CodeComplex: Dataset for Worst-Case Time Complexity Prediction

Anonymous ACL submission

Abstract

Analyzing the worst-case time complexity of a code is a crucial task in computer science and software engineering for ensuring the efficiency, reliability, and robustness of software systems. However, it is well-known that the problem of determining the worst-case time complexity of a program is theoretically undecidable. In response to this challenge, we introduce CodeComplex, a novel source code dataset where each code is manually annotated with a corresponding worst-case time complexity. CodeComplex comprises 4,900 Java codes and an equivalent number of Python codes, all sourced from programming competitions and annotated with complexity labels by a panel of algorithmic experts. To the best of our knowledge, CodeComplex stands as the most extensive code dataset tailored for predicting complexity. Subsequently, we present the outcomes of our experiments employing various baseline models, leveraging state-of-the-art neural models in code comprehension like CodeBERT, GraphCodeBERT, UniXcoder, PLBART, CodeT5, CodeT5+, and ChatGPT. We analyze how the dataset impacts the model's learning in predicting time complexity. We release our dataset[1] and baseline models[2] publicly to encourage the relevant (NLP, SE, and PL) communities to participate in this research.

1 Introduction

The assessment of computational complexity in algorithms, indicated by their worst-case computational complexity, remains crucial for estimating an algorithm's efficiency based on input size. Analyzing the worst-case computational complexity gives us an understanding of how long a task will take for a given instance. The worst-case computational complexity is often referred to as *the* time complexity. Worst-case time complexity, expressed through Big-O notation, offers insights into an algorithm's performance. Unfortunately, it is undecidable to precisely determine the time complexity for an algorithm (Turing, 1936). This limitation leads to the exploration of alternative approaches like static and dynamic code analysis techniques. Static analysis techniques, encompassing metrics like cyclomatic complexity (McCabe, 1976) and the Master theorem (Bentley et al., 1980), provide tractable ways to gauge efficiency. Conversely, dynamic code analysis, involving real-time execution and test cases, offers insights into execution time and bugs but necessitates adequate test cases and code execution (Burnim et al., 2009; Noller et al., 2018; Wei et al., 2018; Saumya et al., 2019; Koo et al., 2019). The evolution of programming comprehension models and AI-assisted programming tools, exemplified by GitHub Copilot[3] and AlphaCode (Li et al., 2022), has altered the landscape of coding practices and education. Despite this advancement, current models still suffer in predicting the time complexities and cannot be used for optimization or education. Recognizing the significant hardness of predicting code complexity, one practical difficulty is that the CoRCoD (Sikka et al., 2020) dataset with 929 codes was the sole dataset available that aimed to facilitate code complexity prediction research.

We introduce the CodeComplex dataset, comprising 9,800 program codes (4,900 Java and 4,900 Python) annotated with complexity classes by algorithm experts. Our paper delineates the challenge of predicting worst-case time complexity for program codes, leveraging cutting-edge deep neural network models and learning algorithms. Baseline performances are established and tested using classical machine learning algorithms and state-of-the-art deep learning models like CodeBERT (Feng

[1] https://anonymous.4open.science/r/CodeComplex-Data-1209

[2] https://anonymous.4open.science/r/CodeComplex-Models-1209

[3] https://copilot.github.com/

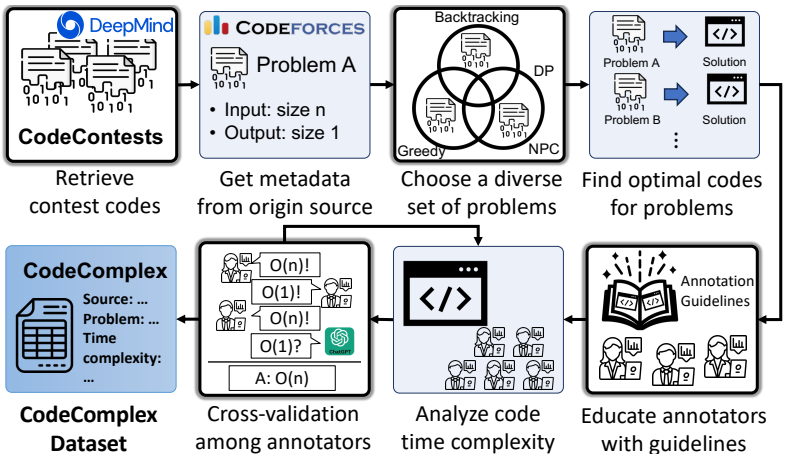

Figure 1: Overview of the CodeComplex dataset creation process.

et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), PLBART (Ahmad et al., 2021), CodeT5 (Yue Wang and Hoi, 2021), CodeT5+ (Wang et al., 2023), ChatGPT (OpenAI, 2024), and Gemini (Google, 2024).

In summary, our contributions are as follows:

1. We have created the large-scale worst-case time complexity dataset for a learning-based complexity prediction task;

2. We have performed experiments with traditional machine learning-based methods, pre-trained programming language models (PLMs), and closed-source LLMs such as ChatGPT and Gemini; and

3. We have carefully analyzed experimental results and suggested limitations of current approaches with promising future directions.

We believe that this paper can fuel advancements in predicting worst-case time complexity for programs which is crucial for both automated optimization of programs and effective education for people who study algorithms and programming.

2 Related Work

In the realm of quantifying program complexity, Bentley et al. (1980) introduced the Master theorem——a useful tool specifically for analyzing the time complexity of divide-and-conquer algorithms. This theorem facilitates the expression of an algorithm's time complexity as a recurrence relation and offers methods to solve this relation.

Recently, Sikka et al. (2020) delved into code complexity prediction using machine learning-based methods. They curated the CoRCoD dataset comprising 929 annotated Java codes. These codes were enriched with various hand-engineered features extracted from the code, encompassing counts of loops, methods, variables, jumps, breaks, switches, and the identification of specific data structures or algorithms like priority queues, hash maps, hash sets, and sorting functions. Employing machine learning classification algorithms such as K-means, random forest, decision tree, SVM, and more, they made predictions based on these diverse features. Additionally, they explored graph2vec, a neural graph embedding framework that operates on a program's AST, and achieved comparable performance results.

Another exploration by Prenner and Robbes (2021) scrutinized the potential of pre-trained programming language understanding models, particularly CodeBERT, for predicting code complexity. Their experiments showcased promising results, suggesting that pre-trained models could serve as a viable solution in this domain. In the most recent development, Moudgalya et al. (2023) tackled the analysis of time and space complexity using language models. They leveraged codes sourced from GeeksForGeeks[4] and CoRCoD, alongside a dataset comprising 3,803 Java codes. Their work showcased the viability of fine-tuning pre-trained language models such as GraphCodeBERT for predicting both time and space complexity, thereby opening new avenues for exploration in this field.

3 The CodeComplex Dataset

The CodeComplex dataset contains a collection of codes written in two languages, Java and Python,

[4] https://www.geeksforgeeks.org/

Table 1: Statistical difference between CoRCoD and CodeComplex. Numbers in parentheses imply the number of codes from CoRCoD.

Class	CoRCoD	CodeComplex	
	Java	Java	Python
$O(1)$	143	750 (+ 62)	791
$O(n)$	382	779 (+ 117)	853
$O(n^2)$	200	765 (+ 48)	657
$O(n^3)$	0	601	606
$O(\ln n)$	54	700 (+ 18)	669
$O(n \ln n)$	150	700 (+ 72)	796
NP-hard	0	605	528
Total	929	4,900 (+ 317)	4,900

from a competitive programming platform. Our dataset originates from Codeforces and collects data from CodeContests, a competitive programming dataset tailored for machine learning applications created by DeepMind. It comprises 9,800 codes, evenly split between Java and Python, with 4,900 codes each. We have categorized these codes into seven distinct complexity classes: constant ($O(1)$), linear ($O(n)$), quadratic ($O(n^2)$), cubic ($O(n^3)$), logarithmic ($O(\ln n)$, $O(n \ln n)$), and NP-hard. Each class contains a minimum of 500 Java and Python codes.

Among the 9,800 codes, we annotated 9,483 codes, as we have confirmed that the remaining 317 Java codes are also found in the CoRCoD dataset. It is worth mentioning that the CoRCoD, a previous dataset used for code complexity prediction, categorizes Java codes into five complexity classes: $O(1)$, $O(n)$, $O(n^2)$, $O(\ln n)$, and $O(n \ln n)$. However, it suffers from imbalanced class distribution, evident in Table 1, with a relatively small size of 929 Java code samples in total. To address this, we expanded the dataset by including both Java and Python languages and extended the complexity classes to seven, representing the most commonly encountered complexities in algorithmic problems. Each class now comprises at least 500 codes, resulting in a dataset of 4,900 codes. This expansion significantly enhances the dataset's value for research, particularly concerning recent DL-based models outlined in Section 4.

3.1 Data Collection

The original corpus of code is from CodeContests, which collected 128 million codes from Codeforces. The corpus only contained information about the contest ID, problem, username, language, acceptance, and statistics (runtime and memory). We extracted the selected problems from this corpus and identified each code's complexity.

Code samples were selected within the matching candidates with the following conditions. First, we checked the relevance of the problem. There are many problems within a coding competition, but not all of them fall into the scope of complexities we seek to compromise. Therefore, the problems were first analyzed to check whether or not they were in the complexity class of our dataset. If the problem was determined to be in one of the seven complexity classes, then we marked the problem as a candidate for the dataset. This helps to establish a clear base dataset for the complexity domain. Second, we checked the completeness and correctness of the code. We filtered codes that are available to pass the given problem in the contest, meaning that the code is functional, self-contained, and correct on the given task. One of the reasons for using code competition data is that we can check if the code is correct for the problem. Lastly, we wanted a large pool of code samples for a given problem. We took code samples from problems with abundant submissions. This helped to clarify the problem's robustness and variation.

Consider the following Python program that solves a problem with $O(1)$ time complexity:

```
buf = input()
hand = buf.split()
t = []
for i in range(3):
    t.append([])
    for j in range(9):
        t[i].append(0)
for x in hand:
    idx = 0
    # Following lines are omitted.
```

Despite the short length of the code, it is not trivial to understand that the time complexity of the above code is actually constant, which implies that the number of instructions for executing the program does not depend on the input size. In fact, the problem description says that the input always consists of three strings separated by whitespace and, therefore, the size of the list hand is actually constant. Hence, it is impossible to correctly calculate the time complexity of a code only by analyzing the code, as the problem description sometimes has a big hint to determine the time complexity.

3.2 Data Preprocessing

Data preprocessing is an important step in preparing datasets for analysis or machine learning tasks.

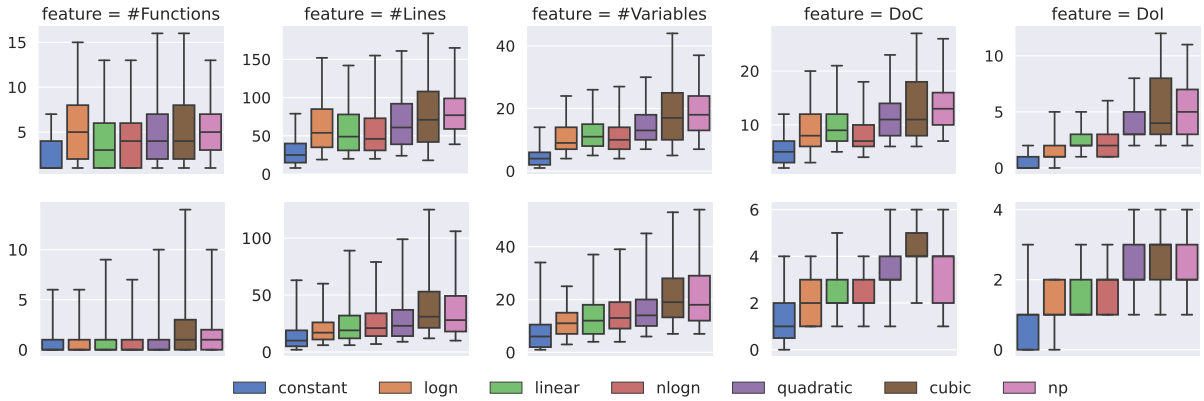

Figure 2: Basic statistics of codes in CodeComplex dataset. The first and second lines show statistics of Java and Python codes, respectively.

In this process, we utilize *dead code elimination* and *comment removal*. Dead code elimination involves removing any code that does not contribute to the functionality or output of the program, thereby reducing unnecessary clutter. From each code, we marked irrelevant codes and unreachable codes as dead codes. Irrelevant code involves variables, functions, and classes that were never used or never called, and unreachable code involves conditional statements that cannot be satisfied and statements that cannot be reached because of control statements such as continue and return.

On the other hand, comment removal entails stripping out any comments within the codebase, which are meant for human understanding. We removed the comments since the fragments could be exploited by the models to improve the accuracy of predicting the time complexity of models.

3.3 Annotation Process

Our primary objective is to create a solid foundation for accurately classifying time complexities. To achieve this, we have meticulously designed a procedure to generate a robust dataset with minimal noise and high quality. We specifically filter 'correct' Java and Python codes, ensuring they pass all test cases, including hidden ones. These codes form the basis of our statistical population. Categorizing problems based on problem-solving strategies involves leveraging annotations from CodeContests. Each problem in the dataset is associated with a plausible problem-solving strategy, such as brute force, dynamic programming, or backtracking, as outlined in CodeContests. Following this initial categorization, a detailed analysis of each problem is conducted. This analysis considers input and output variables, utilized data structures, and the overall workflow of the code. Subsequently, the code for each problem is annotated based on its specific input characteristics. More precisely, we take the largest input variable as the main factor in calculating the overall time complexity. By analyzing the code, we consider each control sequence on the code to determine if the input impacts a control segment or is constant. Note that we assume a *unit-cost RAM model* that requires the same cost for accessing all memory locations for calculating the time complexity. Our core annotation process adheres to four key rules:

1. Consider the input size and the output size as parameters to determine time complexity, with measurement based on the largest parameter among the input variables.

2. Account the impact of used packages and libraries, such as hashmap, sorting, and string-matching algorithms, on time complexity.

3. Treating each test case within a single input separately for complexity measurement.

4. Classifying cases with fixed constants as having a constant time complexity.

The annotation was held by three annotators who have expertise in the algorithm. In the initial annotation step, each annotator annotated each problem independently. Each reasoned on how we judged the input and annotated the time complexity.

During the agreement process, the annotators collaborated closely to reconcile any discrepancies in their annotations. We engaged in thorough discussions, sharing our reasoning and insights to

reach a consensus on the appropriate time complexity classification for each problem. In cases where disagreements arose, the annotators carefully evaluated the evidence and considered alternative perspectives before arriving at a mutually acceptable classification. The involvement of ChatGPT serves as a neutral advisor to validate the annotations and offer additional perspectives on complex cases. Through open communication and collaborative decision-making, the annotators achieved a high level of agreement, ensuring the accuracy and reliability of the final dataset. However, it is essential to note the significant impact of input formats and constraints on the actual time complexity of algorithmic problems. These constraints often lead to deviations from the ideal time complexity. Think of a scenario in which the input exploits the problem constraints in time complexity. Despite the problem of having a quadratic time complexity, the provided input constraints may result in linear running time. Moreover, determining the parameter for complexity measurement becomes crucial when faced with multiple input parameters. Additionally, certain code submissions optimize execution based on problem constraints, thus influencing code complexity assessment.

3.4 Dataset Analysis

The CodeComplex dataset offers a meticulously curated collection of algorithmic problems and corresponding Java and Python code submissions. It serves as a foundation for accurately classifying time complexities and problem-solving strategies.

Figure 2 demonstrates basic statistics of the codes for the number of Lines, number of functions, number of variables, depth of code (DoC) and depth of iterations (DoI). Moreover, when considering the depth of iterations, reflecting nested loops, both Java and Python solutions displayed comparable characteristics. However, a distinctive trait of Python code was its abundance of variables, potentially attributed to Python's lack of explicit variable declaration requirements. This inherent difference in variable declaration mechanisms might contribute to the observed discrepancy in variable counts between the two languages within our dataset.

4 Experiments

As a preliminary study on code complexity prediction using a large-scale dataset, we conduct experiments with well-known machine learning-based solutions. First, we try to replicate the result by Sikka et al. (2020) by employing traditional models such as decision tree (DT), random forest (RF), and support vector machine (SVM). Second, we use pre-trained programming language models (PLMs) such as CodeBERT, GraphCodeBERT, UniXcoder, PLBART, CodeT5, and CodeT5+. Note that we can further categorize these models into two groups where the first group (CodeBERT, GraphCodeBERT, and UniXcoder) only uses encoder architecture, and the second group (PLBART, CodeT5, and CodeT5+) exploits encoder-decoder architecture.

4.1 Experimental Settings

We divide the CodeComplex into training and test datasets using two distinct manners: random split and problem split. As the name implies, the random split involves randomly allocating the data in a 4:1 ratio for both Java and Python. As a result, the training and test datasets comprise 3,920 and 980 codes, respectively. In the case of the problem split, we also randomly split the data in a similar ratio but ensured that the training and test datasets did not share any common problems. The k-fold cross-validation technique is used to avoid any bias by the selected problems.

Code Data Augmentation To deal with the data scarcity problem, we perform several elementary code data augmentations such as the conversion of 'for' loops to 'while' loops, ternary operators to 'if' statements, and loop constructs to out-line list comprehension expressions. We apply these augmentation techniques to resolve the class imbalance without altering the original code's time complexity.

Hyperparameters For all pre-trained models, we use the AdamW (Loshchilov and Hutter, 2019) optimizer with a warmup linear scheduler. The learning rate was set to 2e-6, epsilon to 1e-8, and the weight decay to 1e-2. We applied either the AutoTokenizer or the RobertaTokenizer. The models were fine-tuned for 15 epochs before using them for evaluation.

4.2 Results

We present the following experimental results for various scenarios in Tables 2, 3, and 4. Full experimental results can be found in Appendix D.

Table 2: Performance comparison with two different dataset splits: random split and problem split.

Model	Problem		Random	
	Ja	Py	Ja	Py
Decision Tree	48.6	38.8	49.0	44.5
Random Forest	43.9	40.8	47.8	50.0
SVM	28.1	23.6	28.6	42.8
CodeBERT	60.5	51.2	73.7	73.5
GraphCodeBERT	60.4	**58.1**	86.0	83.7
UniXcoder	57.7	55.0	**89.2**	**86.6**
PLBART	**62.1**	54.0	88.7	82.7
CodeT5	60.7	48.9	85.3	77.3
CodeT5+	58.0	49.8	85.6	84.3

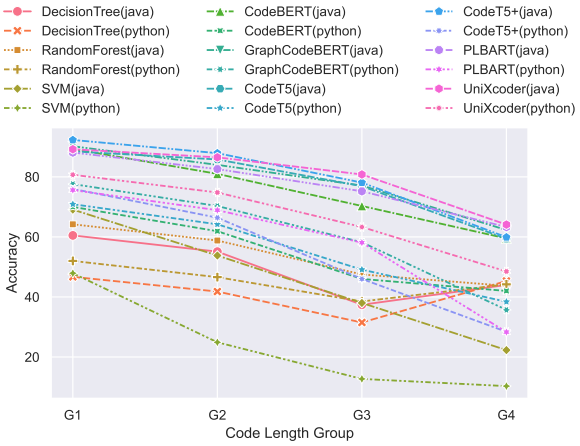

Figure 3: Classification performance of models for different length groups of codes.

5 Analysis & Discussion

5.1 Comparison of Java and Python

Java and Python are both popular programming languages, each with its unique features and characteristics that influence code structures and development practices. One key difference between Java and Python is the syntax typing. Java needs to declare variables with their data types beforehand, but Python variables can be assigned without explicit type declarations. Also, Python has explicit expressions such as list comprehensions that give easy access to elements in the code. This makes the Java code more verbose while Python code tends to be more concise and readable to humans. On the other hand, this difference seems to make it harder for models to understand the underlying structure of the code and predict the time complexity of the code. Since Python gives more freedom in writing code it makes a diverse range of coding styles. Also, users can easily compress or decompress their solution giving the models a harder time getting the right features from the codes.

5.2 Effect of Code Length

Intuitively, it is natural to assume that the shorter the code is, the easier it is to predict the complexity. To confirm our assumption, we categorize codes into four groups according to the number of tokens of the codes. If a code has less than or equal to 128 tokens, then the code falls into the first group (G1). If a code has more than 128 tokens and less than or equal to 256 tokens, then it falls into the second group (G2). The third group (G3) has codes with more than 256 tokens and less than or equal to 512 tokens. Lastly, group G4 has the longest codes, where each code has more than 512 tokens.

Figure 3 shows the experimental results on the four groups. It is easy to see that the experimental results confirm our assumption as the performance gets worse as the code becomes longer.

5.3 Performance Per Complexity Class

As we can see in Tables 3 and 8, models predict the constant class most accurately. Linear time complexities are also relatively accurate since there are direct correlation between the input size and the number of iterations. However, models still face challenges in predicting the complexity of self-referential algorithms and control flows that are not in the main part of the algorithm, such as predicting the $O(n \log n)$ class and $O(n^2)$ class. Many codes in these classes have dedicated function definitions or control variables that decide the exit condition for the loop. This is well seen in the confusion matrix in Figure 4, where we can see many wrong predictions are between classes $O(n)$, $O(n \log n)$, and $O(n^2)$.

5.4 Effect of Dead Code Elimination

Table 4 shows the effect of dead code elimination on prediction performance. Given the nature of codes submitted to competitive programming platforms, there are a lot of redundant variables, methods and even classes in the codes. Due to Java's complicated IO functions and limited built-in data structures, there are many codes related to the implementation of IO and data structures. Removing such fragments helps the models concentrate on the program structure and results in enhanced prediction accuracy. On the other hand, it appears that the dead code elimination does not help improve the performance on Python as the Python codes are already more concise than the Java codes due to its own language design principle.

Table 3: Complexity prediction accuracy of classification methods for each complexity class on Java.

Method	$O(1)$	$O(\ln n)$	$O(n)$	$O(n \ln n)$	$O(n^2)$	$O(n^3)$	NP-hard	Micro	Macro
Decision Tree	64.4	55.9	15.2	65.2	68.8	32.9	34.4	48.6	48.1
Random Forest	66.2	57.7	28.2	68.6	60.2	36.0	62.6	43.9	54.2
SVM	49.7	40.0	65.1	42.7	74.6	23.5	18.0	28.1	44.8
CodeBERT	86.1	60.9	68.1	18.8	33.6	73.1	84.8	60.5	59.4
GraphCodeBERT	88.7	53.5	51.9	28.9	38.8	78.1	85.5	60.4	60.0
UniXCoder	83.1	54.8	54.1	15.9	33.9	76.9	88.1	57.7	56.6
PLBART	86.6	52.7	61.9	34.8	36.4	76.2	88.1	52.1	**61.9**
CodeT5	81.8	43.7	69.4	40.7	33.9	72.0	85.6	**60.7**	60.3
CodeT5+	90.7	50.8	64.2	14.1	27.0	74.9	86.8	58.0	56.1
ChatGPT 3.5[5]	54.0	55.8	74.1	28.0	67.7	79.8	0.0	43.6	35.6
ChatGPT 4.0[6]	64.2	43.4	70.9	72.8	56.2	67.2	0.1	54.8	45.7
Gemini Pro[7]	33.2	15.4	59.4	7.1	72.4	8.1	0.0	30.1	21.4

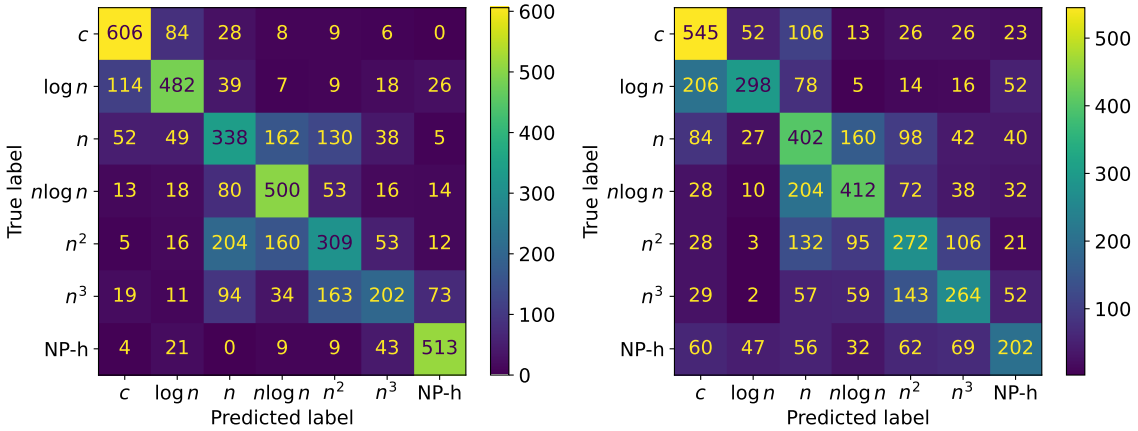

Figure 4: Confusion matrices of prediction results by CodeT5 on Java (left) and Python (right) datasets.

Table 4: Performance comparison before and after dead code processing.

Model	**After**		**Before**	
	Ja	Py	Ja	Py
Decision Tree	**48.6**	**38.8**	47.6	21.1
Random Forest	**43.9**	**40.8**	43.2	23.0
SVM	**28.1**	**23.6**	27.1	21.5
CodeBERT	**60.5**	51.2	59.7	**52.0**
GraphCodeBERT	**60.4**	58.1	57.8	**60.2**
UniXcoder	**57.7**	55.0	57.2	**55.3**
PLBART	**62.1**	54.0	61.2	**55.4**
CodeT5	**60.7**	48.9	60.2	**49.8**
CodeT5+	**58.0**	49.8	57.4	**50.0**
ChatGPT 3.5	**43.6**	41.8	43.3	**43.1**
ChatGPT 4.0	54.8	**51.7**	**55.4**	51.5
Gemini Pro	30.1	**35.1**	**31.2**	33.5

5.5 Experiments with Closed-Source LLMs

We use two well-known closed-source LLMs, OpenAI's ChatGPT (in version 3.5 and 4.0) and Google's Gemini-Pro, to predict the time complexity of the codes. The prompt used for ChatGPT experiments is given in Figure 5 and the prompt for Gemini is provided in Appendix due to space restriction. To summarize the experimental results with LLMs, both models do not perform well on the complexity prediction task as shown in Table 3. In particular, they completely fail to predict NP-hard class. We assume that it is because there is not much available source on the Internet for learning to predict the NP-hardness of a given code.

ChatGPT prompt example

You are a world expert in investigating properties of a code that influences the time complexity.
The given code: "[code]"
Print "ONLY the time complexity in ONE WORD" of the given code in the answer from np, logn, quadratic, constant, cubic, linear and nlogn, do not print any other words in a json format.

Figure 5: LLM prompt examples used in our experiments with ChatGPT.

5.6 Qualitative Error Analysis

After investigating the common errors from extensive experiments with many baseline models, we

find that the following problems are the root causes of most of error cases.

Unused Boilerplate Code Patterns Codes can include parts of codes that are irrelevant to the operation of the code. This can be because of coding habits or template codes for handy development. There are cases where the writer puts in ascii art in the comments. These methods add to the overall recognition load of understanding the codebase and can obscure the true flow of execution. Codes that include unused methods introduce noise, making it harder for models to recognize the structure of the code. Tabel 4 shows that there is some increase in performance when the testing dataset is preprocessed.

Logarithmic Loops The most common error are from the logarithmic complexity class. Loops with logarithmic sizes, such as those found in binary search algorithms, can significantly affect the prediction of a code's time complexity. These loops has similar structures as normal linear loops, but inside the loop they have additional variables or conditions that control the algorithm's flow. Unlike linear loops, this needs a thorough analysis of all contributing factors, ensuring a comprehensive understanding of the algorithm's performance characteristics. It seems like deep learning models yet lack the power of determining the contributing factors and figure out its meaning and impact.

Too Much Conciseness of Python Despite the famous zen of Python[8], it offers various ways to implement a loop such as classical for or while loop, list comprehension, and even lambda function. While the usage of list comprehension and lambda function makes Python codes much more concise and leads to statistics as in Figure 2, it also makes the complexity prediction task more challenging. The tendency is clearly seen especially when compared to Java in Table 2.

5.7 Does Problem Description Help?

A critical aspect in accurately determining the worst-case time complexity of a given code is the comprehensive understanding of the problem specifications. In certain instances, these specifications may indicate that some inputs are constant, significantly influencing the complexity analysis. The absence of a full and detailed specification can lead to an incomplete or incorrect assessment of the worst-case time complexity. Table 5 shows that problem descriptions actually help LLMs perform better as ChatGPT 4.0 is known to have real-time access to the information on the web. Note that the performance becomes worse for ChatGPT 3.5 when problem IDs are provided, as ChatGPT 3.5 does not utilize the problem descriptions, only from problem IDs.

Table 5: Complexity prediction performances of LLMs with and without a problem description in the prompt by the help of information retrieval.

Model	w/o Desc.		with Desc.	
	Ja	Py	Ja	Py
ChatGPT 3.5	**43.38**	**43.14**	42.51	36.55
ChatGPT 4.0	55.42	51.57	**57.61**	**54.28**

6 Conclusions

We have presented CodeComplex, the first large-scale bilingual benchmark dataset for predicting the worst-case time complexity of programs. We have demonstrated the experimental results of several classical machine learning algorithms, pre-trained programming language models, and closed-source LLMs for benchmarking the complexity prediction performance. The results show that while the current state-of-the-art techniques provide promising baselines, there is still a long way to go to achieve a reliable performance for practical use cases.

7 Limitations

There are the following limitations we noticed from our analysis that follow-up research contributions should resolve. First, problem descriptions should be provided as part of the input for the completeness of the specification. As shown in examples in Section 3.1, it is necessary to consider problem specification to determine the intended time complexity of the problem, which will apply to most of the solution codes for the problem. Second, we need to deal with the unintended biases towards problem-specific information rather than implementation details learned due to the characteristics of our dataset. We expect that the code data augmentation that changes an original complexity to an intended complexity would be helpful to increase the diversity of complexities within the pool of solution codes for a single problem and help reduce unintended biases.

[8]There should be one—and preferably only one—obvious way to do it.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668.

Jon Louis Bentley, Dorothea Haken, and James B. Saxe. 1980. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44.

Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 463–473. IEEE.

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 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Google. 2024. Gemini. https://gemini.google.com/. Accessed: 2023.02.12.

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 (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

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 2021, Virtual Event*. OpenReview.net.

Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. 2019. Pyse: Automatic worst-case test generation by reinforcement learning. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 136–147. IEEE.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR*.

T. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, 2(04):308–320.

Kaushik Moudgalya, Ankit Ramakrishnan, Vamsikrishna Chemudupati, and Xing Han Lu. 2023. Tasty: A transformer based approach to space and time complexity.

Yannic Noller, Rody Kersten, and Corina S. Pasareanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 322–332. ACM.

OpenAI. 2024. ChatGPT. https://openai.com/chatgpt/. Accessed: 2024.02.12.

Julian Aron Aron Prenner and Romain Robbes. 2021. Making the most of small software engineering datasets with modern machine learning. *IEEE Transactions on Software Engineering*, pages 5050–5067.

Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi. 2019. XSTRESSOR : Automatic generation of large-scale worst-case test inputs by inferring path conditions. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 1–12. IEEE.

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 2020, Proceedings, Part I*, volume 12035 of *Lecture Notes in Computer Science*, pages 313–325. Springer.

Alan M. Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation.

Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: pattern fuzzing for worst case complexity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 213–223. ACM.

Shafiq Joty Yue Wang, Weishi Wang and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, pages 8696–8708.

A Overview on CodeComplex Dataset

Our dataset construction process owes much to the recently released dataset called the CodeContests[9], a competitive programming dataset for machine learning by DeepMind. We constructed a dataset with the codes from the CodeContests dataset that are again sourced from the coding competition platform Codeforces. Our dataset contains 4,120 codes in seven complexity classes, where there are new 500 Java source codes annotated with each complexity class. The seven complexity classes are constant ($O(1)$), linear ($O(n)$), quadratic ($O(n^2)$), cubic ($O(n^3)$), $O(\ln n)$, $O(n \ln n)$, and NP-hard. We also re-use 317 Java codes from CoRCoD as we confirmed that they also belong to the CodeContests dataset as the other 3803 codes during the dataset creation process.

For constructing the dataset, we asked twelve human annotators who have more than five years of programming experience and algorithmic expertise to inspect the codes manually and classify them into one of the seven complexity classes. Once each human annotator reported the initial result, we collected the annotation results and inspected them once again by assigning the initial result to two different annotators other than the initial annotator. Finally, we have collected 3803 complexity annotated codes in which there are 500 codes for each complexity class.

First, we selected several problems that are expected to belong to one of the considered complexity classes and submitted codes for the problems from Codeforces. The submitted codes contain both correct and incorrect solutions, and they are implemented in various programming languages such as C, C++, Java, and Python. We sorted out only the correct Java codes for our dataset construction.

In the second step, before delving into the time complexity of problems, we divide the problems by the problem-solving strategy such as sorting, DP (dynamic programming), divide-and-conquer, DFS (depth-first search), BFS (breadth-first search), A*, and so on. This is because it is helpful to know the type of problem-solving strategy used to solve the problem for human annotators to analyze the time complexity, and problems solved by the same strategy tend to have similar time complexity.

Third, we uniformly assign problems and correct codes for the problems to human annotators and let them carefully examine the problem-code pairs to label the time complexity of the codes. Notice that there can be solutions with different time complexities for a problem depending on how to actually implement the solutions. We, therefore, provide a specific guideline that contains instructions and precautions to annotators so that human annotators can assign correct and consistent labels to the assigned codes.

After the initial annotation process, we collect the results and assign them to different annotators to carefully cross-check the correctness of the initial annotation results. Primarily, we instruct the annotators again to carefully verify the results in accordance with the precautions provided in the annotation guideline.

A.1 Further Details on CodeComplex Dataset Construction

We gathered 128,000,000 submissions of Codeforces, where 4,086,507 codes are implemented in Java language. After discarding the incorrect codes (that do not pass all the test cases), there are 2,034,925 codes and 7,843 problems. Then the problems are split with their tags (e.g. sorting, dfs, greedy, etc) and given to the annotators with the guidelines in Section A.2. We were able to gather around 500 problems and 15,000 codes for the seven complexity classes.

As the complexity of codes for the same problem can vary depending on the implemented algorithms, it is obvious that the codes we inspect also have various complexity classes. However, we only target seven complexity classes that are the most frequently used complexity classes for algorithmic problems. Accordingly, there were some codes we inspected which belong to other complexity classes such as $O(n^5)$ or $O(\ln \ln n)$. We inspected around 800 problems and found out that the complexity classes of approximately 15 of the problems belong outside the chosen complexity classes. Although it is still possible that one might implement codes with complexity class that falls into the seven complexity classes, we simply rule out the problems from our dataset to ease the annotation process.

During this process, we found out that many codes are not optimal for the given problem and some codes are too difficult to analyze due to their complex code structure. Moreover, there are many codes with a number of methods that are never used, mainly because the codes come from a coding competition

[9] https://github.com/deepmind/code_contests

platform and participants prefer just to include the methods that are frequently used in problem-solving regardless of the actual usage of the methods.

In the section below, we share the detailed guidelines provided to human annotators for a consistent and accurate annotation process.

A.2 Guideline of Production

Annotator Guideline

1. Check the variables that are described in the algorithm problems. Each algorithm implementation can have many variable instances and we only consider the variables that are given as inputs from the problems for calculating the time complexity.

   * For convenience, we use n and m in the guideline to denote the input variable and $|n|$ and $|m|$ to denote the size of n and m.

2. Considering the input variable from the prior step, follow the below instructions for each input type and calculate the time complexity.

   (a) When only a number n is given as an input, calculate the time complexity proportional to n. Do the same thing when there are two or more variables. For instance, when only n is given as an input, the variable used to denote the time complexity of a code is n.

   (b) When a number n and m numeric instances are given as inputs, calculate the time complexity proportional to the one with higher complexity. For instance, when $m = n^2$, we compute the complexity of a code with m. If the implemented algorithm runs in $O(n^2) = O(m)$, it belongs to the linear complexity class.

   (c) If the input is given as constant values, the complexity of a given code also belongs to the constant class. For instance, if an algorithm problem states that exactly 3 numeric values are given as inputs, the solution code only uses the constant number of operations. Therefore, the code belongs to the constant class.

3. Consider the case where the code utilizes the input constraints of the problem. When the input is given by $n \leq a$, the code can use the fixed value a in the problem instead of using n. Mark these codes as unsuitable.

4. Consider the built-in library that the implemented algorithm is using (e.g. HashMap, sort, etc.) to calculate the time complexity of an entire code. For instance, given n numeric instances as inputs, when an implemented algorithm uses $O(n)$ iterations of built-in sort algorithm for n numeric instances, the time complexity for the algorithm is $O(n^2 \ln n)$.

5. When the code has unreachable codes, only consider the reachable code.

6. Mark the item that does not belong to any of the 7 complexity classes.

A.3 Statistics of CodeComplex

Table 6 shows basic statistics in numbers of our CodeComplex dataset.

B Failure Cases

The following example exhibits a failure example where our model predicts $O(2^n)$ for a code with $O(\ln n)$ complexity. We suspect that the primary reason is the usage of bitwise operators. When we filter the codes that use any bitwise operator at least once from our CodeComplex dataset, about 56 of the codes belong to the class $O(2^n)$, which implies NP-hardness. We find that many implementations for NP-hard problems rely on bitwise operators as they can efficiently manage the backtracking process by manipulating bit-level flags.

Table 6: Statistics of codes from CodeComplex dataset. There are two values in each cell where the first value is about the Java codes and the second value is about the Python codes.

Property	$O(1)$		$O(n)$		$O(n^2)$		$O(n^3)$		$O(\ln n)$		$O(n \ln n)$		NP-hard		**Total**	
	Ja	Py	Ja	Py	Ja	Py	Ja	Py	Ja	Py	Ja	Py	Ja	Py	Ja	Py
#Problems	38	50	94	104	12	16	41	46	10	22	60	63	23	36	269	277
#Lines	31.7	19.7	60.9	29.3	72.7	36.6	82.3	48.3	66.0	22.2	59.4	30.7	85.6	43.6	64.5	31.9
#Functions	2.6	1.3	4.5	1.7	5.9	2.0	6.0	3.4	6.0	1.3	4.7	1.6	6.0	2.6	5.0	1.9
#Variables	5.3	9.7	12.2	15.5	15.2	18.6	19.4	24.3	11.2	12.3	11.6	16.0	19.4	24.5	13.2	16.7
DoC	5.7	1.5	10.2	2.5	12.2	3.4	13.6	4.2	9.4	2.3	8.5	2.6	14.2	3.5	10.4	2.8
DoI	0.6	0.5	2.7	1.0	4.0	1.0	5.5	1.0	1.8	0.8	2.4	0.9	5.6	0.9	3.1	0.9

```java
public class mad {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int cura = 0, curb = 0;
        int ver;
        System.out.println("? 0 0");
        System.out.flush();
        ver = sc.nextInt();
        for (int i = 29; i >= 0; i--) {
            System.out.println("? " + (cura + (1 << i)) + " " + curb);
            System.out.flush();
            int temp1 = sc.nextInt();
            System.out.println("? " + cura + " " + (curb + (1 << i)));
            System.out.flush();
            int temp2 = sc.nextInt();
            if (temp1 != temp2) {
                if (temp2 == 1) {
                    cura += (1 << i);
                    curb += (1 << i);
                }
            } else {
                if (ver == 1) cura += (1 << i);
                if (ver == -1) curb += (1 << i);
                ver = temp1;
            }
        }
        System.out.println("! " + cura + " " + curb);
    }
}
```

The following example demonstrates the case when our model predicts constant time complexity $O(1)$ for a code that runs in $O(n)$ time. We suspect that our model may have ignored the existence of the check method which actually determines the $O(n)$ time complexity or considered the argument of check as constant.

```java
public class abc {
    public static int check(StringBuilder s) {
        int countRemove = 0;
        if (!s.toString().contains("xxx")) return countRemove;
        else {
            for (int i = 1; i < s.length() - 1; i++) {
                if (s.charAt(i - 1) == 'x' && s.charAt(i) == 'x' && s.charAt(i + 1) == 'x') {
                    countRemove++;
                }
            }
            return countRemove;
        }
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        String s = sc.next();
        StringBuilder sb = new StringBuilder("");
        sb.append(s);
        System.out.println(check(sb));
    }
}
```

The following is the case where our model predicts the quadratic time complexity $O(n^2)$ when the ground-truth label is $O(n \ln n)$. We guess that our model simply translates the nested `for` loops into the quadratic time complexity. However, the outer loop is to repeat each test case and therefore should be ignored. Then, the $O(n \ln n)$ complexity can be determined by the `sort` function used right before the nested loops.

```
public class round111A {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] coins = new int[n];
        for (int i = 0; i < n; ++i) coins[i] = sc.nextInt();
        Arrays.sort(coins);
        int ans = (int) 1e9;
        for (int i = 1; i <= n; ++i) {
            int sum1 = 0;
            int c = 0;
            int j = n - 1;
            for (j = n - 1; j >= 0 && c < i; --j, ++c) {
                sum1 += coins[j];
            }
            int sum2 = 0;
            for (int k = 0; k <= j; ++k) sum2 += coins[k];
            if (sum1 > sum2) {
                System.out.println(i);
                return;
            }
        }
    }
}
```

The following is the case when our model is confused exponential complexity $O(2^n)$ with quadratic complexity $O(n^2)$. The code actually runs in exponential time in the worst-case but our model simply returns quadratic time complexity as it does not take into account the recursive nature of the method `solve`.

```
public class D {
    static int n, KA, A;
    static int[] b;
    static int[] l;
    static double ans = 0;

    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        n = in.nextInt();
        KA = in.nextInt();
        A = in.nextInt();
        b = new int[n];
        l = new int[n];
        for (int i = 0; i < l.length; i++) {
            b[i] = in.nextInt();
            l[i] = in.nextInt();
        }
        dp = new double[n + 2][n + 2][n * 9999 + 2];
```

```
        go(0, KA);
        System.out.printf("%.6f\n", ans);
    }

    public static void go(int at, int k) {
        if (at == n) {
            ans = Math.max(ans, solve(0, 0, 0));
            return;
        }
        for (int i = 0; i <= k; i++) {
            if (l[at] + i * 10 <= 100) {
                l[at] += i * 10;
                go(at + 1, k - i);
                l[at] -= i * 10;
            }
        }
    }

    static double dp[][][];

    public static double solve(int at, int ok, int B) {
        if (at == n) {
            if (ok > n / 2) {
                return 1;
            } else {
                return (A * 1.0) / (A * 1.0 + B);
            }
        }
        double ret = ((l[at]) / 100.0) * solve(at + 1, ok + 1, B) + (1.0 - ((l[at]) / 100.0)) *
        ↪  solve(at + 1, ok, B + b[at]);
        return ret;
    }

}
```

The following is the case when our model predicts $O(\ln n)$ for a code with $O(n^2)$ complexity. It is easily seen that the `inversions` function determines the quadratic time complexity by the nested `for` loops. We suspect that somehow our model does not take into account the `inversions` function in the complexity prediction and instead focuses on the modulo () operator to draw the wrong conclusion that the complexity is in $O(\ln n)$.

```
public class maestro {
    public static long inversions(long[] arr) {
        long x = 0;
        int n = arr.length;
        for (int i = n - 2; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                if (arr[i] > arr[j]) {
                    x++;
                }
            }
        }
        return x;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        long[] arr = new long[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextLong();
        long m = sc.nextLong();
        long x = inversions(arr) % 2;
        for (int i = 0; i < m; i++) {
            int l = sc.nextInt() - 1;
            int r = sc.nextInt() - 1;
            if ((r - l + 1) % 4 > 1) x = (x + 1) % 2;
```

```
            if (x == 1) System.out.println("odd");
            else System.out.println("even");
        }
    }
}
```

C Further Details on Dead Code Elimination

In a broad sense, the dead code includes redundant code, unreachable code, oxbow code, and so on. We only focus on eliminating unreachable codes, mainly methods and classes that are declared but used nowhere in the code. In order to find such dead codes, we first parse a Java code into an AST and discover methods and classes that do not exist in any method call, class declaration, and arguments of methods. Once we discover such unused methods and classes, we remove the codes corresponding to the declarations of these methods and classes.

The following codes are a running example of the dead code elimination process. From the first code, we can obtain the second code by applying the dead code elimination. It is readily seen that the number of lines decreases from 211 to 101 by the elimination process. In fact, our model predicts $O(\ln n)$ and $O(1)$ for the complexity of the code before and after dead code elimination, respectively, while the actual complexity of the code is $O(1)$.

```
public class Main {
    static long mod = ((long) 1e9) + 7;

    public static int gcd(int a, int b) {
        if (b == 0) return a;
        else return gcd(b, a % b);
    }

    public static long pow_mod(long x, long y) {
        long res = 1;
        x = x % mod;
        while (y > 0) {
            if ((y & 1) == 1) res = (res * x) % mod;
            y = y >> 1;
            x = (x * x) % mod;
        }
        return res;
    }

    public static int lower_bound(int[] arr, int val) {
        int lo = 0;
        int hi = arr.length - 1;
        while (lo < hi) {
            int mid = lo + ((hi - lo + 1) / 2);
            if (arr[mid] == val) {
                return mid;
            } else if (arr[mid] > val) {
                hi = mid - 1;
            } else lo = mid;
        }
        if (arr[lo] <= val) return lo;
        else return -1;
    }

    public static int upper_bound(int[] arr, int val) {
        int lo = 0;
        int hi = arr.length - 1;
        while (lo < hi) {
            int mid = lo + ((hi - lo) / 2);
            if (arr[mid] == val) {
                return mid;
            } else if (arr[mid] > val) {
                hi = mid;
```

```
                ;
            } else lo = mid + 1;
        }
        if (arr[lo] >= val) return lo;
        else return -1;
    }

    public static void main(String[] args) throws java.lang.Exception {
        Reader sn = new Reader();
        Print p = new Print();
        int n = sn.nextInt();
        while ((n--) > 0) {
            int a = sn.nextInt();
            int b = sn.nextInt();
            int small = Math.min(a, b);
            int large = Math.max(a, b);
            long steps = 0;
            while (small != 0) {
                steps += (long) (large / small);
                int large1 = small;
                small = large % small;
                large = large1;
            }
            p.printLine(Long.toString(steps));
        }
        p.close();
    }
}

class Pair implements Comparable<Pair> {
    int val;
    int in;

    Pair(int a, int b) {
        val = a;
        in = b;
    }

    @Override
    public int compareTo(Pair o) {
        if (val == o.val) return Integer.compare(in, o.in);
        else return Integer.compare(val, o.val);
    }
}

class Reader {
    final private int BUFFER_SIZE = 1 << 16;
    private DataInputStream din;
    private byte[] buffer;
    private int bufferPointer, bytesRead;

    public boolean isSpaceChar(int c) {
        return c == ' ' || c == '\n' || c == '\r' || c == '\t' || c == -1;
    }

    public Reader() {
        din = new DataInputStream(System.in);
        buffer = new byte[BUFFER_SIZE];
        bufferPointer = bytesRead = 0;
    }

    public Reader(String file_name) throws IOException {
        din = new DataInputStream(new FileInputStream(file_name));
        buffer = new byte[BUFFER_SIZE];
        bufferPointer = bytesRead = 0;
    }

    public String readLine() throws IOException {
```

783

```
        byte[] buf = new byte[64];
        int cnt = 0, c;
        while ((c = read()) != -1) {
            if (c == '\n') break;
            buf[cnt++] = (byte) c;
        }
        return new String(buf, 0, cnt);
    }

    public String readWord() throws IOException {
        int c = read();
        while (isSpaceChar(c)) c = read();
        StringBuilder res = new StringBuilder();
        do {
            res.appendCodePoint(c);
            c = read();
        } while (!isSpaceChar(c));
        return res.toString();
    }

    public int nextInt() throws IOException {
        int ret = 0;
        byte c = read();
        while (c <= ' ') c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = ret * 10 + c - '0';
        } while ((c = read()) >= '0' && c <= '9');
        if (neg) return -ret;
        return ret;
    }

    public long nextLong() throws IOException {
        long ret = 0;
        byte c = read();
        while (c <= ' ') c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = ret * 10 + c - '0';
        } while ((c = read()) >= '0' && c <= '9');
        if (neg) return -ret;
        return ret;
    }

    public double nextDouble() throws IOException {
        double ret = 0, div = 1;
        byte c = read();
        while (c <= ' ') c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = ret * 10 + c - '0';
        } while ((c = read()) >= '0' && c <= '9');
        if (c == '.') {
            while ((c = read()) >= '0' && c <= '9') {
                ret += (c - '0') / (div *= 10);
            }
        }
        if (neg) return -ret;
        return ret;
    }

    private void fillBuffer() throws IOException {
        bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
        if (bytesRead == -1) buffer[0] = -1;
    }
```

```
    private byte read() throws IOException {
        if (bufferPointer == bytesRead) fillBuffer();
        return buffer[bufferPointer++];
    }

    public void close() throws IOException {
        if (din == null) return;
        din.close();
    }
}

class Print {
    private final BufferedWriter bw;

    public Print() {
        bw = new BufferedWriter(new OutputStreamWriter(System.out));
    }

    public void print(String str) throws IOException {
        bw.append(str);
    }

    public void printLine(String str) throws IOException {
        print(str);
        bw.append("\n");
    }

    public void close() throws IOException {
        bw.close();
    }
}
```

Code after Dead Code Elimination:

```
    static long mod = ((long) 1e9 + 7);

    public static int gcd(int a, int b) {
        if ((b == 0)) return a;
        else return gcd(b, (a % b));
    }

    public static void main(String[] args) throws java.lang.Exception {
        Reader sn = new Reader();
        Print p = new Print();
        int n = sn.nextInt();
        while ((n > 0)) {
            int a = sn.nextInt();
            int b = sn.nextInt();
            int small = Math.min(a, b);
            int large = Math.max(a, b);
            long steps = 0;
            while ((small != 0)) {
                steps += (long) (large / small);
                int large1 = small;
                small = (large % small);
                large = large1;
            }
            p.printLine(Long.toString(steps));
        }
        p.close();
    }
}

class Reader {
    final private int BUFFER_SIZE = (1 << 16);
    private DataInputStream din;
    private byte[] buffer;
```

```
    private int bufferPointer, bytesRead;

    public boolean isSpaceChar(int c) {
        return (((((c == ' ') || (c == '\n')) || (c == '\r')) || (c == '\t')) || (c == -1));
    }

    public Reader() {
        din = new DataInputStream(System.in);
        buffer = new byte[BUFFER_SIZE];
        bufferPointer = bytesRead = 0;
    }

    public Reader(String file_name) throws IOException {
        din = new DataInputStream(new FileInputStream(file_name));
        buffer = new byte[BUFFER_SIZE];
        bufferPointer = bytesRead = 0;
    }

    public int nextInt() throws IOException {
        int ret = 0;
        byte c = read();
        while ((c <= ' ')) c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = (((ret * 10) + c) - '0');
        } while ((((c = read()) >= '0') && (c <= '9')));
        if (neg) return -ret;
        return ret;
    }

    private void fillBuffer() throws IOException {
        bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
        if ((bytesRead == -1)) buffer[0] = -1;
    }

    private byte read() throws IOException {
        if ((bufferPointer == bytesRead)) fillBuffer();
        return buffer[bufferPointer++];
    }

    public void close() throws IOException {
        if ((din == null)) return;
        din.close();
    }
}

class Print {
    final private BufferedWriter bw;

    public Print() {
        bw = new BufferedWriter(new OutputStreamWriter(System.out));
    }

    public void print(String str) throws IOException {
        bw.append(str);
    }

    public void printLine(String str) throws IOException {
        print(str);
        bw.append("\n");
    }

    public void close() throws IOException {
        bw.close();
    }
}
```

D Full Experimental Results

Method	G1	G2	G3	G4	G1	G2	G3	G4
	Java				Python			
Decision Tree	57.2	45.6	40.0	38.2	57.2	45.6	40.0	38.2
Random Forest	62.3	46.8	40.6	26.4	62.3	46.8	40.6	26.4
SVM	48.9	18.1	18.1	16.6	48.9	18.1	18.1	16.6
CodeBERT	72.4	62.8	60.7	48.0	56.9	46.9	37.5	22.8
GraphCodeBERT	74.6	61.7	49.8	39.4	60.3	57.8	44.1	30.8
UniXCoder	58.6	54.4	43.2	31.2	58.6	54.4	43.2	31.2
PLBART	74.3	65.1	62.5	52.8	60.6	49.4	39.9	23.2
CodeT5	69.5	56.5	52.4	42.4	53.6	48.1	36.5	19.5
CodeT5+	72.8	63.5	53.0	44.4	56.4	42.4	30.7	29.8

Table 7: Prediction performance on different code length groups.

E Confusion Matrices for PLM Models

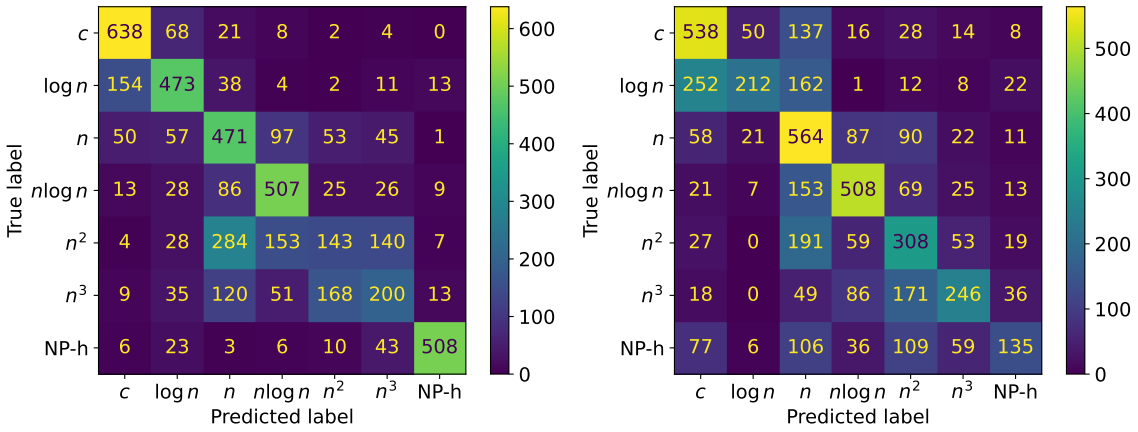

Figure 6: Confusion matrices of prediction results by CodeBERT on Java (left) and Python (right) datasets.

F Classification Results on Python Codes

G LLM Prompt for Gemini

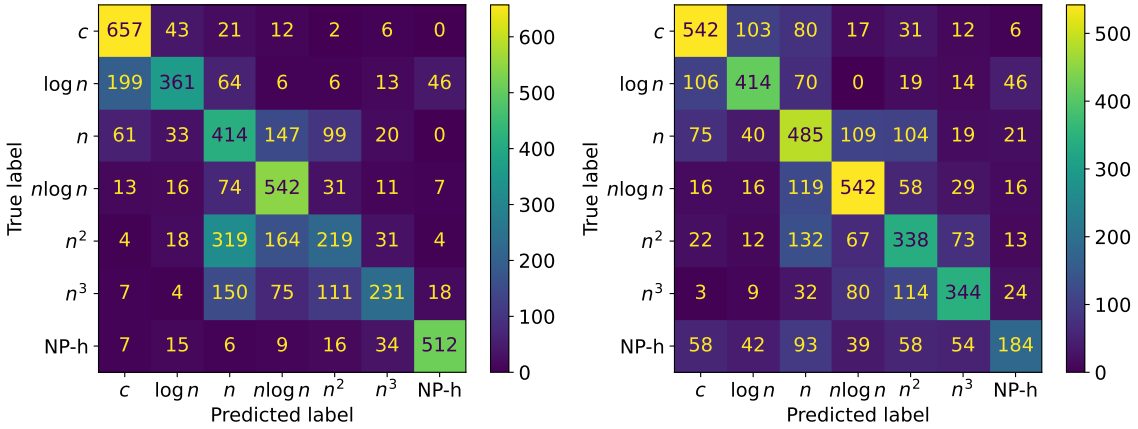

Figure 7: Confusion matrices of prediction results by GraphCodeBERT on Java (left) and Python (right) datasets.

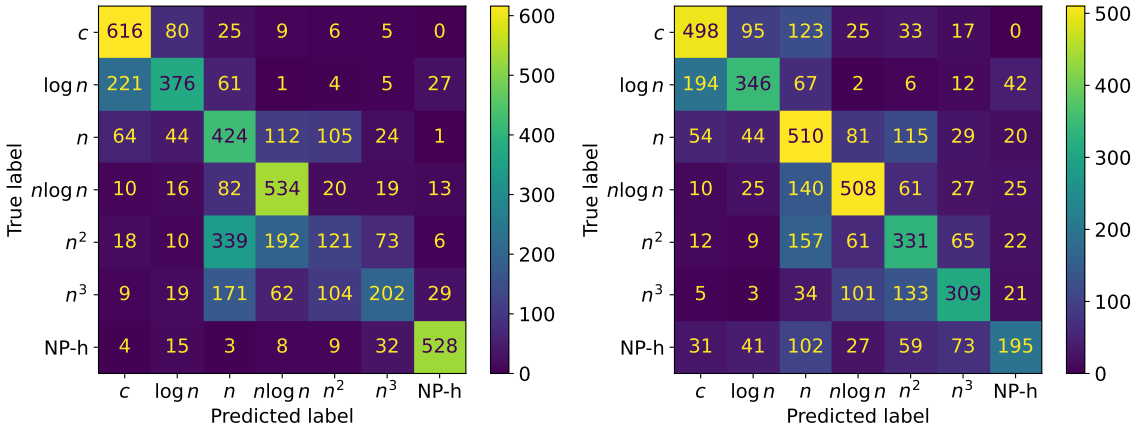

Figure 8: Confusion matrices of prediction results by UniXcoder on Java (left) and Python (right) datasets.

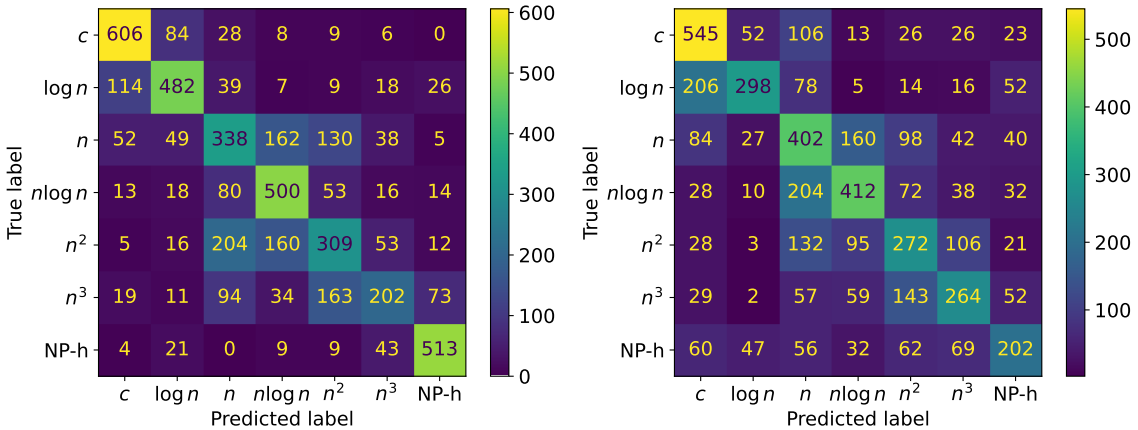

Figure 9: Confusion matrices of prediction results by CodeT5 on Java (left) and Python (right) datasets.

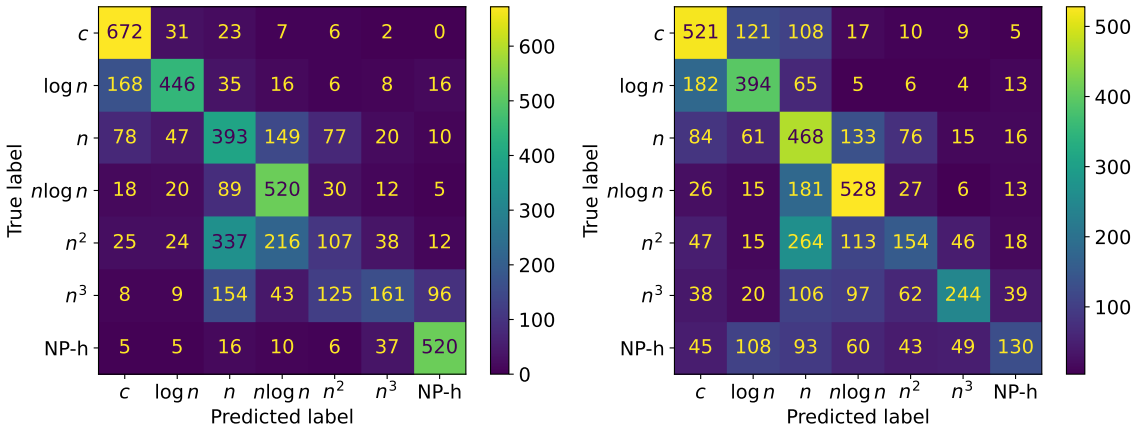

Figure 10: Confusion matrices of prediction results by CodeT5+ on Java (left) and Python (right) datasets.

Table 8: Complexity prediction accuracy of classification methods for each complexity class on Python.

Method	$O(1)$	$O(\ln n)$	$O(n)$	$O(n \ln n)$	$O(n^2)$	$O(n^3)$	NP-hard	Micro	Macro
Decision Tree	45.0	39.8	37.0	62.4	42.1	65.8	6.6	38.8	42.7
Random Forest	52.9	53.4	44.8	63.4	42.0	69.4	18.5	40.8	49.2
SVM	43.1	25.3	78.6	52.1	14.0	20.7	13.7	23.6	35.4
CodeBERT	68.0	66.1	31.7	46.9	40.6	63.8	25.6	51.2	49.2
GraphCodeBERT	68.5	56.9	61.9	51.4	56.8	68.1	34.8	**58.1**	**57.3**
UniXCoder	63.0	59.8	51.7	50.4	51.0	63.8	36.9	55.0	54.4
PLBART	72.1	62.3	51.9	46.3	48.5	59.3	24.2	54.0	52.4
CodeT5	68.9	47.1	44.5	41.4	43.6	51.8	38.3	48.9	48.4
CodeT5+	65.9	54.9	58.9	23.4	40.3	66.3	24.6	49.8	47.7
ChatGPT 3.5	44.4	46.4	83.0	12.3	60.6	29.8	0.0	41.8	35.6
ChatGPT 4.0	54.7	33.0	80.0	39.6	61.4	78.3	0.0	51.7	41.9
Gemini Pro	35.9	19.5	72.0	8.4	61.3	23.2	1.5	35.1	28.5

Gemini prompt example

You are the best programmer in the world.
The given code: "[code]"
Print "ONLY the time complexity in ONE WORD" of the given code in the answer from np, logn, quadratic, constant, cubic, linear and nlogn, do not print any other words in a json format.

Figure 11: LLM prompt examples used in our experiments with Gemini.