# Zero-shot Prompting for Code Complexity Prediction Using GitHub Copilot

Mohammed Latif Siddiq
*University of Notre Dame*
Notre Dame, IN, USA
msiddiq3@nd.edu

Abdus Samee
*Bangladesh University of Eng. and Tech.*
Dhaka, Bangladesh
1805021@ugrad.cse.buet.ac.bd

Sk Ruhul Azgor
*Bangladesh University of Eng. and Tech.*
Dhaka, Bangladesh
1805091@ugrad.cse.buet.ac.bd

Md. Asif Haider[§]
*Bangladesh University of Eng. and Tech.*
Dhaka, Bangladesh
1805112@ugrad.cse.buet.ac.bd

Shehabul Islam Sawraz[§]
*Bangladesh University of Eng. and Tech.*
Dhaka, Bangladesh
1805088@ugrad.cse.buet.ac.bd

Joanna C. S. Santos
*University of Notre Dame*
Notre Dame, IN, USA
joannacss@nd.edu

***Abstract*—Code generation models are gaining popularity because they can produce correct code from a prompt, speeding up the software development process. GitHub Copilot is currently one of the most commonly used tools for code generation. This tool is based on GPT3, a Large Language Model (LLM), and can perform zero-shot prompting tasks *i.e.*, tasks for which the model is not specifically trained. In this paper, we describe a preliminary study that investigates whether GitHub Copilot can predict the runtime complexity of a given program using zero-shot prompting. In our study, we found that GitHub Copilot can correctly predict the runtime complexity 45.44% times in the first suggestion and 56.38% times considering all suggestions. We also compared Copilot to other machine learning, neural network, and transformer-based approaches for code complexity prediction. We observed that Copilot outperformed other approaches for predicting code with linear complexity O($n$).**

***Index Terms*—code generation, computational complexity, transformer, zero-shot prompting, pre-trained model, GitHub copilot**

## I. Introduction

Computational complexity analysis is an essential task in the study of algorithm development [1]. The worst-case runtime complexity of an algorithm denotes the longest time to execute a program, depending on the algorithm's input size. The upper bound of the runtime complexity is mathematically described with the Big O notation ($\mathcal{O}$). For example, if an algorithm takes constant time (*i.e.*, not depending on the algorithm's input size), the upper bound complexity of this algorithm is $\mathcal{O}(1)$, whereas if an algorithm takes a linear time according to its input size $n$, its complexity is $\mathcal{O}(n)$.

Developers can use runtime complexity analysis to learn how much time and space their code will take, as well as find exceptionally efficient or inefficient algorithms and data structures. This can help select the best algorithms for particular tasks and locate and remove pointless actions that might be slowing down the program [1].

[§]These authors equally contributed to this work.

With the release of GitHub Copilot in 2021, automated code generation techniques are increasingly being adopted in the industry [2], [3]. These techniques rely on a user-provided *prompt* to generate code [4]. The prompt specifies the developer's intent and can have varying granularity and structure. They can include code comments, code elements (*e.g.*, function signatures, expressions, identifiers, *etc.*), or a combination of these. Thus, developers can produce a first iteration of code and/or a comment and then rely on these tools to generate the remaining code, saving them time and accelerating the software development process [2], [3], [5].

Although code generation models based on a Large Language Model (LLM) are specifically designed for code generation tasks, it is well known that LLMs can be used for other tasks with *zero-shot prompting* [6]. Zero-shot prompting means the model can be used for tasks that are not explicitly trained but can produce a reasonable solution. In this short paper, we investigate whether the GitHub Copilot, a GPT3-based [7] code generation model [8], can understand the code's structure to help predict the code's computational complexity.

## II. Background

### A. Zero-shot Prompting

Zero-shot prompting is the ability of a natural language processing (NLP) system to generate prompts to elicit the desired response from a user, even in situations where the system has not been explicitly trained on examples of the desired response [9]. This capability is important for NLP systems to be more flexible and adaptable. Researchers have explored unsupervised, transfer, and multitask learning approaches to address the lack of labeled training data for the desired responses [10]–[12]. For example, transfer learning has shown promising results for zero-shot prompting by transferring knowledge from one task or domain to another using a small amount of labeled data from the target task [12].

## *B. Computational Complexity*

Computational complexity refers to the time it takes for a computer to execute an algorithm, considering the input's size $n$ [13]. For example, if the complexity of an algorithm is $\mathcal{O}(n^2)$, the execution time will increase at a much faster rate as the input size grows than if the complexity was $\mathcal{O}(n \times log(n))$. Developers must consider the time complexity when designing and implementing algorithms to create more efficient solutions.

## III. Methodology

In the next sections, we describe *(i)* the dataset we used for evaluation, *(ii)* the systematic steps we followed to use GitHub Copilot to compute the complexity of the collected samples, and *(iii)* how we analyzed the results.

### *A. Dataset*

We collected the dataset from Jeon *et al.* [14], which consists of 4,120 samples from the CodeContests dataset from DeepMind [15], CoRCoD dataset [13] and samples created by the authors. The samples are written in Java and are manually annotated by experts with one of the following complexities: $\mathcal{O}(1)$ (constant), $\mathcal{O}(n)$ (linear), $\mathcal{O}(n^2)$ (quadratic), $\mathcal{O}(n^3)$) (cubic), $\mathcal{O}(ln(n))$ (logarithmic), $\mathcal{O}(n \times ln(n))$ (linear logarithmic), and NP-hard. We used the test set of this dataset which contains **768** samples. Table I presents the distribution of the dataset used in our work.

TABLE I
Class Distribution in the Test Dataset from [14]

| Class | Description | # of Samples |
|---|---|---|
| $\mathcal{O}(1)$ | Constant | 106 (13.80%) |
| $\mathcal{O}(n)$ | Linear | 102 (13.28%) |
| $\mathcal{O}(ln(n))$ | Logarithmic | 114 (14.84%) |
| $\mathcal{O}(n^2)$ | Quadratic | 117 (15.23%) |
| $\mathcal{O}(n^3)$ | Cubic | 112 (14.58%) |
| $\mathcal{O}(n \times ln(n))$ | Linear Logarithmic | 103 (13.41%) |
| NP-hard | - | 114 (14.84%) |

### *B. Workflow*

*1) Data Collection:* We used zero-shot prompting to obtain the complexity class for the Java code. That is, we type the comment "`// Complexity:`" for each sample. This edit action instructs GitHub Copilot to generate code by using as context the method already written. It generates *up to* 10 suggestions, and we collected all of them for analysis.

Listing 1 has an example of a binary search on an array of integer numbers. We highlight the comment (line 13) used as a prompt to Copilot to get the code's complexity. This code is for illustration purposes and is not present in the dataset.

*2) Data Analysis:* After obtaining up to ten suggestions for each sample, we manually went through them to analyze the predicted complexity. We verified the predicted complexity by analyzing the first result only (Top-1) and all the suggestions (Top-all) and compared them to the ground truth.

BinarySearch.java

```
1  class BinarySearch {
2      public boolean binarySearch(int arr[], int first,
3                                  int last, int key) {
4          int mid = (first + last) / 2;
5          while (first <= last) {
6              if (arr[mid] < key) { first = mid + 1; }
7              else if (arr[mid] == key) { return true; }
8              else { last = mid - 1; }
9              mid = (first + last) / 2;
10         }
11         if (first > last) { return false; }
12     }
13     // Complexity:
14 }
```

Listing 1: Using GitHub Copilot to predict the complexity

The suggestions produced by GitHub Copilot could be the best case, worst case, and the average case of the runtime complexity. In our analysis, we used the *worst-case complexity*. As we used zero-shot prompting, there was no mention of the possible classes. Hence, the GitHub Copilot can predict a class that is not present in the seven classes we are considering. In this case, we consider it a negative prediction.

### *C. Evaluation*

We used the following metrics in the evaluation [16]:

- **Accuracy (Acc)**: It is defined as the proportion of correct predictions made by the model out of all predictions made: $Acc = \frac{TP+TN}{TP+FP+TN+FN}$.
- **Precision (P)**: It is calculated by dividing the number of records with correctly predicted labels by the total number of predicted observations in that class: $P = \frac{TP}{TP+FP}$.
- **Recall (R)**: It is computed for each group A by dividing the number of successfully predicted observations in A by the total number of observations in the corresponding class: $R = \frac{TP}{TP+FN}$.
- **F1-Score (F1)**: It is the harmonic mean of the precision and recall: $F1 = \frac{2\times(P\times R)}{P+R}$.

True Positives (TP) are the instances where the model correctly predicted the positive class. True Negatives (TN) are the number of instances where the model correctly predicted any negative classes. False Positives (FP) are the number of instances where the model incorrectly predicted the positive class. False Negatives (FN) are the number of instances where the model incorrectly predicted the negative class.

## IV. Results

Table II compares the Top-1 accuracy result from GitHub Copilot with other machine learning-based approaches. The first three, *i.e.*, Decision Tree, Random Forest, and Support Vector Machine (SVM) are approached with no involvement of classic deep learning approach [13]. Decision Tree works better for *quadratic* complexity prediction. ASTNN [17] is a neural network based on Abstract Syntax Tree (AST) and has a better prediction performance for *cubic* complexity. CodeBERT [18] and GraphCodeBERT [19] are BERT-based encoder pretrained models. PLBART [20] and CodeT5 [21]

TABLE II
ACCURACY RESULTS FOR TOP-1 SUGGESTION FROM GITHUB COPILOT COMPARED TO OTHER APPROACHES

| **Method** | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(ln(n))$ | $\mathcal{O}(n \times ln(n))$ | **NP-hard** |
|---|---|---|---|---|---|---|---|
| Decision Tree | 58.70% | 15.10% | **71.40%** | 35.20% | 47.90% | 67.70% | 33.50% |
| Random Forest | 68.10% | 18.00% | 38.40% | 25.00% | 48.60% | 68.00% | 67.90% |
| SVM | 42.60% | 17.60% | 13.10% | 6.00% | 27.10% | 24.90% | 77.00% |
| ASTNN | 71.40% | 26.70% | 14.20% | **49.50%** | 56.10% | 63.00% | 82.00% |
| CodeBERT | 78.40% | 49.10% | 44.80% | 32.60% | 76.40% | 71.70% | 81.30% |
| GraphCodeBERT | 83.00% | 40.70% | 59.30% | 11.20% | 69.50% | 73.40% | 66.00% |
| PLBART | **83.40%** | 56.50% | 45.10% | 38.20% | 75.00% | 73.10% | 88.40% |
| CodeT5 | 77.50% | 46.00% | 29.90% | 17.30% | 75.90% | 70.20% | 83.20% |
| CodeBERT + HA | 79.30% | 44.00% | 45.10% | 32.20% | 72.20% | 77.90% | **89.70%** |
| (+ Pretrain) | 70.00% | 57.80% | 45.60% | 39.70% | 76.50% | 78.80% | 88.10% |
| (+ dead code elimn.) | 72.90% | 61.00% | 42.20% | 40.00% | **79.60%** | **80.00%** | 88.60% |
| GitHub Copilot(Top-1) | 46.23% | **69.61%** | 49.11% | 16.67% | 23.08% | 66.99% | 50.00% |

use both encoder and decoder for pretraining. Finally, Jeon *et al.* [14] used hierarchical architecture, a pre-trained model, and dead code elimination. These approaches perform better for *constant*, *logarithmic*, *linear logarithmic*, and *NP-hard* complexity prediction than any other models. As we can observe in Table II, GitHub Copilot performed better for *linear complexity* than any other model. It also performed better for *quadratic complexity* than encoder-decoder-based pre-trained models, and Jeon *et al.* [14].

TABLE III
PRECISION, RECALL, F1-SCORE AND ACCURACY FOR EACH CLASS

| **Class** | **Precision** | **Recall** | **F1-Score** | **Accuracy** |
|---|---|---|---|---|
| $\mathcal{O}(1)$ | 0.75 | 0.46 | 0.57 | 46.23% |
| $\mathcal{O}(n)$ | 0.29 | 0.70 | 0.41 | 69.61% |
| $\mathcal{O}(n^2)$ | 0.30 | 0.49 | 0.37 | 49.11% |
| $\mathcal{O}(n^3)$ | 0.53 | 0.17 | 0.25 | 16.67% |
| $\mathcal{O}(ln(n))$ | 0.77 | 0.23 | 0.36 | 23.08% |
| $\mathcal{O}(n \times ln(n))$ | 0.54 | 0.67 | 0.60 | 66.99% |
| NP-hard | 1.00 | 0.50 | 0.67 | 50.00% |
| **Top-1** | 0.52 | 0.40 | 0.40 | 45.44% |
| **Top-All** | - | - | - | 56.38% |

Table III shows the GitHub Copilot's precision, recall, F1-score, accuracy for the Top-1 result, and overall accuracy for Top-1 and Top-All scenarios. For the Top-1 scenario, we presented the macro-average result as the dataset was not balanced. We found that the Top-all performs better than the Top-1 class based on accuracy. This means that GitHub Copilot may include the correct answer in one of its ten suggestions, but it may not always be the first result.

## A. Discussion

Figure 1 shows the confusion matrix for the Top-1 suggestion from GitHub Copilot. We can see that GitHub Copilot confuses *constant* complexity with *linear* complexity. For *logarithmic* complexity, most of them are predicted as *linear* complexity. Most of the *cubic* complexity was predicted as *quadratic* complexity. It also got confused between *NP-Hard* and *quadratic* complexity samples.

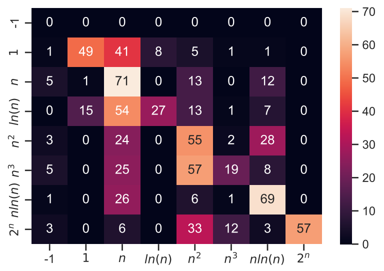

Fig. 1. Confusion Matrix for Complexity Prediction using Top-1 Suggestion

Although OpenAI, the parent company behind the Codex model [8] that powers GitHub Copilot, has a demo for code time complexity prediction using prompt engineering [22], we observed that it falls short in understanding the algorithmic nature of an implementation to predict its time complexity. Even though using zero-shot prompting on GitHub Copilot can be helpful for linear complexity prediction, it falls behind other deep-learning-based models for other computational complexities. These DL models are specifically trained for code complexity prediction, so they perform better than GitHub Copilot (a Codex-based model).

## B. Threats to Validity

An internal validity threat to this work is that we manually collected the top and additional suggestions from the GitHub Copilot and labeled them in corresponding complexity classes. To mitigate this threat, the authors cross-checked the analysis to ensure accuracy. Another threat to this work is that we used GitHub Copilot as an off-the-shelf, closed-source tool whose outputs may change over time and differ across different environments. We use the same environment to extract the complexity predictions to mitigate this threat.

## V. RELATED WORK

Prior works aimed to estimate a code's complexity through different code metrics, such as cyclomatic complexity [23], [24]. Other prior works described ML-based solutions to computational complexity prediction. For example, Sikka *et al.*

[13] created a dataset with over one thousand Java samples and used multiple features (*e.g.*, the number of loops, breaks, ifs, *etc.*) to benchmark different ML methods. Similarly, Prenner *et al.* [25] investigated the possibility of using a pre-trained model like CodeBERT [18] in software engineering tasks, including code complexity prediction. Recently, Jeon *et al.* [14] used hierarchical architecture, pre-training, and dead code elimination approaches in addition to CodeBERT [18] model. All these ML-based approaches fine-tuned the model with a training set and evaluated the performance with a test set. In this paper, however, we did not do any fine-tuning for GitHub Copilot. Instead, we used zero-shot prompting to predict code runtime complexity.

## VI. Conclusion

We investigated whether GitHub Copilot, an LLM-based code generation model, can be useful not only for code generation but also for computational complexity prediction. A correctly predicted time complexity without fine-tuning the model would be a cost-efficient solution. We found that GitHub Copilot can correctly predict a code's complexity half the time. We also observed that the correct complexity was often included in one of the 10 predictions made by Copilot. We plan to extend this work to other code-generation tools and complexity classes in future work.

## References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[2] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: ACM, 2022, p. 21–29. [Online]. Available: https://doi.org/10.1145/3520312.3534864

[3] E. Kalliamvakou, "Research: quantifying github copilot's impact on developer productivity and happiness," 2022. [Online]. Available: https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[4] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[5] F. Sedghi Farooji, "Evaluation of code generation tools," 2014.

[6] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. [Online]. Available: https://openreview.net/forum?id=e2TBb5y0yFf

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan *et al.*, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[9] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2021. [Online]. Available: https://arxiv.org/abs/2109.01652

[10] V. Sanh, A. Webson, C. Raffel, S. H. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, T. L. Scao, A. Raja *et al.*, "Multitask prompted training enables zero-shot task generalization," *arXiv preprint arXiv:2110.08207*, 2021.

[11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[12] H. Pham, Z. Dai, G. Ghiasi, H. Liu, A. W. Yu, M.-T. Luong, M. Tan, and Q. V. Le, "Combined scaling for zero-shot transfer learning," *arXiv preprint arXiv:2111.10050*, 2021.

[13] J. Sikka, K. Satya, Y. Kumar, S. Uppal, R. R. Shah, and R. Zimmermann, "Learning based methods for code runtime complexity prediction," in *Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part I 42*. Springer, 2020, pp. 313–325.

[14] M. Jeon, S. yeop Baik, J. Hahn, Y.-S. Han, and S.-K. Ko, "Deep learning-based source code complexity prediction," 2023. [Online]. Available: https://openreview.net/forum?id=9irBKvxsw9

[15] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[16] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 39.

[17] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st Int'l Conf. on Software Engineering (ICSE)*, 2019, pp. 783–794.

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[19] D. Guo, S. Ren, S. Lu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *9th Int'l Conf. on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[20] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://www.aclweb.org/anthology/2021.naacl-main.211

[21] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conf. on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[22] OpenAI, "Calculate time complexity," 2023. [Online]. Available: https://platform.openai.com/examples/default-time-complexity

[23] S. Henry and C. Selig, "Predicting source-code complexity at the design stage," *IEEE software*, vol. 7, no. 2, pp. 36–44, 1990.

[24] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[25] J. A. Prenner and R. Robbes, "Making the most of small software engineering datasets with modern machine learning," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5050–5067, 2021.