TCProF:Time-Complexity Prediction SSL Framework

Joonghyuk Hahn, Hyeseon Ahn, Jungin Kim, Soohan Lim, Yo-Sub Han*,
Department of Computer Science, Yonsei University, Seoul, Republic of Korea,
{greghahn,hsan,jungin.kim,aness1219,emmous}@yonsei.ac.kr

Abstract

Time complexity is a theoretic measure to determine the amount of time the algorithm needs for its execution. In reality, developers write algorithms into code snippets within limited resources, making the calculation of a code's time complexity a fundamental task. However, determining the precise time complexity of a code is theoretically undecidable. In response, recent advancements have leaned toward deploying datasets for code time complexity prediction and initiating preliminary experiments for this challenge. We investigate the challenge in low-resource scenarios where only a few labeled instances are given for training. Remarkably, we are the first to introduce `TCProF`: a **T**ime-**C**omplexity **Pr**edicti**o**n SSL **F**ramework as an effective solution for code time complexity prediction in low-resource settings. `TCProF` significantly boosts performance by integrating our augmentation, symbolic modules, and a co-training mechanism, achieving a more than 60% improvement over self-training approaches. We further provide an extensive comparative analysis between `TCProF`, ChatGPT, and Gemini-Pro, offering a detailed evaluation of our approach.

1 Introduction

The task of predicting time complexity for code snippets represents a significant challenge in programming efficiency analysis. Time complexity is a crucial benchmark for evaluating algorithm performance across diverse computational domains. However, accurately computing the time complexity of a code snippet remains theoretically undecidable (Asperti, 2008), presenting a substantial obstacle. This issue is particularly crucial in environments such as educational settings, programming competitions, and automated code reviews, where an accurate evaluation of numerous solutions is essential.

In the meantime, the advent of deep learning methodologies presents a promising avenue to address this challenge. Sikka et al. (2020) introduced CorCoD dataset, specifically designed for code time complexity prediction. The dataset consists of code snippets labeled with their time complexity classes. They also provide initial experiments for the code time complexity prediction using both conventional algorithms and basic neural models. Despite these advances (Baik et al., 2024; Moudgalya et al., 2023), the efficiency of such models heavily relies on the availability of extensively annotated datasets. Unfortunately, datasets in this domain are currently scarce due to the problem of data scarcity, as time complexity annotations require professional knowledge. While there are further approaches (Baik et al., 2024; Moudgalya et al., 2023) with considerable potential, their effectiveness is contingent upon the size of annotated datasets.

Addressing the shortage of labeled data, we introduce several innovations. We develop data augmentation techniques specifically tailored to identify key factors that influence the time complexity of the code snippets. We also incorporate a co-training mechanism that leverages both original and augmented data effectively. Additionally, we construct a symbolic module that enhances the accuracy of pseudo-labels compared to the pseudo-labels generated by a model-based approach alone. Collectively, these components form the backbone of `TCProF`[1], our semi-supervised learning (SSL) framework, which is uniquely equipped to address code time complexity prediction in low-resource settings.

Operating under the assumption of limited labeled data and a vast number of unlabeled code snippets, we empirically analyze `TCProF` using publicly available datasets, CorCoD and CodeComplex (Baik et al., 2024). CorCoD includes code

*Corresponding author.

[1]https://github.com/peer0/few-shot-tc

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 9517–9542
April 29 - May 4, 2025 ©2025 Association for Computational Linguistics

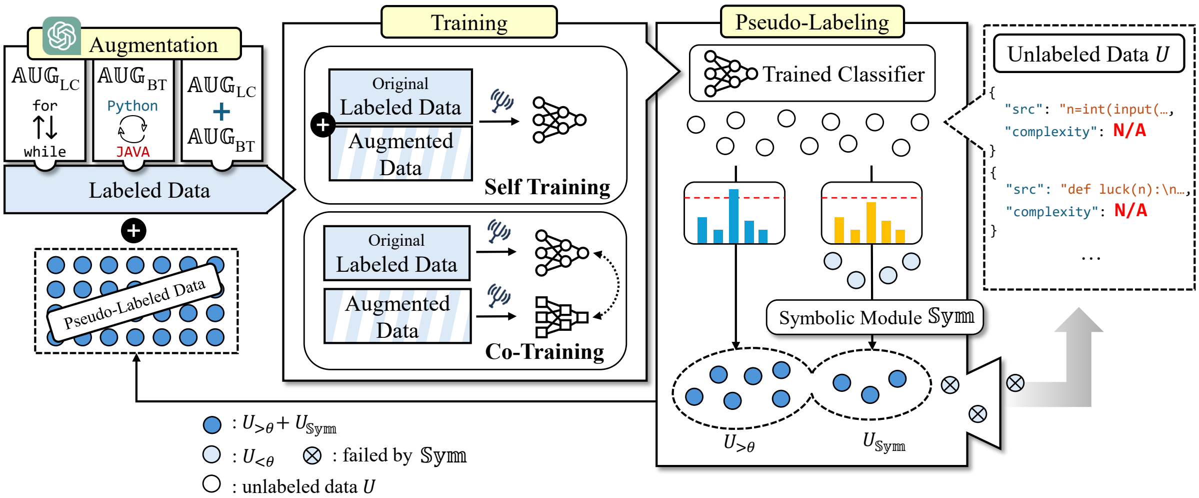


Figure 1: The overall framework of `TCProF`.

snippets categorized into five different complexity classes, $O(1)$, $O(\log N)$, $O(N)$, $O(N \log N)$, and $O(N^2)$. In contrast, CodeComplex consists of codes across seven different complexity classes, $O(1)$, $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$, $O(N^3)$, and $O(2^N)$. For these benchmark datasets, our framework `TCProF` significantly enhances the performance over traditional self-training methods with improvements exceeding 60% in accuracy and F1-scores. In the era of ChatGPT, we further provide comparative studies against off-the-shelf large language models (LLMs). Meanwhile, given the early stage of research in this domain, we provide an in-depth analysis of `TCProF` offering valuable insights into their practical utility. Ultimately, our endeavor specifically targets the practical aspect of code time complexity prediction—the availability of annotated data. By providing a fundamental framework, `TCProF`, for code time complexity prediction in low-resource settings, we aim to lay the groundwork for future research in this domain.

2 Related Works

2.1 SSL for Classification

Data scarcity is a critical problem for various tasks. More specifically, the problem occurs when there are only a few labeled data even though there are tons of unlabeled data. Generating labeled data is costly, making research in these low-resource environments crucial. Semi-supervised learning (SSL) offers an effective solution in such scenarios (Sajjadi et al., 2016; Xie et al., 2020; Chen et al., 2020, 2022; Sohn et al., 2020; Zhang et al., 2021a; Wang et al., 2023a; Zou et al., 2023; Huang et al., 2023; Nie et al., 2024).

One of the popularly used SSL techniques, self-training is a learning mechanism that trains the student model with a few-shot labeled dataset (Geng et al., 2019; Bao et al., 2020; Zhang et al., 2021b) and then subsequently acts as a teacher model for generating pseudo-labels (Lee, 2013). Pseudo-labels are judged based on the predictions of the model for a given unlabeled data. Co-training (Blum and Mitchell, 1998) is also the successful SSL mechanism that simultaneously employs two networks. Jointmatch (Zou and Caragea, 2023) utilizes cross-labeling, inspired by co-training, that uses an additional loss based on pseudo-labels for more reliability instead of augmenting pseudo-labels to the initial labeled dataset for additional training. While this approach has been effective to some degree, it lacks reliability as we cannot guarantee that the model generates 'correct' pseudo-labels. Recent approaches have incorporated symbolic modules to enhance the reliability of pseudo-label generations or data augmentation (Hahn et al., 2021; Kim et al., 2022).

Likewise, data augmentation is also one of the fundamental methods that is effective in the low-resource setting. Data augmentation generates artificial data from the original dataset without changing their labels. Conventional data augmentations are synonym replacements, word insertion or deletion (Wei and Zou, 2019) More advanced methods involve Back-Translation (Edunov et al., 2018) and these days, LLMs have gained popularity in generating various data but with accurate labels. We utilize these insights to develop `TCProF`, an SSL framework for predicting code time complexity in low-resource environments.

2.2 Code Time Complexity Prediction

Computation of code time complexity has long been a theoretically undecidable problem whereas classifying the code time complexity is a recently emerged problem. Sikka et al. (2020) first proposed this task and presented a labeled dataset with five time complexity classes. They propose a dataset named CorCoD, composed of Java codes with $O(1)$, $O(\log N)$, $O(N)$, $O(N \log N)$ and $O(N^2)$ time complexity classes and experimental results on the time complexity classification with baseline neural models.

Similar to CorCoD, CODAIT[2] tried to create good code embeddings by capturing manual features such as number of loops and breaks, and utilized graph-based representations for predicting time complexities. Afterward, Moudgalya et al. (2023) proposed TasTy, consisting of Java and Python data, and Baik et al. (2024) proposed CodeComplex, which consists of also Java and Python data with additional labels. CodeComplex consists of seven different time complexity classes, $O(N^3)$ and $O(2^N)$ in addition to those of CorCoD.

While time complexity prediction has been explored in these datasets, the scarcity of labeled data remains a significant challenge. Unlike runtime-based datasets from online judge platforms, which can be influenced by hardware, input distributions, and implementation-specific optimizations (Ishimwe et al., 2021; Zhang et al., 2023), these datasets provide explicit theoretical complexity labels for the code snippets. However, annotating such datasets requires expert knowledge, making them inherently low-resource. We propose `TCProF`, the SSL framework designed to alleviate this data scarcity challenge and enhance the accuracy of time complexity prediction.

3 Methodology

3.1 Overview

`TCProF` is a robust SSL framework designed for code time complexity prediction in a low-resource setting illustrated in Fig. 1. `TCProF` comprises three primary components:

[2]https://community.ibm.com/community/user/ai-datascience/blogs/sepideh-seifzadeh1/2021/10/05/ai-for-code-predict-code-complexity-using-ibms-cod

Augmentation module ($\mathbb{AUG}$): This module employs augmentation involving loop representation conversion (LC) and back-translation conversion (BT), and a combined ensemble of these methods to enhance the diversity of augmented data.
Training mechanism: Utilizing a co-training approach, `TCProF` trains two models simultaneously; one with original data and the other with augmented data. This mitigates the error accumulation problem of self-training.
Pseudo-label module: Integrating our symbolic module ($\mathbb{S}\mathtt{ym}$) with the classifier, this module generates more precise pseudo-labels.

Algorithm 1 describes a detailed procedure of our framework.

3.2 Self Training

In our experiments, we have a dataset T for training, a dataset V for validation and a dataset E for evaluation. we split T into a labeled dataset $L = \{l_1, l_2, \dots, l_M\}$ of size M and an unlabeled dataset $U = \{u_1, u_2, \dots, u_N\}$ of size N for self-training:

$$L = \{l \mid l = (d, c)\},\ U = \{u \mid u = (d, \lambda)\}$$

where d is a code and c is a complexity class for d. λ in u represents that u does not contain a complexity class for d.

For the first iteration, we train the baseline model B with L. With the trained B, we predict the complexity class c for d in each $u \in U$. If the confidence score $s_u(c)$ of u for c passes the predefined threshold θ, we pseudo-label each u with its corresponding label c_u and then add it to L:

$$L' = L + \{u \mid u = (d, c_u) \text{ where } s_u(c_u) \geq \theta\}.$$

For the next iterations afterward, we train B with the updated labeled dataset L' and then pseudo-label the unlabeled data to update L' again.

3.3 Co-Training

We mitigate the error accumulation problem of self-training by implementing co-training as our learning mechanism. Our implementation of co-training involves two different models B and B_{aug} which are trained with two different datasets L and L_{aug}. Likewise to the self-training in Section 3.2, L is a labeled dataset of size M. L_{aug} is a labeled dataset of size M generated by our data augmentation strategy in Section 3.7:

$$\begin{aligned} L_{aug} = \{l_{aug} \mid l_{aug} = (d_{aug}, c),\ &\text{where} \\ d_{aug} \text{ is augmented from } d \text{ in } l = (d, c)\}&. \end{aligned}$$

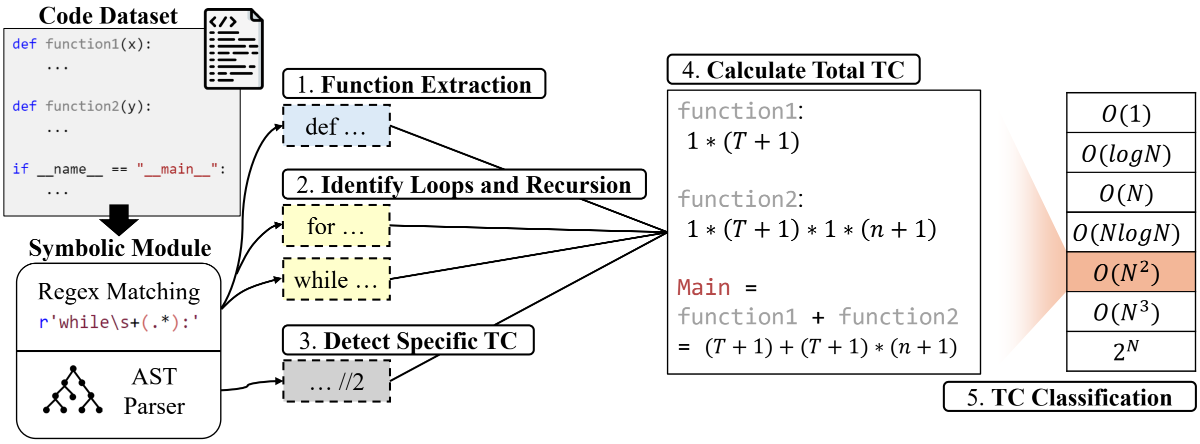


Figure 2: A procedural illustration of our symbolic module $\mathbb{Sym}$.

Each model B and B_{aug} is trained by L and L_{aug}, respectively. Following the pseudo-label strategy from Sections 3.4 and 3.5, two models then generate pseudo-labels and update labeled dataset L' and L'_{aug}, respectively:

$$L' = L + \{u \mid u = (d, c_u) \text{ where } u \text{ is pseudo-labeled by } B_{aug}\},$$

$$L'_{aug} = L_{aug} + \{u \mid u = (d, c_u) \text{ where } u \text{ is pseudo-labeled by } B\}.$$

3.4 Model-based Pseudo-Labels

The conventional pseudo-labeling procedure is done by prediction of B. The confidence score of unlabeled data with its corresponding class that B outputs is the one and only component for model-based pseudo-labels. We can adjust the threshold value for more precise pseudo-labels.

3.5 Symbolic Pseudo-Labels

We introduce a computational module, designed to predict the temporal complexity of source code through symbolic analysis, employing the computational results as pseudo-labels to enhance the accuracy of predictions. We have named this module, the **symbolic module** $\mathbb{Sym}$. The symbolic module $\mathbb{Sym}$ is designed to predict the time complexity of code by identifying specific patterns and structural elements, without relying on neural networks. This approach effectively complements language models that, by their nature, do not account for the hierarchical organization inherent in programming code (Allamanis et al., 2018; Chen et al., 2021; Zhang et al., 2022). The module primarily employs Regular Expressions (Regex) and Abstract Syntax Trees (ASTs) as essential tools for a detailed analysis of the source code. This aims to describe the time complexity of iterations, function calls, recursive and iterative calls, and additional relevant constructs, thereby facilitating a comprehensive understanding of the code's structure and execution flow. Our module outputs a formula consisting of the above components and regarding the size of inputs in the source code, subsequently aggregating these elements to categorize the time complexity class of the source code.

Our approach is systematically organized into five distinct phases and is illustrated in Figure 2:

Function Extraction: We employ Regex to identify and extract function definitions from source code. Each function forms the basic unit of analysis as an independent code block. If the source code does not contain functions, this step is omitted.
Identify Loops and Recursions: We utilize Regex to identify the presence of repetitive statements (such as for loops and while loops) and recursive functions to determine the frequency of loop iterations and function calls.
Detect Specific Time Complexity (TC): We utilize ASTs to detect operations that modify the input size. By identifying the presence of sorting or binary traversal, we classify source codes according to their time complexities such as $O(\log N)$ and $O(N \log N)$.
Calculate Total TC: The final time complexity is calculated by summing the complexity of each iteration and function call, based on the patterns identified using Regex, and the code structure analyzed by ASTs. In this step, we calculate the overall complexity, ensuring a robust analysis of recursive

Algorithm 1 Procedure for the SSL framework `TCProF`($\mathcal{L}, \mathcal{U}, \mathbf{B}, \mathbf{B}_{\text{aug}}, \mathbb{S}\text{ym}, \mathbb{AUG}$). Inputs include labeled dataset $\mathcal{L}$, unlabeled dataset $\mathcal{U}$, baseline model $\mathbf{B}$, co-training model $\mathbf{B}_{\text{aug}}$, symbolic module $\mathbb{S}\text{ym}$, and augmentation module $\mathbb{AUG}$. The set of complexity classes is denoted as $\mathcal{C}$.

```
function PSEUDO-LABEL(B, Sym, U)                        ▷ Sections 3.4, 3.5
    U' ← ∅
    for each u = (d, λ) ∈ U do
        c_u ← arg max_{c∈C} B(u, c)
        U' ← U' ∪ {(d, c_u) : B(u, c_u) ≥ θ}
        U' ← U' ∪ {(d, Sym(d, u)) : B(u, c_u) < θ}
    end for
    return U'
end function
procedure TIMECOMP(L, U, B, B_aug, Sym, AUG)
    L_aug ← AUG(L)                                      ▷ Section 3.7
    for e ← 1 to epoch_number do
        if self-train then                              ▷ Section 3.2
            L ← L ∪ L_aug
            B ← fine-tune(B, L)
            U_pl ← PSEUDO-LABEL(B, Sym, U)
            L ← L ∪ U_pl
        else if co-train then                           ▷ Section 3.3
            B ← fine-tune(B, L)
            B_aug ← fine-tune(B_aug, L_aug)
            U_pl ← PSEUDO-LABEL(B, Sym, U)
            U_aug-pl ← PSEUDO-LABEL(B_aug, Sym, U)
            L ← L ∪ U_aug-pl
            L_aug ← L_aug ∪ U_pl
        end if
    end for
end procedure
```

relationships within the source code.

TC Classification: The calculated time complexity is categorized into the pre-defined complexity classes. These assigned classes are utilized as pseudo-labels.

In Appendix H, we present Figure 9 as a running example of the symbolic pseudo-labeling.

3.6 Merge Pseudo-Labeling

We have two pseudo-label modules based on model confidence and the symbolic module. Illustrated in Figure 1 and Algorithm 1, we first pseudo-label the unlabeled data by the model confidence. Then, we use the symbolic module to pseudo-label the unlabeled data that failed pseudo-labeling by the model confidence.

3.7 Data Augmentation

We introduce a data augmentation module $\mathbb{AUG}$, designed to complement the lack of labeled data. Our augmentation strategies leverage the ChatGPT API, specifically employing the `gpt-3.5-turbo-0125` model, to augment our experiments' CorCoD and CodeComplex datasets. The objective is to create precise augmentations that respect the intrinsic properties of the code, ensuring semantic integrity while introducing syntactic variability. We ensure that the augmented code snippets are free from syntactic errors, reinforcing their reliability for further analysis. We present two augmentation methods specifically designed to augment code snippets that preserve the original time complexity:

Back-Translation (BT): This method involves translating a code snippet into another programming language and then back to the original language to maintain its semantic essence. For instance, Java code snippets are translated into Python and then back into Java. This process gives syntactic variation while retaining the context.

Loop-Conversion (LC): Loop structures are the primary components that determine the time complexity of code snippets. This technique modifies the loop structures to different but semantically equivalent forms. Using regular expressions, we filter codes containing "for" or "while" loops and then convert these loops by employing For2While and While2For transformation rules, preserving the original logic of the code snippet. For instance, a while loop can be converted into a for loop (While2For) and vice versa (For2While), depending on the context. If the original code snippet contains both for and while loops, we leverage While2for and For2while respectively.

These augmentation methods are detailed in the prompts listed in Appendix K and are integrated into `TCProF` to enhance the robustness. The augmented data from BT and LC methods supplement the initial labeled data in three distinct experimental configurations: 1) $\mathbb{AUG}_{BT}$ incorporates back-translation augmented data, 2) $\mathbb{AUG}_{LC}$ incorporates loop-conversion augmented data, and 3) $\mathbb{AUG}_{BT+LC}$ combines both back-translation and loop-conversion augmented data.

There are two experimental setups for $\mathbb{AUG}_{LC}$ as there are code snippets without any loop structures: 1) $\mathbb{AUG}_{LC\_Natural}$ uses naturally sampled data as the initial labeled data without specific preconditions and 2) $\mathbb{AUG}_{LC\_Artificial}$ selects initial labeled data specifically containing loop structures to maximize the use of augmented data.

These configurations for $\mathbb{AUG}_{LC}$ are also applied to the combined augmentation strategy $\mathbb{AUG}_{BT+LC}$, with corresponding $\mathbb{AUG}_{BT+LC\_Natural}$ and $\mathbb{AUG}_{BT+LC\_Artificial}$ settings. Table 2 employs $\mathbb{AUG}_{BT+LC\_Natural}$ as

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	5	10	5	10	5	10
SSL Baselines						
ST (UniXcoder)	15.44$\pm$ 6.30	31.77$\pm$ 0.81	26.02$\pm$11.72	40.98$\pm$ 1.14	37.89$\pm$ 6.57	45.61$\pm$11.59
ST (CodeT5+)	18.87$\pm$ 3.21	31.69$\pm$14.00	28.76$\pm$17.47	38.32$\pm$18.82	35.79$\pm$ 6.90	45.26$\pm$ 6.32
JointMatch (CodeT5+)	14.62$\pm$ 4.26	24.68$\pm$ 2.60	20.97$\pm$ 3.90	21.04$\pm$ 4.33	36.49$\pm$ 4.98	42.11$\pm$ 6.57
JointMatch (UniXcoder)	9.62$\pm$ 5.23	19.39$\pm$ 3.86	14.68$\pm$11.78	20.76$\pm$10.46	35.44$\pm$ 6.85	48.42$\pm$ 8.22
`TCProF`(CodeT5+)	38.63$\pm$ 1.32	41.98$\pm$ 2.92	44.74$\pm$ 3.26	59.29$\pm$ 3.71	50.53$\pm$ 0.86	51.93$\pm$ 3.88
`TCProF`(UniXcoder)	**52.50**$\pm$ 1.56	**53.85**$\pm$ 3.63	**54.64**$\pm$ 3.77	**70.29**$\pm$ 2.06	**55.44**$\pm$ 0.99	**63.16**$\pm$ 2.27

Table 1: Accuracy performance of SSL baselines and `TCProF`. ST refers to Self-Training. `TCProF`(CodeT5+) represents `TCProF` implemented to the baseline CodeT5+ and vice versa to UniXcoder. The scores are averaged from three runs with different seeds. We report the full result in Table 5.

the baseline for the $\mathbb{AUG}$ setup as it is more natural and common compared to $\mathbb{AUG}_{BT+LC\_Artificial}$. An extensive analysis of augmentation strategies is provided in Appendix J.

3.8 Implementation Details

We use UniXcoder (Guo et al., 2022) and CodeT5+ (Wang et al., 2023b) as our baselines for `TCProF`. We assign the batch of size 7, the number of epochs as 20, and set the learning rate of co-training as `1e-5` and `2e-6`. We use `1e-5` learning rate for self-training. Our confidence score threshold θ is 0.7. We conduct experiments using the NVIDIA RTX 3090 for training `TCProF`.

4 Experimental Setup

We use CodeComplex (Baik et al., 2024) and CorCoD (Sikka et al., 2020) datasets. CodeComplex consists of 4,900 Java and 4,900 Python codes, and CorCoD consists of 929 Java codes. We follow Baik et al. (2024) to split train and test data. For low-resource settings, we perform 5- and 10-shot experiments where we pick 5 and 10 data for each label from the train dataset and use the remaining train dataset as an unlabeled dataset for pseudo-labels, respectively.

Code time complexity prediction is a recent code-related task and we are the first to perform the task in low-resource scenarios. We use four pre-trained code language models, CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), and CodeT5+ (Wang et al., 2023b) as our baseline models.

We also include JointMatch as an SSL baseline for comparison. Recently, Zou and Caragea (2023) proposed JointMatch for a state-of-the-art SSL framework for low-resource settings, where the authors employ cross-labeling to enhance pseudo-labeling. While JointMatch is effective for text classification tasks in low-resource settings, we analyze whether it is effective for code time complexity prediction in low-resource settings and compare the performance with our framework `TCProF`.

5 Results and Analysis

We present our main results in Table 1, indicating that our model outperforms baselines. In Section 5.1, we suggest possible intuitions that can be derived from the performance result of Table 1. We demonstrate ablation studies in Section 5.2 and provide error analyses in Section 5.3. Then, for the extensive analyses of `TCProF`, we show how TCProF approaches the fully fine-tuned models in Section 5.4 and provide comparisons with commercial LLMs in Section 5.5.

5.1 Comparison with Baselines

Table 1 presents the Self-Training (ST) results for the base models, UniXcoder and CodeT5+. The table illustrates that the average accuracy and standard deviation scores for ST of base models, which serve as baselines, are generally low. Specifically, on the CodeComplex datasets, the accuracy scores of these baselines are below 40%, except for ST (UniXcoder) on CodeComplex (Python). Furthermore, the high standard deviation scores indicate that the performance of these models is unstable and unreliable.

Our augmentation, symbolic pseudo-labeling, and co-training strategies mitigate this problem. We pick UniXcoder and CodeT5+ as the baseline model to implement our strategies[3]. `TCProF` out-

[3]Appendix B demonstrates the detailed analysis of the

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	5	10	5	10	5	10
CodeT5+						
+ AUG	26.85$\pm$ 2.4	35.86$\pm$ 4.53	33.94$\pm$ 3.82	57.44$\pm$ 1.17	36.14$\pm$ 4.98	43.85$\pm$ 6.34
+ Sym	21.92$\pm$ 2.54	35.50$\pm$11.18	32.10$\pm$ 6.78	46.38$\pm$ 6.58	36.84$\pm$ 1.72	42.81$\pm$ 2.76
+ Sym + AUG	37.88$\pm$ 1.65	38.26$\pm$ 2.40	41.80$\pm$ 1.49	58.88$\pm$12.37	37.89$\pm$ 2.58	44.21$\pm$ 2.27
TCProF(CodeT5+)	38.63$\pm$ 1.32	41.98$\pm$ 2.92	44.74$\pm$ 3.26	59.29$\pm$ 3.71	50.53$\pm$ 0.86	51.93$\pm$ 3.88
UnixCoder						
+ AUG	34.22$\pm$12.32	39.00$\pm$ 6.64	51.63$\pm$12.42	63.04$\pm$ 8.75	44.21$\pm$ 3.16	51.57$\pm$ 5.57
+ Sym	41.76$\pm$ 2.62	45.34$\pm$ 2.91	39.82$\pm$ 5.13	51.02$\pm$ 2.14	50.88$\pm$ 8.48	54.04$\pm$ 6.56
+ Sym + AUG	43.70$\pm$ 2.00	45.49$\pm$ 1.10	54.03$\pm$ 5.52	67.55$\pm$ 1.06	50.18$\pm$ 4.24	58.60$\pm$ 4.73
TCProF(UniXcoder)	**52.50**$\pm$ 1.56	**53.85**$\pm$ 3.63	**54.64**$\pm$ 3.77	**70.29**$\pm$ 2.06	**55.44**$\pm$ 0.99	**63.16**$\pm$ 2.27

Table 2: Ablation studies of TCProF(CodeT5+) and TCProF(UniXcoder) accuracy (%) performance. AUG represents $\mathbb{AUG}_{BT+LC}$. The full result is in Table 5 and extensive augmentation result is in Table 11.

performs the baselines illustrated in Table 1. Especially, TCProF(UniXcoder) accomplishes 64.81% improvements on average compared to the best performance of baselines. We discuss the analysis of our strategies in Section 5.2 as ablation studies.

We also include experiments on JointMatch, a state-of-the-art SSL baseline, to compare with TCProF. While JointMatch is the state-of-the-art approach for text classification in low-resource settings, the case is quite different in a code time complexity prediction. Table 1 indicates that JointMatch is generally less effective than self-training baselines and TCProF(UniXcoder) outperforms JointMatch by 131.09% on average. This is because cross-labeling of JointMatch depends on unlabeled data to alleviate the pseudo-label noise and the strategy is not quite applicable to code time complexity prediction data. We provide a detailed analysis in Appendix E illustrating that TCProF is more appropriate than the cross-labeling of JointMatch.

5.2 Ablation Studies

In our ablation studies, delineated within Table 2, we evaluate the impact of each module within TCProF, as introduced in Figure 1 and Section 3. Our framework integrates the augmentation module AUG, the symbolic module Sym, and implements both self-training and co-training strategies for code time complexity prediction in low-resource settings. The underlying hypothesis points that each component incrementally improves upon basic self-training baseline models. The empirical result from Table 2 confirms our hypothesis, demonstrating performance enhancements in our baselines, CodeT5+ and UniXcoder, as additional modules are integrated.

Particularly, the interaction between AUG and Sym shows a clear synergistic effect. While AUG alone significantly boosts performance compared to the self-training baselines, it mostly shows high standard deviation scores, which are then relaxed when applied with Sym. This is especially noticeable in CodeComplex (Python). AUG is effective considering its enhancements from the performance of self-training baselines in Table 1, but it also shows high standard deviation scores, mostly over 10%. However, this problem is substantially mitigated when AUG is combined with Sym, reducing the average standard deviation to 5%.

This enhancement is further developed when we implement the co-training strategy, solidifying not only the accuracy but also reducing the standard deviation. These observations are particularly pronounced in Python datasets rather than Java datasets. The variance between Python and Java can primarily be attributed to the differential effectiveness of AUG across these programming languages. The less stringent syntax of Python compared to Java allows greater variability in augmented Python code snippets, contributing to this performance boost. This phenomenon is further supported by Tables 11 and 12 in Appendix J. Despite these differences, a closer analysis of the F1-scores in Table 6 shows that both languages exhibit substantial improvements, each around 50%.

Furthermore, implemented all together with a co-training strategy, TCProF strengthens the per-

comparison between UniXcoder and CodeT5+.

formance both in accuracy and standard deviation. We also provide detailed analysis on F1-scores in Appendix A.

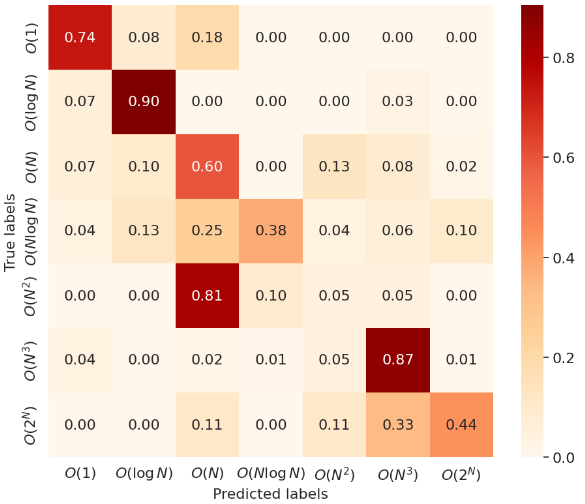


Figure 3: Confusion matrix of ours for CodeComplex (Python) 10-shot.

5.3 Error Analysis

Furthermore, we examine the types of errors to which our method is particularly vulnerable for extensive analysis. Figure 3 illustrates that our approach is especially weak on $O(N^2)$ class. We notice that the codes with this class are mostly predicted as $O(N)$. Codes in both classes involve loops and usually the only difference is the depth of the loops. However, precisely, the core factor that makes the difference is the loops involving the input length. For instance, codes in $O(N)$ class can contain multiple-depth loop statements where only a single loop statement involves the length of the input. The model needs to precisely determine whether the iteration number of each loop is proportional to the input size for the correct prediction. Likewise, the class $O(NlogN)$ involving both loops linear and logarithmic to the input size and the class $O(2^N)$ involving exponential iterations are also relatively erroneous compared to the other classes. We present examples of code instances, demonstrating the errors in Appendix I.

5.4 Comparison to the full-train result

`TCProF` demonstrates a promising result in a few-shot settings, improving the baselines on a large scale. We conduct further experiments on comparing the best 10-shot performance of `TCProF` to the baselines trained with the whole train datasets in Table 3. For CodeComplex datasets, `TCProF`(UniXcoder) exceeds the accuracy of all the baselines except for CodeT5+ and the performance gap with CodeT5+ is small. Take CodeComplex (Python) for instance, `TCProF`(UniXcoder) achieves 70.29% accuracy and the accuracy of CodeT5+ fine-tuned with the full train dataset is 72.88%. Our approach catches up to 96.45%.

FULL	CodeComplex Java	CodeComplex Python	CorCoD
	Acc.	Acc.	Acc.
CodeBERT	$39.82_{\pm 6.90}$	$66.39_{\pm 1.43}$	$74.74_{\pm 1.72}$
GraphCodeBERT	$46.46_{\pm 2.18}$	$67.83_{\pm 8.36}$	$72.98_{\pm 1.79}$
UniXcoder	$46.76_{\pm 1.91}$	$68.44_{\pm 4.66}$	$\mathbf{77.54}_{\pm 1.31}$
CodeT5+	$\mathbf{55.11}_{\pm 2.18}$	$\mathbf{72.88}_{\pm 1.95}$	$75.79_{\pm 2.27}$

Table 3: Accuracy of base models trained with the full-train dataset.

Appendix C presents the full results of accuracy and F1 performance in Table 7. Overall to all the datasets, `TCProF` catches up to 91% and 94% for accuracy and F1-scores, respectively, compared to the base models fine-tuned with the full train dataset. This indicates that `TCProF`, a framework designed for low-resource settings, is highly effective in achieving competitive performance even with limited labeled data. By leveraging co-training, symbolic pseudo-labels, and data augmentation, `TCProF` enhances model generalization and robustness, making it a viable alternative to fully supervised approaches, particularly in scenarios where the amount of labeled data is scarce or costly to obtain. We also present Figures 7a and 7b in the appendix for clear visualization of the results.

	CodeComplex Java	CodeComplex Python	CorCoD
	Acc.	Acc.	Acc.
Gemini-pro	49.54	31.05	61.91
GPT-3.5	62.15	32.55	69.42
GPT-4	**64.01**	53.04	**78.86**
`TCProF`(UniXcoder)	53.85	**70.29**	63.16

Table 4: Comparison with the performance of LLM for Java and Python datasets of CodeComplex, and CorCod dataset.

5.5 Comparison with LLMs

LLMs are known to be effective in general NLP tasks and we extend the analysis into comparing our performance with LLMs. We evaluate the LLMs in a 5-shot in-context learning setting by the prompt demonstrated in Appendix L. Our baseline LLMs are Gemini-pro (Anil and et al., 2023), GPT3.5, and GPT-4 (OpenAI, 2023). The results of Table 4 indicates that TCProF shows competitive performance to these commercial LLMs. These LLMs perform much better than CodeT5+ and UniXcoder fine-tuned with the full train dataset for CodeComplex (Java) and CorCoD referring to Table 3. As TCProF takes CodeT5+ and UniXcoder as the baseline models, competitive performance of TCProF compared to LLMs is remarkable.

Furthermore, TCProF performs better than any of these LLMs for CodeComplex (Python). In Appendix F, we also present Table 9 that includes both the accuracy and F1 performance of LLMs.

6 Conclusion

Code time complexity prediction remains a largely unexplored yet critical task, and our framework, TCProF, is the first attempt to tackle this challenge within low-resource environments. We have developed TCProF as a robust SSL framework for predicting code time complexity, showing through comprehensive analyses that it significantly improves upon the performance of baseline models. As we reflect on the capabilities of TCProF, it is important to recognize that our current focus has been on establishing an effective framework that performs well in constrained environments. Looking ahead, expanding the generalization capabilities of TCProF to accommodate a wider range of programming languages is a vital next step. We believe TCProF will serve as a significant milestone, propelling forward the research in code time complexity prediction, especially in low-resource settings.

Limitation

Application Scope TCProF is specifically designed for few-shot settings and thus, its application would not be effective in fine-tuning with the full dataset. One possible exploration is employing the augmentation module of TCProF for fine-tuning the full dataset. We provide the analyses in Appendix J. Another limitation of TCProF is the space complexity prediction. This is also due to the absence of datasets involving the space complexity. Thus, we will be happy to apply and develop our framework for the space complexity prediction when the datasets are released.

Framework Adaptability to Zero-Shots As our framework targets few-shot settings where 'several' data instances for each class are provided, We do not provide initial configuration for zero-shot settings. However, we can adapt the zero-shot setting by altering the structure of TCProF to incorporate the symbolic module for generating initial training data with pseudo-labels. It is essential to clarify that our objective is not zero-shot learning but few-shot learning.

Dynamic Calculation TCProF is focused on classifying time complexity into seven discrete classes, rather than dynamically calculating time complexities across a continuous range. We are eager to develop further modules and implement generative models for targeting the time complexity 'calculation', extending from the time complexity 'prediction'.

Acknowledgment

This research was supported by the NRF grant (RS-2023-00208094) and the AI Graduate School Program (RS-2020-II201361) funded by the Korean government (MSIT).

References

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR*.

Rohan Anil and et al. 2023. Gemini: A family of highly capable multimodal models. *preprint*, arXiv: 2312.11805.

Andrea Asperti. 2008. The intensional content of Rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 113–119.

Seung-Yeop Baik, Joonghyuk Hahn, Jungin Kim, Aditi, Mingi Jeon, Yo-Sub Han, and Sang-Ki Ko. 2024. CodeComplex: Dataset for worst-case time complexity prediction. *preprint*, arXiv: 2401.08719.

Yujia Bao, Menghua Wu, Shiyu Chang, and Regina Barzilay. 2020. Few-shot text classification with distributional signatures. In *8th International Conference on Learning Representations, ICLR*.

Avrim Blum and Tom M. Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory, COLT*, pages 92–100.

Hui Chen, Wei Han, and Soujanya Poria. 2022. SAT: improving semi-supervised text classification with simple instance-adaptive self-training. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 6141–6146.

Jiaao Chen, Zichao Yang, and Diyi Yang. 2020. Mixtext: Linguistically-informed interpolation of hidden space for semi-supervised text classification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL*, pages 2147–2157.

Zimin Chen, Vincent J. Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. PLUR: A unifying, graph-based view of program learning, understanding, and repair. In *Advances in Neural Information Processing Systems 34*, pages 23089–23101.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 489–500.

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 Association for Computational Linguistics: EMNLP*, pages 1536–1547.

Ruiying Geng, Binhua Li, Yongbin Li, Xiaodan Zhu, Ping Jian, and Jian Sun. 2019. Induction networks for few-shot text classification. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*, pages 3902–3911.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, ACL*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR*.

Joonghyuk Hahn, Hyunjoon Cheon, Kyuyeol Han, Cheongjae Lee, Junseok Kim, and Yo-Sub Han. 2021. Self-training using rules of grammar for few-shot NLU. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 4576–4581.

Zhuo Huang, Li Shen, Jun Yu, Bo Han, and Tongliang Liu. 2023. FlatMatch: Bridging labeled data and unlabeled data with cross-sharpness for semi-supervised learning. In *Advances in Neural Information Processing Systems 36*.

Didier Ishimwe, KimHao Nguyen, and ThanhVu Nguyen. 2021. Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–23.

Hazel H. Kim, Daecheol Woo, Seong Joon Oh, Jeong-Won Cha, and Yo-Sub Han. 2022. ALP: data augmentation using lexicalized pcfgs for few-shot text classification. In *Thirty-Sixth AAAI Conference on Artificial Intelligence*, pages 10894–10902.

Dong-Hyun Lee. 2013. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on challenges in representation learning, ICML*, page 896.

Kaushik Moudgalya, Ankit Ramakrishnan, Vamsikrishna Chemudupati, and Xing Han Lu. 2023. TASTY: A transformer based approach to space and time complexity. *preprint*, arXiv: 2305.05379.

Wenhua Nie, Lin Deng, Chang-Bo Liu, Jialing Wei, Ruitong Han, and Haoran Zheng. 2024. STSPL-SSC: semi-supervised few-shot short text clustering with semantic text similarity optimized pseudo-labels. In *Findings of the Association for Computational Linguistics, ACL*, pages 12174–12185.

OpenAI. 2023. GPT-4 technical report. *preprint*, arXiv: 2303.08774.

Mehdi Sajjadi, Mehran Javanmardi, and Tolga Tasdizen. 2016. Regularization with stochastic transformations and perturbations for deep semi-supervised learning. *Advances in Neural Information Processing Systems 29*, 29.

Jagriti Sikka, Kushal Satya, Yaman Kumar, Shagun Uppal, Rajiv Ratn Shah, and Roger Zimmermann. 2020. Learning based methods for code runtime complexity prediction. In *Advances in Information Retrieval - 42nd European Conference on IR Research, ECIR Proceedings, Part I*, volume 12035, pages 313–325.

Kihyuk Sohn, David Berthelot, Nicholas Carlini, Zizhao Zhang, Han Zhang, Colin A Raffel, Ekin Dogus Cubuk, Alexey Kurakin, and Chun-Liang Li. 2020. Fixmatch: Simplifying semi-supervised learning with consistency and confidence. *Advances in Neural Information Processing Systems 33*, 33:596–608.

Yidong Wang, Hao Chen, Qiang Heng, Wenxin Hou, Yue Fan, Zhen Wu, Jindong Wang, Marios Savvides, Takahiro Shinozaki, Bhiksha Raj, Bernt Schiele, and Xing Xie. 2023a. Freematch: Self-adaptive thresholding for semi-supervised learning. In *The Eleventh International Conference on Learning Representations, ICLR*.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1069–1088.

Jason W. Wei and Kai Zou. 2019. EDA: easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*, pages 6381–6387.

Qizhe Xie, Zihang Dai, Eduard Hovy, Thang Luong, and Quoc Le. 2020. Unsupervised data augmentation for consistency training. *Advances in Neural Information Processing Systems 33*, 33:6256–6268.

Bowen Zhang, Yidong Wang, Wenxin Hou, Hao Wu, Jindong Wang, Manabu Okumura, and Takahiro Shinozaki. 2021a. Flexmatch: Boosting semi-supervised learning with curriculum pseudo labeling. *Advances in Neural Information Processing Systems 34*, 34:18408–18419.

En Zhang, Fan Wu, and Xuesong Lu. 2023. An intelligent scheduling system for large-scale online judging. In *Computer Science and Education. Computer Science and Technology - 18th International Conference, ICCSE, Proceedings, Part I*, volume 2023 of *Communications in Computer and Information Science*, pages 265–279.

Jian-Guo Zhang, Trung Bui, Seunghyun Yoon, Xiang Chen, Zhiwei Liu, Congying Xia, Quan Hung Tran, Walter Chang, and Philip S. Yu. 2021b. Few-shot intent detection via contrastive pre-training and fine-tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1906–1912.

Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC*, pages 378–389.

Henry Peng Zou and Cornelia Caragea. 2023. JointMatch: A unified approach for diverse and collaborative pseudo-labeling to semi-supervised text classification. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 7290–7301.

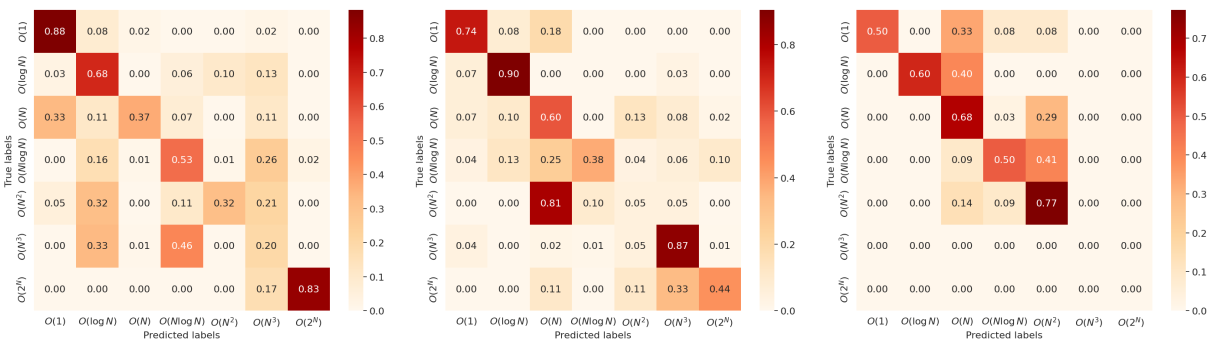
Henry Peng Zou, Cornelia Caragea, Yue Zhou, and Doina Caragea. 2023. Semi-supervised few-shot learning for fine-grained disaster tweet classification. In *Proceedings of the 20th International ISCRAM Conference*.

A TCProF Full Result

TCProF consists of three core parts. The first is a symbolic module $\mathbb{Sym}$ for pseudo-labeling. The second is a data augmentation module $\mathbb{AUG}$ which employs Back-Translation (BT), Loop-Conversion (LC), and both (BT+LC). Finally, the third one is a co-training module. Our hypothesis is that the more components implemented, the better the performance and eventually, TCProF, with all modules implemented, performs the best. The result of ST(CodeBERT) and ST(GraphCodeBERT) were omitted in Table 1, and the result of $\mathbb{AUG}_{BT}$ and$\mathbb{AUG}_{LC}$ were omitted in Table 2. We present the full experimental results regarding all the three components of TCProF in Tables 5 and 6, each for the accuracy and F1 performance.

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	5	10	5	10	5	10
SSL Baselines						
ST(CodeBERT)	22.82$\pm$ 0.00	26.47$\pm$ 6.33	16.33$\pm$ 0.24	15.51$\pm$ 1.18	35.79$\pm$ 0.00	35.79$\pm$ 0.00
ST(GraphCodeBERT)	22.89$\pm$ 0.13	20.51$\pm$ 5.67	16.19$\pm$ 0.00	23.57$\pm$ 12.78	36.14$\pm$ 0.61	35.79$\pm$ 0.00
ST(CodeT5+)	18.87$\pm$ 3.21	31.69$\pm$14.00	28.76$\pm$ 17.47	38.32$\pm$ 18.82	35.79$\pm$ 6.90	45.26$\pm$ 6.32
ST(UniXcoder)	15.44$\pm$ 6.30	31.92$\pm$ 6.14	26.02$\pm$ 11.72	40.98$\pm$ 1.14	37.89$\pm$ 6.57	45.61$\pm$ 11.59
JointMatch (CodeT5+)	14.62$\pm$ 4.26	24.68$\pm$ 2.60	20.97$\pm$ 3.90	21.04$\pm$ 4.33	36.49$\pm$ 4.98	42.11$\pm$ 6.57
JointMatch (UniXcoder)	9.62$\pm$ 5.23	19.39$\pm$ 3.86	14.68$\pm$ 11.78	20.76$\pm$ 10.46	35.44$\pm$ 6.85	48.42$\pm$ 8.22
CodeT5+						
+ $\mathbb{AUG}_{BT}$	20.13$\pm$ 3.31	26.77$\pm$ 5.38	39.34$\pm$19.17	45.36$\pm$11.58	34.73$\pm$ 7.59	38.94$\pm$ 2.79
+ $\mathbb{AUG}_{LC}$	26.47$\pm$ 5.49	29.53$\pm$ 5.83	32.92$\pm$14.47	53.89$\pm$16.03	35.78$\pm$ 5.47	42.45$\pm$ 7.97
+ $\mathbb{AUG}_{BT+LC}$	26.85$\pm$ 2.4	35.86$\pm$ 4.53	33.94$\pm$ 3.82	57.44$\pm$ 1.17	36.14$\pm$ 4.98	43.85$\pm$ 6.34
+ $\mathbb{Sym}$	21.92$\pm$ 2.54	35.50$\pm$11.18	32.10$\pm$ 6.78	46.38$\pm$ 6.58	36.84$\pm$ 1.72	42.81$\pm$ 2.76
+ $\mathbb{Sym}$ + $\mathbb{AUG}_{BT+LC}$	37.88$\pm$ 1.65	38.26$\pm$ 2.40	41.80$\pm$ 1.49	58.88$\pm$12.37	37.89$\pm$ 2.58	44.21$\pm$ 2.27
TCProF(CodeT5+)	38.63$\pm$ 1.32	41.98$\pm$ 2.92	44.74$\pm$ 3.26	59.29$\pm$ 3.71	50.53$\pm$ 0.86	51.93$\pm$ 3.88
UnixCoder						
+ $\mathbb{AUG}_{BT}$	29.75$\pm$ 3.17	33.40$\pm$ 14.43	50.31$\pm$19.27	52.93$\pm$10.66	33.15$\pm$ 3.72	50.17$\pm$ 9.43
+ $\mathbb{AUG}_{LC}$	30.72$\pm$ 8.40	33.03$\pm$ 6.55	44.53$\pm$13.40	53.55$\pm$ 4.10	38.94$\pm$ 3.16	47.01$\pm$ 4.38
+ $\mathbb{AUG}_{BT+LC}$	34.22$\pm$12.32	39.00$\pm$ 6.64	51.63$\pm$12.42	63.04$\pm$ 8.75	44.21$\pm$ 3.16	51.57$\pm$ 5.57
+ $\mathbb{Sym}$	41.76$\pm$ 2.62	45.34$\pm$ 2.91	39.82$\pm$ 5.13	51.02$\pm$ 2.14	50.88$\pm$ 8.48	54.04$\pm$ 6.56
+ $\mathbb{Sym}$ + $\mathbb{AUG}_{BT+LC}$	43.70$\pm$ 2.00	45.49$\pm$ 1.10	54.03$\pm$ 5.52	67.55$\pm$ 1.06	50.18$\pm$ 4.24	58.60$\pm$ 4.73
TCProF(UniXcoder)	**52.50**$\pm$ 1.56	**53.85**$\pm$ 3.63	**54.64**$\pm$ 3.77	**70.29**$\pm$ 2.06	**55.44**$\pm$ 0.99	**63.16**$\pm$ 2.27

Table 5: Accuracy performance comparisons. The scores are averaged from three runs with different seeds.



(a) CodeComplex (Java). (b) CodeComplex (Python). (c) CorCoD.

Figure 4: Confusion matrices of 10-shot TCProF(UniXcoder) performance.

Both results are consistent with our hypothesis. Comparing the performance of $\mathbb{AUG}_{BT}$ and $\mathbb{AUG}_{LC}$ with $\mathbb{AUG}_{BT+LC}$, the tendency of performance scores demonstrates that using both augmentation methods is superior to using the single method. Additional implementation of $\mathbb{Sym}$ also improves the scores. It is notable that some performance scores of $\mathbb{AUG}_{BT+LC}$ with high standard deviation scores show lesser standard deviation scores when implemented with $\mathbb{S}\mathrm{ym}$. This indicates that the usage of symbolic pseudo-labels and augmentation complements each other and enhances the model performance. Finally, with all modules and co-training strategy implemented, `TCProF`, performs the best.

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	5	10	5	10	5	10
SSL Baselines						
ST(CodeBERT)	5.31 ± 0.00	8.89 ± 6.20	4.27 ± 0.51	4.50 ± 0.90	10.54 ± 0.00	10.54 ± 0.00
ST(GraphCodeBERT)	5.52 ± 0.29	12.33 ± 11.97	3.98 ± 0.00	9.89 ± 10.24	12.82 ± 3.94	10.54 ± 0.00
ST(CodeT5+)	18.94 ± 3.69	29.20 ± 10.37	23.74 ± 12.26	28.48 ± 11.05	26.87 ± 17.65	45.17 ± 7.10
ST(UniXcoder)	13.52 ± 6.65	34.91 ± 5.73	21.61 ± 9.43	34.57 ± 0.80	38.92 ± 7.24	51.16 ± 8.61
JointMatch (CodeT5+)	7.43 ± 1.26	25.24 ± 1.51	13.61 ± 6.05	14.46 ± 3.18	30.31 ± 8.01	45.86 ± 7.78
JointMatch (UniXcoder)	9.70 ± 5.41	10.38 ± 5.20	7.87 ± 8.82	13.02 ± 8.85	18.98 ± 7.43	39.44 ± 11.93
CodeT5+						
+ $\mathbb{AUG}_{BT}$	22.90 ± 1.60	27.31 ± 3.13	31.94 ± 9.17	35.83 ± 4.52	35.80 ± 10.98	43.85 ± 2.99
+ $\mathbb{AUG}_{LC}$	25.33 ± 3.56	26.42 ± 4.02	26.60 ± 5.77	42.50 ± 8.51	39.36 ± 6.15	46.76 ± 6.72
+ $\mathbb{AUG}_{BT+LC}$	26.23 ± 2.07	30.69 ± 8.08	31.12 ± 5.61	42.38 ± 1.11	41.78 ± 5.12	47.98 ± 4.70
+ $\mathbb{S}\mathrm{ym}$	19.54 ± 1.76	30.52 ± 10.61	23.18 ± 2.75	34.69 ± 3.51	39.86 ± 1.77	46.66 ± 4.20
+ $\mathbb{S}\mathrm{ym}$ + $\mathbb{AUG}_{BT+LC}$	29.11 ± 1.64	32.72 ± 1.48	31.99 ± 1.64	42.95 ± 10.01	42.36 ± 8.08	48.09 ± 1.78
`TCProF`(CodeT5+)	28.58 ± 0.37	33.60 ± 0.98	34.49 ± 4.55	44.03 ± 2.74	45.13 ± 5.27	49.60 ± 8.12
UnixCoder						
+ $\mathbb{AUG}_{BT}$	29.18 ± 3.95	33.52 ± 10.03	31.43 ± 5.98	39.24 ± 3.60	38.82 ± 9.97	52.37 ± 8.00
+ $\mathbb{AUG}_{LC}$	27.24 ± 2.77	36.26 ± 16.28	29.48 ± 3.87	34.82 ± 4.26	40.33 ± 8.51	48.20 ± 13.78
+ $\mathbb{AUG}_{BT+LC}$	31.02 ± 14.22	34.21 ± 10.92	33.53 ± 5.73	45.95 ± 8.69	43.42 ± 1.29	52.57 ± 5.91
+ $\mathbb{S}\mathrm{ym}$	31.88 ± 1.28	44.74 ± 2.10	24.50 ± 3.48	40.77 ± 2.56	48.77 ± 6.32	54.12 ± 6.47
+ $\mathbb{S}\mathrm{ym}$ + $\mathbb{AUG}_{BT+LC}$	39.66 ± 0.44	45.31 ± 4.48	37.64 ± 4.90	51.58 ± 1.61	47.45 ± 4.93	56.87 ± 10.95
`TCProF`(UniXcoder)	**42.89** ± 3.09	**49.45** ± 0.97	**38.15** ± 4.09	**53.17** ± 2.13	**56.56** ± 3.52	**63.57** ± 3.09

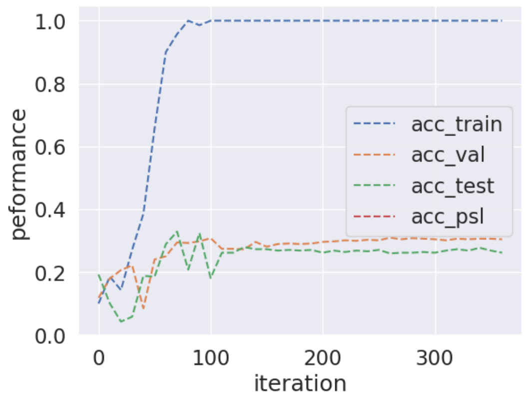
Table 6: F1-score performance comparisons. The scores are averaged from three runs with different seeds.

It is remarkable that our hypothesis holds on both the accuracy and F1 performance of `TCProF`. Intriguing facts are that while the SSL baselines tend to have higher standard deviation scores for F1 performance than accuracy, `TCProF` generally shows less standard deviation scores for F1 performance. This is because F1-scores evaluate how well the model performs for 'all' classes while accuracy does not consider whether the model is biased on specific classes. We provide confusion matrices of 10-shot `TCProF`(UniXcoder) for CodeComplex (Java), CodeComplex (Python), and CorCoD in Figures 4a, 4b, and 4c, respectively. We can see that `TCProF` produces unbiased performance compared to those seen in Figures 6b and 8b.

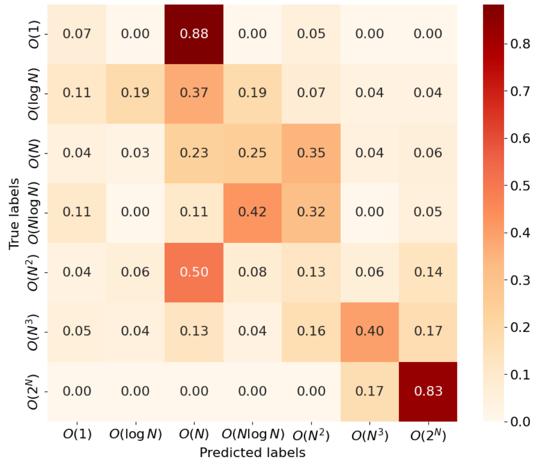
B Baseline Selection for Self-Training

From the four well-known code language models, CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5+, we experimented with which baseline models would fit for semi-supervised learning (SSL) in low-resource settings. Generally, UniXcoder and CodeT5+ are better than CodeBERT and GraphCode-BERT. We can also see the same tendency comparing the performance of each model trained with the full data in Table 8. Additionally analyzing for low-resource settings, we selected two best-performing models, UniXcoder and CodeT5+ to implement `TCProF`.

CodeBERT and GraphCodeBERT have satisfactory accuracy for 5-shot CodeComplex (Java) and CorCoD compared to the other two models as shown in Table 5. However, Table 6 reveals that CodeBERT and GraphCodeBERT perform significantly lower than that of the other two models. The large performance gap of CodeBERT and GraphCodeBERT between Tables 5 and 6 and the low F1 performance of these two models indicate that they are unsuitable to implement our methodology.

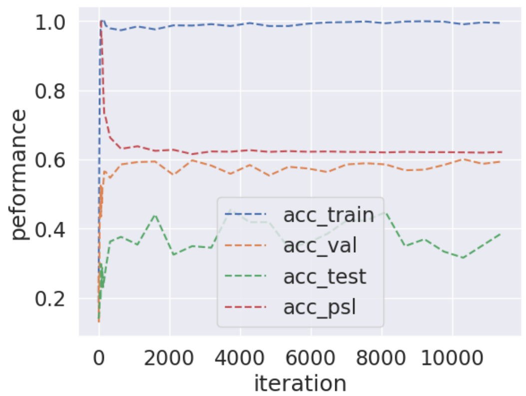


(a) Learning curves and accuracy shifts in CodeT5+ self-training

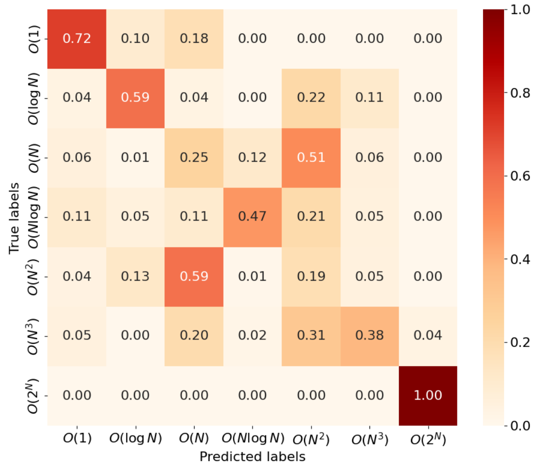


(b) Confusion matrix of CodeT5+ self-training

Figure 5: CodeT5+ self-learning performance visualization for CodeComplex (Java) 10-shot.



(a) Learning curves and accuracy shifts in UniXcoder self-training



(b) Confusion matrix of UniXcoder self-training

Figure 6: UniXcoder self-learning performance visualization for CodeComplex (Java) 10-shot.

The relatively high performance of UniXcoder and CodeT5+ in Tables 5 and 6 empirically prove that they are suitable for low-resource settings. From confusion matrices of the two models in Figures 5b and 6b, we can also easily see that UniXcoder and CodeT5+ do not overfit to a certain class. Furthermore, we depict the learning processes of these two models in Figures 5a and 6a. In the figures, `acc_train`, `acc_val`, `acc_test`, and `acc_psl` indicate the accuracy of the train, valid, test, and pseudo-labeled datasets, respectively. These figures demonstrate that the converging patterns of `acc_train`, `acc_val`, `acc_test` are similar, indicating that the two models are both suitable for training in low-resource settings. From these two models, we also analyze which model is more suitable for low-resource settings, and eventually `TCProF`.

In Figure 5a, `acc_psl` is not shown in the graph, meaning that CodeT5+ does not produce pseudo-labels in the self-training at all. This is because pseudo-labels are produced based on the confidence score and the confidence score of CodeT5+ does not match the pre-defined threshold. It is quite surprising as CodeT5+ has high `acc_train`. We have run an experiment of training CodeT5+ continuously to inspect when CodeT5+ generates pseudo-labels and the model produced biased pseudo-labels for only a few classes for CodeComplex (Python). Thus, independent to the performance, CodeT5+ is not suitable for pseudo-labeling when implemented with only self-training.

UniXcoder, on the other hand, surely generates pseudo-labels referring to Figure 6a. For all datasets, UniXcoder generates pseudo-labels for all 7 classes, indicating that the model shows consistent performance without being biased toward a specific class. Figure 6b also confirms that UniXcoder shows relatively well-distributed performance for each class in test data. This confirms that UniXcoder is more suitable that CodeT5+ in a low-resource setting.

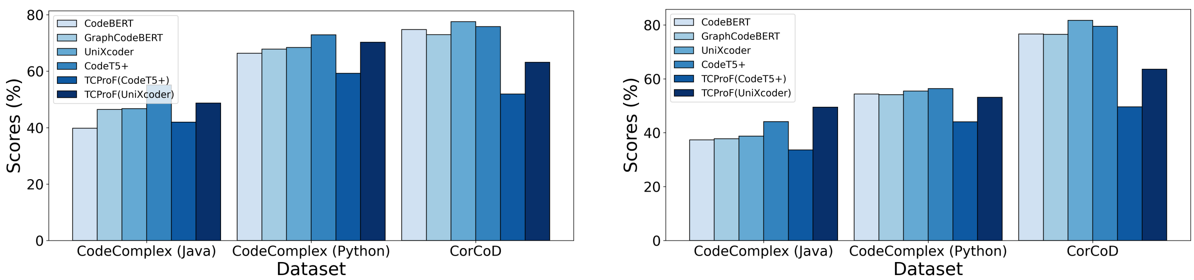
Referring to Table 1 in Section 5.1, we can see that our proposed framework TCProF effectively improves the performance of both UniXcoder and CodeT5+ but we can also notice that UniXcoder is more effective for TCProF and eventually, low-resource settings.

C Comparison to Baselines Fine-Tuned with Full Dataset

We have provided a comparative analysis of TCProF and baselines fine-tuned with the entire train dataset in Section 5.4. Detailed results for accuracy and F1-scores are shown in Table 7 and Figures 7a and 7b provide full results of accuracy and F1 performance.

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	Acc.	F1	Acc.	F1	Acc.	F1
Baselines trained with the Full train dataset						
CodeBERT	39.82± 6.90	37.35± 3.56	66.39± 1.43	54.34± 0.93	74.74± 1.72	76.64± 3.07
GraphCodeBERT	46.46± 2.18	37.75± 1.18	67.83± 8.36	54.13± 7.25	72.98± 1.79	76.48± 2.26
UniXcoder	46.76± 1.91	38.76± 0.39	68.44± 4.66	55.45± 3.33	**77.54**± 1.31	**81.69**± 1.43
CodeT5+	**55.11**± 2.18	44.14± 3.54	**72.88**± 1.95	**56.39**± 2.25	75.79± 2.27	79.51± 1.96
10-shot performance of TCProF						
TCProF(CodeT5+)	41.98±2.92	33.60 ±0.98	59.29±3.71	44.03±2.74	51.93± 3.88	49.60± 8.12
TCProF(UniXcoder)	53.85±3.63	**49.45**±0.97	70.29±2.06	53.17±2.13	63.16± 2.27	63.57± 3.09

Table 7: Performance comparison of baselines trained with the full-train dataset and TCProF 10-shot performance.



(a) Accuracy comparison.

(b) Fl-score comparison.

Figure 7: Visualized performance of baselines trained with full train dataset and TCProF 10-shot.

We can see that TCProF effectively catches up with both the performance of accuracy and F1-scores. It is remarkable TCProF even exceeds the F1 performance of CodeT5+ for CodeComplex (Java), accomplishing the best F1-scores for the dataset.

D Baselines Fine-Tuned with the Augmented Full Dataset

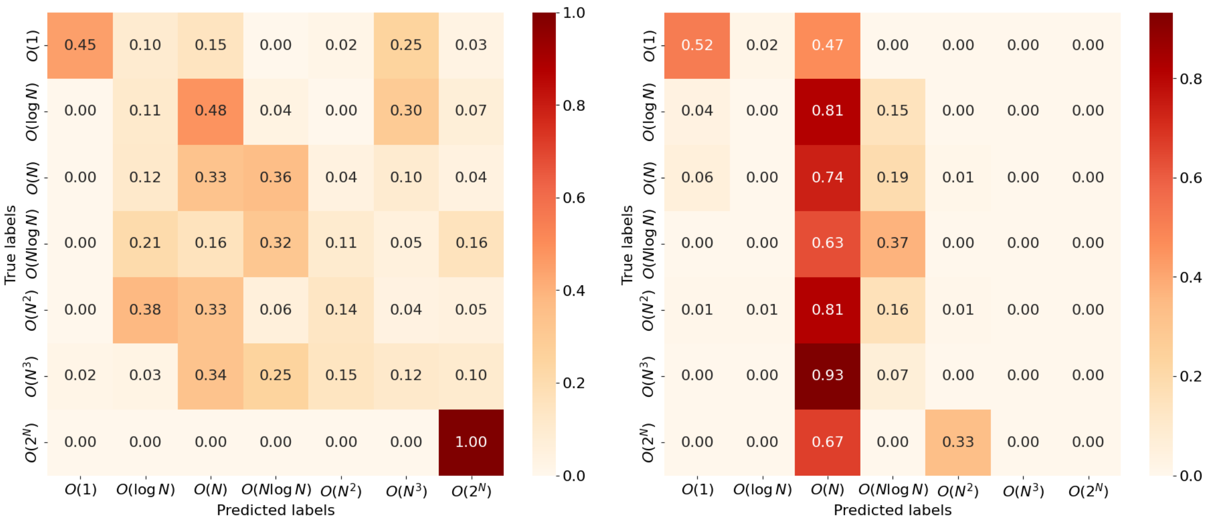
We have implemented $\mathbb{AUG}_{BT+LC}$, which combines Back-Translation (BT) and Loop-Conversion (LC) to investigate the impact of data augmentation on the full dataset. The results in Table 8 demonstrate the effectiveness of augmentation across the baselines and datasets. Notably, CodeT5+ shows a significant enhancement, particularly in Python, where it achieves over 90% accuracy and 85% F1-scores. This improvement is consistent across different models, indicating that $\mathbb{AUG}_{BT+LC}$ is beneficial in predicting time complexity more accurately and reliably.

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	Acc.	F1	Acc.	F1	Acc.	F1
Original						
CodeBERT	39.82 ± 6.90	37.35 ± 3.56	66.39 ± 1.43	54.34 ± 0.93	74.74 ± 1.72	76.64 ± 3.07
GraphCodeBERT	46.46 ± 2.18	37.75 ± 1.18	67.83 ± 8.36	54.13 ± 7.25	72.98 ± 1.79	76.48 ± 2.26
UniXcoder	46.76 ± 1.91	38.76 ± 0.39	68.44 ± 4.66	55.45 ± 3.33	77.54 ± 1.31	81.69 ± 1.43
CodeT5+	55.11 ± 2.18	44.14 ± 3.54	72.88 ± 1.95	56.39 ± 2.25	75.79 ± 2.27	79.51 ± 1.96
Augmented						
CodeBERT	50.41 ± 2.84	42.11 ± 1.99	81.83 ± 1.79	69.20 ± 4.07	78.60 ± 0.99	80.06 ± 0.69
GraphCodeBERT	58.69 ± 2.89	46.96 ± 2.25	85.31 ± 0.59	77.82 ± 0.62	78.60 ± 0.99	81.08 ± 0.80
UniXcoder	**63.91** ± 1.28	**49.07** ± 1.18	86.13 ± 1.58	77.47 ± 2.80	78.95 ± 0.00	82.22 ± 0.50
CodeT5+	59.51 ± 0.80	48.79 ± 1.38	**90.91** ± 2.25	**85.67** ± 6.11	**79.65** ± 0.50	**82.65** ± 0.62

Table 8: Performance of baselines trained with augmented train dataset. The scores are averaged from three runs with different seeds.

E Comparison to JointMatch

JointMatch is a recognized SSL approach. It displays the state-of-the-art performance for text classification in low-resource settings. However, we found this unsuitable for the code time complexity prediction task. Tables 5 and 6 illustrates that JointMatch underperforms compared to the standard self-training approaches. Confusion matrices in Figures 8a and 8b provide further details that JointMatch is either biased on several classes or is underfitted. This is even more remarkable as Figures 5b and 6b displays much better performance. This poor performance of JointMatch is that the model is primarily developed for text classification datasets such as AG News, Yahoo! Answers, and IMDB. These datasets are completely different from the code time complexity prediction datasets and thus, it is reasonable that the learning mechanism of JointMatch is not effective on the time complexity prediction.



(a) Confusion matrix of CodeT5+ JointMatch

(b) Confusion matrix of UniXcoder JointMatch

Figure 8: CodeT5+, UniXcoder JointMatch performance visualization for CodeComplex (Java) 10-shot.

F LLM Comparison

Table 9 illustrates the accuracy and F1 performance of LLMs and `TCProF`. Likewise to the analysis from Section 5.5, `TCProF` demonstrates performance competitive to the LLMs for F1-scores as well.

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	Acc.	F1	Acc.	F1	Acc.	F1
GPT3.5	62.15	37.96	32.55	29.24	69.42	43.29
GPT4	**64.01**	46.90	53.04	45.15	**78.86**	56.95
Gemini-pro	49.54	29.73	31.05	29.76	61.91	41.26
`TCProF`(UniXcoder)	53.85	**49.45**	70.29	53.17	63.16	63.57

Table 9: Comparison with the performance of LLM and 10-shot `TCProF`.

G Usage of Sym as a Code Time Complexity Classifier

Our symbolic module, Sym, is specifically designed to enhance the pseudo-labeling process within the `TCProF`, aiming to improve the code time complexity prediction accuracy. This module leverages symbolic reasoning to generate more reliable pseudo-labels, empirically proven in Tables 1 and 2. For an extensive analysis, we present the experimental results in Table 10, using Sym alone to classify the time complexity of given code snippets.

	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
	Acc.	F1	Acc.	F1	Acc.	F1
Sym	49.69	40.83	55.76	41.27	53.68	37.53

Table 10: The performance of Sym as a time complexity classifier of the code snippets.

Notably, Sym shows relatively better performance on Java datasets compared to Python datasets. This difference can be attributed to the strict syntax of Java, which aids Sym in more effective identification and process of loop structures—a critical aspect for time complexity analysis. Although the overall performance of Sym is promising, its integration with the broader `TCProF`, demonstrates the most effective results. Appendix H illustrates with examples, the detailed procedure of how Sym operates.

While Sym does not capture every potential operation involved in computing time complexity, its design is strategically tailored to assist the baseline model's capability in producing more precise pseudo-labels. Moving forward, we recognize the potential for further enhancing Sym by expanding its scope to include a broader range of conditions. This development is part of our future research aimed to broaden our understanding of code time complexity prediction.

H Symbolic Pseudo-Labeling Running Example

In the symbolic module Sym, the first step involves employing regular expressions (Regex) to ascertain the existence of functions within the source code. Figure 9 presents a Python code snippet that contains a function named `solve()`. The module recognizes `solve()` as a function due to the presence of the keyword `def`. Subsequently, Sym extends its analysis to detect loops and recursion, again utilizing Regex.

In the given example, Sym identifies two for-loops due to the the keyword `for`: one within the `solve()` function and another in the main code section. In the main code section, Sym checks `solve()` is called only once. Through Regex matching, the module employs the keyword `in`, along with the associated range variables `aa` and `n`, to determine the size of each loop. As depicted in Figure 9, Sym calculates the time complexity of for loop in `solve()` as $O(N)$. Additionally, the module detects the use of the keyword `sorted`, which leads to the derivation of the combined time complexity for `solve()` as $O(N) + O(N \log N)$.

Finally, Sym computes the overall time complexity for the main code section. Given the presence of an additional for loop in this code section, the module calculates the total time complexity as $O(N) + O(N) + O(N \log N) = O(N \log N)$.

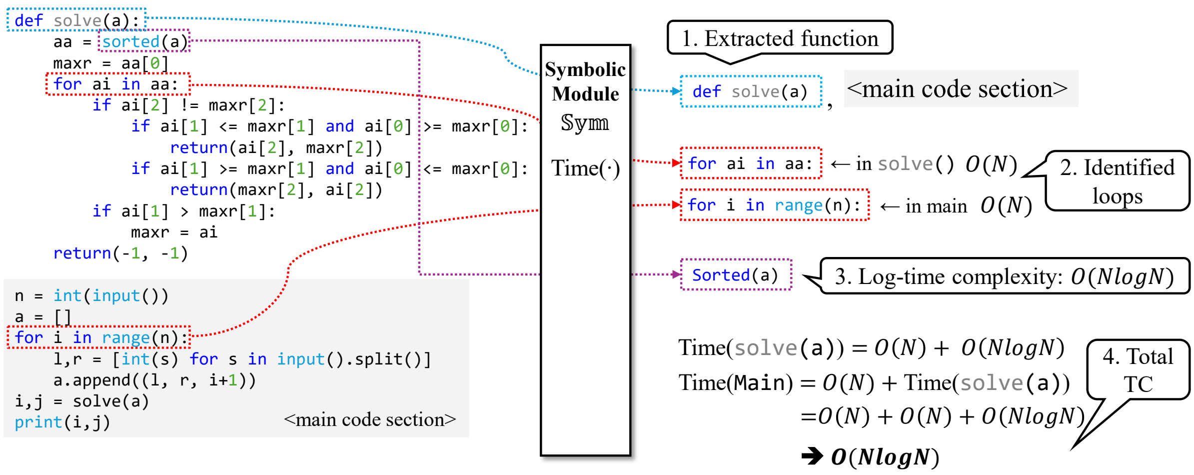
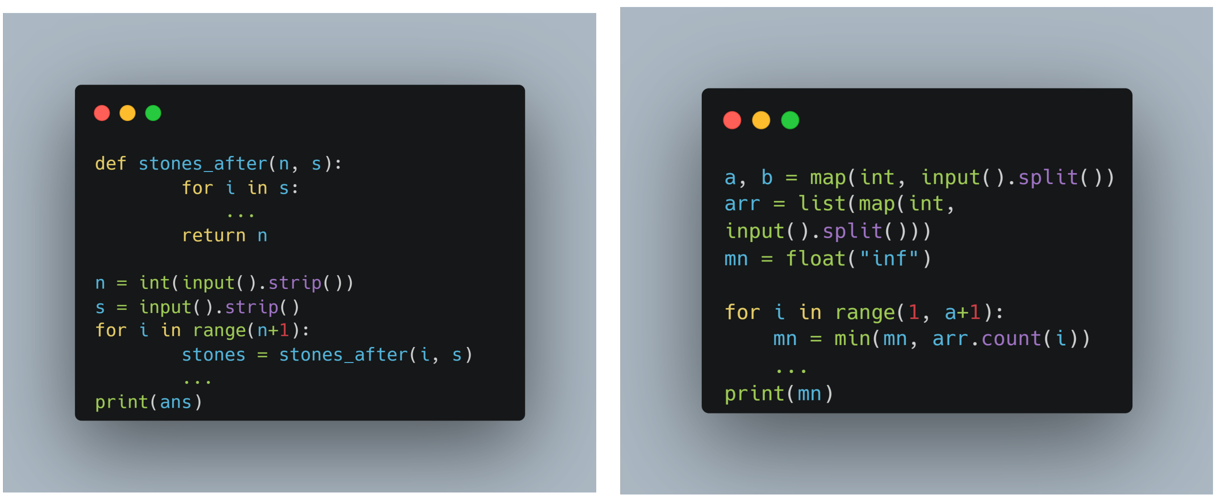


Figure 9: Execution process of the Symbolic Module using Python code snippet.

I Error Analysis

From the test cases, we have analyzed the errors, and as discussed in Section 5.3, $O(N^2) \rightarrow O(N)$ errors occur the most. Without sufficient attention to the input size, it is sometimes confusing, even for human experts, to differentiate two codes, each in $O(N)$ and $O(N^2)$ classes, respectively. For instance, the code instance in Figure 10a is rather straightforward. The `stones_after` function is called in the main part which runs linear to the input size. The main part calls the function n times and thus, the time complexity of the whole code in $O(N^2)$. However, the code in Figure 10b contains `count` operation which runs linear to the size of `arr`. If the code does not have the knowledge on the operation, the model is likely to predict its time complexity as $O(N)$, which is wrong. This is mostly seen in the codes of class $O(N^2)$ and thus, the majority of errors are the $O(N^2) \rightarrow O(N)$.



(a) Correct code example predicted $O(N^2)$.

(b) Incorrect code example predicted $O(N)$.

Figure 10: Code examples of $O(N^2)$ class.

J Extensive Analyses on Augmentation

We have performed an extensive analysis of our augmentation strategy across four models: CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5+. As stated in Appendix B, CodeBERT and GraphcodeBERT are unsuitable for self-training. Similarly, in this experiment, we notice that CodeBERT and Graphcode-BERT have no consistent tendency of the results. In contrast, UniXcoder and CodeT5+ exhibit more

Models	Augmentation	CodeComplex (Java) 5	10	CodeComplex (Python) 5	10	CorCoD 5	10
CodeBERT	$\mathbb{AUG}_{BT}$	21.85±2.61	29.23±3.18	25.06±8.16	43.44±5.19	35.78±9.18	38.94±5.27
	$\mathbb{AUG}_{LC\_Natural}$	26.32±4.20	30.20±6.21	39.82±1.56	50.61±6.05	33.33±5.40	34.03±2.43
	$\mathbb{AUG}_{LC\_Artificial}$	17.52±8.20	20.06±8.13	38.86±7.60	45.42±3.64	28.42±2.11	33.68±1.05
	$\mathbb{AUG}_{BT+LC\_Natural}$	25.72±0.23	38.10±1.31	39.82±8.30	55.53±10.25	36.13±1.61	41.05±1.82
	$\mathbb{AUG}_{BT+LC\_Artificial}$	24.75±6.28	39.30±4.21	43.24±6.76	43.30±3.86	28.77±5.40	40.00±4.22
GraphCodeBERT	$\mathbb{AUG}_{BT}$	29.00±10.99	36.91±4.62	34.22±7.28	47.40±7.63	32.65±3.82	44.56±5.30
	$\mathbb{AUG}_{LC\_Natural}$	25.87±3.07	24.76±1.03	29.64±2.39	47.06±7.40	36.49±2.49	45.26±8.42
	$\mathbb{AUG}_{LC\_Artificial}$	22.07±1.05	30.42±2.93	33.46±15.61	46.30±6.85	34.38±3.38	41.75±8.83
	$\mathbb{AUG}_{BT+LC\_Natural}$	25.80±2.40	33.85±2.29	34.15±12.08	47.88±3.80	38.94±2.78	43.86±7.62
	$\mathbb{AUG}_{BT+LC\_Artificial}$	21.10±1.79	35.46±8.07	45.21±7.89	52.59±4.69	37.89±4.59	40.65±7.40
CodeT5+	$\mathbb{AUG}_{BT}$	20.13±3.31	26.77±5.38	39.34±19.44	45.36±11.58	34.73±7.59	38.94±2.79
	$\mathbb{AUG}_{LC\_Natural}$	26.47±5.49	29.53±5.83	32.92±14.47	53.89±16.03	35.78±5.47	42.45±7.97
	$\mathbb{AUG}_{LC\_Artificial}$	21.25±8.79	31.09±6.45	33.33±17.78	60.86±8.22	27.37±2.78	41.40±5.80
	$\mathbb{AUG}_{BT+LC\_Natural}$	26.85±1.14	35.86±4.53	33.94±3.82	57.44±1.17	36.14±4.98	43.85±6.34
	$\mathbb{AUG}_{BT+LC\_Artificial}$	24.53±7.18	36.16±5.24	40.43±17.16	59.15±1.60	31.93±2.19	40.69±4.26
UniXcoder	$\mathbb{AUG}_{BT}$	29.75±3.17	33.40±14.43	50.31±19.27	52.93±10.66	33.15±3.72	50.17±9.43
	$\mathbb{AUG}_{LC\_Natural}$	30.72±8.40	33.03±6.55	44.53±13.40	53.55±4.10	38.94±3.16	47.01±4.38
	$\mathbb{AUG}_{LC\_Artificial}$	22.81±6.02	42.28±16.47	40.91±5.95	50.75±8.10	37.89±8.42	43.15±3.16
	$\mathbb{AUG}_{BT+LC\_Natural}$	34.22±12.32	39.00±6.64	51.63±12.42	63.04±8.75	44.21±3.16	51.57±5.57
	$\mathbb{AUG}_{BT+LC\_Artificial}$	32.73±7.54	40.19±15.17	47.33±13.84	59.15±7.17	38.59±9.78	51.93±6.77

Table 11: Accuracy of augmentation strategies, $\mathbb{AUG}_{BT}$ and $\mathbb{AUG}_{LC}$ and $\mathbb{AUG}_{BT+LC}$ in natural and artificial settings. The scores are averaged from three runs with different seeds.

consistent performance tendencies. Therefore, this section focuses on the accuracy and F1-score results of UniXcoder and CodeT5+.

$\mathbb{AUG}_{BT}$ involves augmenting data through Back-Translation of the original labeled code. $\mathbb{AUG}_{LC\_Natural}$ denotes the experimental condition where the initial labeled data is used without specific constraints. In contrast, $\mathbb{AUG}_{LC\_Artificial}$ restricts the initial labeled data to code containing either for loops or while loops, ensuring that all augmented Loop-Conversion data can be generated for every initial labeled data. The combined strategy $\mathbb{AUG}_{BT+LC\_Natural}$ integrates Back-Translation augmentation with Loop-Conversion augmentation under natural conditions. Similarly, $\mathbb{AUG}_{BT+LC\_Artificial}$ combines Back-Translation augmentation with Loop-Conversion augmentation under artificial conditions.

We analyzed the results in two parts: $\mathbb{AUG}_{LC\_Natural}$ vs. $\mathbb{AUG}_{LC\_Artificial}$ and $\mathbb{AUG}_{BT}$ vs. $\mathbb{AUG}_{LC\_Natural}$.

The first comparison, $\mathbb{AUG}_{LC\_Natural}$ and $\mathbb{AUG}_{LC\_Artificial}$, highlights two different augmentation strategies within $\mathbb{AUG}_{LC\_Natural}$. Notably, $\mathbb{AUG}_{LC\_Natural}$ significantly enhances model performance compared to $\mathbb{AUG}_{LC\_Artificial}$, particularly in the UniXcoder model. The average number of data points used for training in the $\mathbb{AUG}_{LC\_Natural}$ is 65, while the $\mathbb{AUG}_{LC\_Artificial}$ uses 70 data in 5-shot settings on the CodeComplex dataset, which has seven labels. Despite having less training data, $\mathbb{AUG}_{LC\_Natural}$ achieves higher performance.

The second comparison, $\mathbb{AUG}_{BT}$ vs. $\mathbb{AUG}_{LC\_Natural}$, evaluates the differences between $\mathbb{AUG}_{BT}$ and $\mathbb{AUG}_{LC\_Natural}$. In Tables 11 and 12, $\mathbb{AUG}_{LC\_Natural}$ consistently outperforms $\mathbb{AUG}_{BT}$ in both accuracy and F1-score across all evaluated models while maintaining lower standard deviations, indicating more reliable and stable performance. The average number of data points used for training in the $\mathbb{AUG}_{LC\_Natural}$ is 65, while the $\mathbb{AUG}_{BT}$ uses 70 data in 5-shot settings on the CodeComplex dataset, which has seven labels. Despite having less training data, $\mathbb{AUG}_{LC\_Natural}$ achieves higher performance.

For the CodeT5+ model, $\mathbb{AUG}_{LC\_Natural}$ achieves higher accuracy except in the CodeComplex Python 5-shot setting, where a high standard deviation of 19.44% was observed in $\mathbb{AUG}_{BT}$. Similarly, for the UniXcoder model, $\mathbb{AUG}_{LC\_Natural}$ achieves higher accuracy in most cases, except in the CodeComplex

Models	Augmentation	CodeComplex (Java)		CodeComplex (Python)		CorCoD	
		5	10	5	10	5	10
CodeBERT	$\mathbb{AUG}_{BT}$	$19.94_{\pm 5.56}$	$27.48_{\pm 6.51}$	$19.90_{\pm 5.09}$	$25.46_{\pm 7.78}$	$33.45_{\pm 8.85}$	$35.44_{\pm 10.80}$
	$\mathbb{AUG}_{LC\_Natural}$	$28.54_{\pm 0.60}$	$34.38_{\pm 1.04}$	$22.75_{\pm 5.23}$	$33.69_{\pm 5.40}$	$33.09_{\pm 10.91}$	$24.47_{\pm 3.93}$
	$\mathbb{AUG}_{LC\_Artificial}$	$15.22_{\pm 7.17}$	$15.78_{\pm 5.74}$	$27.81_{\pm 8.68}$	$33.00_{\pm 6.67}$	$18.37_{\pm 3.60}$	$32.31_{\pm 7.57}$
	$\mathbb{AUG}_{BT+LC\_Natural}$	$23.95_{\pm 1.42}$	$28.23_{\pm 0.36}$	$28.34_{\pm 3.43}$	$35.95_{\pm 7.99}$	$29.80_{\pm 5.94}$	$46.39_{\pm 3.01}$
	$\mathbb{AUG}_{BT+LC\_Artificial}$	$22.45_{\pm 7.29}$	$34.14_{\pm 6.96}$	$28.19_{\pm 7.91}$	$34.12_{\pm 6.02}$	$22.30_{\pm 1.45}$	$41.05_{\pm 4.25}$
GraphCodeBERT	$\mathbb{AUG}_{BT}$	$24.77_{\pm 12.26}$	$29.14_{\pm 5.32}$	$25.78_{\pm 4.32}$	$33.16_{\pm 2.92}$	$27.36_{\pm 6.60}$	$44.89_{\pm 5.06}$
	$\mathbb{AUG}_{LC\_Natural}$	$26.43_{\pm 6.25}$	$22.30_{\pm 2.24}$	$25.75_{\pm 9.66}$	$33.39_{\pm 1.79}$	$34.29_{\pm 8.78}$	$43.88_{\pm 4.57}$
	$\mathbb{AUG}_{LC\_Artificial}$	$15.59_{\pm 6.34}$	$22.18_{\pm 3.79}$	$26.07_{\pm 5.80}$	$36.00_{\pm 3.21}$	$40.69_{\pm 4.65}$	$41.71_{\pm 7.23}$
	$\mathbb{AUG}_{BT+LC\_Natural}$	$20.69_{\pm 4.23}$	$35.64_{\pm 7.23}$	$27.33_{\pm 5.72}$	$35.99_{\pm 4.39}$	$38.17_{\pm 1.78}$	$40.76_{\pm 6.64}$
	$\mathbb{AUG}_{BT+LC\_Artificial}$	$20.98_{\pm 1.78}$	$34.34_{\pm 2.00}$	$31.30_{\pm 3.69}$	$42.97_{\pm 2.52}$	$40.96_{\pm 4.84}$	$36.77_{\pm 5.18}$
CodeT5+	$\mathbb{AUG}_{BT}$	$22.90_{\pm 1.60}$	$27.31_{\pm 3.13}$	$31.94_{\pm 9.17}$	$35.83_{\pm 4.52}$	$35.80_{\pm 10.98}$	$43.85_{\pm 2.99}$
	$\mathbb{AUG}_{LC\_Natural}$	$25.33_{\pm 3.56}$	$26.42_{\pm 4.02}$	$26.60_{\pm 5.77}$	$42.50_{\pm 8.51}$	$39.36_{\pm 6.15}$	$46.76_{\pm 6.72}$
	$\mathbb{AUG}_{LC\_Artificial}$	$16.67_{\pm 4.92}$	$27.68_{\pm 8.33}$	$34.73_{\pm 5.42}$	$43.37_{\pm 4.71}$	$27.92_{\pm 3.56}$	$44.87_{\pm 7.29}$
	$\mathbb{AUG}_{BT+LC\_Natural}$	$26.23_{\pm 2.07}$	$30.69_{\pm 8.08}$	$31.12_{\pm 5.61}$	$42.38_{\pm 1.11}$	$41.78_{\pm 5.12}$	$47.98_{\pm 4.70}$
	$\mathbb{AUG}_{BT+LC\_Artificial}$	$21.70_{\pm 5.73}$	$32.92_{\pm 3.43}$	$33.27_{\pm 2.25}$	$43.07_{\pm 1.69}$	$30.77_{\pm 8.12}$	$48.19_{\pm 4.78}$
UniXcoder	$\mathbb{AUG}_{BT}$	$29.18_{\pm 3.95}$	$33.52_{\pm 10.03}$	$31.43_{\pm 5.98}$	$39.24_{\pm 3.60}$	$38.82_{\pm 9.97}$	$52.39_{\pm 8.03}$
	$\mathbb{AUG}_{LC\_Natural}$	$27.24_{\pm 2.77}$	$36.26_{\pm 16.28}$	$29.48_{\pm 3.87}$	$34.82_{\pm 4.26}$	$40.33_{\pm 8.51}$	$48.20_{\pm 13.78}$
	$\mathbb{AUG}_{LC\_Artificial}$	$28.08_{\pm 13.97}$	$35.58_{\pm 8.97}$	$31.11_{\pm 7.11}$	$35.81_{\pm 4.66}$	$38.41_{\pm 9.57}$	$44.36_{\pm 7.43}$
	$\mathbb{AUG}_{BT+LC\_Natural}$	$31.02_{\pm 14.22}$	$34.21_{\pm 10.92}$	$33.53_{\pm 5.73}$	$45.95_{\pm 8.69}$	$43.42_{\pm 1.29}$	$52.57_{\pm 5.91}$
	$\mathbb{AUG}_{BT+LC\_Artificial}$	$29.34_{\pm 6.21}$	$38.99_{\pm 5.56}$	$30.94_{\pm 6.00}$	$42.07_{\pm 4.98}$	$40.62_{\pm 10.50}$	$51.83_{\pm 5.38}$

Table 12: F1-scores of augmentation strategies, $\mathbb{AUG}_{BT}$ and $\mathbb{AUG}_{LC}$ and $\mathbb{AUG}_{BT+LC}$ in natural and artificial settings. The scores are averaged from three runs with different seeds.

Python 5-shot setting, where it shows 6%p lower accuracy. These large performance differences are likely due to the inherent variability in the CodeComplex Python data results, contributing to the instability observed across all models. Despite this 6%p lower standard deviation compared to $\mathbb{AUG}_{BT}$ in the CodeComplex Python 5-shot setting, indicating more stable performance even with a slight drop in accuracy. In the 10-shot setting, where the data is more comprehensive, $\mathbb{AUG}_{LC\_Natural}$ outperforms $\mathbb{AUG}_{BT}$ in accuracy.

There are some exceptions, such as in the CodeComplex Java 10-shot setting, likely due to the limited diversity of training data. $\mathbb{AUG}_{LC\_Natural}$ focuses on converting loop parts rather than making substantial code changes, while $\mathbb{AUG}_{BT}$ enhances the diversity of the training data. When $\mathbb{AUG}_{BT}$ is added to the training data, it helps bridge the performance gap observed in these exceptional cases, as evidenced by the results for $\mathbb{AUG}_{BT+LC\_Artificial}$ in CodeComplex Java 10-shot setting. These findings suggest that $\mathbb{AUG}_{LC\_Natural}$ is the most effective augmentation strategy for enhancing the performance of code models. Moreover, combining $\mathbb{AUG}_{BT}$ with $\mathbb{AUG}_{LC\_Natural}$ further boosts the performance of the models.

K Augmentation Prompts

The augmentation prompts presented in this section are designed to enhance the capabilities of language models in code Back-Translation and Loop-Conversion tasks. The first prompt, Back-Translation, challenges the model to translate Java code into Python and then back into Java, ensuring functional equivalence without syntactical similarity to the original code. The second prompt, Loop-Conversion, requires the model to convert all ‘for’ loops to ‘while’ loops and vice versa in the given Java code, even at the cost of readability. To illustrate the effectiveness of these methods, we provide both original code and augmented code examples for each augmentation technique. Note that we only show parts of the codes because they are too long to display.

K.1 Back-Translation Examples

The following is an original example of a Java code.

```
import ...
public class ProblemD {
        public static void main(String[] args) throws IOException {
                BufferedReader s = new BufferedReader(new InputStreamReader(System.in));
                PrintWriter out = new PrintWriter(System.out);
                String[] data = s.readLine().split(" ");
                String[] line = s.readLine().split(" ");
            double ans = solve(a, v, l, d, w);
                out.println(String.format("\%.07f", ans));
                out.flush();
        }
        private static double solve(double a, double v, double l, double d, double w) {
                double maxSpeedAtD = Math.sqrt(2 * d / a) * a;
                if (v <= w || maxSpeedAtD <= w) {
                        // okay. just go
                ...
                }
                double time = 0.0d;
                double maxSpeedTime = Math.sqrt((d / a) + (w * w / (2 * a * a)));
                double maxSpeed = maxSpeedTime * a;
                if (maxSpeed <= v) {
                ...
                }

                // after that, just go.
                double timeToV = (v - w) / a;
                double timeToVLen = timeToV * w + 0.5 * timeToV * (v - w);
                if (timeToVLen <= l - d) {
                ...
                }
                return time;
        }
}
```

The following is a transformed example of a Java code.

```
import ...

public class ProblemD {
        public static void main(String[] args) throws IOException {
                BufferedReader s = new BufferedReader(new InputStreamReader(System.in));
                PrintWriter out = new PrintWriter(System.out);

                String[] data = s.readLine().split(" ");
                String[] line = s.readLine().split(" ");

                double ans = solve(a, v, l, d, w);
                out.println(String.format("%.07f", ans));

                out.flush();
        }

        private static double solve(double a, double v, double l, double d, double w) {
                double maxSpeedAtD = Math.sqrt(2 * d / a) * a;
                if (v >= w && maxSpeedAtD > w) {
                ...
                }
        }

        private static double findTimeToReachL(double a, double v, double l) {
            ...
                return timeToMaxSpeed + leftDist / v;
        }

        private static double calculateTimeToReachDestination(double a, double v, double l,
        ↪  double d, double w) {
```

```
        ...
            return maxSpeed <= v ? maxSpeedTime + (a * maxSpeedTime - w) / a :
            ↪  calculateTimeForCaseThree(a, v, d, w, maxSpeedTime);
    }

    private static double calculateTimeForCaseThree(double a, double v, double d, double w,
    ↪  double maxSpeedTime) {
        ...
            return v > w ? v / a + time : v / a + vtime + (v - w) / a;
    }
}
```

K.2 Loop-Conversion Examples

The following is an original example of a Java code.

```
import java.util.*;
import java.io.*;

public class Soldiers {
        public static void main(String[] args) throws IOException {
                new Soldiers().run();
        }

        void run() throws IOException {
                br = new BufferedReader(new InputStreamReader(System.in));
                pw = new PrintWriter(System.out);

                int n = nextInt();
                pw.println(3 * (n / 2));

                pw.close();
        }

        BufferedReader br;
        StringTokenizer st;
        PrintWriter pw;

        String next() throws IOException {
                while (st == null || !st.hasMoreTokens()) {
                        st = new StringTokenizer(br.readLine());
                }
                return st.nextToken();
        }

        int nextInt() throws IOException {
                return Integer.parseInt(next());
        }

        double nextDouble() throws IOException {
                return Double.parseDouble(next());
        }
}
```

The following is a transformed example of a Java code.

```
import java.util.*;
import java.io.*;

public class Soldiers {
        public static void main(String[] args) throws IOException {
                new Soldiers().run();
        }
```

```
	void run() throws IOException {
		br = new BufferedReader(new InputStreamReader(System.in));
		pw = new PrintWriter(System.out);

		int n = nextInt();
		pw.println(3 * (n / 2));

		pw.close();
	}

	BufferedReader br;
	StringTokenizer st;
	PrintWriter pw;

	String next() throws IOException {
		for(; st == null || !st.hasMoreTokens(); st = new
		↪  StringTokenizer(br.readLine())) {
		}
		return st.nextToken();
	}

	int nextInt() throws IOException {
		return Integer.parseInt(next());
	}

	double nextDouble() throws IOException {
		return Double.parseDouble(next());
	}
}
```

K.3 Back-Translation Prompt

Back-Translation Prompt

You are a powerful AI that can translate Java code into Python code and then back into Java code.
I will provide you with a piece of Java code.
Your task is to generate an equivalent Python code and then translate it back into Java.
When translating back into Java, the code should not be identical to the original Java code, but it should still be functionally equivalent.
Please ensure that the output is valid Java and does not contain any syntax or constructs from the other language. The Java output should be in the following JSON format:
{
"back-translation": "[Your transformed Java code here]"
}.
Please note that "[Your transformed Java code here]" should be replaced with your transformed Java code.
Do not include any other keys in the "back-translation" value.
Please ensure that you generate the complete transformed Java code, do not stop halfway through.
Do not generate Python code, only Java code.

Given Java Code:

```
for (int i = 0; i < 5; i++) {
 System.out.println("Number is " + i);
}
```

Your Transformed Python Code(Java to Python):

```
for i in range(5):
    print("Number is " + str(I))
```

Your Transformed Java Code(Python to Java):

```
int i = 0;
while (i < 5) {
    System.out.println("Number is " + i);
    i++;
}
```

Here is the Java code:"[Original Java Code]"
Please translate this Java code into Python code and then back into Java code, and generate your answer in the following JSON format:
{{
"back-translation": "[The transformed Java code]"
}}

Don't cut me off in the middle, create it all the way through. The output should not contain any text, only code. This includes avoiding explanations, comments, or any other form of text.
Please ensure that the transformed code is properly indented for readability.
Do not generate Python code, only Java code.

Figure 11: LLM prompt examples used in Back-translation.

K.4 Loop-Conversion Prompt

Loop-Conversion Prompt

...
Your task is to convert all ‘for’ loops into ‘while’ loops and all ‘while’ loops into ‘for’ loops.
For example, if the Java code contains a ‘for’ loop, you should change it into a ‘while’ loop.
If the Java code contains a ‘while’ loop, you should change it into a ‘for’ loop.
If the Java code contains both ‘for’ and ‘while’ loops, you should change all ‘for’ loops into ‘while’ loops and all ‘while’ loops into ‘for’ loops.
Please note that all ‘while’ loops should be converted into ‘for’ loops, even if it breaks the readability of the code. Please note that all ‘while’ loops should be converted into ‘for’ loops, even if it only runs once. This may result in ‘for’ loops with empty initialization or increment sections.
The output should be valid Java code and should not contain any syntax or constructs from other languages.
The Java output should be in the following JSON format:
Please ensure that your transformed Java code is enclosed in double quotes and ends with a closing quote.
{
"forwhile": "[Your transformed Java code here]"
}.
...
Here is an example of how to convert a ‘for’ loop to a ‘while’ loop in Java:
...
And here is an example of how to convert a Java code that contains both ‘for’ and ‘while’ loops:

```
Given Java Code:

for(int i = 0; i < 10; i++) {
    System.out.println("For loop: " + i);
}
while(!st.hasMoreTokens()) {
    st = new StringTokenizer(in.readLine());
}
```

Your Transformed Java Code:

```
int i = 0;
while(i < 10) {
    System.out.println("For loop: " + i);
    i++;
}
for(; !st.hasMoreTokens(); st = new StringTokenizer(in.readLine())) {
}
```

Here is the Java code: "[Original Java Code]".
Please convert all ‘for’ loops into ‘while’ loops and all ‘while’ loops into ‘for’ loops, and generate your answer in the following JSON format:
{{
"forwhile": "[The transformed Java code]"
}}.
Don’t cut me off ...

Figure 12: LLM prompt examples used in Loop-conversion.

L LLM 5-shot In-Context Learning Prompts

LLM Prompt

You are the best programmar in the world. You will be asked to determine the time complexity of the following code. For the time complexity, choose one time complexity from the following options 'constant', 'logn', 'linear', 'nlogn', 'quadratic', 'cubic', and 'exponential'. Do not hesitate to use any other supplementary materials you need for the task. I will first give you the code. After you read the code, I will ask you to compute the time complexity of the code. The following are the demonstrations of the time complexity for codes:

```
n = int(input())
for i in range(n): print(i)
```

"complexity": linear

```
print(int(input()))
```

"complexity": constant

```
print("*")
print("**")
print("***")
```

"complexity": constant

```
n = int(input())
items = list(map(int, input().split()))
items.sort()
```

"complexity": nlogn

```
def powerset(items):
    n = len(items)
    for i in range(1 << n):
        subset = []
        for j in range(n):
            if i \& (1 << j):
                subset.append(items[j])
        print(subset)
items = list(map(int, input().split()))
powerset(items)
```

"complexity": exponential

Please output the time complexity of the whole code in a json format. Json format should be
{
"complexity": time complexity of the whole code
}.

Figure 13: LLM prompt used for 5-shot in-context learning evaluation.