Optimizing Class-Level Code Generation: Enhancing In-Context Learning in Large Language Models with Pruning Techniques

Anonymous ACL submission

Abstract

Recently, many large language models (LLMs) have been proposed, showing advanced proficiency in code generation. Such generation focuses on generating independent and often method-level code, thus leaving it unclear how LLMs perform in generating more complicated tasks. To fill this research gap, researchers have studied to construct prompt to achieve good performance in complicated code generation, i.e., class-level code generation, and they have launched a corresponding benchmark, ClassEval, on generating and evaluating class-level code. However, the obvious difficulty of class-level code generation prompt construction lies in that class-level prompt has longer texts than those of method-level code generation, and at the same time the input length of the released models always has a limitation, such as: GPT-3.5 and GPT-4 with 4096 tokens and 8192 tokens limitations respectively.

Therefore, it is important to research how to construct the class-level prompt by pruning some code tokens. Through the pruning strategy, we add more code examples into prompt to deliver as many semantic information as possible to LLMs. We introduce a new pruning strategy, namely attention-guided strategy, to this research point. By this pruning strategy, we conduct experiments on code generation by GPT-3.5, a kind of LLM proved to achieve excellent performance on generation tasks. In our work, we adopt ClassEval benchmark dataset specialized for class-level code generation to conduct our experiments. Additionally, we evaluate the strategy both in method-level and class-level metrics, finding that this pruning strategy is effective to prune appropriate tokens for LLM to generate class-level code. Above all, attention-guided strategy outperforms the randomly pruning strategy with 4.2%, 10.2% and 13% higher class-level code generation accuracy by LLM. We also analyze the impact of the quantity of code reduction on the quality of code generation in LLM, concluding that pruning under 40% of code snippets with extra 4 examples included can take great advantage of the intelligence of LLM to contribute to perfect class-level code generation.

1 Introduction

With the rapid advancement of large language models (LLMs), code generation from natural language descriptions has seen significant progress in recent studies(Kang et al., 2023a,b; Vikram et al., 2023). Researchers have developed numerous LLMs, such as SantaCoder(Allal et al., 2023), InCoder(Fried et al., 2023), StarCoder(Li et al., 2023), GPT-4(OpenAI, 2024), Instruct-CodeGen(Yuan et al., 2023), Instruct-StarCoder(Du et al., 2024), and CodeBERT(Feng et al., 2020a), all targeting code generation by training on vast quantities of general code corpus.These state-of-the-art models have impressive capacities, capable of handling up to 2048 tokens with WizardCoder(Luo et al., 2023) and 8192 tokens with GPT-4(OpenAI, 2024). However, the prevalent use of LLMs to generate short, independent code snippets underutilizes their potential, particularly in handling complex class-level code generation tasks(Qin et al., 2024).

Current approaches often focus on generating short code snippets with limited tokens, which fails to leverage the full capacity of modern LLMs. Despite their advanced capabilities, these models are often restricted by input length limitations, posing a challenge for generating longer, more complex code structures. The introduction of the ClassEval benchmark(Du et al., 2023) aims to address this by targeting class-level code generation with longer and interdependent code snippets. However, managing overlong inputs remains a critical concern, as every LLM has inherent limitations on input length.

Pruning strategies are essential for managing long input code snippets(Lu and Debray, 2012), yet existing methods have limitations. Traditional code pruning methods, often based on delta debugging prototypes(Zeller and Hildebrandt, 2002), require auxiliary deep models(Rabin et al., 2021; Suneja et al., 2021), making them complex and resource-intensive. Effective pruning must balance trimming excessive tokens while preserving essential semantic and structural information. This is particularly challenging in class-level code generation, where dependency and contextual information are crucial. This paper introduces an attention-guided pruning strategy, leveraging the attention mechanism(Bahdanau et al., 2016) from CodeBERT to selectively prune tokens from input prompts. By trimming 10%, 20%, 30%, and 40% of input tokens, the strategy allows for the inclusion of more ground-truth code generation examples, thereby enriching the prompt. Experiments conducted with GPT-3.5 on the ClassEval benchmark demonstrate the strategy's effectiveness, showing improved performance in class-level code generation tasks. The attention-guided pruning method consistently outperforms random pruning, maintaining stable performance with minimal information loss when pruning up to 40% of tokens. Our results indicate advancements ranging from 4.2% to 13% in $Pass@k$ evaluations, highlighting the strategy's potential in enhancing LLM's code generation capabilities. However, it also underscores the greater complexity of class-level code generation compared to method-level tasks due to higher dependency and contextual requirements.

The key contributions of this paper are as follows.

- **Pruning Strategy.** We propose the simplification strategy of LLM's prompt, named attention-guided strategy.
- **Efficacy on LLM's code generation.** This simplification method optimizes the length of prompts for contextual learning in large language models, allowing them to encounter richer contextual scenarios. Consequently, this enhancement improves the effectiveness of class-level code generation.
- **Open Source.** We have open-sourced all the code on https://zenodo.org/records/11640097, making it available for subsequent research to facilitate deeper replication and further exploration.

2 Background

In this section, we introduce the recent state-of-the-art LLMs for code generation. We introduce the benchmarks for code generation and focus on class-level benchmark ClassEval in Appendix A.

2.1 Large Language Models for Code Generation

Code generation is a classical task in computer science which is to construct code snippets with description in the human beings natural languages and is nowadays been widely studied(Kang et al., 2023a,c; Vikram et al., 2023). The recent LLMs, pre-trained by large quantities of text corpora with enormous number of parameters, achieve fantastic performances in various NLP tasks (Chang et al., 2024; Clark et al., 2020; Lample and Conneau, 2019) including code generation, such as: ChatGLM(Du et al., 2021) and the well-known GPT-4(OpenAI, 2024). The prevalent LLM, GPT-4, outperforms other models on HumanEval benchmark(Luo et al., 2023), and it achieves the highest correctness score among other ten models on ClassEval benchmark(Du et al., 2023). Hence, more and more researchers tend to exploit the LLMs to accomplish code generation tasks(Chen et al., 2021; Shen et al., 2023). The code LLMs, which are pre-trained by plenty of code snippets on purpose, have stronger capacity than general LLM in code generation tasks(Luo et al., 2023; Zan et al., 2023; Christopoulou et al., 2022). From now on, a large number of code LLMs have been proposed including CodeBERT(Feng et al., 2020b), WizardCoder(Luo et al., 2023), Instruct-StarCoder(Du et al., 2024), Instruct-CodeGen(Yuan et al., 2023), etc. Specifically, we take CodeBERT into detailed introduction. CodeBERT is a bimodal pre-trained language model by semantic representation from several programming languages(Feng et al., 2020b). The trained capacity of CodeBERT can be utilized to solve kinds of downstream tasks including code search and code summarization accomplished in (Feng et al., 2020b). It has been demonstrated that CodeBERT has a great advantage on understanding semantics of code snippets than other deep learning models such as code2vec(Alon et al., 2019) and ASTNN(Zhang et al., 2019). It is an encoder on Transformer(Vaswani et al., 2017) structure. With pre-trained CodeBERT, we can adopt it to downstream tasks by fine-tuning.

Additionally, prompt(Liu et al., 2023b) is widely-used to inspire LLM's creativity and intelligence so prompt engineering arouses researchers' attention recently aiming to craft a well-structured prompt tailored to a LLM and execute predictions with anticipated high performance. Prompt is potential to be well self-adapted, proposing alternative prompts to elicit further information or generate associated artifacts(White et al., 2023). There are two main prompt engineering tasks which are prompt template engineering and prompt answer engineering(Liu et al., 2023a).

3 Method

As mentioned before, the more tokens we feed into input, the much more capability of understanding requirements the LLM will achieve and the more exactly the LLM will respond to your requests(Minaee et al., 2024). Every large-scale language model is subject to input token length restrictions, such as GPT-4 with a limit of 8192 tokens and GPT-3.5 with a limit of 4096 tokens. Therefore, we wonder how to deliver input prompt into LLMs as many as possible meanwhile the LLM won't return an exceeded maximum length error back.

In this section, we introduce the pruning method to prune input tokens delivered into LLMs. Our principle is to remove some unimportant tokens and statements from the input of LLM, however, retain essential information of input. Meanwhile, we can add more practical code examples into the pruned short prompt leading LLM to produce a exact code answer.

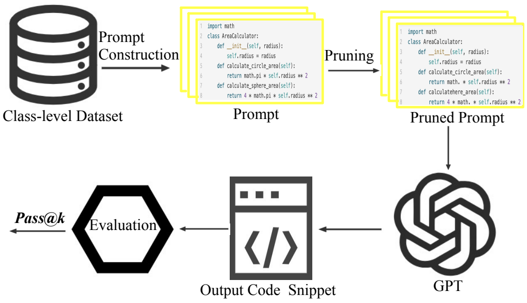
To be more specific, we define a code snippet $C = \{t_1; ... : t_{|C|}\}$, consisting of $|C|$ tokens. Our goal is to produce a pruned code snippet C_p which retains only the max limitation length of L LLM input tokens.

3.1 Implementation Process

The work of this paper contributes to prompt pruning in order to convey more code generation examples into LLM. And the prompt inspires the model to produce more specific code snippet to pass the more test cases. There are two main processes in the entire implementation including the prompt pruning and evaluate output code snippet produced by LLM as depicted in Figure 1.

At the beginning of the implementation, we construct the prompt as mentioned in Appendix B and add as many code generation examples as possible from ClassEval dataset into the prompt. These examples are composed in the form of input-output pairs which are the instruction of code generation as input and the ground truth code snippet as output. Certainly, these examples do not contain desired code snippet and are chosen randomly. Then, we conduct the emphasis of the work to prune the prompt to restrict the length under the limitations of token numbers of certain LLM, specifically the "GPT-3.5-turbo" model[1], with the methods introduced in Section 3. The maximum input token numbers of "GPT-3.5-turbo" model is set to 4096, in other words we have to prune the instruction part and examples part of the prompt other than testing part in Appendix B due to the consistency of the LLM input.

The second process is to request LLM with temperature of 0 for the stability of LLM's output, namely GPT-3.5-turbo, to generate the code solution snippet for each input processed prompt. Finally, we evaluate the fresh produced code snippet by calculating the metrics $Pass@1$,$Pass@3$ and $Pass@5$ as illustrated in Section 4.2.



```
import math
class AreaCalculator:
    def __init__(self, radius):
        self.radius = radius
    def calculate_circle_area(self):
        return math.pi * self.radius ** 2
    def calculate_sphere_area(self):
        return 4 * math.pi * self.radius ** 2
```

```
import math
class AreaCalculator:
    def __init__(self, radius):
        self.radius = radius
    def calculate_circle_area(self):
        return math. * self.radius ** 2
    def calculatehere_area(self):
        return 4 * math. * self.radius ** 2
```

Figure 1: Implementation Process On Prompt Pruning and Evaluation

3.2 Attention-guided Pruning

The frequency-based selection strategy actually differentiates the common used tokens and uncommon used ones with great effectiveness in pruning input code tokens, while it doesn't perform well in choosing the input code tokens of great significance and semantic meaningfulness. In order to prune tokens with regard for the semantic importance of each token, we introduce an attention-guided pruning strategy that chooses input tokens from code snippet based on the attention weights produced by BERT model, such as: CodeBERT and etc. The tokens with high attention are not exactly matching the tokens which occur frequently, especially some

[1] GPT-3.5-turbo is an advanced large language model developed by OpenAI, which serves as an enhanced version of the GPT-3 series. And it is suitable for a variety of natural language processing tasks.

method or class names obtain high attention when feed into BERT model while these tokens just appear several times in the code snippet.

Algorithm 1 displays the simplified code programming process. Firstly, we generate the attention of each input code token through the BERT model according to the empirical studying. Then, we select tokens separated from input code prompt with higher attention after acquiring the tokens and its corresponding attention. The algorithm is divided into two phases: generating tokens' attentions and selecting tokens. In the first phase of the above algorithm, generating attention phase, first we divide the input code snippet into input tokens which the BERT model requires with the certain tokenizer. Then we send several batches of input tokens into BERT model and receive the output of the model, and the each batch size m depends on the limitation of the BERT model input length. Last, we can extract the attention of each input token correspondingly.

Second phase will contribute to the well-pruned code snippet based on the attentions produced during the first phase. We adapt a 0-1 backpack strategy(Martello and Toth, 1987) to choose input tokens, where the separated input tokens can be considered as the items to be collected into the backpack, with the attention of each token being the values, and the token numbers being the weights because it is the number of tokens that really acts when LLM preforms. Then chosen input code tokens will be detokenized into original input code and concatenated into the pruned input code snippet and finally the pruned input code snippet will be sent into LLM which generates code we wanted without exceeded input limitation error.

```
Algorithm 1: Attention-guided Pruning
 Notation:
   • $C = t_1, t_2, \ldots, t_{|C|}$: Input code snippet
   • $\mathbf{A}_t$: Attention scores from the BERT model
   • $C_p = t'_1, t'_2, \ldots, t'_{|C|}$: Pruned code snippet
 Procedure:
   1. Generate attention scores
      for each $t_1, t_2, \ldots, t_m \in C$ do
          $output \leftarrow$ BERT$(t_1, t_2, \ldots, t_m)$;
          $\mathbf{A}_{t_1,t_2,\ldots,t_m} \leftarrow output.attentions$;
   2. Conduct 0-1 knapsack optimization
      $\{t\}_{t \in C_0} \leftarrow$ 0-1 knapsack(values $=$
      $\{\mathbf{A}(t)_{t \in C}\}$, items $= \{t\}_{t \in C}$, weights $=$
      $\{|t|\}_{t \in C}$, capacity $=$
      Model Input Length Limit);
   3. Generate the final code snippet
      $C_p \leftarrow t'_1, t'_2, \ldots, t'_{|C_0|}$, where $C_0 \in$ C;
```

The time complexity of attention-guided pruning algorithm is $O(N * L_T)$, where L_T denotes the target numbers of tokens. The cost of time mainly owes to the 0-1 backpack algorithm. Additionally, the 0-1 backpack algorithm demands to handle two-dimensional array when programming dynamically, costing space in memory.

4 EMPIRICAL STUDY

As mentioned before, we utilize the state-of-the-art ClassEval benchmark to evaluate our methods of class-level prompt engineering.

4.1 Models

In this work, our goal is to generate code in the accordance with the processed prompt by means of the method in Section 3, therefore we adopt the prevalent GPT series models.

The core of GPT series models lies in the Transformer architecture(Ghojogh and Ghodsi, 2020), a deep neural network framework renowned for its proficiency in handling sequential data, particularly in capturing long-range dependencies. Typically, GPT models consist of multiple Transformer blocks, each comprising self-attention mechanisms and feed-forward neural networks, stacked together to form the entire model(OpenAI, 2024). The uniqueness of GPT models lies in their pre-training process. During pre-training, GPT models employ unlabeled, large-scale text data for self-supervised learning. By predicting the next token in a sequence task, GPT models learn semantic and syntactic information from the text data, enabling them to exhibit strong generalization capabilities in downstream tasks. Masked language modeling is commonly employed as a pre-training task for GPT models, which requires the model to predict masked tokens based on context. The two most commonly used models are GPT-3.5 and GPT-4 which just differ slightly in performance, so we choose GPT-3.5 to generate the final output code demanded by ClassEval benchmark. Table 1 presents the model we chose with their specific

Table 1: Expeimental Settings

Item	Value
Model Name	GPT-3.5-turbo-0613
Context Window	4096 Tokens
Temperature	0

parametres.

4.2 Metrics

To evaluate the output of LLMs, we adopt the commonly-used $Pass@k$(Chen et al., 2021) metric to measure the performance of LLMs, which calculate the pass percentage of k generated code snippet samples for each task. The calculation formula is:

$$\textbf{Pass@k} = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \binom{n-c}{k} / \binom{n}{k} \right] \quad (1)$$

In Eq. 1, n stands for the total number of code samples, c denotes the number of passed code samples, and k represents the number of samples chosen from n code samples which is k in $pass@k$. In our work, although we pay more attention on class-level code generation performance by LLMs, we also take both the class-level $Pass@k$ and the method-level $Pass@k$ into consideration with class granularity and method granularity respectively. The generated class-level code sample is determined to be correct only if it has passed all the method-level and class-level test cases.

For each method we test, we conducted sampling to generate code samples and evaluate the performance of each method by $Pass@k$ with $k = \{1, 3, 5\}$. We calculate the success rate of code generation based on two kinds of $Pass@k$. The first one, **all success**, indicates that the code snippets generated in both attempts passed all test cases. The second one, **partial success**, indicates that the code snippets generated in one of the two attempts passed all test cases without any exceptions or errors. More details of implementation process are displayed in Section 3.1.

4.3 Baselines

In this section, we introduce the baseline experiments. To perform a strong and effective validation, the baselines are conducted under the same implementation pipeline as depicted in Section 3.1.

The first experiment we conduct, compared with the pruning strategy we propose, is few-shot learning of LLM without pruning. Compared with the experiments we test out pruning strategy, this baseline is conducted under the process depicted in Figure 1 without the pruning process. In other words, the prompt constructed from the ClassEval dataset is directly fed into GPT. Furthermore, the prompt includes only a single code generation example in its examples part as described in Appendix B.

The second baseline experiment is conducted for ablation study, named random pruning. This baseline experiments are completely conducted under the implementation same as the process depicted in Figure 1. The core of this experiments to be conducted is to prune stochastically the generated prompt from ClassEval dataset with certain stochastical distribution. This experiment is designed to demonstrate whether our proposed strategy is effectual when pruning for this experiment also conduct prompt pruning but with random.

4.4 Experiment Settings

To validate the effectiveness of our method, we tested the effects of the strategy under the same baseline while ensuring consistent external conditions for each strategy, including LLM hyperparameters, the number of experiments, and other factors. To simplify the complexity of our experiments, we utilized the ClassEval benchmark, as it is the most comprehensive for class-level code generation, featuring a class-level dataset, pipeline, and evaluation framework. In our work, we take all 100 class -level code generation tasks in ClassEval into experiments. The prompt consists of examples in the form of input-output pairs which are the instruction as input and the ground truth code snippet as output. In addition, the desired code snippet does not emerge in those examples, chosen stochastically, in the prompt. With several tries of pruning tokens from prompt, we find that pruning over 40% tokens can destroy both semantics and structure of prompt and the prompt just contains some meaningless and unrelated tokens. Therefore the the pruning strategy is tested with the set of prune percentages: 10%, 20%, 30%, and 40% which averagely supply 4 code generation examples to the prompt. For the attention-based pruning strategy, we used the CodeBERT model with default parameters to generate specific attention values and set the window length to 500 tokens.

During the code generation phase, we employed the GPT-3.5 model from the GPT series, specifically using the official model name GPT-3.5-turbo-0613, with a temperature setting of 0.

5 Results

5.1 Overall Evaluation Accuracy

Through the above experiments, the attention-guided strategy performs really well on class-level code generation especially when the pruning proportion is 10% to 30%. Figure 2 shows the class-level and method-level $Pass@1$ of LLM, GPT-3.5-turbo, on ClassEval benchmark by pruning prompt. Considering space limits, we just present the class-level $Pass@1$ and, certainly, method-level $Pass@1$ with attention-guided pruning strategy. Table 2, 3, 4, 5 present the evaluation results, $Pass@k$, of both the class-level generation and the method-level generation with nucleus sampling[2] on ClassEval benchmark by means of the pruning algorithm. Those tables present the strategy we propose, the ablation study and few-shot experiments without pruning which generates code snippets with one example fed into the LLM without any pruning. As indicated in tables, the achieved results are presented in the form of "A/B", where A represents the $Pass@k$ when each generated example passes through all test cases, while B represents the $Pass@k$ when the generated examples only satisfy a subset of the test cases in the ClassEval benchmark. In accordance with Figure 2 and Table 2, 3, 4, 5 , we have the following observations.

Comparison among pruning quantities. As shown in Figure 2, it is obvious to conclude that LLM performs similarly when we prune a little fraction of input prompt at the range of percentage from 10% to 30%. The Class-level $Pass@k$ is around 8.7% and the method-level $Pass@k$ stabilises at around 20.1% with the max difference of 0.5% and 1.3% respectively. This is mainly because we feed as many examples as possible into the input prompt when cutting some tokens out. Therefore, the LLM reserves the enough understanding of the input prompt and obtains abundant code generation information from the input prompt even pruned. However, the LLM performs inferior when the pruning percentage of input prompt augments to 40% with the only $Pass@k$ of Class-level and Method-level, 3.4% and 10.4% respectively. The sharp declination of the LLM performance is on account of the excessive pruning of the input prompt which contributes to the destruction of prompt semantics, misunderstanding the LLM to generate code snippets casually.

Finding 1: The performance of both method-level and class-level code generation stabilizes when pruning the input prompt of LLM in a small portion mainly because of its structural integrity and appended extra examples. In addition, pruning really works in code generation tasks compared to the few-shot learning without pruning. Whereas huge amount of prompt pruning destroys the basic structure of input prompt and limits the intelligence of LLM to generate code as wanted which causes the lower $Pass@k$. These findings indicate that it is unnecessary to pay more attention on pruning quantities of input prompt tokens.

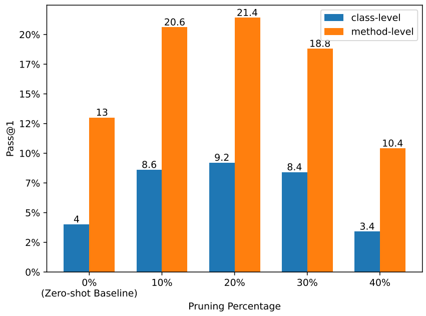


Figure 2: *Pass@1* on ClassEval with Attention-guided Pruning

5.2 Ablation Study and Analysis

We conduct ablation study with the simplest and most straightforward strategy, the lexical exclusion, which randomly selects some tokens from initial code snippet C_p and drop them out to meet the input length limitation of LLM. For every token $t \in T$, we define the variable p_w to imply whether the token chosen will be kept($r_w = 1$) or dropped($r_w = 0$) which follows the Bernoulli distribution:

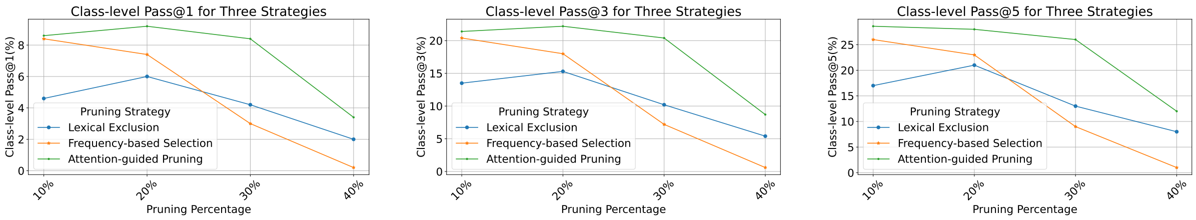
$$\begin{aligned} & r_w \sim Bernoulli(p) \\ & C_p = \{w | w \in Cand r_w > 0\} \end{aligned} \tag{2}$$

where $p = L/|C|$is the probability of choosing a token from the code snippet. From the above equation, lexical exclusion has been proved to be robust

[2] Nucleus sampling is also known as top-p sampling. By selecting the next token from a subset of top-probability tokens, it ensures the generated content is both coherent and contextually rich.

Table 2: *Pass@k* of Code Generation by Pruning Strategy with 20% Reduction

Experiments	Class-level All Success/Partial Success			Method-level All Success/Partial Success		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Few-shot Without Pruning	4%/7.0%	9%/16.5%	10%/20%	13.0%/15.9%	29.4%/36.2%	33.0%/41.2%
Random Pruning	6%/10.8%	15.3%/27.0%	21%/36%	15.0%/19.7%	36.6%/47.9%	47.4%/61.2%
Attention-guided Pruning	**9.2%/17.0%**	**22.2%/40.8%**	**28%/51%**	**21.4%/26.9%**	**50.2%/63.1%**	**60.2%/75.7%**



(a) Pass@1

(b) Pass@3

(c) Pass@5

Figure 3: Class-level Results of Three Strategies

to diverse networks and models, meanwhile it could prune input tokens exceeding max length limitation effectively. With the same implementation process described in Section 3.1, lexical exclusion is conducted as the ablation study for its random choices of tokens from input prompt. Through comparison between their results, it is clear whether the attention-guided strategy is actually effective.

After conducting the above ablation study, we take this study and the attention-guided strategy into comparison. Table 2, 3, 4 present the performance of the prompt pruning strategy introduced in Section 3 with the proportion of pruning from 10% to 40% under two generation strategies, class-level and method-level, on ClassEval benchmark. Based on results presented in these tables, the pruning strategy deliver quite great performances on LLM code generation.

Ablation study results show that attention-guided pruning strategy works well. On the one hand, the attention-guided pruning strategy achieve the better performance of both the class-level and method-level generation, when the pruning percentage is lower than 40% leading to the effective input prompt as explained before (*i.e.,* the improvements in **all-success** class-level generation range from 1.4% to 4.2% on $Pass$@1, from 3.3% to 10.2% on $Pass$@3 and from 4% to 13% on $Pass$@5). In addition, the attention-guided pruning strategy remains better or approximate performance than random pruning on method-level generation(*i.e.,* the improvements up to 6.5% on $Pass$@1, up to 13.6% on $Pass$@3 and up to 16.4% on $Pass$@5). Even though the pruning percentage comes up to 40%, the attention-guided pruning still remains good performance on code generation. The main reason of the advantages of attention-guided pruning strategy lies in two aspects.

Firstly, attention-guided strategy focuses on each token of input prompt rather than an entire word or statement as illustrated in Section 3.2. Therefore, attention-guided strategy takes the smaller granularity of input prompt into consideration compared to the ablation study, which avoids conducting prompt roughly. As two examples depicted in Figure 5, there are some meaningless tokens like "_" are pruned under strategy we propose as well as some tokens can be inferred from the context like "item". Additionally, the tokens we conducted through attention-guided pruning strategy are generated from certain tokenizers and the LLM recognizes the input text practically in this perspective. Therefore pruning tokens from input prompt is an advanced manner resulting in awesome performance of the attention-guided strategy.

Secondly, it is attention of input tokens that attention-guided pruning strategy really concerns with. The attention is produced by CodeBERT model through delivering the batch by batch context of input prompt into BERT model. Therefore, the attention value represents the significance of corresponding token amid the context. The higher a token's corresponding attention value is, the more semantically meaningful the token is and the more necessary it is to contribute to the understanding of LLM on code generation. In other words, the attention value is kind of measure scale on the importance of its corresponding token in model's view.

Finding 2: Attention-guided pruning strategy is the effective pruning strategy we proposed at the certain pruning percentage. This strategy performs well on code generation task mainly because of the small granularity of pruning and the contextual consideration related to the attention score which generated from CodeBERT.

Comparing Class-level code generation and Method-level code generation. As depicted in Figure 2, both the class-level code generation and the method-level generation performs well especially when the pruning percentage is under 40% for it causes huge damage in prompt semantics. Specifically, the method-level generation and the class-level generation performs best when pruning 20% input prompt tokens on ClassEval benchmark, which $Pass@1$ is 21.4% and 9.2% respectively. Even with the pruning percentage set as high as 40%, the accuracy performance is sustained at 10.4% for class-level generation and 3.4% for method-level generation. However, on the one hand, the improvement at $Pass@k$ in the class-level code generation is not equivalent to an equal improvement in the method-level experimental results. It is because a certain improvement in class-level code tasks means several methods in this class performs well. Therefore a little improvement in class-level code generation task is of great significance which means several methods are improved through certain strategy. It can be concluded that the method-level code generation has a better performance $Pass@k$ than the other one according to the Table 2, 3, 4, 5 does not mean our strategy is more suitable for method-level code generation tasks.

On the other hand, improvements of $Pass@k$ in class-level code generation can lead to higher improvements in method-level code generation tasks, because a class passing all test cases implies that every method within the class successfully passes all corresponding cases in method-level tasks. A 4.6%, 5.2%, and 4.4% increase in the final class-level results contribute to a 7.6%, 8.4%, and 5.8% increase in the final method-level results, respectively. It is the improvements in class-level code generation tasks that really make sense in our code generation tasks which take contextual information into consideration.

Finding 3: A large difference lying between the class-level code generation and the method-level code generation is the complexity in a class including a large number of methods and, most importantly, the context and relation among methods in a class. Improvements in class-level code generation tasks are of significance and can result in the larger improvements in method-level code generation. Through pruning strategy we proposed class-level code generation performs great improvements leading to method-level tasks' improvements either.

6 conclusion

This work proposes a novel attention-guided pruning strategy for tackling challenging class-level token-level code generation, where attention is used to optimise LLMs' input prompts. We report experiments on the challenging ClassEval benchmark code generation tasks, where our attention-guided pruning outperforms random pruning on class-level code generation when pruning less than 40% of tokens from inputs. Taking advantage of the context and focusing on the fine-grained token-level adjustment demonstrates that attention-guided pruning is significantly better than random pruning for both class-level and well-studied method-level code.

Our results illustrate the potential of attention-guided pruning to boost the performance of LLMs. Future work will continue to evaluate the application of this approach under various programming languages and more diverse coding contexts, and develop a reliable system that effectively utilises pruning in delivering code, both human- and machine-sounding, efficiently.

7 Limitations

This paper proposes an attention-guided prompt pruning strategy and demonstrates through experiments its efficacy in enabling large language models (LLMs) to generate desired class-level code with validated accuracy. However, there are several limitations to this study. Firstly, the study employs only the ClassEval benchmark dataset for code generation and evaluation, without applying and testing the proposed strategy on other class-level datasets. Secondly, the evaluation metric used is $Pass@k$, without conducting subjective manual assessments from a programmer's perspective on class-level code generation. Lastly, this work utilizes only GPT-3.5 for class-level code generation, lacking validation of the strategy's generalizability across various state-of-the-art LLMs. These aspects will be potential directions for future research.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. Santacoder: don't reach for the stars! *Preprint*, arXiv:2301.03988.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2016. Neural machine translation by jointly learning to align and translate. *Preprint*, arXiv:1409.0473.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2024. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3).

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. Pangu-coder: Program synthesis with function-level language modeling. *Preprint*, arXiv:2207.11280.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2021. All NLP tasks are generation tasks: A general pretraining framework. *CoRR*, abs/2103.10360.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020a. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020b. Codebert: A pre-trained model for programming and natural languages. *Preprint*, arXiv:2002.08155.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. *Preprint*, arXiv:2204.05999.

Benyamin Ghojogh and Ali Ghodsi. 2020. Attention mechanism, transformers, bert, and gpt: tutorial and survey.

Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023a. Explainable automated debugging via large language model-driven scientific debugging. *Preprint*, arXiv:2304.02195.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023b. Large language models are few-shot testers: Exploring llm-based general bug reproduction. *Preprint*, arXiv:2209.11515.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023c. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323.

Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023a. Improving chatgpt prompt for code generation. *arXiv e-prints*, pages arXiv–2305.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023b. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Gen Lu and Saumya Debray. 2012. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 31–40. IEEE.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Silvano Martello and Paolo Toth. 1987. Algorithms for knapsack problems. *North-Holland Mathematics Studies*, 132:213–257.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *Preprint*, arXiv:2402.06196.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Libo Qin, Qiguang Chen, Xiachong Feng, Yang Wu, Yongheng Zhang, Yinghui Li, Min Li, Wanxiang Che, and Philip S. Yu. 2024. Large language models meet nlp: A survey. *Preprint*, arXiv:2405.12819.

Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21. ACM.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *Preprint*, arXiv:2307.14936.

KR Srinath. 2017. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357.

Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. *Preprint*, arXiv:2011.14934.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can large language models write good property-based tests? *Preprint*, arXiv:2307.04346.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *Preprint*, arXiv:2302.11382.

Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. *Preprint*, arXiv:2212.09420.

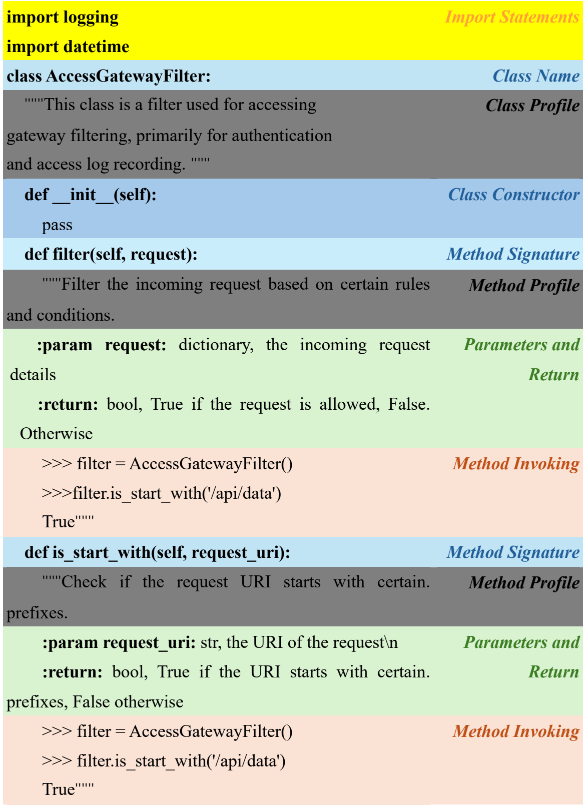
Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering*, 28(2):183–200.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE.

A ClassEval Benchmark and Class Skeleton in ClassEval

A benchmark typically encompasses various NLP tasks, and likewise, a code benchmark requests a natural language description of the task and provides the ground truth code snippet as output. ClassEval benchmark(Du et al., 2023) is one such code benchmark designed specifically for class-level code generation. This benchmark comprises constructed class skeletons, test suites, and canonical solutions, collectively forming the ClassEval class-level code generation benchmark(Du et al., 2023). All 100 class-level code tasks within ClassEval were manually constructed in Python due to its prevalence (Srinath, 2017) incurring nearly 500 person-hours of effort. These class-level code generation tasks encompass practical programming scenarios prevalent in industry and are derived from three sources. Firstly, they draw upon the precedent of code tasks from previously proposed benchmarks such as HumanEval(Luo et al., 2023) and MBPP(Austin et al., 2021). Secondly, they leverage the Python Package Index (PyPI), which houses a vast array of Python development packages that can be used to design various code tasks manually. Thirdly, they are shaped by the insights of experienced programmers with 2-8 years of Python development experience.

To streamline the construction of test suites, ClassEval adopts the widely-used unittest framework of Python along with diverse assertion APIs. For class-level benchmarking, ClassEval test cases encompass all methods within a class, ensuring that each method is invoked at least once during testing. A notable feature of the ClassEval benchmark is the design of the class skeleton format, as illustrated in Figure 4. The manual skeleton structure has been carefully crafted based on the consensus of experienced authors, adhering to four key principles: dependency, class constructor, method functionality, and method parameter and return value.



```
import logging
import datetime
class AccessGatewayFilter:
    """This class is a filter used for accessing
gateway filtering, primarily for authentication
and access log recording. """
    def __init__(self):
        pass
    def filter(self, request):
        """Filter the incoming request based on certain rules
and conditions.
        :param request: dictionary, the incoming request
 details
        :return: bool, True if the request is allowed, False.
  Otherwise
        >>> filter = AccessGatewayFilter()
        >>>filter.is_start_with('/api/data')
        True"""
    def is_start_with(self, request_uri):
        """Check if the request URI starts with certain.
prefixes.
        :param request_uri: str, the URI of the request\n
        :return: bool, True if the URI starts with certain.
prefixes, False otherwise
        >>> filter = AccessGatewayFilter()
        >>> filter.is_start_with('/api/data')
        True"""
```

Figure 4: An Example of Class Skeleton in ClassEval

B Prompt Design

Then we describe the designation of the original prompt without pruning in the class-level code generation task. The prompt can be divided into three parts as follows:

- **Instruction**
  The instruction part is the core of the whole input prompt, which contains the name and skeleton of the target code. Here is the construction of the instruction part with the name and skeleton of the ground truth class.

  > Please complete the class ${Class Name} in the subsequent code. ${Class Skeleton}

- **Examples**
  As mentioned in Section 2, we feed examples into the input prompt as many as possible in order to lead LLMs to better understanding of code generation. In this way, the Examples part is necessarily contained with the longest length of the three parts. Our pruning strategy is conducted in this part to shorten the whole length of the prompt. The following is the

sample of an example, which the whole examples' part consists of several diverse examples from ClassEval benchmark dataset. Certainly, each example contains instruction part, similar as presented before, and solution part from ClassEval benchmark dataset.

> Example n:
> Please complete the class ${Example Class Name} in the following code.
> ${Example Class Skeleton} *//To be pruned.*
> The solution is:
> ${Example Class Solution} *//To be pruned.*

- **Testing**
  The testing part is the final prompt fed into LLM with the required class skeleton and examples of code generation. The ultimate structure of the input prompt is shown as follows:

> #You are an expert programmer.Here are some coding examples, you can learn from these examples:
> Example 1:
> Please complete the class xxx in the following code.
> ***class sample1:***
> ***def m1(p1,. . .):***
> . . .
> ***def m2(t1,. . .):***
> . . .
>
> . . .
>
> #Below is an instruction that describes a task. Write a response that appropriately completes the request.
> ### Instruction:
> ***class xxx:*** *// Required class*
> ***def m1(p1,. . .):***
> *// Method Introduction*
> ***def m2(t1,. . .):***
> *// Method Introduction*
> ### Response:

C Attention-guided Pruning Visualization

As presented in the paper, attention-guided pruning performs effectively in class-level code generation. Below, we visually demonstrate the effect of the attention-guided pruning strategy on an initial class within the ClassEval benchmark dataset. The two images of Figure 5a and 5c show the original code of the class examples we selected from the ClassEval dataset. The two images Figure 5b and 5d display the code after applying the aforementioned attention-guided pruning strategy. We set the pruning percentage to 20% (our previous results indicate that pruning 20% of the code tokens achieves the best performance with Attention-guided pruning). As shown in Figure 5, the attention-guided pruning strategy indeed removes some tokens, but essentially retains the structure of the original class. Pruned by our proposed strategy, the prompt still includes the class name, method names, and method signatures. Specifically, some meaningless tokens like "_" in methods name, some tokens can be easily inferred from context like "item" and other unnecessary symbols like "*" are pruned according to Figure 5 in red lines. As we can see, the pruned class remains almost complete semantics which are "to manage shopping items, their prices, quantities, and allows to for add, remove, view items, and calculate the total price" and "to calculate the area of different shapes, including circle, sphere, cylinder, sector and annulus" respectively in Figure 5b and 5d. Also it remains the main structure of the original class. This is mainly because the attention-guided pruning strategy is based on the LLM's understanding of the text, specifically the attention mechanism, to perform the pruning.

D Experiment Results on Different Pruning Proportion

This section presents, Table 3,4,5, the other results of our experiments of both the class-level generation and the method-level generation which the pruning proportions are 10%, 30% and 40%.

Table 3: *Pass@k* of Code Generation by Pruning Strategy with 10% Reduction

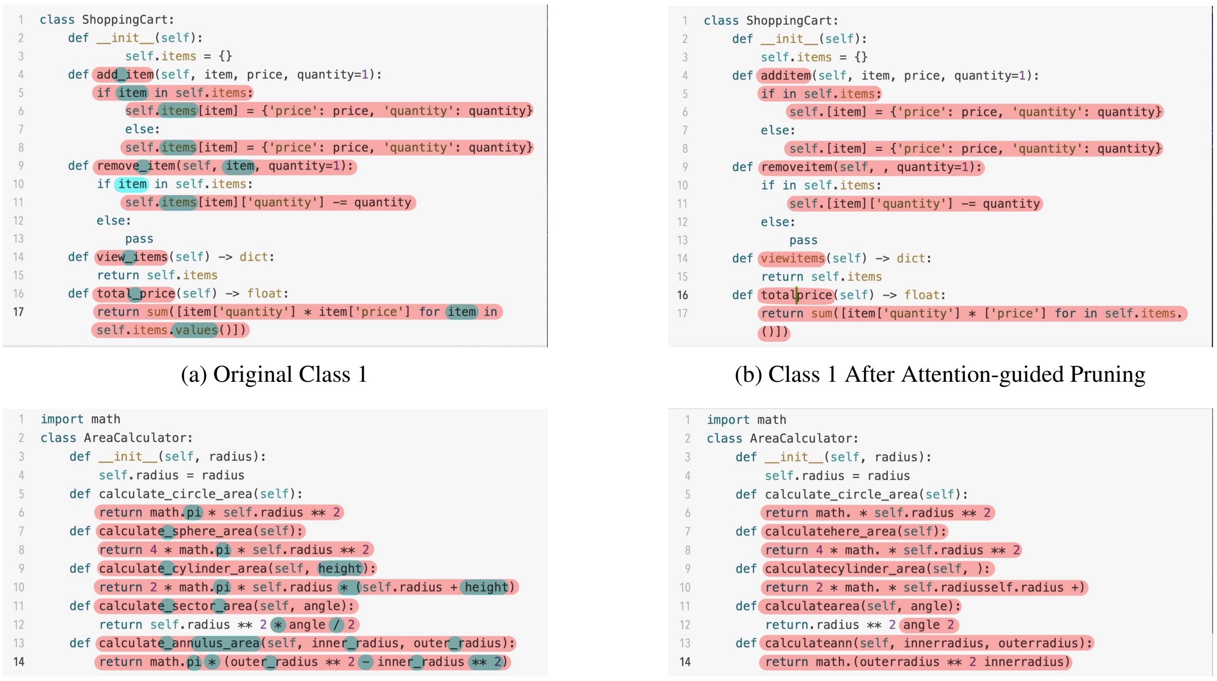
Experiments	Class-level All Success/Partial Success			Method-level All Success/Partial Success		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Few-shot Without Pruning	4%/7.0%	9%/16.5%	10%/20%	13.0%/15.9%	29.4%/36.2%	33.0%/41.2%
Random Pruning	5.6%/11.2%	13.5%/27.0%	17%/34%	15.8%/21.0%	38.4%/50.6%	49.0%/63.7%
Attention-guided Pruning	**8.6%/16.7%**	**21.4%/39.2%**	**28.6%/47.3%**	**20.6%/26.5%**	**48.9%/62.3%**	**60.1%/74.9%**

Table 4: *Pass@k* of Code Generation by Pruning Strategy with 30% Reduction

Experiments	Class-level All Success/Partial Success			Method-level All Success/Partial Success		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Few-shot Without Pruning	4%/7.0%	9%/16.5%	10%/20%	13.0%/15.9%	29.4%/36.2%	33.0%/41.2%
Random Pruning	4.2%/7.6%	10.2%/18.3%	13%/23%	13.3%/18.9%	31.7%/44.9%	39.0%/55.0%
Attention-guided Pruning	**8.4%/15.6%**	**20.4%/37.5%**	**26%/47%**	**18.8%/23.3%**	**44.9%/55.6%**	**55.4%/68.5%**

Table 5: *Pass@k* of Code Generation by Pruning Strategy with 40% Reduction

Experiments	Class-level All Success/Partial Success			Method-level All Success/Partial Success		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Few-shot Without Pruning	**4%/7.0%**	**9%/16.5%**	**10%/20%**	**13.0%/15.9%**	**29.4%/36.2%**	**33.0%/41.2%**
Random Pruning	2%/4.6%	5.4%/11.4%	8%/15%	11.1%/16.0%	27.43%/38.2%	35.3%/47.4%
Attention-guided Pruning	3.4%/7.0%	8.7%/17.4%	12%/23%	10.4%/14.9%	25.9%/36.3%	34.1%/46.6%



```
class ShoppingCart:
    def __init__(self):
            self.items = {}
    def add_item(self, item, price, quantity=1):
        if item in self.items:
            self.items[item] = {'price': price, 'quantity': quantity}
            else:
            self.items[item] = {'price': price, 'quantity': quantity}
    def remove_item(self, item, quantity=1):
        if item in self.items:
            self.items[item]['quantity'] -= quantity
        else:
            pass
    def view_items(self) -> dict:
        return self.items
    def total_price(self) -> float:
        return sum([item['quantity'] * item['price'] for item in
        self.items.values()])
```

(a) Original Class 1

```
class ShoppingCart:
    def __init__(self):
        self.items = {}
    def additem(self, item, price, quantity=1):
        if in self.items:
            self.[item] = {'price': price, 'quantity': quantity}
        else:
            self.[item] = {'price': price, 'quantity': quantity}
    def removeitem(self, , quantity=1):
        if in self.items:
            self.[item]['quantity'] -= quantity
        else:
            pass
    def viewitems(self) -> dict:
        return self.items
    def totalprice(self) -> float:
        return sum([item['quantity'] * ['price'] for in self.items.
        ()])
```

(b) Class 1 After Attention-guided Pruning

```
import math
class AreaCalculator:
    def __init__(self, radius):
        self.radius = radius
    def calculate_circle_area(self):
        return math.pi * self.radius ** 2
    def calculate_sphere_area(self):
        return 4 * math.pi * self.radius ** 2
    def calculate_cylinder_area(self, height):
        return 2 * math.pi * self.radius * (self.radius + height)
    def calculate_sector_area(self, angle):
        return self.radius ** 2 * angle / 2
    def calculate_annulus_area(self, inner_radius, outer_radius):
        return math.pi * (outer_radius ** 2 - inner_radius ** 2)
```

(c) Original Class 2

```
import math
class AreaCalculator:
    def __init__(self, radius):
        self.radius = radius
    def calculate_circle_area(self):
        return math. * self.radius ** 2
    def calculatehere_area(self):
        return 4 * math. * self.radius ** 2
    def calculatecylinder_area(self, ):
        return 2 * math. * self.radiusself.radius +)
    def calculatearea(self, angle):
        return.radius ** 2 angle 2
    def calculateann(self, innerradius, outerradius):
        return math.(outerradius ** 2 innerradius)
```

(d) Class 2 After Attention-guided Pruning

Figure 5: Two pairs of examples of original class(left) and pruned class(right) by attention-guided pruning