Using Transformer-based Pre-trained Language Model for Automated Evaluation of Comments to Aid Software Maintenance

Debjyoti Paul[1,*,†], Bitan Biswas[2,*,†] and Rudrani Paul[3,***,†]

[1]*Data Scientist (Amazon), Independent Researcher*
[2]*Student at St. Xavier's College, Kolkata*
[3]*Software Engineer(TCS), Independent Researcher*

Abstract

With the increasing number of software applications, evaluating code repositories is of paradigm importance for building elegant software systems. The quality of a code repository is dependent on the readability of the code and the easiness with which it can be understood by other developers. Code commenting is a key requirement to ensure that code is readable, reusable, and reproducible. While useful comments can help software developers write better software, irrelevant and ambiguous comments can be overwhelming and misguiding to developers. Automatic software maintenance can help evaluate code repository and associated comments to provide meaningful insight into code quality and also help improve code comprehension by flagging not useful comments and highlighting useful ones. This paper explores the task of identifying code comment usefulness using a pre-trained transformer-based language model. The data for evaluation has been sourced from FIRE 2022, December 9-13, 2022, Kolkata, India[1]. The paper tries to understand how pre-trained language models like BERT and GPT-2 trained on large text corpora including code repositories can help understand comment usefulness. Such a model can help identify the comment usefulness with a minimum requirement of feature engineering and thus be integrated into an automatic software maintenance system to generalize across code repositories. The proposed model achieves an F1-score 90.15% and accuracy of 91.21% in the test set with 90.12% precision and 90.47% recall. The paper also explores explainable AI techniques like LIME and Attention visualizer to understand how these transformer-based models are identifying comment usefulness and establishing a sense of trust for these black box language models to be used in building automated software maintenance technologies.

Keywords

Transformers, GPT-2, BERT, Automatic Software Maintenance

Forum for Information Retrieval Evaluation, December 9-13, 2022, India
*Corresponding author.
†This work has been carried out independently by the author and is not part of Amazon's work and doesn't represent Amazon in any way. The ownership of opinion and research elucidated reside solely on the author and not any of his current or previous employer.
debjyoti93.paul@gmail.com (D. Paul); bitanbiswas99@gmail.com (B. Biswas); rudranipaul27@gmail.com (R. Paul)
https://github.com/dpaul0501/irse_codebert (D. Paul)


1. Introduction

Code comments should be such that developers need minimal additional context information to understand the code block or line the comment is referring to. Additional pointers can always be given informs of links/documents but too many hinder the flow of code readability. Useful comments are usually to the point and unambiguous in nature explaining the expected input and output of the line of codes for which the comment has been made. Comments which are ambiguous and irrelevant or too short are rendered not useful by developers. Not useful comments lead to poor software engineering experience and increase the time spent on code maintenance by many folds and a developer has to understand the code flow by repeated testing and writing customized unit tests. Such steps increase the time and effort spent on a given task.

Software development and maintenance tasks are one of the important and growing needs across the globe. A large chunk of a software developer's time is spent on improving and maintenance of existing code repositories. Often the person maintaining or performing bug fixes is not the same person who wrote the code in the first place. The process of code commenting is hence quite important to improve the readability of the code. Useful and to-the-point comments help maintenance tasks like feature addition and bug fixes easy as developers readily understand the code functionality and spend more time on designing the maintenance process.

With the improvement of predictive modeling techniques, automated systems can be made to perform different software maintenance tasks like code quality and comment quality monitoring. Recommending better code design or commenting can help augment software developers for better software maintenance.

To help promote research in the area of automated software maintenance using state of art language technology and information retrieval methods, FIRE 2022, December 9-13, 2022, Kolkata, India[1] [2] [3] has organized a binary classification task to classify source code comments as Useful or Not Useful for a given comment and associated code as input.

The complexity of software design lies in that software can be developed in various coding languages and platforms which have different syntactic styles leading to heterogeneous software design. Hand-engineered features while being straightforward and more intuitive, might not generalize well across variable coding languages and design as they might not capture the full complexity of features required. Since the software best practices and design principles are uniform everywhere, it is intended that the predictive model is designed in such a way that it has contextual information. Pre-trained large language models being trained on diverse corpora especially those trained on bimodal data of both natural language and programming language can generate more generalized feature vectors. But such language models are black boxes in nature and the explainability of the model is lost. In this research study, we attempt to find a solution for the same. The novel contributions made by this paper are the following.

1. Leveraging transformer-based architecture for building code usefulness predictive model without hand-engineered features
2. using explainable AI techniques to probe into the model for gaining insight into what constitutes a useful or not useful comment to instill a sense of trust in the modeling approach.

2. Related Studies

Application of machine learning techniques to automate and improve different software engineering tasks like source code analysis, software testing, coverage, and vulnerability analysis has been an active area of research [4]. Numerous researchers have applied predictive techniques to understand code quality. The quality of code depends on the readability and comprehension aspects and code comments play an intricate role in that. Since documentation and code share similar vocabulary, NLP and text-based models can be used for analysis and for automated software maintenance [5] [3]. [6] [2] [7] explore use of machine learning and text-based techniques for code comment usefulness prediction. [6] has performed in-depth feature analysis to understand what constitutes useful code comments and insights into irrelevant and inconsistent comments. [6] has also proposed a knowledge graph-based approach to represent finite code ontologies and correlate with the natural language and programming language mix vocabulary in code comments. Features like comment length, and number of code-related terms as compared to stop words are important markers of useful comments. Besides code comments, code reviews are an integral part of software development. [] explores the usage of NLP techniques for enforcing consistent use of identifiers.

Code reviews can help enforcement of software design best practices. To reduce the burden on human software developers to perform code reviews, automated software can be developed to perform code reviews as explored by [9] [10]. [11] explores the usage of language models for placement and log description in the code for uniform and meaningful log statements generated.

With the advancement of generative language model for text generation task, researchers have also explored the scope of code generation [12] [13] [14][15] . Such models can be useful for code-recommendation and completion tasks and guide developers to adhere to good programming principles. Text summarization techniques have been explored by [16] for effective documentation of source code.

3. Method

The main research question that is explored by this paper is if pre-trained language model based on transformer architecture can be fine-tuned for automatic evaluation of usefulness of code comment. The motivation of this research question comes from the fact that these pre-trained language model doesn't need explicit feature engineering and can generate contextual feature embedding for text across various domains. A basic exploration of the data reveals that the not useful comments are mostly around comments which doesn't contain any relevant information - like use of symbols to demarcate code blocks,. Fig1 shows the top common words associated with useful and not usefull class. Fig:2 shows the class distribution are almost balanced.

3.1. Modeling Approach

The code comment classification task consists of a unique challenge. The usefulness of code comments depends on the context of the code. While there is overlap between natural language (NL) and programming language (PL), there is syntactic difference in how they are used. Also, PL is a structured and finite vocabulary language as compared to NL which is unstructured.

	Common_words	count
1	/**********/	142
2	*	74
3	/*----------*/	72
4	file	68
5	/*The	59
6	data	57
7	use	56
8	/**********/	51
9	read	50

(a) Not Useful class

	Common_words	count
1	*	830
2	@param	407
3	The	316
4	-	272
5	data	230
6	*	218
7	file	210
8	function	196
9	=	195

(b) Useful class

Figure 1: Common Words associated with Useful and Not Useful class

Code comments contain a mix of PL and NL vocabulary and hence lead to complications of word sense disambiguation and understanding the contextual information.

In recent years, transformer [17]-based model has achieved a remarkable benchmark in NLP and text-related tasks as compared to RNN/LSTM [18]-based model. Pre-trained large language models based on this transformer architecture, (trained on large text corpus and have a large number of parameters, often in billions) have gained popularity as these models have been able to generalize across multiple downstream tasks in both classification and generation. Word and Sentence embedding technologies [19][20] have been at the forefront to solve NLP and text-related machine learning tasks as such techniques don't require explicit hand engineering of features. Such embeddings can be also fine-tuned for task-specific applications. The pre-trained transformer-based models leverage multi-head attention architecture to embed knowledge from large training corpora and embed knowledge using some pre-training tasks. This leads to the generation of superior embedding. This is evident from the benchmark achieved by these models. There are primarily two types of such pre-trained language models - autoencoding and autoregressive. BERT [21] is a bidirectional autoencoding based transformer model. Bert uses the encoder architecture of a transformer and consists of 12 Transformer encoding blocks with 12 self-attention heads and an embedding size of 768. BERT accepts a maximum of 512 tokens as input and accepts two special tokens [CLS] and [SEP]. [CLS] token is used to mark the beginning of the input token while [SEP] is used as a demarcating sentence boundary. For text classification tasks, BERT final hidden embedding of [CLS] token can be used as hidden representation and a classification layer on top of it can be used.

$$p(c|h) = softmax(W, h) \tag{1}$$

where W is the task-specific parameter matrix for the classification layer. The full set of Bert parameters are finetuned as part of the training process.

CodeBERT [15] is a language model based on Transformer based architecture that is trained on bimodal data of NL and PL. CodeBERT attempts to learn and bridge the gap between language

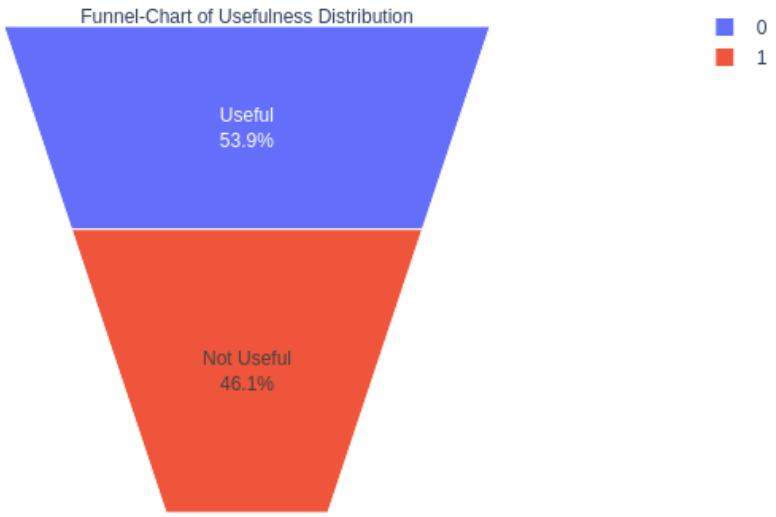

Figure 2: Useful and Not useful comment, class distribution

models for downstream tasks which contain both NL and PL tasks. This model is quite suitable for this application to understand the usefulness of code comments and can be also used to do automated code reviews or comment suggestions. CodeBert is based on Roberta [22] architecture which is an optimized and improved training to the vanilla Bert model. Roberta improves over Bert-base training by introducing CC-News dataset and removing the next sentence prediction task. It also tries to improve performance of Bert by training over longer sequences and larger batch sizes and dynamic masking.

GPT-2 [23] is an autoregressive generative model introduced by Open-AI. Unlike BERT, GPT-2 is built using the decoder architecture of the transformer. GPT-2 is trained on WebText corpus which consists of curated webpages from Reddit. Several studies [13] [12] [24] has proposed using GPT-2 for code generation task as well. Given Reddit contains lots of programming blogs, it might be a source of why GPT-2 has performed well on these tasks in the intersection of natural language and programming language.

The goal of this paper is to leverage these pre-trained language models described above for the task of code comment usefulness classification. Our assumption is that these language model being trained on large and diverse corpus possesses the context required for understanding the relationship between code and comment. Hence these models can be directly applied to obtain rich feature embedding without explicit engineering. We treat the Bert-base model as

a baseline, as it is mainly trained on NL vocabulary. Models like codeBert and GPT-2 having the context of both NL and PL can directly embed the relation between code and associated comment. This will make the modeling approach agnostic of a software system and coding language or platform. Given any code and comment, along with fine-tuning data, the modeling approach can adapt to the application.

3.2. Training Data Input Format

Each input sample consists of two feature strings: <Code string> - representing the code and <Comment text> which is the associated comment. For Bert based model we provide the input to the model in the format [CLS] <Comment text> [SEP] <Code string>. For GPT2 model the input format used is Comment: <Comment text> Code: <Code string>

Models are trained on training data and we do hyperparameter tuning on validation data for learning rate, batch size, and optimization technique. We have used AdamW as the optimization technique, with a learning rate of $10^{-}5$, and a batch size of 20 for the final model. We have reported the model performance on the test dataset provided as a part of the IRSE track. The code for all the proposed model and vizualizations are available through this text link

4. Results

The following table shows the comparison of the transformer-based models for code comment usefulness classification. Finetuned Code-Bert model performs slightly better when compared with Finetuned GPT-2. Both Code-Bert and GPT-2 perform significantly better than the Bert-Base model. Overall transformer-based architecture generalizes quite well. Using Code-Bert (Roberta) based model improves performances over Bert-base and it beats the performance achieved by GPT-2. Code-Bert model is explicitly pre-trained on both NL and PL and hence this improvement of the baseline is expected. Both Code-Bert and GPT-2 model outperforms the performance of Precision and Recall scores of For all the model we have used AdamW [25] optimizer, and cross entropy loss with Xavier initialization [**?**]. The classification layer is built on top of the top of average sentence vector as obtained from the final layer of transformer. The other parameters of the model are kept to as the standard configuration of each of the model used.

Model	Precision	Recall	F1-Score	Accuracy
BERT-Base	84.23%	84.19%	83.31%	84.16%
RoBERTa-Code-bert	90.12%	90.47%	90.15%	91.21%
GPT-2	88.86%	88.73%	88.69%	91.15%

While the transformer-based model doesn't require explicit feature engineering, given they are trained on a large corpus and the pre-training mechanism leads to the learning of generalized embedding, these models do lack explainability. When dealing with applications like software maintenance, model accuracy is not the only paradigm requirement. One might also need to understand why certain comments are deemed as not useful and what leads to useful commenting. One might also look at the task of generating code comments given codes. Existing explainable AI techniques like LIME [26], SHAP [27] and transformer attention visualization

[28] [29] can be leveraged to probe into such black box models and understand what kind of features the model is learning to achieve the classification task.

By studying the features learned by the transformer model one can also draw a parallel with the hand-engineered features-based model and generate a sense of trust in the transformer-based model's performance. We investigate the feature learned by the best performing model - Code-Bert. Our observations are similar when investigating the feature learned by GPT-2.

We sampled from the training data Useful and Not Useful comments and observed applied Lime text explainer. Fig:3 Fig:5 show the LIME interpretation for the model decision of useful comment where the original label was also useful comment Fig:4Fig:6. For the 1st useful comment Fig:4 sample coding construct like **memchunk** has been found relevance with **allocation** and hence the model predicted the comment as useful comment. Similarly, for the 2nd useful comment sample Fig:6 the coding ontology of beg: end is found to be related to length defined **(process length)** which shows that the comment and the associated code are contextually linked.

Fig:7 Fig:9 show the LIME interpretation for the model decision of useful comment where the original label was also useful comment 810. For the 1st not useful comment Fig:8 sample, the comment was mostly dashed lines and hence the model predicted the comment as not useful comment. Similarly for the 2nd not useful comment sample Fig:10 the coding term `NEW_ITY_DRIVER` is directly copied in the comment. A developer reading the code for the first time will have a difficult time understanding the purpose of the line of code from the such a short comment.

We also explored the attention visualization Fig:11 Fig:12 of intermediate BERT and GPT-2 layer. The token-token attention score can provide insight into features extracted by the transformer-based models. For Useful comment 1 Fig:11 we found that attention score are higher for words like allocation, initialize etc from comment text and **memchunk**, **malloc** from code chunk. Lower attention is received by unrelated code line **printf**. For Useful comment 1 Fig:12 we found that the attention score is lower for dashed line-based comment which doesn't add any relevant meaning to the code associated.

5. Conclusion

With the ever increasing demand of building technology solutions, maintaining of existing code bases is a surmounting challenge. Code commenting is an important step in any software development. Adopting best practices for commenting code leads to improve readability and comprehensiveness of the code . Since most of the time, a developer spend in maintaining existing code - refactoring, feature addition or bug fixes, relevant and informative comments can help improve productive of the programmer and better maintenance of code base. It can also increase adoption of similar code bases and thus can lead to reduction of repetitive effort. In this paper we have explored pre-trained language model for code usefulness classification. We have also explored explainable techniques like LIME and attention vizualization to probe into the model learned and understand what what are the insights for writing useful comments. Our analysis shows the these pre-trained model are able to capture the context of comment and correlate with the associated code. Having Programming language related data in pre-training

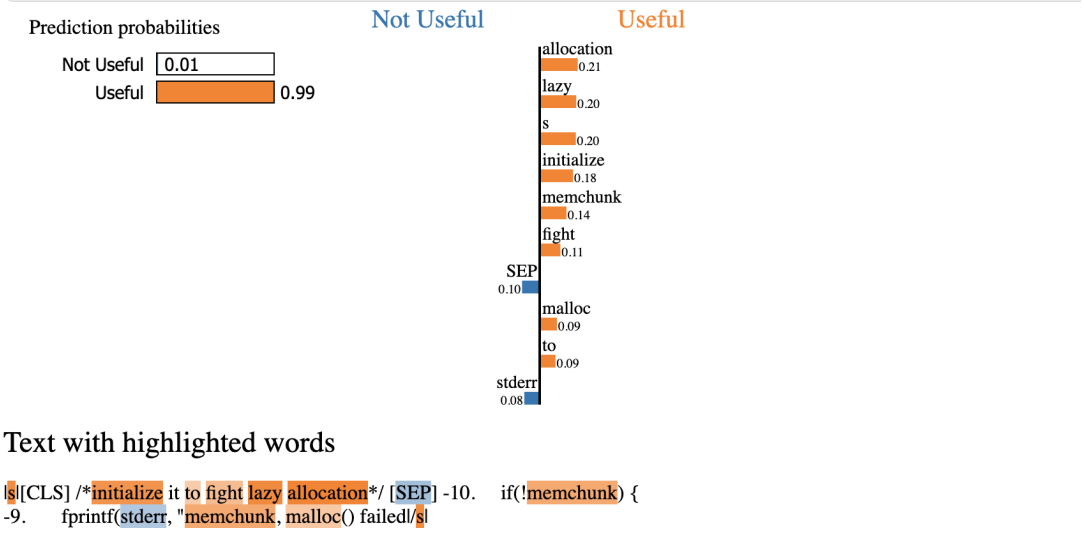

Figure 3: Lime Explanation for a random Useful comment sample 1

```
[CLS] /*initialize it to fight lazy allocation*/ [SEP] -10.     if(!memchunk) {
-9.       fprintf(stderr, "memchunk, malloc() failed\n");
-8.       nitems /= 2;
-7.     }
-6.   } while(nitems && !memchunk);
-5.   if(!memchunk) {
-4.     store_errmsg("memchunk, malloc() failed", errno);
-3.     fprintf(stderr, "%s\n", msgbuff);
-2.     return -4;
-1.   }

 /*initialize it to fight lazy allocation*/

1.   fprintf(stderr, "initializing memchunk array\n");
2.   for(i = 0; i < nitems; i++)
3.     memchunk[i] = -1;
```

Figure 4: Useful comment sample 1

helps in better contextual representation of the code and comment and association of the underlying vocabulary. In future research we would like to extend the model to new and more extensively used programming language like C++, Java and Python. We will also explore the application of model of comment suggestion given a code snippet.

References

[1] S. Majumdar, A. Bandyopadhyay, P. P. Das, P. D Clough, S. Chattopadhyay, P. Majumder, Overview of the IRSE subtrack at FIRE 2022: Information Retreival in Software Engineering, in: Working Notes of FIRE 2022 - Forum for Information Retrieval Evaluation, ACM, 2022.

[2] 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.

[3] S. Majumdar, S. Papdeja, P. P. Das, S. K. Ghosh, Comment-mine—a semantic search

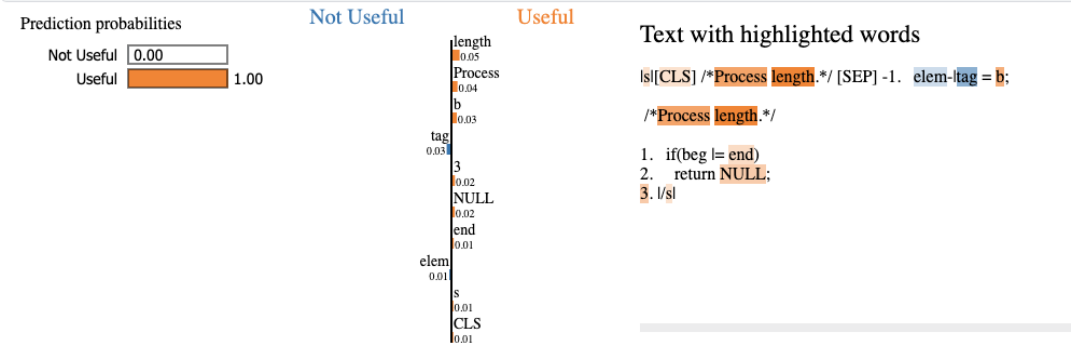

Figure 5: Lime Explanation for a random Useful comment sample 2

```
[CLS] /*Process length.*/ [SEP] -1.   elem->tag = b;

 /*Process length.*/

1.   if(beg >= end)
2.     return NULL;
3.   b = (unsigned char) *beg++;
4.   if(!(b & 0x80))
5.     len = b;
6.   else if(!(b &= 0x7F)) {
```

Figure 6: Useful comment sample 2

approach to program comprehension from code comments, in: Advanced Computing and Systems for Security, Springer, 2020, pp. 29–42.

[4] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, F. Sarro, A survey on machine learning techniques for source code analysis, arXiv preprint arXiv:2110.09610 (2021).

[5] N. Khamis, R. Witte, J. Rilling, Automatic quality assessment of source code comments: the javadocminer, in: International Conference on Application of Natural Language to Information Systems, Springer, 2010, pp. 68–79.

[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] X. Sun, Q. Geng, D. Lo, Y. Duan, X. Liu, B. Li, Code comment quality analysis and improvement recommendation: An automated approach, International journal of software engineering and knowledge engineering 26 (2016) 981–1000.

[8] S. K. Kumar, On weight initialization in deep neural networks, arXiv preprint arXiv:1704.08863 (2017).

[9] Y. Arafat, S. Sumbul, H. Shamma, Categorizing code review comments using machine learn-

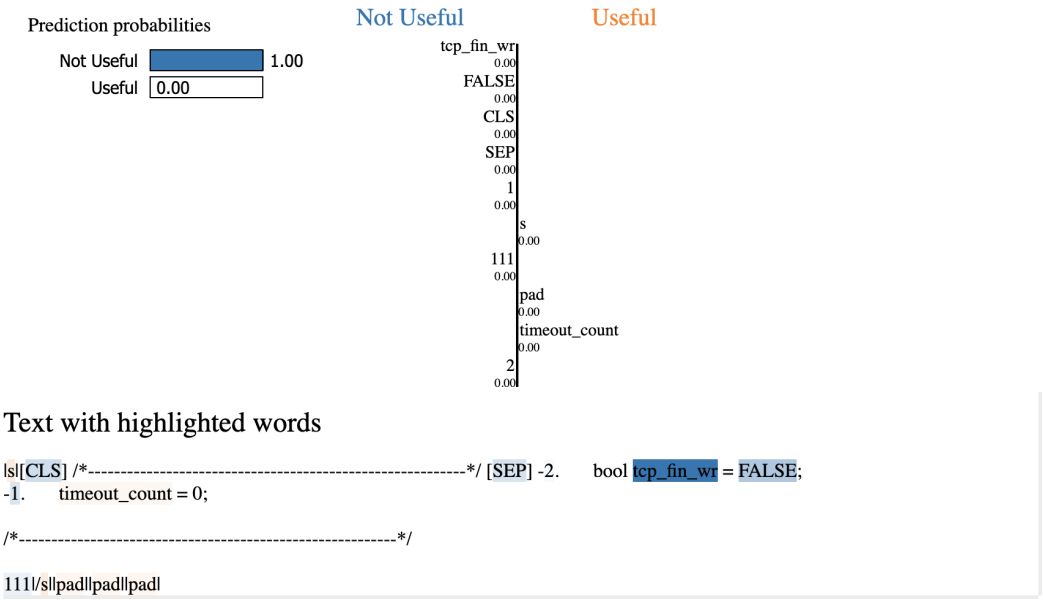

Figure 7: Lime Explanation for a random Not Useful comment sample 1

```
[CLS] /*----------*/ [SEP] -2.       bool tcp_fin_wr = FALSE;
-1.       timeout_count = 0;

/*----------*/

111
```

Figure 8: Not Useful comment sample 1

ing, in: Proceedings of Sixth International Congress on Information and Communication Technology, Springer, 2022, pp. 195–206.

[10] 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).

[11] S. Gholamian, P. A. Ward, Borrowing from similar code: A deep learning nlp-based approach for log statement automation, arXiv preprint arXiv:2112.01259 (2021).

[12] L. Perez, L. Ottens, S. Viswanathan, Automatic code generation using pre-trained language models, arXiv preprint arXiv:2102.10535 (2021).

[13] I. Paik, J.-W. Wang, Improving text-to-code generation with features of code graph on gpt-2, Electronics 10 (2021) 2706.

[14] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, G. Bavota, An empirical study on the usage of bert models for code completion, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 108–119.

[15] 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).

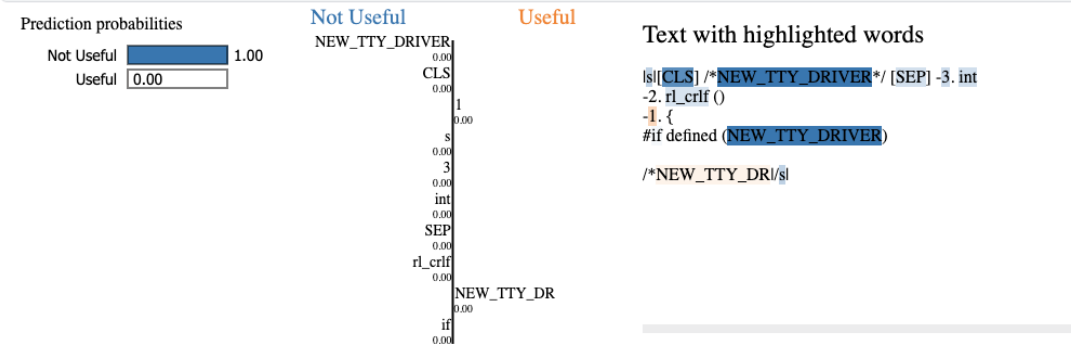

Figure 9: Lime Explanation for a random Not Useful comment sample 2

```
[CLS] /*NEW_TTY_DRIVER*/ [SEP] -3. int
-2. rl_crlf ()
-1. {
#if defined (NEW_TTY_DRIVER)

/*NEW_TTY_DRIVER*/

1.    if (_rl_term_cr)
2.      tputs (_rl_term_cr, 1, _rl_output_character_function);
```

Figure 10: Not Useful comment sample 2

[16] M. P. Arthur, Automatic source code documentation using code summarization technique of nlp, Procedia Computer Science 171 (2020) 2522–2531.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[18] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735–1780.

[19] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Advances in neural information processing systems 26 (2013).

[20] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.

[21] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).

[22] 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).

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., Language models are unsupervised multitask learners, OpenAI blog 1 (2019) 9.
[24] A. Svyatkovskiy, S. K. Deng, S. Fu, N. Sundaresan, Intellicode compose: Code generation using transformer, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1433–1443.
[25] I. Loshchilov, F. Hutter, Decoupled weight decay regularization, arXiv preprint arXiv:1711.05101 (2017).
[26] M. T. Ribeiro, S. Singh, C. Guestrin, " why should i trust you?" explaining the predictions of any classifier, in: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, 2016, pp. 1135–1144.
[27] S. M. Lundberg, S.-I. Lee, A unified approach to interpreting model predictions, Advances in neural information processing systems 30 (2017).
[28] J. Vig, Bertviz: A tool for visualizing multihead self-attention in the bert model, in: ICLR Workshop: Debugging Machine Learning Models, 2019.
[29] S. Abnar, W. Zuidema, Quantifying attention flow in transformers, arXiv preprint arXiv:2005.00928 (2020).

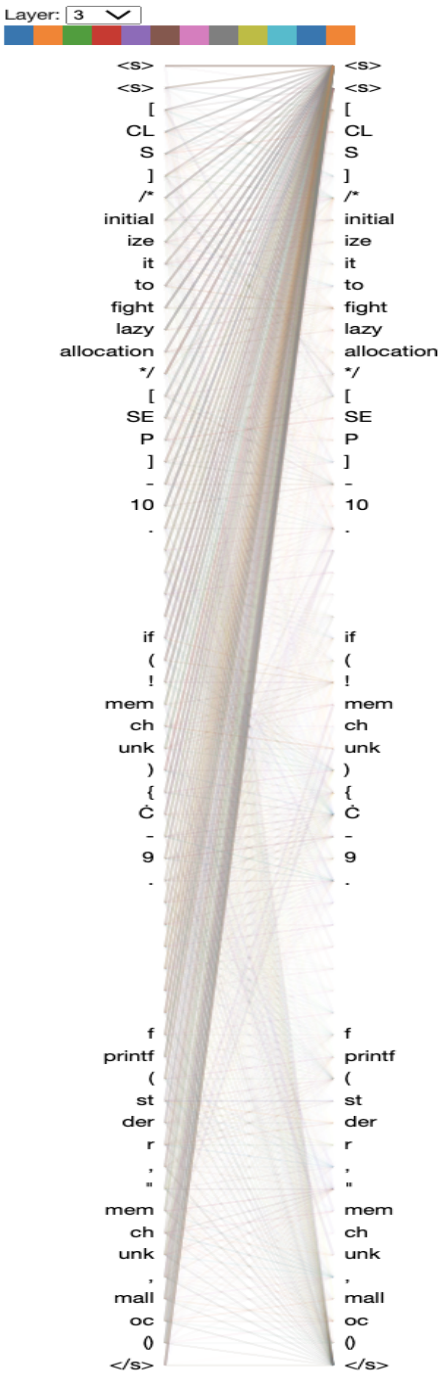

Figure 11: Attention Vizualization of Code-Bert Layer-3 for Useful Comment Sample 1

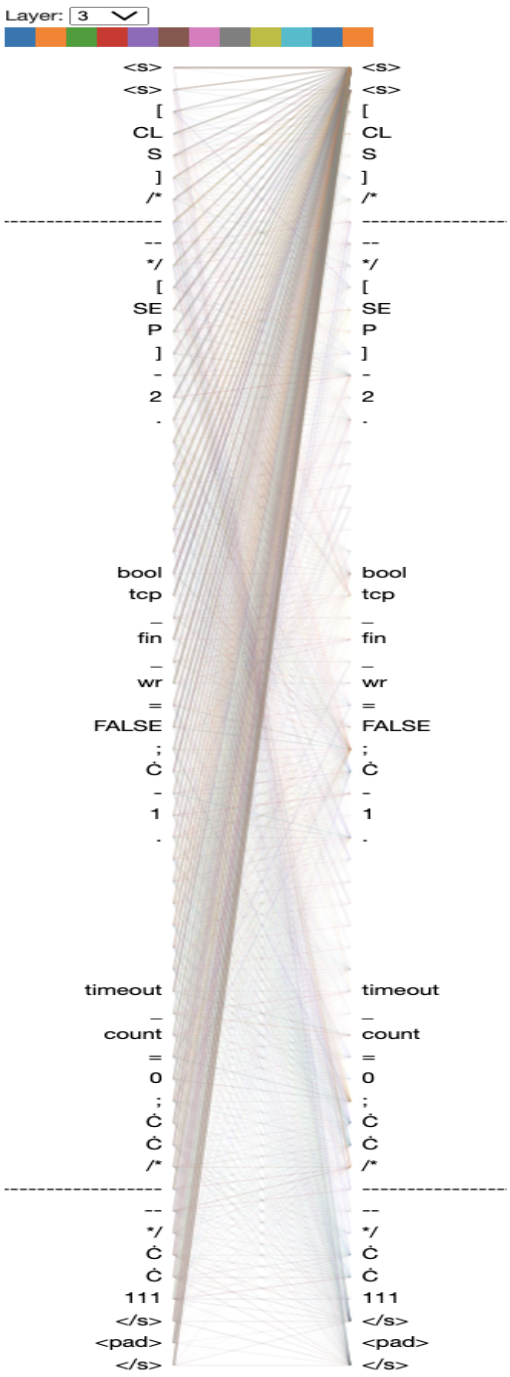

Figure 12: Attention Vizualization of Code-Bert Layer-3 for Not Useful Comment Sample 1