LATE-IIMAS at Semeval-2026 Task 13: Evaluating GNNs, PLMs, LLMs, and Stylometry for Automatic Code Identification

Andric Valdez[1], Emmanuel Ancona[2], Sebastián Bernardino[3], Fabian Herrera[4], Fazlourrahman Balouchzahi[4], Helena Gomez-Adorno[4]

[1]Posgrado en Ciencia e Ingeniería de la Computación,
[2]Facultad de Ciencias,
[3]Facultad de Estudios Superiores Acatlán,
[4]Instituto de Investigaciones en Matemáticas y en Sistemas,
Universidad Nacional Autónoma de México (UNAM)

Correspondence: andric.valdez@gmail.com

Abstract

The generation of source code via Artificial Intelligence has become a prevalent practice in both academia and industry, posing significant challenges to academic integrity and authorship attribution. In this work, we address SemEval-2026 Task 13: Detecting Machine-Generated Code by evaluating the effectiveness of four distinct methodologies: Graph Neural Networks (GNNs), Pre-trained Language Models (PLMs), Large Language Models (LLMs), and Stylometric Feature Engineering using XGBoost. Our approach focuses on three specific scenarios: Subtask A (Binary Detection), Subtask B (Multi-Class Authorship), and Subtask C (Hybrid Code Detection). While our models achieved high performance during the validation phase, the transition to the final test set revealed substantial challenges in generalization, likely due to the increased diversity of programming languages and generators in the unseen data. This work serves as a foundational first step, identifying critical gaps in model robustness and highlighting the need for more sophisticated methodologies to bridge the performance gap in complex, real-world environments.

1 Introduction

Nowadays, LLMs can create robust scripts, which are useful for speeding up coding production (Endla et al., 2025). They are doing so well that it is a big challenge to distinguish them from those made by humans, and we need a system to achieve authorship classification (Pan et al., 2024).

For this reason, we tackle SemEval-2026 Task 13 (Orel et al., 2026): Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios, which consists of three subtasks, A: Binary Machine-Generated Code Detection, B: Multi-Class Authorship Detection, and C: Hybrid Code Detection.

We evaluated four main approaches: 1) Graph Neural Networks (GNNs), 2) Pretrained Language Models (PLMs), 3) Large Language Models (LLMs), and 4) Stylometric Features. With the GNN approach, we leverage structural information from the code co-occurrence graphs; for the stylometric approach we used a XGBoost ensemble with fixed stylometric features; in the PLMs approach, we integrate a Domain Adversarial Neural Network (DANN) with the CodeBERT model (Feng et al., 2020), which incorporates two new independent classification heads; and finally in the LLM approach, we ask for Command A (Team et al., 2025) to classify each code, adding to each prompt 2 examples of how to perform the classification[1].

2 Related Work

Graph Neural Networks have become an increasingly used technique in text analysis for classification in document clustering, sentiment analysis, and semantic similarity, among other classic NLP tasks, as detailed in the survey by Wu et al. (2023). In this context, Abstract Syntax Trees (AST) are a useful tool for representing code as graphs, as done by code2vec, explained by Alon et al. (2019), since, unlike text in NLP, code presents hierarchical structures that are better captured using this structure. It is also worth mentioning the previous work carried out by our institution in the analysis of languages of the Iberian Peninsula at IberAuTexTification 2024 for stylometric analysis using Graph Neural Networks proposed by Valdez-Valenzuela et al. (2025), which was a starting point for this work. These approaches are important because PLMs such as

[1]Code repository: https://github.com/PLN-disca-iimas/code-detect-Semeval2026

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2689–2696

CodeBERT (Feng et al., 2020) tend to ignore the logical hierarchy inherent to source code, regardless of the language; being based on the transformer architecture, CodeBERT captures semantic dependencies in the code, which allows for a different perspective compared to the syntactic analysis of ASTs.

Another approach that has addressed this problem is feature extraction for training classical Machine Learning models, specifically focusing on XGBoost ensemble models. As shown in the work of (Shi et al., 2025), certain features, such as lexical diversity (measured through line counts, number of unique tokens, and vocabulary, among others), as well as feature extraction aided by AST, are utilized. With this, we compare the results against those obtained with PLMs and GNNs.

Our approach consists of comparing the results obtained from classifying the code snippets using the different techniques described in this section.

3 Dataset

In this task, the organizers provided a dataset derived from enriched data from our previous work (Orel et al., 2025b) (Orel et al., 2025a). For each subtask, three subsets, training, validation, and test, compose the dataset; the training instances ranged from 500,000 to 900,000. All datasets consist of four columns:

- code - program code.
- generator - the model which generated the code (or 'human' if it is human-written).
- language - programming language of the code.
- label - A numerical value which represents the class, i.e., the author of the code (classes are different between tasks).

The following labels are presented for each task:

- **Task A**: Binary classification problem. Two labels, 0 (human) and 1 (AI).
- **Task B**: Multiclass classification problem. Eleven labels: 0 (human), 1 (DeepSeek-AI), 2 (Qwen), 3 (01-ai), 4 (Bigcode), 5 (Gemma), 6 (Phi), 7 (meta-LLaMA), 8 (IBM-granite), 9 (Mistral), and 10 (OpenAI).
- **Task C**: Multiclass classification problem. Four labels, 0 (human), 1 (AI), 2 (Hybrid), 3 (Adversarial)

Depending on the task, the codes are written in up to 8 programming languages: Python, Java, JavaScript, C#, C++, Go, PHP, and C. A common pattern is that the majority of the codes in both the training data and the test data are in Python and Java. Also, the LLM authors come from the same LLM families mentioned in the previous paragraph. e.g., GPT-4o and GPT-4o-mini are unique authors derived from the same LLM family.

Counting all kinds of characters (including white spaces), the code length is, in general, around 1,250 characters, also counting that a lot of them have comments, the actual coding is somewhat short. However, the dataset has a good amount of medium and large codes.

Appendix A shows more information about the label, programming language, and code length distribution for all task subsets.

3.1 Preprocessing

Due to the quantity and imbalance of some of the datasets, we modify the training subset to accomplish a better generalization and representation of the data.

In task A, labels are balanced, but programming languages are highly imbalanced (Java, C++, and Python, with an excess of the latter). We made two partitions to address this issue: equitable and Python undersampling. To address this issue, we created two partitions: Equitable and Python Undersampling. In the Equitable partition, we reduced the number of Python instances to match the count of other languages while preserving the original author proportions. Python Undersampling is a variation of this strategy that further reduces the volume of Python data. However, instead of fully balancing all programming languages, we only decrease Python samples enough to narrow the gap in their proportions. Table 1 shows the partitions and their characteristics.

In task B, unlike task A, the authors are not balanced, produced primarily by the large amount of human codes compared to the others. We decided not to do undersampling, but to divide the data into two parts: Human vs LLM and LLM only. In Human vs LLM, we changed the labels into 0 (humans) and 1 (machine); the machine class incorporates the remaining authors. As a result, we improved the authors' imbalanced situation. For the LLM only, we removed the human class completely. Table 2 summarizes the final dataset for task B, with brief additional information.

Table 1: Train and validation partitions for task A.

Partition	Description	Present languages	Training Instances
Equitable	- Balance labels - Balance languages	Python, Java, C++	∼62,000
Python under-sampling	- Balance labels - Undersampling on python - Reduced language imbalance	Python, Java, C++	∼143,000

Table 2: Dataset preprocessing on task B

Partition	Description	Training Instances
Original	No changes	∼500,000
Human vs LLM	The labels are changed 0 (human) 1 (machine)	∼500,000
LLM only	The human instances are removed	∼57,904

4 System Overview

In this section, we present the different approaches explored for the Machine-Generated Code detection task. We first describe in Section 4.1 the traditional baselines and, in Section 4.2, the stylometric feature-based methods, followed by Section 4.3 and 4.4, the domain-adaptive pre-trained language models and prompting strategies with large language models, respectively. Finally, Section 4.5 introduces our Graph Neural Network approach, which leverages text graph representations and graph-based deep learning to model structural relationships within code for classification.

4.1 Baselines

The organizers provided CodeBERT as a baseline, an encoder capable of processing and understanding natural language and programming language ('bimodal' pretrained model), which has 125 million parameters, considered a small model for today's standards.

4.2 Stylometric Features

Stylometric analysis consists of extracting features from the source code in our dataset based on its stylistic properties; once these features are obtained, an XGBoost model is trained for classification. For code tokenization, we used regular expressions that search for sequences of letters (including underscores to detect function and variable naming conventions), numbers, and special structural and operational characters. With this in mind, we consider the following features to construct our final feature vector.

1. **Length Properties**
   - *Code Length:* Total number of characters in the code calculated via string methods.
   - *Average Line Length:* The mean number of characters per line across the entire code snippet.
2. **Lexical Variety**
   - *Total Token Count:* Total number of tokens identified after tokenization.
   - *Unique Tokens:* Number of distinct tokens in the snippet.
   - *Token Ratio:* The ratio of unique tokens to total tokens.
3. **Comments**
   - *Comment Count:* Identification of comments using common prefixes (`#`, `//`, `/*`, `--`, `;`). We verify both line-start comments and inline comments by checking for trailing characters after the prefix.
   - *Comment Density:* The ratio of identified comments to the total number of lines.
4. **Indentation**
   - *Average Indentation:* The mean indentation value calculated after expanding tabs into a fixed number of spaces.
   - *Standard Deviation:* The variation of indentation across all lines.
   - *Indentation Levels:* The count of unique indentation depths present in the code.

Feature vectors are standardized via *Z-score* before training an **XGBoost** model, which predicts the class with the highest probability ($argmax$).

4.3 Pre-trained Language Models

This system combines the baseline with a Domain-Adversarial Neural Network (Ganin et al., 2016). The DANN model incorporates two independent classification heads after the main architecture. Given a sample, one head classifies the main target, while the secondary head identifies the domain to which the sample belongs. During training, both heads compute their own loss function separately. Then, in the backward propagation step, a gradient reversal layer is inserted between the secondary head and the main body. This layer multiplies the sign of the domain gradient by minus one. Consequently, the main label gradient is backpropagated

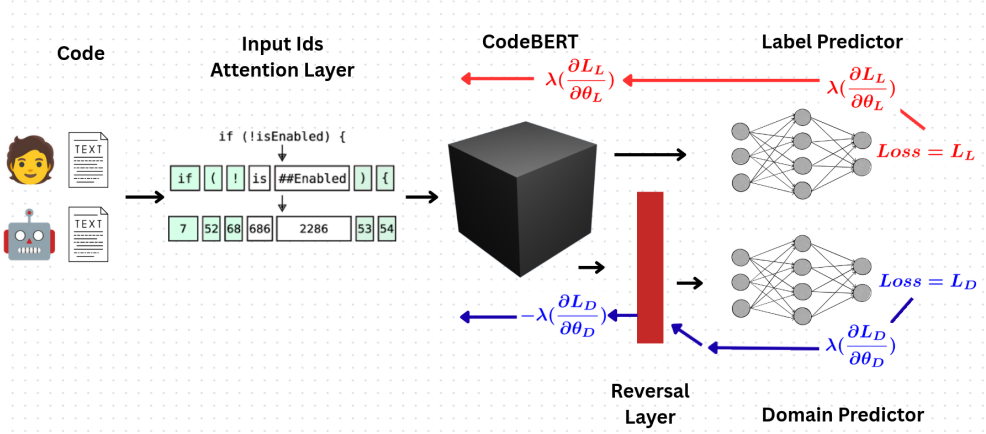

Figure 1: System overview for Pre-trained Language Model

normally through the principal classifier and the main body, whereas the secondary gradient propagates normally only along the secondary classifier and in the opposite direction through the main body. The objective is to learn a model that generalizes well across domains. For this task, we consider the programming language as our domain set (which is highly imbalanced across all three tasks), with Python prevailing over the others. Hence, we expect to generalize well to other, less frequent programming languages such as Go.

Figure 1 shows the approach used to solve the Machine-Generated Code task. In Task A, our best model was trained on the partitioned Python-undersampling data 1. For Task B, we used a cascade system to make predictions: we trained two DANN models, the first was a binary identification model trained on the Human vs LLM set (which makes the initial prediction), and the second was a multi-classifier DANN trained on the LLM only set 2. Finally, for Task C, as mentioned before, we used the entire dataset.

4.4 Large Language Models

We used the LLM Command A from Cohere (Team et al., 2025) to tackle subtask A. We extract two examples from the training dataset and use them as prompts for each classification. We tested different temperature values, and the best results were obtained at 0.3. We used the following prompt:

```
prompt = f""" code: "{code_examples}"
Label: {examples_label}
You must classify the next code as human
    made or artificial intelligence
    generated.
Your response must be only one of this
    labels: 0 for human made and 1 for
    AI generated.
Code:
{code_block}
Now output the label with no extra text:
"""
```

4.5 Graph Neural Networks

Figure 2 shows the GNN approach. This pipeline transforms each code document into a graph representation and learns node-level embeddings for classification. First, we applied text preprocessing: we cleaned the raw code, balanced the training distribution via subsampling, and included an out-of-distribution approach.

After that, we build a co-occurrence graph per document using a special tokenizer: extracts code units (identifiers, operators, punctuation symbols, numbers, and strings), optionally removing comments and normalizing literals (e.g., mapping strings to <STR> and numbers to <NUM>), while preserving structural tokens such as brackets and operators. These tokens represent the graph nodes, and undirected edges connect tokens that co-occur within a fixed window in the token sequence. Figure 3 illustrates an example of the Co-Occurrence Graph generated from the following Python code:

```
(b, a) = map(int, input().split())
div = int(a // b)
rem = a % b
if rem == 0:
    print(int(div))
else:
    print(div + 1)
```

Using this graph representation as input, we initialize node features with contextual embeddings from a pre-trained language model (e.g., CodeBERT) to provide semantic information before graph learning. The obtained graph is processed by a Graph Neural Network to refine node representations by leveraging their structural co-occurrence

Table 3: Results of the best F1-scores obtained by each strategy on the validation and final test datasets. The last row shows the top-ranked system on the final leaderboard.

Subtask	*A*		*B*		*C*	
Approach	*Validation*	*Test*	*Validation*	*Test*	*Validation*	*Test*
Baseline	**0.9875**	0.3053	0.3500	0.2285	0.7000	0.4812
GNN	0.8412	0.5381	**0.8675**	0.1550	N/A	N/A
PLM	0.9774	0.4219	0.5265	**0.3324**	**0.8425**	**0.5677**
LLM	0.6192	**0.5641**	N/A	N/A	N/A	N/A
Stylometry	0.8400	0.3200	N/A	N/A	N/A	N/A
Best ranked score	*0.9971*		*0.5081*		*0.7855*	

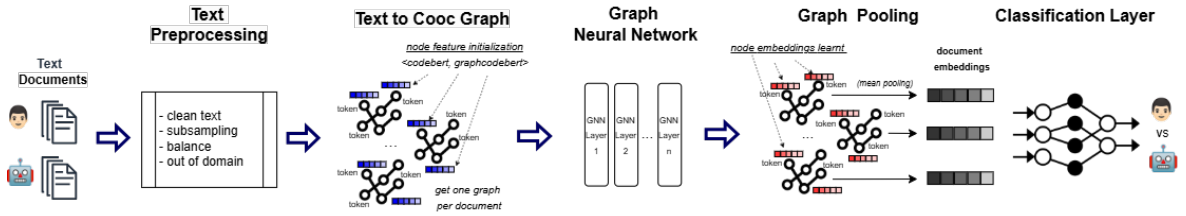

Figure 2: System overview using GNN approach

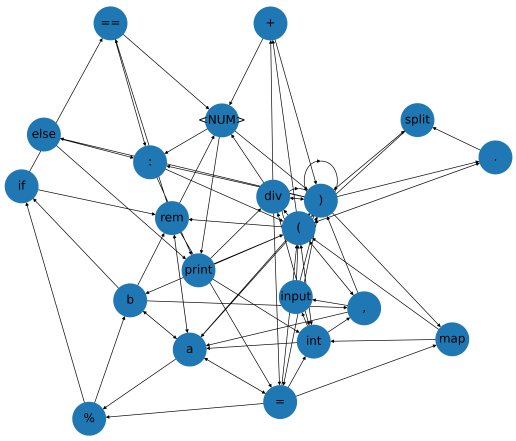

Figure 3: Co-Occurrence Graph

patterns. Finally, we aggregate node embeddings with mean pooling to obtain a single graph (document) embedding, which is passed to a simple neural classifier to predict whether the input code was written by a human or generated by a machine.

5 Results

This section presents the F1-scores for the following approaches: GNN, PLM, LLM, Stylometry, and the baseline. Table 3 summarizes our results for each task on both the validation and test sets. We include both scores to highlight the substantial discrepancy that can arise between the approaches. In task A, we used the undersampling partition 1. Note that not every approach was applied to every subtask; in such cases, the result is marked as N/A.

All our models outperform the baseline on task A, and the LLM system achieves the best performance, with an F1-score of 0.5641, indicating that few-shot strategies are more effective. Followed for the GNN approach, achieving a 0.5381 F1-score, demonstrating the effectiveness of graph representations for this task. Finally, we ranked 38th in Subtask A on the final leaderboard using the GNN approach. For Subtasks B and C, using the PLM approach, we ranked 25th and 22nd, respectively.

6 Conclusions

This research presents four independently developed systems for SemEval 2026 Task 13, each of which exhibits different strategies: Graph Neural Networks, Pretrained Language Models, Large Language Models, and Stylometry. The limited representation and imbalance of programming languages in the dataset made it difficult for our models to generalize effectively. However, for all of our systems, applying a dataset partitioning strategy helped improve performance. Specifically for stylometry, recognizing new languages poses a significant challenge for the way our solution was designed. For tasks B and C, we used a PLM and GNN, obtaining better results than the baseline. Future work includes combining these systems, exploring other encoders (e.g., GraphCodeBERT), and exploring different graph representations.

Acknowledgments

Andric Valdez (CVU-927264) thanks the Secretaría de Ciencia, Humanidades, Tecnología e Innovación (SECIHTI) graduate degree scholarship program. The authors also thank Adrian Durán Chavesti, Ricardo Villareal, and Rita Rodriguez of the Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS) for their support with the computational resources (access, configuration, and administration) used to run the experiments. Fazlourrahman Balouchzahi acknowledges the support from the SECIHTI, Mexico, through the Postdoctoral Fellowship Program (EPM 2025).

References

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–29. ACM New York, NY, USA.

Purushotham Endla, Jayendra Gopal Thatipudi, Madhavi Latha Talluri, Pradeep K Joshi, Amita Joshi, and M.Anuradha. 2025. Autoopticode-llm: An autonomous large language model framework for intelligent code generation and optimization. In *2025 6th International Conference on IoT Based Control Networks and Intelligent Systems (ICICNIS)*, pages 1471–1478.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page Ref. paper.

Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario March, and Victor Lempitsky. 2016. Domain-adversarial training of neural networks. *Journal of machine learning research*, 17(59):1–35.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025a. CoDet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10570–10593, Vienna, Austria. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026. SemEval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. In *Proceedings of the 20th International Workshop on Semantic Evaluation (SemEval-2026)*, San Diego, USA. Association for Computational Linguistics.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025b. Droid: A resource suite for ai-generated code detection. *Preprint*, arXiv:2507.10583.

Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. Assessing ai detectors in identifying ai-generated code: Implications for education. In *Proceedings of the 46th international conference on software engineering: software engineering education and training*, pages 1–11.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2025. Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1628–1639, Los Alamitos, CA, USA. IEEE Computer Society.

Cohere Team, A Ahmadian, M Ahmed, J Alammar, M Alizadeh, Y Alnumay, S Althammer, A Arkhangorodsky, V Aryabumi, D Aumiller, and 1 others. 2025. Command a: An enterprise-ready large language model. *arXiv preprint arXiv:2504.00698*.

Andric Valdez-Valenzuela, Helena Gómez-Adorno, and Manuel Montes-y Gómez. 2025. Text graph neural networks for detecting AI-generated content. In *Proceedings of the 1stWorkshop on GenAI Content Detection (GenAIDetect)*, pages 134–139, Abu Dhabi, UAE. International Conference on Computational Linguistics.

Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, and Bo Long. 2023. Graph neural networks for natural language processing: A survey. *Foundations and Trends in Machine Learning*, 16(2):119–328.

A Dataset Statistics

In this section, we present the label, programming language, and code length distributions for each subtask (A, B, and C) in the training and validation subsets. These statistics were heavily considered when constructing all the models and defined how we approached each subtask, because the distributions differ notably between them.

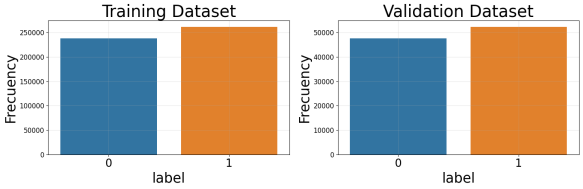

Figure 4: Label distribution for train and validation datasets on task A

Figure 4 shows a very equitable label distribution for the binary classification task A in both training and validation. Class 1 (AI) is slightly more abundant than class 0 (human). If needed, balancing this task completely is easy, since each class has approximately 250,000 instances. After subsampling, the amount of data remains sufficient to train all the models.

Figure 5 shows the language distribution for task A. Python, Java, and C++ are the only languages present, with Python overwhelmingly dominating the other two (over 90%). This has important implications for the models: they may not generalize well to new data, especially if written in programming languages never seen before, and they could learn spurious correlations—for instance, associating Java or C++ with a specific class. To mitigate this, we applied undersampling strategies to better balance the distribution.

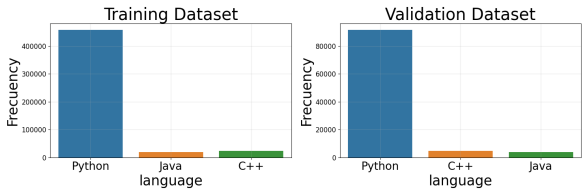

Figure 5: Language distribution for train, validation, and test sample datasets on task A

Figure 6 shows the code length distribution for task A. Code length refers to the number of characters, giving us an idea of how many tokens each code contains. This is very important for GNN, PLM, and LLM models, as they can only process a limited number of tokens—meaning that long codes can only be partially analyzed, potentially affecting predictions. Overall, the codes are short, but long codes are still present in the datasets.

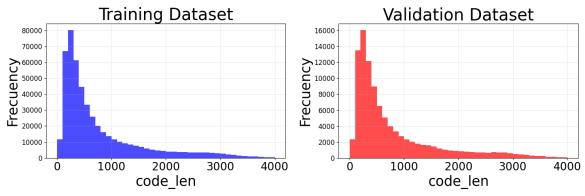

Figure 6: Code length distribution for train and validation datasets on task A

Figure 7 shows the label distribution for the multiclass classification problem of task B. The distribution only considers LLM generators for better visualization. Human-written codes constitute more than 80% of the training instances. The label distribution is highly imbalanced: some generators represent less than 1% of the total data. Imbalance also exists among the AI generators themselves. Performing undersampling for this task is more complicated.

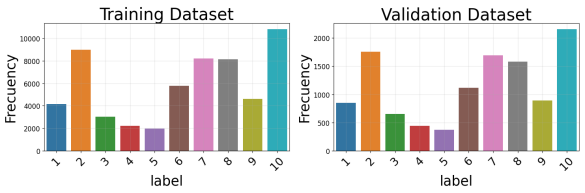

Figure 7: Label distribution for train and validation datasets on task B

Figure 8 shows the language distribution for task B. This task features more diversity in programming languages, with eight in total. The most prominent languages are Python and Java, which appear in very similar amounts. Even so, due to the amount of data, each programming language has a sizable number of instances.

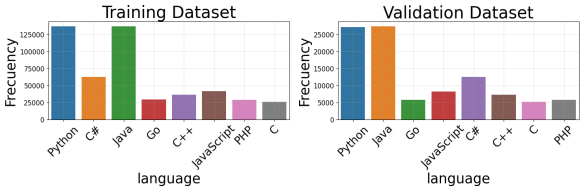

Figure 8: Language distribution for train and validation datasets on task B

Figure 9 shows the code length distribution for task B. Here, the codes are generally larger than in task A, with very long codes still present. For GNN, PLM, and LLM models, it is recommended to use a high maximum token cap to analyze all or at least the majority of each code.

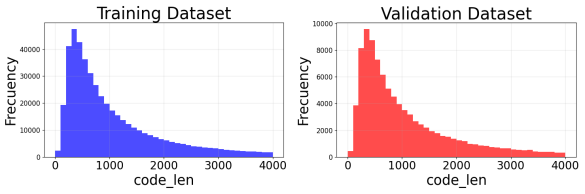

Figure 9: Code length distribution for train and validation dataset on task B

Figure 10 shows the label distribution for the multiclass classification problem of task C. Label 0 (human) is more prominent than the others, indicating imbalance among the four classes. However, the class with the fewest instances (label 2, hybrid) still has almost 100,000 examples. Undersampling

techniques can balance this task completely while leaving ample training data.

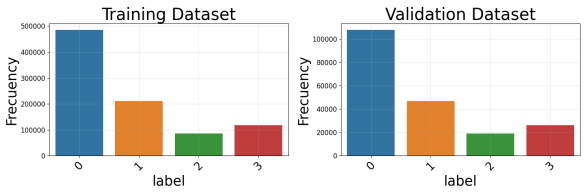

Figure 10: Label distribution for train and validation datasets on task C

Figure 11 shows the language distribution for task C. This distribution is very similar to that of task B, presenting eight programming languages, with Python and Java being more dominant than the others. Even so, due to the amount of data, each programming language has a sizable number of instances.

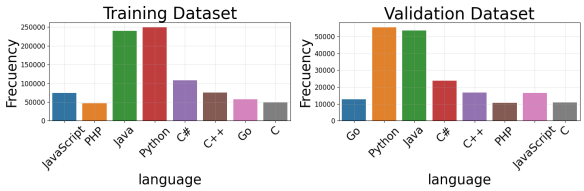

Figure 11: Language distribution for train and validation datasets on task C

Figure 12 shows the code length distribution for task C. Overall, this task has longer texts than tasks B and A. This distribution makes it harder for GNN, PLM, and LLM models due to the risk of not analyzing the codes completely.

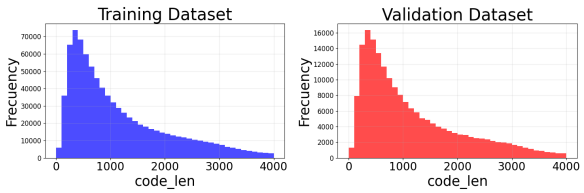

Figure 12: Code length distribution for train and validation dataset on task C.