

A ML-LLM pairing for better code comment classification

Hanna Abi Akl^{1,2}

¹Data ScienceTech Institute (DSTI), 4 Rue de la Collégiale 75005 Paris, France

²Université Côte d'Azur, Inria, CNRS, I3S

Abstract

The "Information Retrieval in Software Engineering (IRSE) ¹" at FIRE 2023 shared task introduces code comment classification, a challenging task that pairs a code snippet with a comment that should be evaluated as either useful or not useful to the understanding of the relevant code. We answer the code comment classification shared task challenge by providing a two-fold evaluation: from an algorithmic perspective, we compare the performance of classical machine learning systems and complement our evaluations from a data-driven perspective by generating additional data with the help of large language model (LLM) prompting to measure the potential increase in performance. Our best model, which took second place in the shared task, is a Neural Network with a Macro-F1 score of 88.401% on the provided seed data and a 1.5% overall increase in performance on the data generated by the LLM.

Keywords

Natural Language Processing, Machine Learning, Information Retrieval, Large Language Models, Code Comprehension, Comment Quality

1. Introduction

In software development, code and documentation go hand-in-hand. Writing code is crucial to maintaining existing code bases, developing new features and fixing bugs. Documentation helps developers make sense of the logic behind written code and provides a steady set of guidelines to iterate over it [1]. Code commenting is a form of documentation whereby comments written in natural language are inserted in the code [1]. The advantage of this method is that it helps clarify parts of the code without affecting performance since comments are ignored by compilers [1]. It also provides an easy way to reflect updates on code changes without having to modify the entire documentation [1].

From the practice of writing code comments comes the challenge of identifying useful comments [2]. Writing comments is not always accurate science, and some comments can be outdated or ambiguous [2]. This can be problematic for developers who rely on these comments to understand and alter the code. There is then a real need for code comment

¹<https://sites.google.com/view/irse2023/home>


Forum for Information Retrieval Evaluation, December 15-18, 2023, India


* Forum for Information Retrieval Evaluation, December 15-18, 2023, India

*Corresponding author.

✉ hanna.abi-akl@dsti.institute (H. A. Akl)

ORCID [0000-0001-9829-7401](https://orcid.org/0000-0001-9829-7401) (H. A. Akl)

 © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

 CEUR Workshop Proceedings (CEUR-WS.org)

checking. This need has framed the task of collecting code comments from real projects in a code-comment database to aid in the task of classifying useful versus not useful comments [3]. Code comment classification is still a relatively new task that explores the possibility of accurately discriminating between comments that bring added value to the corresponding code and comments that are not pertinent with respect to the surrounding code [4].

Recent research has aimed to answer this challenge by compiling a semantic code-comment base by scraping and collecting code and surrounding comments from real projects in C [3]. Researchers have also explored applying machine and deep learning techniques to solve this binary classification problem by considering useful comments (i.e., informative of the surrounding code) as a class and non-useful comments (i.e., redundant, uninformative or ambiguous) as another [5, 6].

On the other hand, the rise of large language models (LLM) [7] and their ability to pose as a jack-of-all-trades by solving a wide range of machine learning and deep learning problems, coupled with their wealth of training data, make them an interesting entry point for the code comment classification task [8]. Based on the Transformers model [9], they are able to create robust embeddings from text, which helps them tackle problems based on natural language [10]. Another recent breakthrough in LLMs is in generative artificial intelligence, where users combine pre-trained models with different prompting techniques to generate output data (e.g., text) [11]. This prompting ability is at the heart of prompt engineering, a method that can redirect a LLM into focusing its generation on a specific need. This need can be in the form of answering specific questions, solving certain tasks (e.g., a classification problem) or even producing data in a pre-defined format [12]. The latter use case plays a detrimental role in data augmentation, whereby users can couple the power of LLMs with a pre-existing dataset to enrich it and overcome data scarcity [13, 14].

The IRSE at FIRE 2023 shared task proposes to measure the effects of leveraging LLMs in the context of solving the code comment classification problem [15]. Specifically, challengers are asked to use the generative capabilities of LLMs to enrich an existing dataset of code comments and compare the performance of classical machine learning models on the classification task before and after data augmentation [15].

In this paper, we show how prompting LLMs effectively can increase model performance on the code comment classification problem. The rest of the paper is organized as follows. In section 2, we discuss some of the related work. In section 3, we present the experimental setup. In section 4, we discuss the results. Finally, we present our conclusions in section 5.

2. Related Work

This section discusses some of the proposed strategies in the literature to classify code comments by quality.

2.1. Baseline models for code comment classification

Paul [16] leveraged classical machine learning models to solve the code comment classification task on a C language dataset of code and comments. They extracted text-level features like

comment length and comment position within the source code and found a comparable performance between a logistic regression and a support vector machine binary classifier [16]. Das and Chatterjee [17] studied the performance of deep learning models by proposing a fusion transformer system based on BERT and CodeBERT. Their system combined text-based features with dense embeddings and outperformed all other baseline models on the code comment classification task [17].

2.2. Embedding techniques for code comment classification

Basu et al. [18] compared both classical machine learning models and transformer-based models with different embedding techniques and found that the bag-of-words representation can outperform transformer-based embeddings on the code comment classification problem. Their findings could not be generalized and were limited by the size of the dataset they used for their runs [18]. Majumdar et al. [19] examined the effects of using embeddings to tackle the code comment pair classification challenge by developing and training a low-dimensional contextualized word embeddings model based on masked language models. The resulting model captured semantic code concepts better and resulted in a boost in their binary classification systems when compared to vanilla word embeddings models [19]. Other areas of research suggest an inclination toward specializing software engineering terms and building a domain vocabulary to produce more representative word models. Mishra and Sharma [20] proposed a methodology for crawling and scraping Wikipedia as a base for collecting software engineering terms. Gonzalez-Perez and Henderson-Sellers [21] laid the groundwork for the construction of such an ontology in terms of completeness, clarity, generalizability and extensibility. Simmons and Dillon [22] proposed an open-source architecture designed to act as both an ontology and a knowledge base meta-model for software development semantics.

3. Experiments

This section describes the framework of our experiments in terms of data, models and training process.

3.1. Dataset description

The dataset considered for this shared task is divided in two parts: a seed data provided by the task organizers and a LLM-generated dataset to complement it. We introduce both datasets in the following subsections.

3.1.1. Seed Data

The data provided by the task organizers consists of 11452 pairs of code and comments written in C, labeled as either Useful or Not Useful. The data contains 7063 Useful rows and 4389 Not Useful rows. The comments and surrounding code snippets are extracted from GitHub. For every comment, the label (Useful or Not Useful) was generated by a team of 14 annotators. Each comment was annotated by 2 annotators. Cohen's metric was used for inter-annotator

agreement with a kappa score of 0.734. The annotation process was supervised by weekly meetings and peer review sessions. Sample data is shown in Figure 1.

#	Comment	Code	Label
1	<code>/* uses png_malloc defined in pngpriv.h*/</code>	<code>/* uses png_malloc defined in pngpriv.h*/ PNG_FUNCTION(png_const_structp png_ptr) { if (png_ptr == NULL info_ptr == NULL) return; png_malloc(png_ptr); ...}</code>	Useful
2	<code>/* serial bus is locked before use */</code>	<code>static int bus_reset (. . .) /* serial bus is locked before use*/ { .. update_serial_bus_lock (bus * busR); }</code>	Not Useful
3	<code>// integer variable</code>	<code>int DeleteVendor; // integer variable</code>	Not Useful

Figure 1: Example of Seed Data

3.1.2. Data Augmentation

Participants are required to generate an additional dataset to complement the provided seed data. The generated dataset consists of code and comment pairs with labels generated using a LLM model of choice. In our experiments, we chose ChatGPT as our LLM and prompted it to generate data that aligns with the criteria of the given dataset, i.e., the generated code snippets should be written in the C programming language and the corresponding comments should be a mixture of useful and not useful. Additionally, we asked ChatGPT¹ to label each code-comment pair with the corresponding class (Useful or Not Useful). The ablation study performed on the LLM-generated dataset can be found in the Appendix. Using this method, we were able to generate 421 new code-comment pairs with 411 being labeled as Useful and 10 labeled as Not Useful by ChatGPT. Figure 2 shows an example output from ChatGPT.

3.2. System description

This section introduces the methodology used in our experimental runs. It describes the machine learning models as well as the features employed in our experiments.

3.2.1. Model Choice

Since the challenge explicitly limits participants to classical machine learning models (including neural networks but not extending to recurrent neural networks or more modern architectures like large language models), we based our experiments on 3 systems: Random Forest (RF), Voting Classifier (VC) and Neural Network (NN).

For the Random Forest model, we configured the following parameters: number of estimators = 100, criterion = gini, minimum samples split = 2, minimum samples leaf = 1, maximum features = sqrt, and bootstrap = True.

¹<https://chat.openai.com/share/6538a7f4-0a19-4e54-b5d8-d246dac3781a>



Figure 2: Example of ChatGPT Ggenerated Data

The Voting Classifier is based on 3 estimator models: a Random Forest, a Neural Network and a Linear SVC. The Random Forest has the following parameters: number of estimators = 100, criterion = gini, minimum samples split = 2, minimum samples leaf = 1, maximum features = sqrt, bootstrap = True. The Neural Network has the following configuration: number of hidden layers = 2, hidden layers sizes = (20,10), activation = relu, solver = adam, alpha = 0.0001, learning rate = constant, initial learning rate = 0.001, maximum iterations = 200, shuffle = True, tolerance = 0.0001, momentum = 0.9, nesterov's momentum = True, beta 1 = 0.9, beta 2 = 0.999, epsilon = 0.00000001. The Linear SVC is configured as follows: penalty = L2, loss = squared hinge, dual = True, tolerance = 0.0001, C = 1.0, fit intercept = True, maximum iterations = 1000. The voting strategy is set to hard.

The last model is a standalone Neural Network with the following configuration: number of hidden layers = 2, hidden layers sizes = (20,10), activation = relu, solver = adam, alpha = 0.0001, learning rate = constant, initial learning rate = 0.001, maximum iterations = 200, shuffle = True, tolerance = 0.0001, momentum = 0.9, nesterov's momentum = True, beta 1 = 0.9, beta 2 = 0.999,

epsilon = 0.00000001.

All models are implemented using the scikit-learn ² package in Python.

3.2.2. Features

For the feature engineering phase, we concatenate code-comment pairs and embed the resulting input strings. We use the flax-sentence-embeddings/st-codesearch-distilroberta-base ³ model trained with the Hugging Face sentence-transformers ⁴ library on the CodeSearchNet ⁵ dataset compiled from code and documentation strings in the Go, Java, Javascript, PHP, Python and Ruby programming languages [23]. The result is one 768 dimensional embedding vector for every code-comment input string. These embeddings constitute our final feature set and are fed to the different models.

3.2.3. Experimental Setup

We divide our experiment in two phases: seed data run and seed + LLM data run. The setup is identical for both phases and the only difference is the input data used. In the seed data run, only the seed data provided by the task organizers is used to assess model performance. In the seed + LLM data run, the data generated by ChatGPT is added to the seed data and the resulting augmented dataset is used as the input for our models.

In both phases, analyzing the data at our disposal shows a class imbalance where the Useful class is over-represented at 61.6% in the seed data and 97.6% in the LLM-generated data. We use the SMOTE [24] technique to balance the datasets and restore class parity by synthetically generating rows of Not Useful data to achieve a 50-50 percent class distribution.

Next, we split our data using the scikit-learn Repeated Stratified K-Fold cross validator ⁶ with 10 folds and 3 allowed repetitions. We use the Accuracy, Precision, Recall and F1 scores as metrics for evaluating our models. All experiments are performed on a Dell G15 Special Edition 5521 hardware with 14 CPU Cores, 32 GB RAM and NVIDIA GeForce RTX 3070 Ti GPU.

4. Results

Table 1 demonstrates the performance of each model on the seed data. For each scoring metric, the best score is marked in bold for both the Useful (U) and Not Useful (NU) classes. On the majority of the scoring metrics, the Neural Network outclasses the Random Forest and the Voting Classifier models. The Voting Classifier outperforms the Neural Network on the Recall of the Useful class and the Precision of the Not Useful class which shows that applying different non-linear models together can compensate for the shortcomings of one model's blind spots and classify more instances of Useful and Not Useful data correctly.

²<https://scikit-learn.org/stable/index.h>

³<https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base>

⁴<https://huggingface.co/sentence-transformers>

⁵https://huggingface.co/datasets/code_search_net

⁶https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedStratifiedKFold.html

The results of Table 2 are consistent with these findings. The Neural Network model is the overall best model since it outperforms the other systems in 5 scoring metrics out of 8 over both classes, while the Voting Classifier retains the best scores in F1 (U), Recall (U) and Precision (UN). We also note that the scores are consistently high for both classes, which is in large part helped by the SMOTE data augmentation technique.

Having balanced both classes in our experiments allows us to have a better baseline when measuring the impact of the additional data generated by ChatGPT. By comparing the scores of Tables 1 and 2, we see that fixing the models and augmenting the data yields a 1.5% increase in scores overall. Particularly, this solidifies the claim that the data generated by the LLM aligns with the data expected for this challenge and can further aid in solving it.

Model	Macro-F1 (U)	Precision	Recall	Accuracy	Macro-F1 (NU)	Precision	Recall	Accuracy
RF	84.727	83.263	86.257	84.454	84.168	85.758	82.651	84.454
VC	88.133	88.071	88.215	88.123	88.111	88.211	88.031	88.123
NN	88.401	89.484	87.380	88.536	88.664	87.692	89.693	88.536

Table 1
Model performance comparison on seed data

Model	Macro-F1 (U)	Precision	Recall	Accuracy	Macro-F1 (NU)	Precision	Recall	Accuracy
RF	85.587	84.397	86.818	85.381	85.168	86.438	83.943	85.381
VC	88.539	88.705	88.395	88.560	88.578	88.454	88.725	88.560
NN	88.489	90.007	87.066	88.678	88.856	87.506	90.290	88.678

Table 2
Model performance comparison on seed + LLM data

5. Conclusion

In this shared task, we evaluate the impact of generating LLM data to improve model performance. We explore the effects of this data generation by augmenting the existing code comment dataset and measuring the increase in the model classification scores. In the future, we plan to incorporate other data generation mechanisms such as ontology or knowledge graph integration into our LLM prompting technique to further our study of the impact of a refined data augmentation pipeline on classification performance.

References

- [1] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 1 ed., Prentice Hall PTR, USA, 2008.

- [2] Y. Shinyama, Y. Arahori, K. Gondow, Analyzing code comments to boost program comprehension, in: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2018, pp. 325–334.
- [3] S. Majumdar, S. Papdeja, P. P. Das, S. K. Ghosh, Comment-mine—a semantic search approach to program comprehension from code comments, *Advanced Computing and Systems for Security: Volume Twelve (2020)* 29–42.
- [4] FIRE '22: Proceedings of the 14th Annual Meeting of the Forum for Information Retrieval Evaluation, Association for Computing Machinery, New York, NY, USA, 2022.
- [5] S. Majumdar, A. Bandyopadhyay, P. P. Das, P. Clough, S. Chattopadhyay, P. Majumder, Can we predict useful comments in source codes?-analysis of findings from information retrieval in software engineering track@ fire 2022, in: Proceedings of the 14th Annual Meeting of the Forum for Information Retrieval Evaluation, 2022, pp. 15–17.
- [6] K. Ghosh, A. Senapati, Information retrieval in software engineering utilizing a pre-trained bert model (2022).
- [7] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, J.-R. Wen, A survey of large language models, 2023. [arXiv:2303.18223](https://arxiv.org/abs/2303.18223).
- [8] J. Kocoń, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydło, J. Baran, J. Bielaniewicz, M. Gruza, A. Janz, K. Kanclerz, et al., Chatgpt: Jack of all trades, master of none, *Information Fusion (2023)* 101861.
- [9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, 2023. [arXiv:1706.03762](https://arxiv.org/abs/1706.03762).
- [10] 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](https://arxiv.org/abs/1910.03771).
- [11] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, G. Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, 2021. [arXiv:2107.13586](https://arxiv.org/abs/2107.13586).
- [12] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, J. Ba, Large language models are human-level prompt engineers, 2023. [arXiv:2211.01910](https://arxiv.org/abs/2211.01910).
- [13] C. Whitehouse, M. Choudhury, A. F. Aji, Llm-powered data augmentation for enhanced crosslingual performance, 2023. [arXiv:2305.14288](https://arxiv.org/abs/2305.14288).
- [14] J. Yuan, R. Tang, X. Jiang, X. Hu, Large language models for healthcare data augmentation: An example on patient-trial matching, 2023. [arXiv:2303.16756](https://arxiv.org/abs/2303.16756).
- [15] S. Majumdar, S. Paul, D. Paul, A. Bandyopadhyay, B. Dave, S. Chattopadhyay, P. P. Das, P. D. Clough, P. Majumder, Generative ai for software metadata: Overview of the information retrieval in software engineering track at fire 2023, in: *Forum for Information Retrieval Evaluation*, ACM, 2023.
- [16] S. Paul, Source code comment classification using logistic regression and support vector machine (2022).
- [17] M. Das, S. Chatterjee, Exploring transformer-based models for automatic useful code comments detection (2022).
- [18] T. Basu, et al., Identification of the relevance of comments in codes using bag of words

- and transformer based models, arXiv preprint arXiv:2308.06144 (2023).
- [19] S. Majumdar, A. Varshney, P. P. Das, P. D. Clough, S. Chattopadhyay, An effective low-dimensional software code representation using bert and elmo, in: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2022, pp. 763–774.
 - [20] S. Mishra, A. Sharma, Crawling wikipedia pages to train word embeddings model for software engineering domain, in: 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference), 2021, pp. 1–5.
 - [21] C. Gonzalez-Perez, B. Henderson-Sellers, An ontology for software development methodologies and endeavours, in: Ontologies for software engineering and software technology, Springer, 2006, pp. 123–151.
 - [22] G. L. Simmons, T. S. Dillon, Towards an ontology for open source software development, in: IFIP International Conference on Open Source Systems, Springer, 2006, pp. 65–75.
 - [23] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, 2020. arXiv:1909.09436.
 - [24] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, SMOTE: Synthetic minority over-sampling technique, Journal of Artificial Intelligence Research 16 (2002) 321–357. URL: <https://doi.org/10.1613/jair.953>. doi:10.1613/jair.953.

A. Appendix

This section presents the ablation study done on the ChatGPT prompt to understand its inherent data generation mechanisms. Overall, we asked ChatGPT to provide 3000 additional lines of code-comment data split equally between Useful and Not Useful.

From the generated dataset, we’ve had to trim down rows for duplicates (i.e., the same code-comment pairs generated multiple times), incompleteness (i.e., the generation was missing parts of the code snippet or comment) and ambiguity (i.e., the code was not written in proper C language or the comment was not in proper English). These rules ensure the generated dataset mimics as faithfully as possible the original seed data with human supervision.

Additionally, we’ve attempted to retro-engineer ChatGPT to explain the process of its generated data. The following extract transcribes the code-comment generation mechanism adopted by the LLM.

User: *can you explain how you came up with these examples?*

ChatGPT: *Certainly! The examples were generated programmatically using a combination of randomization and predefined patterns. Here’s a breakdown of the process:*

1. *Randomization: To create random lines of C code, I utilized random number generators available in programming languages. For this task, I used the built-in random number generator provided by the programming environment. The generator is initialized with a seed (which may be the system time), and it generates random numbers within a specified range. This allowed me to*

create random integers for variables, loop limits, and other numerical values.

2. Predefined Patterns: I used predefined patterns to structure the C code snippets. These patterns include simple variable declarations, conditional statements, loops, arithmetic operations, function calls, and string operations. By combining these patterns with the randomly generated numbers, I formed meaningful C code lines.

3. Comments: For the comments, I used a mixture of random phrases and predefined descriptions. Half of the comments were matched with the corresponding lines of code to provide meaningful explanations of the code's purpose or functionality. The other half of the comments were randomly generated phrases, ensuring a mix of relevant and unrelated comments.

4. Iteration: To generate 3000 examples, I repeated the randomization and predefined patterns generation process 3000 times. Each iteration produced a unique line of C code along with a comment, resulting in a diverse set of examples.

It's important to note that these examples are randomly generated and may not necessarily represent valid or complete programs. The goal was to provide a mix of code snippets and comments to showcase various C language constructs and coding patterns, as well as to demonstrate the use of comments for code documentation. The examples are intended for illustrative purposes and should not be used in a production environment without proper validation and testing.