useful
2	/* serial bus is locked before use */	static int bus_reset (. . .) /* serial bus is locked before use*/ { .. update_serial_bus_lock (bus * busR); }	Not Useful
3	// integer variable	int Delete_Vendor; // integer variable	Not Useful

Table 1
Example of Some *Useful* and *Not-Useful* Comments

code [2]. *A **code comment** is a programmer-readable explanation or annotation in a computer program's source code, added to make the source code manageable for humans to comprehend, and is commonly ignored by compilers and interpreters.*[2]

Commenting code is not always as easy as described in the definition. Depending on the products and company guidelines, the technique of writing comments may differ. Although, to a large extent, the commonality among all the guidelines is that a comment should be informative and meaningful and represent whatever is written in actual coding without any ambiguity. Nevertheless, it is undeniable that comments can be noisy, unstable, and may not evolve with the source code [3]; still, source code and associated comments can play an essential role [4] in understanding the code rapidly for ongoing maintenance.

In addition, merely placing irrelevant comments in the codes does not add any significance to enhance the readability of the source code. Thus it is necessary to identify useful comments in the source code given a code snippet. Therefore to engage and facilitate the research around useful comment detection, the organizers of the "*Information Retrieval in Software Engineering (IRSE)*" [5] shared task at **FIRE 2022** have introduced a dataset for useful comments classification given the associated code snippet. The objective of the shared task is to devise methodologies to detect useful comments automatically. We show some examples of Useful and Not-Useful comments in Table 1.

To this end, several strategies have been explored to classify comments based on various hand-crafted features such as explicit syntactic information, presence of specific tags (e.g., @param, @deprecated, etc.), words and symbols; or implicit elements, such as comment length, parts-of-speech of comment words or the cosine similarity of words in code-comment pairs [6, 7, 8, 9]; however, the recent advancement of transformer-based models(e.g., BERT [10], CodeBERT [11]) are still under-explored.

In this paper, we attempt to use the existing transformer-based models for our classification task, which have already been seen to outperform several baselines and stand as a state-of-the-art model for various downstream tasks [12, 13, 14] in Natural language processing. The rest of the paper is organized as follows. In section 2, we discuss some of the related work. In section

[2]https://en.wikipedia.org/wiki/Comment_(computer_programming)

3, we present the dataset description. In Section 4, we introduce the system description that we propose. Finally, we discuss the results in section 5 and the conclusion in Section 6.

2. Related Works

This section discusses some of the proposed strategies in the literature to investigate and evaluate comments quality by detecting inconsistencies and classifying comments.

2.1. Detecting inconsistencies with source code

Tan et al. [15] devised a tool iComments, to analyze comments written in natural language to extract implicit program rules and used those rules to automatically detect inconsistencies between comments and source code, indicating either bugs or irrelevant comments. For this purpose, the author incorporates Machine Learning, Natural Language Processing(NLP), Statistics, and Program Analysis methods and evaluates the tools on four large code bases: Linux, Mozilla, Wine, and Apache. Ratol el al. [16] designed a new rule-based approach called Fraco to detect fragile comments. It incorporates the identifier's type, its morphology, the identifier's scope, and the location of the comment. The author evaluated the method by comparing its precision and recall against hand-annotated benchmarks created for six targets Java systems and compared the results against the performance of Eclipse's automatic in-comment identifier replacement feature.

2.2. Comment classification & quality evaluation

Haouari et al. [8] empirically studied existing comments in different open source Java projects from both a quantitative and a qualitative point of view. The authors proposed a taxonomy of comments based on comment scope (inline, method, constructor), comment type (application, implementation, and the like), and comment style for their analysis. Padioleau et al. [17] manually examined 1,050 comments randomly from operating systems code written in C from three open source projects: Linux, FreeBSD, and OpenSolaris (started as closed software) due to their overwhelming complexity and the critical essence of trustworthiness. The authors studied the comments from several dimensions and categorized them based on memory, locks, data-structure related, errors, control flow, TODO or FIXME, etc. Aman et al. [18] collected Java methods (programs) from six popular open source products and conducted analyses on words that emerged in their comments. The authors showed that a method with longer comments (more words) tends to be more change-prone and requires more fixes after releases. Steidl et al. [9] presented a semi-automatic approach for quality analysis and assessment of code comments. The method furnishes a model for comment quality based on different comment categories(copyright, header, member, inline, section, code, and application task). The authors explored machine learning models to categorize comments on Java and C/C++ programs. Additionally, they presented a quality model that filters out useless and uninformative comments by examining the similarity of words in code-comment pairs using the Levenshtein distance and comment length. Majumdar et al. [6] proposed CommentProbe for automatic classification and quality evaluation of code comments of C codebases based on how they can assist in understanding existing code.

Split	Useful	Not-Useful	Total
Train	4,337	3,710	8,047
Test	282	719	1001
Total	4,619	4,429	9,048

Table 2
Dataset statistics

For this purpose, the authors have collected 20,206 comments from open-source GitHub projects and annotated them with assistance from industry experts. The authors handcrafted several features to analyze comments semantically and using machine learning models, classified them as *Useful*, *Partially Useful*, and *Not Useful*.

Although the existing methodologies established several baselines for meaningful code comments analysis, none of the approaches used the recent advancement of transformer-based models. Hence to fill the research gap, in this work, we propose a fusion-based technique using pre-trained transformer-based models to classify *Useful* comments based on the dataset shared by the organizers of *IRSE*.

3. Dataset Description

The shared task on Useful Comment Classification (given the surrounding code snippet) in *Information Retrieval in Software Engineering (IRSE)* [5] at **FIRE 2022** is based on a classification problem to evaluate the usefulness of code comments to improve the readability of the codes for the developers. The primary focus of the shared task is to improvise methodologies for *Useful* code comments detection. For this purpose, the organizers developed a dataset by labeling code comments as *Useful* and *Non-Useful* based on the associated code. In total, a team of 14 annotators was assigned for the task, and two individual annotators labeled each code comment as *Useful* and *Non-Useful*. To supervise the annotation process, the organizer conducted weekly meetings with the annotators. To evaluate the quality of the annotated dataset, the organizers used the cohen kappa score and achieved a kappa (k) value of 0.734, which shows substantial agreement among the annotators. We show the class distribution of the shared dataset in Table 2. The training set consists of 8,047 code comments (out of which 4,337 comments were labeled as *Useful*), and the test set consists of 1,001 Comments.

4. System Description

This section discusses the methodology we followed for detecting Useful code comments. The detail of the pipeline is shown in Figure 1.

4.1. Pre-Processing

While manually going through the dataset, we observe the code comments contain lots of special characters, blank spaces, newlines, etc. Therefore we apply pre-processing steps to

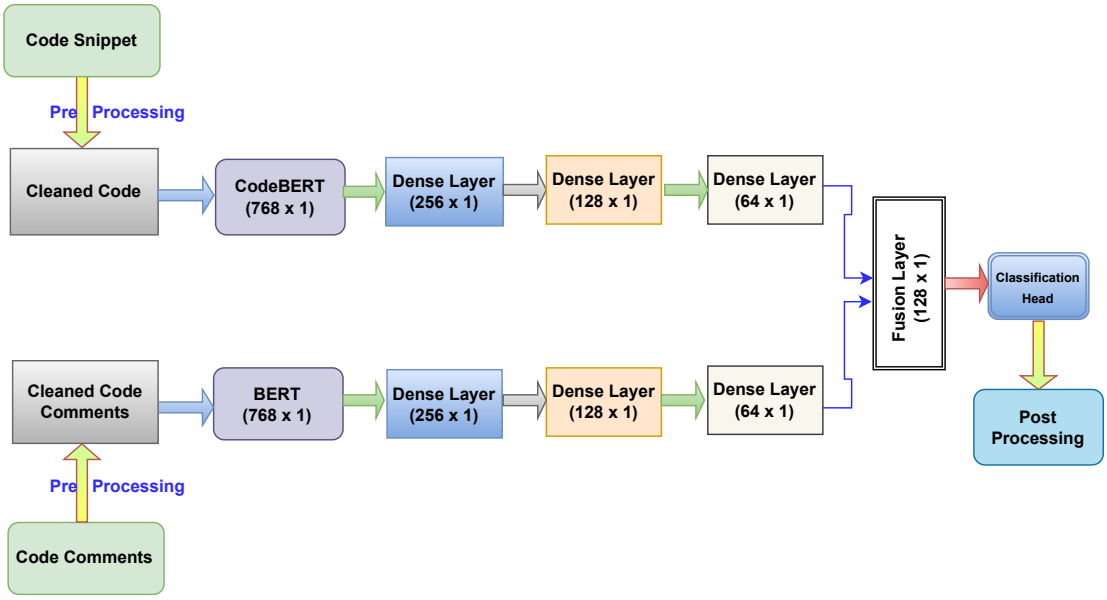

Figure 1: Architecture of the Fusion-based BERT+CodeBERT(PP) Model.

remove all the non-English characters. Further, we observed the comments are associated with the code snippet, so we removed the code comments from the code snippet as well.

4.2. Uni-modal Text-based BERT Model

As part of our initial experiments, we pose the problem as a unimodal text classification task. Here, instead of using the comments along with their associated codes, we only use the code comments to find out whether the comment is useful or not. The idea is that although the code associated with the comment is not utilized explicitly for the classification, sometimes developers write code snippets in the comment to make it more transparent, which can indicate the model to determine the usefulness of the comments. For this purpose, we use the transformer-form model **BERT** [10].

BERT[3], which stands for Bidirectional Encoder Representations from Transformers, is pre-trained on a large corpus of English data utilizing masked language modeling (MLM) and Next sentence prediction (NSP) objectives in a self-supervised manner. It consists of a stack of transformer encoder layers with 12 “attention heads," i.e., fully connected neural networks augmented with a self-attention mechanism. The model can handle a maximum of 512 tokens as input. To fine-tune BERT, we add a fully connected layer with the output corresponding to the CLS token in the input. This CLS token output usually carries the representation of the sentence provided to the model.

[3]https://huggingface.co/bert-base-uncased

4.3. Fusion Model

As discussed above, the uni-modal model does not explicitly consider the associated code snippet along with the code comments. However, to decide whether a comment is useful, the use of surrounding code is crucial. Therefore we design a fusion-based model to take into consideration both code comments and code snippets to understand better the relationship between the comments and surrounding code snippets.

Although programming languages are primarily written in the English language, the grammar of programming languages does not follow the rule of natural language. Hence using a model like BERT, which is pre-trained on natural language, won't be an excellent choice to represent the code. Thus we use the model **CodeBERT** [4] [11], a multi-programming-lingual model pre-trained on NL-PL pairs in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go). This model is initialized with Roberta-base and trained with MLM and Replaced Token Detection(RTD) objective. Similar to BERT, the illustration of [CLS] works as the aggregated sequence representation passed through the model.

Here we extract the 768-dimensional features vector from the last layer of the BERT and CodeBERT model using the code comments and code snippets, respectively. Then these feature vectors are separately provided through three dense intermediate layers of size 256, 128, and 64, respectively. Finally, we concatenate all the nodes(BERT+CodeBERT) and reduce them to a feature vector of length 2(*Useful* or *Not-Useful*).

4.4. Post-Processing

We further apply the following post-processing step to improve the classification performance of the models. We interviewed two software developers and asked how they felt about the short-length comments. Based on their prior experience, both of them have noticed that shorter code comments are mostly Not-Useful. Thus as a post-processing step, we relabeled all the comments less than five tokens to *Not-Useful*.

4.5. Experimental Setup

We have trained the models for ten epochs with binary cross entropy loss for both the uni-modal and fusion-based models. We have used the Adam optimizer with an initial learning rate of 2e-5 and epsilon of 1e-8. For the unimodal text-based BERT model, we have used the batch size of 16 and maximum token length of 100; for the fusion-based model, we have used the batch size of 32. Additionally, as no validation set was given for the experiments, we divided the training data points into 85% and 15% split and used the 15% as a validation set. We predict the test set for the best validation performance. We have used HuggingFace[19] and PyTorch [20] for implementing all the models.

[4] https://huggingface.co/microsoft/codebert-base

Model	Accuracy	Macro-F1	F1(U)	P(U)	R(U)
BERT	89.810	87.595	82.352	80.405	**84.397**
BERT(PP)	<u>92.607</u>	<u>90.482</u>	<u>85.984</u>	<u>92.276</u>	80.496
BERT+CodeBERT	90.909	88.804	83.950	83.508	**84.397**
BERT+CodeBERT(PP)	**92.807**	**90.739**	**86.363**	**92.682**	<u>80.851</u>

Table 3
Performance Comparisons of All the Models. U: *Useful* Class. PP: Post-Processing. The best performance in each column is marked in **bold** and second best is <u>underlined</u>

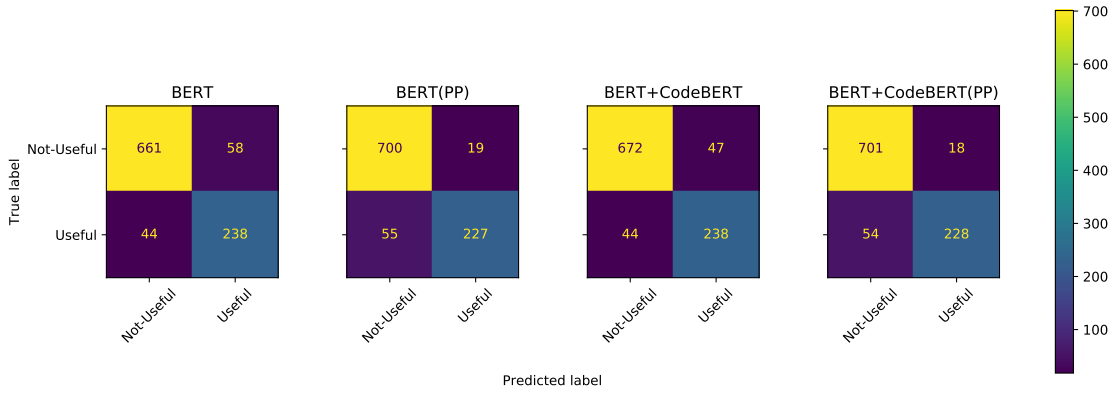

Figure 2: Confusion Matrix on Test Data for Each Model

5. Results

Table 3 demonstrates the performance of each model. As expected, the BERT+CodeBERT model(Accuracy: 90.909, Macro F1: 88.804), which utilizes features from code comments and snippets, performs better than the standalone BERT(Accuracy: 89.810, Macro F1: 87.595) model. Further, we observe post-processing improves the model's performance. In conclusion, BERT+CodeBERT(PP) model performed the best in terms of accuracy(92.807) and macro F1(90.739) score. We plot the confusion matrix in Figure 2 to further assess the models. Although post-processing makes the majority of the correction for Not-Useful classes; however, some of the Useful classes' test data points got misclassified, which further reduces the recall for Useful classes, as shown in Table 3. The observation holds true for both the unimodal text-based model and the fusion-based model. Nonetheless, the post-processing technique improves the overall performance.

While Majumdar et al. [6] achieved a macro F1 score of 86.34, our fusion-based techniques achieved a Macro F1 score of 90.73. Although one thing to keep in mind is that Majumdar et al. [6] performed the analysis considering three classes, and here we have two classes; therefore, the comparison is not entirely precise.

6. Conclusion

In this shared task, we deal with a novel problem of classifying **Useful** code comments given the surrounding code snippet. We evaluated the state-of-the-art transformer-based models. We found that the fusion-based model BERT+CodeBERT, which uses features from code comments and snippets, performs better than the standalone text-based BERT model. In the future, we plan to explore other transformer-based models, such as Roberta [21], CodeT5 [22], etc., for code understanding and useful comments detection. We also plan to explore knowledge graphs relevant to Software Engineering to improve the classification performance.

References

[1] S. Majumdar, S. Papdeja, P. P. Das, S. K. Ghosh, Comment-mine—a semantic search approach to program comprehension from code comments, in: Advanced Computing and Systems for Security, Springer, 2020, pp. 29–42.

[2] L. H. Etzkorn, C. G. Davis, L. L. Bowen, The language of comments in computer software: A sublanguage of english, Journal of Pragmatics 33 (2001) 1731–1756.

[3] Z. M. Jiang, A. E. Hassan, Examining the evolution of code comments in postgresql, in: Proceedings of the 2006 international workshop on Mining software repositories, 2006, pp. 179–180.

[4] B. Stroustrup, H. Sutter, C++ core guidelines, Web. Last accessed February (2018).

[5] S. Majumdar, A. Bandyopadhyay, P. P. Das, P. D Clough, S. Chattopadhyay, P. Majumder, Overview of the IRSE track at FIRE 2022: Information Retrieval in Software Engineering, in: Forum for Information Retrieval Evaluation, ACM, 2022.

[6] S. Majumdar, A. Bansal, P. P. Das, P. D. Clough, K. Datta, S. K. Ghosh, Automated evaluation of comments to aid software maintenance, Journal of Software: Evolution and Process 34 (2022) e2463.

[7] L. Pascarella, 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.

[8] D. Haouari, H. Sahraoui, P. Langlais, How good is your comment? a study of comments in java programs, in: 2011 International Symposium on Empirical Software Engineering and Measurement, IEEE, 2011, pp. 137–146.

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

[10] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, in: NAACL, 2019.

[11] 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).

[12] S. Banerjee, M. Sarkar, N. Agrawal, P. Saha, M. Das, Exploring transformer based models to identify hate speech and offensive content in english and indo-aryan languages, arXiv preprint arXiv:2111.13974 (2021).

[13] M. Das, P. Saha, R. Dutt, P. Goyal, A. Mukherjee, B. Mathew, You too brutus! trapping hateful users in social media: Challenges, solutions & insights, in: Proceedings of the 32nd ACM Conference on Hypertext and Social Media, 2021, pp. 79–89.

[14] M. Das, S. Banerjee, P. Saha, Abusive and threatening language detection in urdu using boosting based and bert based models: A comparative approach, arXiv preprint arXiv:2111.14830 (2021).

[15] L. Tan, D. Yuan, G. Krishna, Y. Zhou, /* icomment: Bugs or bad comments?*, in: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007, pp. 145–158.

[16] I. K. Ratol, M. P. Robillard, Detecting fragile comments, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 112–122.

[17] Y. Padioleau, L. Tan, Y. Zhou, Listening to programmers—taxonomies and characteristics of comments in operating system code, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 331–341.

[18] H. Aman, S. Amasaki, T. Yokogawa, M. Kawahara, Empirical analysis of words in comments written for java methods, in: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2017, pp. 375–379.

[19] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush, Huggingface's transformers: State-of-the-art natural language processing, 2020. arXiv:1910.03771.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, 2019. arXiv:1912.01703.

[21] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, Roberta: A robustly optimized bert pretraining approach, arXiv preprint arXiv:1907.11692 (2019).

[22] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.