

Dopamin: Transformer-based Comment Classifiers through Domain Post-Training and Multi-level Layer Aggregation

Nam Le Hai
namlh35@fpt.com
FPT Software AI Center
Hanoi, Vietnam

Nghi D. Q. Bui
nghi.bui@fulbright.edu.vn
Fulbright University
Ho Chi Minh City, Vietnam

ABSTRACT

Code comments provide important information for understanding the source code. They can help developers understand the overall purpose of a function or class, as well as identify bugs and technical debt. However, an overabundance of comments is meaningless and counterproductive. As a result, it is critical to automatically filter out these comments for specific purposes. In this paper, we present **Dopamin**, a Transformer-based tool for dealing with this issue. Our model excels not only in presenting knowledge sharing of common categories across multiple languages, but also in achieving robust performance in comment classification by improving comment representation. As a result, it outperforms the STACC baseline by 3% on the NLBSE'24 Tool Competition dataset in terms of average F1-score, while maintaining a comparable inference time for practical use. The source code is publicly available at <https://github.com/FSoft-AI4Code/Dopamin>.

ACM Reference Format:

Nam Le Hai and Nghi D. Q. Bui. 2024. Dopamin: Transformer-based Comment Classifiers through Domain Post-Training and Multi-level Layer Aggregation. In *2024 ACM/IEEE International Workshop on NL-based Software Engineering (NLBSE '24)*, April 20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3643787.3648044>

1 INTRODUCTION

In the intricate world of software development, source code comments play a crucial role, serving as the backbone of applications by elucidating the functionality and intent behind code segments. These comments vary widely in their utility, ranging from summarizing the purpose of functions or classes, aiding in code maintenance, to identifying instances of technical debt, as highlighted in studies like [7, 9, 12]. However, not all code comments are equally beneficial, or different information in comments can be used for different development tasks. With the increasing complexity of software projects, the task of discerning valuable comments from the less pertinent ones has become more critical.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

NLBSE '24, April 20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0576-2/24/04
<https://doi.org/10.1145/3643787.3648044>

Table 1: NLBSE'24 Tool Competition dataset properties

Language	Number of categories	Number of data per category
Java	7	10.555
Python	5	2.555
Pharo	7	1.765

Current tools for comment classification, as discussed in references like [1, 10, 11], predominantly focus on categorizing comments based on their apparent utility in the coding workflow. Despite their effectiveness, these tools often lack the nuanced understanding needed to differentiate between subtly different types of comments, limiting their practical utility. Recognizing this gap, this paper introduces **Dopamin** - a Transformer-based Comment Classifier that utilizes a code language model for an enhanced classification process.

Dopamin takes a novel approach, relying on domain post-training on diverse comment types across various coding environments, and adopting a multi-level layer aggregation strategy inspired by Karimi et al. [5]. The post-training procedure incorporates data from all programming languages, facilitating knowledge transfer across different languages and leveraging the high-resource language (Java) to improve the less resource-intensive languages (Python). Meanwhile, layer aggregation methodology enables Dopamin to not only classify comments but also understand the nuanced semantic information they carry, as the higher layers in BERT are adept at capturing intricate semantic features, a concept supported by Jawahar et al. [3]. By doing so, Dopamin significantly improves the relevance and accuracy of comment classification, catering to the evolving complexity of software development.

The efficacy of Dopamin is evident in our experimental results, where it achieves an F1-score of 0.74, surpassing the 0.71 F1-score of the existing STACC [1] baseline. Through Dopamin, we aim to redefine the standards in code comment classification, providing a tool that is effective in distinguishing various types of comments on multiple programming languages.

2 DATA PREPARATION

In this section, we illustrate the NLBSE'24 Tool Competition dataset introduced by Kallis et al. [4], detailing our approach to processing comments and splitting the training data for model selection.

2.1 Dataset statistic

The competition provided binary comment classification data of three languages (Python, Java, and Pharo). In total, there are 19 categories corresponding to 19 classifiers required to build. Overall

Table 2: Different model results on Validation set

Model	Precision	Recall	F1
CodeBERT	0.7921	0.8438	0.8133
RoBERTa	0.7900	0.8304	0.8063
ALBERT	0.6695	0.7769	0.7108

information of the dataset is shown in Table 1 and details on the competition repository¹.

2.2 Data preprocess

Input feature: Follow Al-Kaswan et al. [1], we concatenate the class name and comment sentence to serve as the input to the model, employing "</s>" as the separator between them.

Data splitting: The training data is divided into a training set and a validation set for refining the optimal model. Instead of employing the validation set for hyperparameter tuning, it is utilized to choose the best checkpoint during the training process. Due to the modest amount of data, we select only 10% of the training set of each category to serve as the validation set. We employ stratified sampling to maintain the label distribution in both sets.

3 METHODOLOGY

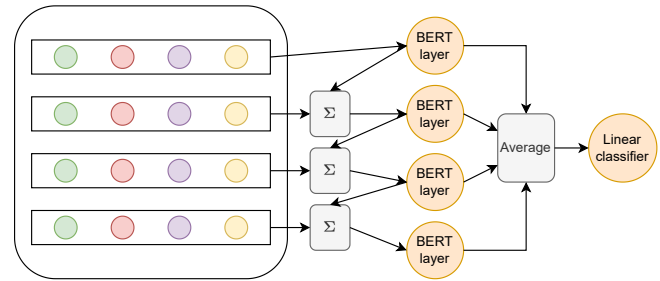
This section describes the Dopamin methodology, including model selection, the methodology for obtaining the optimal checkpoint, domain post-training procedures, and layer aggregation techniques.

3.1 Model selection

We investigate several candidates as the backbone model. Since the primary source of classification information comes from comments, which are in natural language form, we choose RoBERTa [8] and ALBERT [6] as candidates. Additionally, the comments primarily contain syntax related to the coding domain. Therefore, we are also considering CodeBERT [2], which is a language model pretrained on large code corpus, as an option. We opt for these architectures because they are based on Transformer encoders, commonly employed for classification tasks. Additionally, the base versions of these models share the same size as the baseline (STACC), resulting in fair comparison and no additional overhead in inference time, which is a key consideration for evaluation score. The performance of each model on the validation set is presented in Table 2. As a result, we select CodeBERT - the model that achieves the best F1 score on the validation set, as the backbone model.

3.2 Domain post-training

Table 1 shows that Python and Pharo have much fewer examples compared to Java. Besides, some categories are contained in both Java and Python languages such as *Expand*, *Summary*, and *Usage*. Therefore, we combined the data of all languages to finetune the CodeBERT backbone model in the data domain (post-training), facilitating the transfer of knowledge across languages before individual training of models for each category in the target domain.

**Figure 1: Hierarchical aggregation**

During the post-training procedure, we did not concatenate the class name and comment sentence as input to the model as mentioned in Section 2.2. Instead, we concatenate the category to the comment sentence since the model is required to predict for all categories, necessitating the inclusion of category information in the input.

Following the post-training of the model in the target domain, we utilize it as the initial state for the model to be trained individually for each category using the procedure in Section 3.4 and input feature in Section 2.2.

3.3 Multi-level layer aggregation

Jawahar et al. [3] previously showed that the upper layers in BERT yield rich semantic features of linguistic information and distinct layers can exhibit distinct capabilities in encoding semantic information. Hence, combining these layers can obtain a more comprehensive representation for the input text. Therefore, we adopt the Hierarchical aggregation (HSUM) introduced by Karimi et al. [5] to enrich the comment representation. The illustration of HSUM is shown in Figure 1. Specifically, we combined the top four layers of the model to obtain the final representation for comment.

3.4 Optimal checkpoint

Given the high cost of hyperparameter search for 19 categories, we choose to **keep the hyperparameters constant** throughout the training process for each category. Instead, we use the validation set to determine the best checkpoint step. This strategy is similar to early stopping, which aims to prevent the model from overfitting. We employ it as a heuristic to determine the optimal step for the stage of training the model on the original training set without validation. Specifically, there are two stages in the training process.

- Stage 1: the model is trained on the training set and the best checkpoint (*optimal_step*) is obtained based on the F1-score of the validation set.
- Stage 2: Considering the limited amount of data, training the model on the entire original training set is essential. After obtaining the *optimal_step* in stage 1, we train the model on the original training set and acquire the final model at step *optimal_step* + *extra_steps*. The *extra_steps* represents the additional steps due to the incorporation of more data during training.

For example, after *stage 1* the model for the *Java - Deprecation* category attains its highest F1 score at step 200. The *extra_steps* is set to 100, thus after training on the full original dataset in *stage*

¹<https://github.com/nlbse2024/code-comment-classification>

2, we obtain the checkpoint at step 300 as the final model for the *Deprecation* category of Java.

4 EXPERIMENT SETUP

4.1 Training hyperparameters

For reproducibility, we set the random seed as 0. The hyperparameters are selected based on references from prior studies [2, 6, 8] that involved fine-tuning models on downstream tasks.

- **Post-training stage:** In this stage, we train the backbone model during 10 epochs with the learning rate of $2e - 5$, batch size of 64, and evaluation step of 500.
- **Individual classifier training stage:** Each model is trained for 10 epochs for Java categories and 20 epochs for Python or Pharo categories. We assign a lower number of epochs to Java due to its higher volume of training examples, resulting in a greater number of training steps within a single epoch. The learning rate is set to $1e-5$, the batch size is 64, and the evaluation step is 50. We use the *extra_steps* of 100 for all categories.

4.2 Implementation

We use the HuggingFace transformers² and PyTorch packages³ to implement Dopamin. All experiments are conducted using two Nvidia A100 GPUs with 80GB of VRAM. During the evaluation on the provided test set, we utilize Google Colab T4, adhering to the competition’s specifications, to acquire the inference time.

4.3 Metrics

For evaluation, we employ the metrics outlined by the competition, considering a category c . Specifically, we calculate the recall R_c , precision P_c , and F1 score $F1_c$ for each category. These metrics are defined as follows:

$$P_c = \frac{TP_c}{TP_c + FP_c}, R_c = \frac{TP_c}{TP_c + FN_c}, F1_c = 2 \cdot \frac{P_c \cdot R_c}{P_c + R_c}$$

in which, TP_c , FP_c , and FN_c are the true positives, false positives, and false negatives for a category c , correspondingly.

Finally, the submission score of the competition using both the average F1-score and the inference time is defined as:

$$\text{submission_score} = 0.75(\text{avg.F1}) + 0.25 \frac{\text{max_avg_runtime} - \text{measured_avg_runtime}}{\text{max_avg_runtime}}$$

5 EXPERIMENTAL RESULTS

In this section, we present the performance comparison of Dopamin against the STACC baseline. Table 4 shows the performance comparison between Dopamin and STACC across various categories and languages demonstrates Dopamin’s enhanced capabilities in comment classification. Below is a summary of the key findings from the table.

5.0.1 Overall Performance. Dopamin attains a comprehensive F1-score of **0.74**, balancing Precision at 0.73 and Recall at 0.75. This marks an enhancement compared to STACC’s overall F1-score of

²<https://huggingface.co/docs/transformers/index>

³<https://pytorch.org/>

Table 3: Ablation study on our proposed components: Domain Post-training (PoT) and Layer aggregation (LA). All the models use CodeBERT as the backbone.

PoT	LA	Validation set			Test set		
		Precision	Recall	F1	Precision	Recall	F1
✓	✓	0.844	0.905	0.869	0.735	0.745	0.738
✓	✗	0.851	0.892	0.866	0.712	0.729	0.717
✗	✓	0.820	0.824	0.820	0.727	0.720	0.719
✗	✗	0.792	0.844	0.813	0.728	0.713	0.714

0.71. Moreover, Dopamin incurs no additional inference time overhead compared to STACC. Consequently, the submission score reaches **0.703** compare to 0.675 of STACC baseline. Overall, the result indicates a consistent and significant enhancement in classification effectiveness.

5.0.2 Performance by Language.

- **Java:** Dopamin performs better in categories like *Pointer*, *Summary*, *Ownership*, *Rational*, and *Usage*, with notable improvements in F1-scores. For instance, Dopamin achieves F1-score at 0.90 compared to STACC’s 0.78 in the *Summary* category.
- **Pharo:** Dopamin shows improved performance in categories like *Classreferences*, *Collaborators*, and *Example*. For example, in the *Classreferences* category, Dopamin’s F1-score is 0.68 compared to STACC’s 0.52. However, Dopamin exhibits a weakness in 4 out of 7 categories when compared to STACC in this language. The explanation might stem from Pharo being the language with the lowest data volume per category (under 2000). Given the limited data, the few-shot approach (STACC) appears more effective than fine-tuning with classification loss (Dopamin). Moreover, the absence of overlap categories between Pharo and other languages could restrict the advantages of the Post-training stage.
- **Python:** Dopamin also outperforms STACC in the Python category, particularly in *Parameters*, *Summary*, and *Usage* categories. These categories overlap with those in Java, suggesting the effectiveness of knowledge transfer during our Post-training process.

5.0.3 Category-Specific Performance. Dopamin excels particularly in categories where understanding the context and semantics is crucial, like *Summary*, *Usage*, and *Ownership*. In some categories, like *Java - Expand* and *Pharo - Keyimplementationpoints*, Dopamin’s performance is lower than STACC. This suggests room for further optimization in certain specific categories.

In summary, Dopamin generally exhibits a balanced improvement in both precision (P_c) and recall (R_c) across various categories. For example, in the *Java - Usage* category, Dopamin shows a precision of 0.87 and recall of 0.95, compared to STACC’s 0.64 and 0.92, respectively. Dopamin also demonstrates a notable improvement over STACC in most categories and languages, reflecting its advanced capability in understanding and classifying code comments.

6 ABLATION STUDY

In this section, we present the experimental outcomes obtained by systematically trimming individual components within Dopamin,

Table 4: Performance of Dopamin against the STACC baseline

Language	Category	STACC			Dopamin			$\Delta F1_c$
		P_c	R_c	$F1_c$	P_c	R_c	$F1_c$	
Java	Deprecation	0.81	0.94	0.87	0.81	0.88	0.85	-0.02
Java	Pointer	0.82	0.78	0.80	0.89	0.81	0.85	+0.05
Java	Summary	0.93	0.67	0.78	0.94	0.87	0.90	+0.12
Java	Expand	0.57	0.73	0.64	0.52	0.63	0.57	-0.07
Java	Ownership	0.97	1.0	0.99	1.0	0.99	1.0	+0.01
Java	Rational	0.49	0.51	0.5	0.49	0.59	0.54	+0.04
Java	Usage	0.64	0.92	0.76	0.87	0.95	0.91	+0.15
Pharo	Classreferences	0.47	0.57	0.52	0.76	0.62	0.68	+0.16
Pharo	Example	0.93	0.89	0.91	0.93	0.93	0.93	+0.02
Pharo	Keyimplementationpoints	0.69	0.79	0.73	0.5	0.69	0.58	-0.15
Pharo	Collaborators	0.36	0.91	0.51	0.57	0.64	0.60	+0.09
Pharo	Intent	0.87	0.89	0.88	0.87	0.85	0.86	-0.02
Pharo	Keymessages	0.79	0.91	0.85	0.86	0.81	0.83	-0.02
Pharo	Responsibilities	0.67	0.63	0.65	0.59	0.62	0.61	-0.04
Python	Developmentnotes	0.43	0.54	0.48	0.45	0.44	0.44	-0.04
Python	Parameters	0.78	0.86	0.81	0.88	0.85	0.86	+0.05
Python	Summary	0.62	0.64	0.63	0.8	0.69	0.74	+0.11
Python	Expand	0.52	0.56	0.54	0.51	0.53	0.52	-0.02
Python	Usage	0.69	0.77	0.73	0.72	0.79	0.76	+0.03
Overall		0.69	0.76	0.71	0.73	0.75	0.74	+0.03

showcasing the efficacy of each element. The results of this study on Validation and Test sets are illustrated in Table 3.

On the validation set, improvements are evident when adopting Post-training or Layer Aggregation independently, as opposed to fine-tuning the unmodified CodeBERT model. Especially, the post-training method makes up significant enhancement (over 5% on F1 score). Meanwhile, on the test set, HSUM demonstrates its efficacy in enhancing F1 score performance. This underscores the effectiveness of each proposed component. Consequently, the combination of these two methodologies, encapsulated in Dopamin, attains the highest performance of F1 score on both evaluation sets.

7 CONCLUSION

We proposed Dopamin, a novel approach for code comment classification, which demonstrates notable advancements in the benchmark. By selecting CodeBERT as the backbone model and implementing an optimal checkpoint process, we have optimized the classification accuracy across various programming languages. The domain post-training procedure significantly enhances performance for low-resource languages, illustrating the model's adaptability. Additionally, the use of multi-level layer aggregation, specifically the Hierarchical aggregation (HSUM) technique, enriches the semantic representation of comments, contributing to Dopamin's superior performance over the STACC baseline in most categories. Overall, Dopamin's methodology and results mark a significant improvement in the efficiency and precision of code comment classification, offering valuable insights and tools for software communities.

REFERENCES

- [1] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. 2023. STACC: Code Comment Classification using SentenceTransformers. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*. 28–31.
- [2] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.
- [3] Ganesh Jawahar, Benoit Sagot, and Djamel Seddah. 2019. What does BERT learn about the structure of language?. In *Association for Computational Linguistics*.
- [4] Rafael Kallis, Giuseppe Colavito, Ali Al-Kaswan, Luca Pascarella, Oscar Chaparro, and Pooja Rani. 2024. The NLBSE'24 Tool Competition. In *Proceedings of The 3rd International Workshop on Natural Language-based Software Engineering (NLBSE'24)*.
- [5] Akbar Karimi, Leonardo Rossi, and Andrea Prati. 2021. Improving BERT Performance for Aspect-Based Sentiment Analysis. In *4th International Conference on Natural Language and Speech Processing, Trento, Italy, November 12-13, 2021*, Mourad Abbas and Abed Alhakim Freihat (Eds.). Association for Computational Linguistics, 196–203.
- [6] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).
- [7] Yikun Li, Mohamed Soliman, and Paris Avgeriou. 2023. Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering* 28, 3 (2023), 1–38.
- [8] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
- [9] Dung Nguyen Manh, Nam Le Hai, Anh T. V. Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi D. Q. Bui. 2023. The Vault: A Comprehensive Multilingual Dataset for Advancing Code Understanding and Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, 4763–4788.
- [10] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE.
- [11] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of systems and software* 181 (2021), 111047.
- [12] Hung Quoc To, Nghi D. Q. Bui, Jin L. C. Guo, and Tien N. Nguyen. 2023. Better Language Models of Code through Self-Improvement. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 12994–13002.