MLAlgo-Bench: Can Machines Implement Machine Learning Algorithms?

Yunfei Wang[1,2]*, **Yeqin Zhang**[1,2,*], **Yuyang Wu**[1,2],
Liang Lu[1,2], **Phi Le Nguyen**[3], **Xiaoliang Wang**[1], **Cam-Tu Nguyen**[1,2]†
[1] State Key Laboratory for Novel Software Technology, Nanjing University, China
[2] School of Artificial Intelligence, Nanjing University, China
[3] Institute for AI Innovation and Societal Impact, Hanoi University of Science and Technology
{woilfwang, zhangyeqin, wuyuyang, luli}@smail.nju.edu.cn
lenp@soict.hust.edu.vn, {waxili, ncamtu}@nju.edu.cn

Abstract

As machine learning (ML) application continues to expand across diverse fields, there is a rising demand for ML code generation. In this paper, we aim at a critical research question: *Can machines autonomously generate ML code for sophisticated, human-designed algorithms or solutions?* To answer this question, we introduce a novel benchmark, MLAlgo-Bench, which includes two challenging tasks: 1) Generating code for ML algorithms including both traditional ML and modern deep learning-based methods, and 2) Giving humans solution sketches, writing ML code for solving practical tasks in Kaggle competitions. This benchmark is unique in its focus on the challenges of interpreting intricate human instructions and producing multi-step, high-complexity code, offering a rigorous test for current Large Language Model (LLM) capabilities. We introduce an automatic evaluation framework with comprehensive metrics such as task pass rate, relative performance metric, and time overhead. Currently, the top-performing models (Claude3.5-Sonnet) achieve a 48.8% task completion rate on realizing machine learning algorithms, and a 21.6% rate for completing Kaggle competitions. Further analysis suggests substantial room for improvement. The data and code are available at https://github.com/WoilfWang/MLAlgo-Bench-Main.

1 Introduction

Recent advancements in large language models (LLMs) (Touvron et al., 2023; OpenAI, 2023; Team et al., 2023) have demonstrated remarkable performance across various domains. In this context, the code generation capabilities of LLMs (Chen et al., 2021; Roziere et al., 2023; Team et al., 2024) have attracted significant attention thanks to their potential to greatly enhance the efficiency of professionals in everyday tasks. As a result, numerous benchmarks (Chen et al., 2021; Jimenez et al., 2024; Lai et al., 2023; Zan et al., 2022a,b) have been introduced to assess the performance of LLMs from different aspects of software engineering.

Recently, new interest has emerged in benchmarking the capabilities of LLM agents to perform ML tasks (Huang et al., 2024; Chan et al., 2024). This area of research holds great promise, as ML is closely linked to the rapidly advancing field of artificial intelligence. While recent ML benchmarks (Huang et al., 2024; Chan et al., 2024; Nathani et al., 2025) provide valuable insights, they primarily evaluate LLMs' ability to solve ML tasks autonomously without human-designed solutions or algorithms. In this paper, we contend that the development of effective ML agents fundamentally depends on their capacity to follow sophisticated, human-designed solutions. Mastery of this ability opens new opportunities for LLMs to assist human practitioners and researchers in translating high-level ideas—such as those found in ML research papers—into practical implementations, significantly reducing the effort required for experimentation.

To comprehensively evaluate the capabilities of LLMs in the implementation of ML methods, we propose a benchmark called MLAlgo-Bench. Our benchmark consists of two settings. Setting 1 is designed to assess whether LLMs can successfully implement ML modules based on detailed algorithm descriptions. It includes a total of 121 tasks, covering not only traditional machine learning algorithms like Random Forest and SVM but also a wide range of deep learning algorithms, such as Transformer modules and their various variants. Setting 2 focuses on evaluating whether LLMs can successfully complete machine learning competition by generating the full workflow code based on a task description and the corresponding solution ideas, which are provided by human top competitors. For each task in our benchmark, we provide

*Equal contribution.
†Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 14298–14329
November 4-9, 2025

Benchmark	Time	Source	Avg Input Lens	Num	Metrics
CoNaLa (Yin et al., 2018)	2018	Stack Overflow	9.6	2879	BLEU
HumanEval (Chen et al., 2021)	2021	-	131.3	164	Pass@k
MBPP (Austin et al., 2021)	2021	-	16.1	974	Pass Rate
PandasEval (Zan et al., 2022c)	2022	StackOverflow	68.3	101	Pass@k
NumpyEval (Zan et al., 2022c)	2022	StackOverflow	69.1	101	Pass@k
DS1000 (Lai et al., 2023)	2022	StackOverflow	282.4	1000	Pass@k + SF-contrains
SWE-Bench (Jimenez et al., 2024)	2023	Github	480.9	2294	Pass Rate + Apply
ClassEval (Du et al., 2024)	2024	Manual	123.7	100	Pass@k
ML-Bench (Tang et al., 2023)	2023	Github	781.6	9641	Pass Rate/ Pass@k
MLAgentBench (Huang et al., 2024)	2023	Kaggle and others	108.7	13	Pass Rate
MLGym-Bench (Nathani et al., 2025)	2025	Canonical tasks	387.5	13	AUP Scores
MLE-Bench (Chan et al., 2024)	2024	Kaggle	2106.7	76	Pass@k / Kaggle Ranks
MLAlgo-Bench (ours)	2024	scikit-learn/ LabML-Github/ Kaggle	1449.8	218	Pass Rate/ Performance/ Time Overhead / Instruction-Following

Table 1: Avg Input Lens denotes the average token length of all task prompts. In the table, the pass@k evaluation metric is proposed by (Kulal et al., 2019). SF-contrains is the evaluation metrics proposed by (Lai et al., 2023). The pass rate represents the proportion of successfully solved tasks. AUP Score is introduced by (Roberts et al., 2023). For ML-Bench, we only calculate the length of instruction + oracle.

automated evaluation scripts and introduce novel metrics tailored to ML method implementation.

Compared to existing code generation benchmarks (Chen et al., 2021; Jimenez et al., 2024; Lai et al., 2023; Zan et al., 2022a), MLAlgo-Bench presents greater challenges for LLMs. First, previous benchmarks typically involve generating only short code snippets (function-level code) such as HumanEval (Liu et al., 2024b), ML-Bench (Tang et al., 2023). In contrast, each task in our benchmark requires module-level implementation, with more lengthy output code. Second, the tasks in our benchmark generally involve more complex code instructions. For Setting 1, the instructions often include intricate workflows and mathematical formulas. In Setting 2, our instructions encompass detailed descriptions of machine learning tasks, datasets, and solutions – a composite ML method with feature engineering. In other words, *MLAlgo-Bench places higher demands on their ability to understand and follow technical instructions*. Please refer to Table 1 for a summary of MLAlgo-Bench and other benchmarks.

We evaluate the code generation capabilities of several LLMs on our benchmark. In Setting 1, the best-performing LLM is Claude-3.5-Sonnet, achieving a pass rate of 48.8%. In Setting 2, the top performer is also Claude-3.5-Sonnet, with a pass rate of 21.6%. We provide a detailed analysis of the common error types made by LLMs, and conduct an in-depth investigation into several factors: 1) The impact of methodology specification on ML method implementation; 2) A detailed evaluation and analysis was conducted on the instruction-following capabilities of LLMs; 3) The impact of an agentic framework on the ability to correctly implement a given solution. In summary, the paper makes the following contributions:

- We propose a *new benchmark MLAlgo-Bench* for evaluating the ability to realize ML solutions. Our benchmark is characterized by long input and long output (module-level code), with high demand for instruction following, as well as math, and logic capabilities.
- We provide an evaluation framework and evaluate the performance of contemporary LLMs and recognize common error types such as math errors and data understanding.
- Further analysis also reveals the limitations of current LLM in instruction following, and in the capability to call external ML libraries.

2 MLAlgo-Bench Benchmark

2.1 Problem Formalization

Given a detailed description of a ML algorithm or problem with a solution specified by S, and implementation instructions I, an ML research assistant $\mathcal{A}$ is tasked with generating ML code C. The code C must strictly follow I and accurately implement S. The instruction I defines the necessary inputs and outputs, assuming $\mathcal{A}$ is familiar with standard ML and numerical libraries. The specification S outlines the problem, describes the methodology,

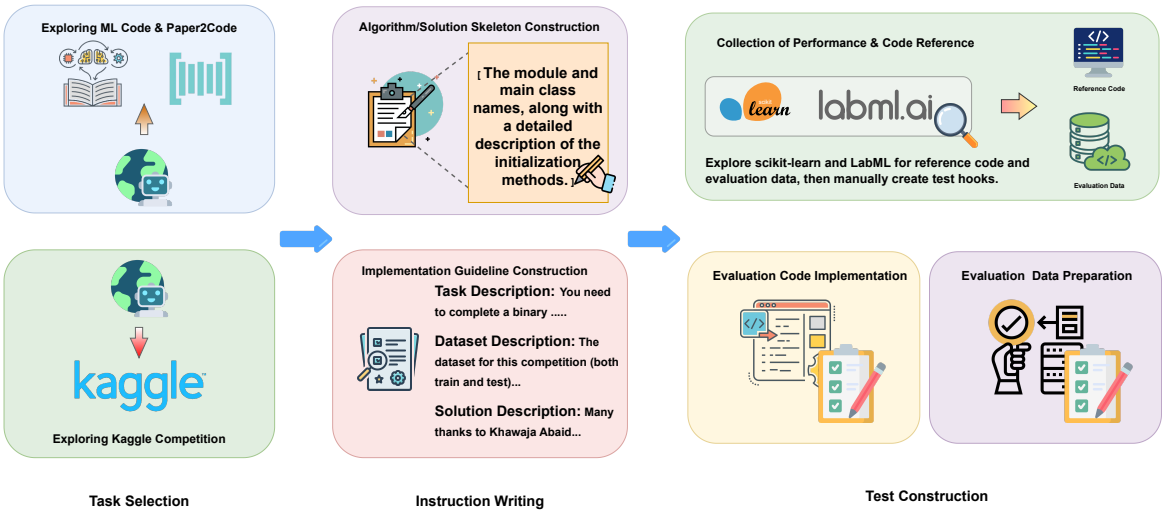


Figure 1: The data construction pipeline of MLAlgo-Bench

and, depending on the task, may include details of the evaluation dataset.

2.2 Dataset Preparation and Construction

To achieve these objectives, we design our benchmarks based on several key principles: 1) *Diversity*: The dataset should include a variety of ML algorithms and methods; 2) *Detailed and self-contained instructions*: Since the focus is on methodology implementation, we assume that the environment is properly set up for running the LLMs implementation. This is different from tasks ML-Bench (Tang et al., 2023), which require setting up ML Github environment; 3) *Clear evaluation framework*: The goal is not just to produce runnable code, but to ensure the code executes correctly and achieves reasonable performance. With such principles, we construct our dataset according to the pipeline in Figure 1 with details in the following.

2.2.1 Step I: Task Selection

For task collection, we initially explore ML Githubs and Paper2code[1], aiming to isolate the method in papers and corresponding Github source code for benchmarking. Most papers are too abstract and lack implementation details; our paper2code experiments show LLMs struggle to understand them, leading to no successful implementations. We hence switch to simpler tasks that belong to two types: *Setting 1* that requires LLMs to implement well-described Machine Learning algorithms; and *Setting 2* that allows LLMs to use ML libraries but implement human-designed solutions to finish a ML or data science task.

Setting 1: ML Algorithm Implementation We manually explore scikit-learn and LabML (Varuna Jayasiri, 2020) for a wide range of ML algorithms. Scikit-learn includes implementations of traditional methods, ranging from simpler models like KNN to more complex ones such as Support Vector Machines (SVM), Random Forests, and Gradient Boosting Decision Trees. In contrast, LabML is an educational platform, that offers deep learning algorithm implementations with detailed explanations. The algorithms include various Transformer variants (Vaswani et al., 2017; Dai et al., 2019; Fedus et al., 2022; Schlag et al., 2021), Diffusion models (Ho et al., 2020), and several deep learning optimizers (Reddi et al., 2018). The explanations in LabML are simpler versions of those from the corresponding papers, providing clearer and more accessible technical descriptions. We select a diverse set of tasks that are sufficiently intricate for comprehensive evaluation, but not so complex that hinders deep analysis.

Setting 2: ML Solution Implementation Setting 2 is designed to evaluate whether LLMs can effectively implement human-designed solutions for ML competitions. We explore Kaggle, a leading platform for data science and machine learning competitions, known for its high-quality and challenging tasks. From a pool of 600 Kaggle competition tasks, we select only those that provide high-level solution sketches from the top-5 competitors in the leaderboard. These solutions include detailed descriptions of feature engineering

[1]https://paperswithcode.com/

methods and/or ML techniques, typically involving an ensemble of well-known ML methods like XGBoost, Random Forest, etc. To reduce storage and evaluation costs, we further filter the tasks to include only those with smaller datasets. Notably, we carefully reviewed the license statements of all task data and confirmed that they are free to share.

2.2.2 Step II: Instruction Writing

For each task, whether in Setting 1 or Setting 2, the research team — comprising graduate-level students and researchers specializing in Artificial Intelligence — writes instructions for the task specification S and the implementation details I.

Setting 1 The task specification S provides a detailed description of the selected algorithms, which are rewritten based on scikit-learn documentation and LabML explanations. The rewriting is to ensure that the specifications are clear and standardized, while also mitigating potential data leakage issues. For the implementation details I, we explicitly define the module and main class names, along with a detailed description of the initialization methods. These specifications are necessary to ensure that the generated code can be seamlessly integrated into our evaluation framework. In addition to the initialization method, we specify key functions that are needed by the evaluation code (Step III). For instance, an implementation of a deep learning model needs the forward and backward functions, which are called by the evaluation code for training and testing the output code.

After completing the initial annotation of the instructions for each task in Setting 1, a senior researcher will review each instruction to ensure its quality. Our review focused on the following aspects: 1) Checking for any errors in the description of algorithmic modules and ensuring that the description thoroughly explains the principles and corresponding steps of the algorithm; 2) Verifying whether the code format requirements in the instructions align with the format used in the evaluation framework to avoid any formatting mismatches during the evaluation process. For instructions that don't meet the standards, we will make necessary revisions to ensure their quality.

Setting 2 The specification S includes the following components: *task description*, *dataset description*, and *solution description*. The task and dataset descriptions are from the competition details provided by Kaggle. For the solution description, we gather high-level ideas of teams ranked within top-5 on the private leaderboard. To ensure the quality of the reference solutions, we have cleaned and rewritten the collected solutions. We require a qualified solution to include a complete workflow for solving machine learning tasks, including data processing and a detailed description of the methodology. Low-quality solutions, such as those with overly concise method descriptions, are directly discarded. Additionally, many solutions contain images or external links; after careful review, we convert them into detailed descriptions in natural language and incorporate them into the instructions. Regarding implementation details I, we provide guidelines on permissible ML libraries and specify the required outputs, typically the labels for a specific test set (see Step III).

2.2.3 Step III: Test Construction

To evaluate ML algorithms or solutions, it is crucial to define two key components: the evaluation datasets and the code/performance reference.

Setting 1 We explore scikit-learn and LabML for reference code and evaluation data. The reference code can be run on the evaluation dataset for performance reference. We then manually create test hooks (evaluation code script) that call the (LLM) generated code for training and testing on the relevant data. The evaluation code script can be run successfully only if the implementation strictly follows the implementation guideline I. Although LLMs may encounter scikit-learn or LabML during pretraining, the instruction rewriting process helps obfuscate the link between the instruction and the original implementation.

Setting 2 The dataset associated with each competition is used for evaluation. However, as the test sets are hidden, we split the public training set to create a new train/test set for our benchmark. As previously mentioned, we obtain the solution sketch from top-5 in the leaderboard. If a solution is associated with an implementation code, we run it to obtain a performance reference on our test set. If no implementation is provided, we use the leaderboard score of the solution as the (human) performance reference. Although there might be a distribution shift between the leaderboard testset and our new test set, the performance reference is sufficient to measure a task's difficulty and approximate the performance score range for the corresponding solutions. Similar to Setting 1, we also

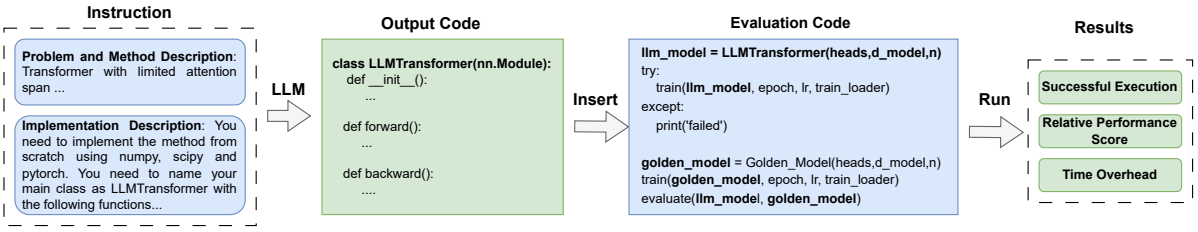


Figure 2: The complete evaluation process in MLAlgo-Bench.

Settings	#Tasks	Input Len	#Avg Test
Setting 1	121 (66 + 55)	931±442	1381
Setting 2	97	2097±802	708,615

Table 2: Data characteristics of MLAlgo-Bench. Note that Setting 1 includes 66 tasks from scikit-learn and 55 Deep learning related tasks from LabML. #Avg Test indicates the average number of testing data samples.

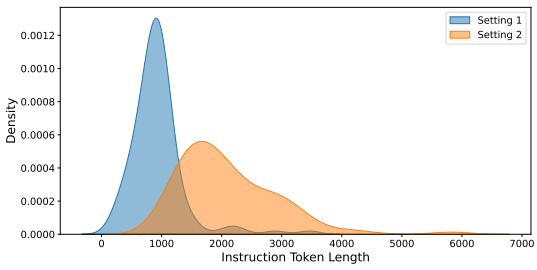


Figure 3: MLAlgo-Bench: input length distribution.

write the evaluation code scripts for the evaluation framework described in Section 2.4.

2.3 Statistical Analysis

Table 2 and Figure 3 show data characteristics of MLAlgo-Bench. It is observable that Setting 2 has a much longer input length compared to Setting 1 due to the lengthy problem and data descriptions. It is also noted that, although the number of tasks is moderate, it is still comparable to those in other benchmarks. To put this into perspective, the number of tasks in MLAlgo-Bench surpasses that of the well-known HumanEval dataset (see Table 1).

2.4 Evaluation Framework

The complete evaluation pipeline is illustrated in Figure 2. First, given an instruction (S + I), an LLM is invoked to generate an output code, which is then automatically integrated into an evaluation code script for testing. The evaluation code script loads the specified dataset and executes the output code in a sandbox environment (e.g., a Docker container needs to be set up). The execution process verifies whether the code runs successfully, computes the task pass rate, and compares the performance against human performance reference.

Overall Pass Rate The overall pass rate is defined as the proportion of LLM code that can be executed successfully and achieves performance, after training and testing, that is comparable to the performance reference.

Relative Performance Score (RPS) We define a Relative Performance Score (ΔScore) to evaluate the performance of executable LLM implementations, where performance is measured relative to a task-specific reference. The reference performance is obtained either from executing the official reference code (Setting 1) or from the solution sketch's leaderboard score on Kaggle (Setting 2). To ensure comparability across tasks with different metrics (e.g., MSE, F1, ROC-AUC), we apply *min-max normalization* to both the LLM-generated scores and the reference scores, scaling them to the $[0, 1]$ interval. Note that for metrics where lower values indicate better performance (e.g., MSE), scores are first inverted prior to normalization. Normalization strategies are adapted per setting:

- **Setting 1**: The maximum and minimum values for each task are defined based on theoretical performance bounds. For instance, in an n-class classification task, the maximum is 1.0 (perfect accuracy), while the minimum is set to the accuracy of random guessing, $\frac{1}{n}$.
- **Setting 2**: The maximum is set to the top leaderboard score for the task, and the minimum is the score at the bottom 5% of the leaderboard rankings.

The formula used to compute the relative performance score ΔScore is as follows:

$$nScore_m = \frac{Score_m - min_m}{max_m - min_m} \quad (1)$$

$$\Delta Score = \frac{1}{|\Omega|} \sum_{m \in \Omega} \frac{nScore_m^{LLM}}{nScore_m^{refer}} \quad (2)$$

	LLM	Setting 1				Setting 2		
		ΔScore	ΔTime	Pass Rate (%)	EScore	ΔScore	Pass Rate (%)	EScore
Closed Source LLM	GPT-4o-mini	0.951 (0.136)	0.943(3.91)	27.3	0.260	0.950(0.051)	10.3	0.098
	GPT-4o	0.939 (0.22)	1.16 (3.71)	38.0	0.357	0.966 (0.028)	16.4	0.158
	Claude-3.5-Sonnet	**1.02 (0.36)**	0.733 (3.20)	**48.8**	0.485	0.812(0.168)	**21.6**	0.175
Open Source LLM	Qwen-2.5-72B	0.952 (0.171)	1.14 (3.47)	34.7	0.325	0.946 (0.039)	7.2	0.068
	LLaMA-3.1-405B	0.967 (0.073)	0.852 (3.55)	25.6	0.250	0.952 (0.027)	5.2	0.050
	Qwen-2.5-7B	0.977(0.089)	0.621(3.86)	18.2	0.178	**0.998 (0.016)**	4.1	0.041
	Qwen-Coder-2.5-7B	0.967 (0.129)	**0.076 (4.61)**	19.0	0.184	0.940 (0.040)	4.1	0.039
	LLaMA-3.1-8B	0.745 (0.290)	0.131 (4.38)	14.0	0.104	\	\	\

Table 3: The main experimental results of different LLMs on our benchmark. The numbers in parentheses represent the standard deviation. We do not report the results of llaMA-8B in Setting 2 due to its weaker instruction-following abilities and performance.

where Ω denotes the set of *passed* LLM code, $nScore_m^{LLM}$ and $nScore_m^{golden}$ represent the performance score of code m generated by LLM and the performance reference score min-max normalized by using Eq 1. min_m and max_m denote the minimum and maximum scores, respectively, for a given task m.

Relative Time Overhead (RTS) is used to assess the efficiency of executable code generated by LLMs. In some cases, the time overhead of LLM-generated code is significantly higher than that of the reference code. To mitigate the impact of these outliers on the average time overhead, we apply a log transformation to the results:

$$\Delta Time = \frac{1}{|\Omega|} \sum_{m \in \Omega} \log \frac{Time_m^{LLMs}}{Time_m^{golden}} \quad (3)$$

where Ω denotes the set of algorithms generated by LLMs that can be successfully executed, $Time_m^{LLMs}$ and $Time_m^{golden}$ represent the time cost of algorithm m generated by LLMs and golden solution's time cost, respectively. It is noteworthy that if LLM code is more efficient than the referer-ence code, the RTS is smaller than 0.

Effective Score (EScore) EScore provides a unified metric that combines the pass rate and the Relative Performance Score (RPS), which is computed only for passed tasks. Essentially, Escore is calculated by assigning a score of zero to all failed tasks. The formula for EScore is as follows:

$$EScore = \Delta Score \times \text{pass-rate} \quad (4)$$

3 Experiments

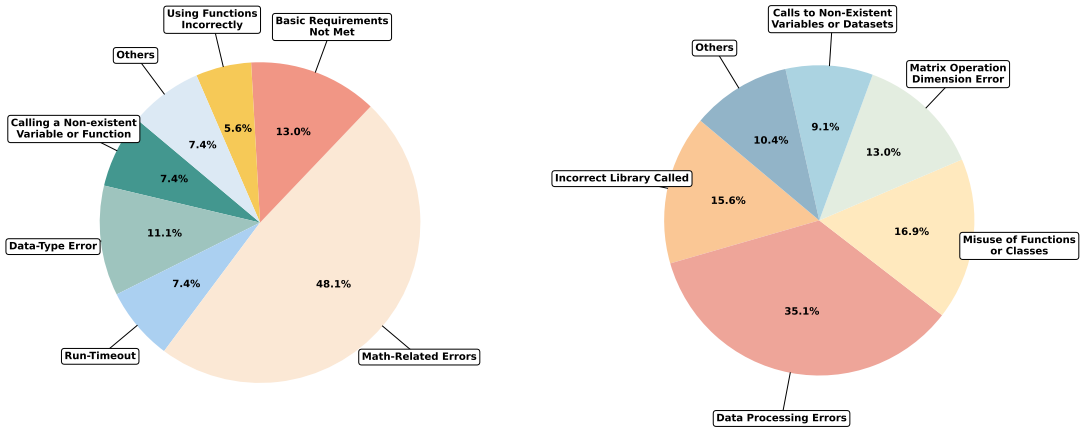
We evaluate three closed-sourced LLMs (GPT-4o-mini, GPT-4o, Claude-3.5-Sonnet) and five open-sourced LLMs (Qwen-2.5-72B, LLaMA-3.1-405B, Qwen-2.5-7B, Qwen-2.5-7B-coder, LLaMA-3.1-8B). The inclusion of such LLMs is decided based on their the instruction following capabilities and the coding capabilities tested on the general benchmarks. The decoding hyperparameters for these LLMs are all set to temperature=0.2 and top_p=0.9.

The evaluation framework invokes the evaluation script to run LLM-generated code as shown in Figure 2.4. This evaluation stage is conducted in a uniform environment, a single A100-40GB GPU, with the Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz and 503 GiB of memory.

3.1 Quantitative Results

Claude 3.5-Sonnet achieves the best overall performance both in pass-rate and EScore across Setting 1 and Setting 2. Among open-source LLMs, Qwen-2.5-72B demonstrates the strongest performance, even outperforming GPT-4o-mini on Setting 1—highlighting its potential as a strong alternative for future research. The RPS (ΔScore) and RTS (ΔTime) results indicate that even when LLMs generate successful implementations, their code still underperforms compared to human-written reference solutions. Notably, a high pass rate does not always correlate with a higher RPS (ΔScore). This is especially evident in Setting 2, where Claude 3.5-Sonnet achieves the highest pass rate but the lowest ΔScore. Two main factors contribute to this discrepancy:

- **Task Difficulty**: For more challenging tasks (i.e., those where weaker LLMs struggle to generate runnable code), there is a greater likelihood of producing suboptimal implementations that pass basic checks but underperform relative to the reference.
- **Shortcut Solutions**: LLMs may achieve a



(a) Setting 1: ML Algorithms (b) Setting 2: ML Solutions

Figure 4: Common error types of Qwen-2.5-72B

high pass rate by opting for simpler implementations that ignore the intended method descriptions (Setting 1) or provided solutions (Setting 2). While these solutions may run successfully, they tend to yield lower scores when compared against the performance of more sophisticated, reference-aligned solutions.

3.2 Error Analysis

The most common errors of Qwen-2.5-72B are shown in Figure 4 (See the Appendix for errors of GPT-4o-mini). By analyzing the results of LLMs in Setting 1, we find that current LLMs are more capable of implementing relatively simple machine learning algorithms, such as logistic regression, decision trees, or basic neural networks. For more complex algorithms, such as gradient boosting trees, SVMs, Transformer variants, and deep learning optimization techniques, LLMs often encounter implementation errors. In Setting 2, our analysis also shows that LLMs perform well when directly applying machine learning algorithms. They can utilize open-source libraries like XGBoost, LightGBM, and scikit-learn with minimal errors. However, LLMs frequently struggle with dataset processing. The common types of errors made by LLMs are demonstrated in detail in Figure 4.

3.3 Instruction-Following Ability

This section evaluates whether the code generated by LLMs faithfully follows the descriptions provided in the algorithm modules or reference solutions. Evaluation scores within [4–5] indicate implementations that closely adhere to most steps of the reference, whereas the range of [0–2] means that the code significantly deviates from or fails to follow the instructions. The detailed evaluation criteria are presented in Table 12 and Table 11. We employ both LLM-as-judge and human evaluation for this task. For LLM-as-judge, we select two top-performing models—GPT-4o and Claude-3.5-Sonnet—to act as evaluation experts. The final scores are obtained by averaging the results from both models. To enhance reliability, each model is prompted to provide a justification for its assigned score. For human evaluation, we rely on three graduate students who are co-authors of the paper, majoring in Artificial Intelligence. They are qualified to perform this task as they have completed advanced machine learning courses and are familiar with the relevant algorithms. A total of 30 tasks are randomly sampled from each setting (Setting 1 and Setting 2), resulting in 60 tasks for evaluation.

The experimental results are presented in Table 4. Overall, Claude-3.5-Sonnet demonstrates stronger instruction-following capabilities, achieving the highest scores in both LLM-based and human evaluations. Notably, the open-source model Qwen-2.5-72B performs comparably to Claude-3.5-Sonnet and even surpasses it in human evaluation scores, highlighting its potential as a competitive alternative. Additionally, the instruction-following scores for all models are generally lower in Setting 2 compared to Setting 1. This can be attributed to the fact that Setting 2 generally has longer inputs compared to that of Setting 1.

To verify the correlation between LLM-based evaluation and human assessments, we rank correlation (Kendall and Smith, 1939; Seo et al., 2025) and show results presented in Figure 5. The observed agreement scores range from 0.5 to 0.6, and

LLM	LLM Evaluation		Human Evaluation	
	Setting 1	Setting 2	Setting 1	Setting 2
Qwen-2.5-72B	4.45	3.80	4.23	4.15
GPT-4o	4.45	3.59	4.31	3.86
LLaMA-3.1-405B	4.20	3.23	3.97	3.57
Claude-3.5-Sonnet	4.62	3.89	4.48	3.93

Table 4: The instruction following ability of LLM

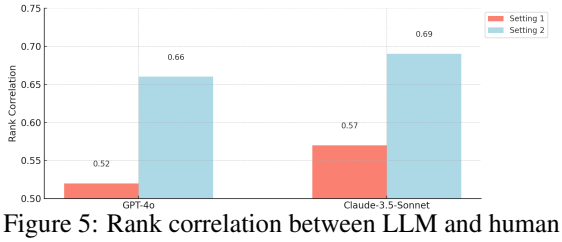


Figure 5: Rank correlation between LLM and human evaluations

Method	ΔScore	Pass Rate(%)
GPT-4o-mini	0.950	10.3
GPT-4o-mini + AIDE	0.931	37.1
GPT-4o	0.966	16.4
GPT-4o + AIDE	0.932	42.3

Table 5: AIDE Framework in MLAlgo-Bench Setting 2

Method	ΔScore	Instruct
GPT-4o-mini	0.950	3.31
GPT-4o-mini + AIDE	0.936	2.18
GPT-4o	0.961	3.62
GPT-4o + AIDE	0.939	2.78

Table 6: RPS ΔScore of tasks that succeeded in both direct generation and AIDE-based generation. **Instruct** denotes the instruction-following score.

approach 0.7 in Setting 2 with Claude-3.5-Sonnet. For the average human score and the average LLM score, the correlation is 0.67 in Setting 1 and 0.72 in Setting 2. To show that this correlation is statistically significant, we apply a two-tailed t-test with the null hypothesis that "there is no correlation between human evaluation and LLM evaluation." With a sample size of $N = 120$ (30 tasks in Setting 2 across 4 LLMs) and correlation $r = 0.72$, the t-statistic is $t = 11.27$. At $\alpha = 0.05$ and $df = 118$, the critical value is ± 1.98[2]. Since the observed t-value exceeds the critical threshold, we reject the null hypothesis at the 95% confidence level. Therefore, the correlation between human evaluation and LLM evaluation is statistically significant. Regarding human evaluation, we also compute human inter-agreement using the rank correlation score. For Setting 1, the rank correlation score of human inter-agreement is 0.54, which reflects a moderate level of agreement (Landis and Koch, 1977), while for Setting 2, the score is 0.76, indicating substantial agreement (Landis and Koch, 1977).

3.4 The Performance of Agentic Framework on MLAlgo-Bench Setting 2

This section evaluates the performance of code agent frameworks on MLAlgo-Bench under Setting 2. We conducted experiments using the open-source code generation agent framework AIDE (Dominik Schmidt and Wu, 2024), with GPT-4o-mini and GPT-4o as the base LLMs. The maximum number of generation attempts is set to five. The results, presented in Table 5, show a substantial improvement in task pass rate when using the AIDE framework. This improvement can be attributed to AIDE's ability to incorporate compiler feedback and enable the LLM to iteratively revise its code based on runtime error logs. However, we also observe a decline in relative performance scores, suggesting that AIDE may encourage shortcut solutions that pass the task criteria without fully adhering to the intended solution strategy.

To further investigate the above observation, we focus on tasks that were successfully completed by both direct generation and the AIDE framework and evaluate their instruction-following ability and relative performance scores within this shared subset. For instruction-following evaluation, we adopt the same LLM-based method described in Section 3.3. Specifically, there are 12 such tasks for GPT-4o and 10 for GPT-4o-mini. The results are summarized in Table 6. It is observable that instruction-following scores exhibit a noticeable decline when using the AIDE framework. Relative performance scores also drop to some extent. This outcome is expected, as the reference solutions provided are among the top 5 on the Kaggle leaderboard. By adhering to these instructions, direct generation can achieve better performance within a narrower exploration space. In contrast, AIDE's trial-and-error approach may yield functional but less aligned or optimal solutions.

[2] https://numiqo.com/tutorial/t-distribution

4 Conclusion

This paper introduces MLAlgo-Bench for evaluating LLMs in the task of implementing machine learning (ML) methods. We provide detailed instruction annotations, an automated evaluation framework, as well as code and performance reference to facilitate future research. Additionally, we evaluate the performance of current mainstream LLMs on MLAlgo-Bench, offering a comprehensive analysis of their capabilities in this domain.

A deeper analysis reveals several areas for improvement, including the enhancement of mathematical understanding, better comprehension of data descriptions, improved function call capabilities, and more robust instruction-following abilities. Addressing these issues would enable LLMs to generate more precise and reliable ML implementations. Such improvements could significantly benefit ML agents (Huang et al., 2024), allowing them to complete ML tasks more effectively with fewer interactive steps.

5 Limitations

The ultimate goal of our work is to develop an AI research assistant capable of translating human-designed ideas into implementations. While this task is still far from fully achievable, our work seeks to set the first step in this process. To enable an AI system to bring human ideas to life, such as those outlined in research papers, it must go beyond just processing text. The system needs to be able to understand and interpret other modalities of information, such as method diagrams, flowcharts, and visual representations of algorithms. These multimodal inputs provide crucial context and enhance the AI's ability to accurately translate abstract concepts into working code. In our current work, we have only explored text as a single modality. In the future, we will focus on investigating how to enable LLM Agents to leverage multimodal information to better help humans realize their ideas.

Ethical Considerations

The collection of all reference code in our benchmark, as well as the construction of evaluation code, is based on currently available open-source libraries and does not involve any copyright infringement. The prompts for each algorithm in the benchmark are designed to ensure they do not contain personal privacy issues, raise ethical concerns, or encourage LLMs to generate harmful content.

References

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

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

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. A scalable and extensible approach to benchmarking nl2code for 18 programming languages.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.

Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988.

Zhengyao Jiang Dominik Schmidt and Yuxiang Wu. 2024. Introducing weco aide.

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.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine Learning*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. Dsbench: How far are data science agents to becoming data science experts? *arXiv preprint arXiv:2409.07703*.

Maurice G Kendall and B Babington Smith. 1939. The problem of m rankings. *The annals of mathematical statistics*, 10:275–287.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

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. *arXiv preprint arXiv:2306.08568*.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, et al. 2025. Mlgym: A new framework and benchmark for advancing ai research agents. *arXiv preprint arXiv:2502.14499*.

OpenAI. 2023. Gpt-4 technical report.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. 2018. On the convergence of adam and beyond. In *International Conference on Learning Representations*.

Nicholas Roberts, Samuel Guo, Cong Xu, Ameet Talwalkar, et al. 2023. Automl decathlon: Diverse tasks, modern methods, and efficiency at scale. In *NeurIPS 2022 Competition Track*, pages 151–170. PMLR.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. 2021. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*.

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

Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, et al. 2023. Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code. *arXiv e-prints*, pages arXiv–2311.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Adithya Narasinghe Lakshith Nishshanke Varuna Jayasiri, Nipun Wijerathne. 2020. labml.ai: A library to organize machine learning experiments.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pages 476–486. IEEE.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2022a. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 277–288.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, et al. 2022b. CERT: continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org.

Daoguang Zan, Bei Chen, Dejian Yang, et al. 2022c. CERT: continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

A Related works

A.1 Large Language Models Meet NLP2Code

The task of generating code from natural language, referred to as NLP2Code (Zan et al., 2023), has been a long-standing challenge in the field of code intelligence. Recently, there has been a growing trend of using large language models (LLMs) for the NLP2Code task. These LLMs can be broadly categorized into two types: *general LLMs* and *Code LLMs* (Zan et al., 2023; Du et al., 2024).

General LLMs, which typically have billions of parameters, are trained on a combination of text and code corpora. Models such as GPT-4 (OpenAI, 2023), LLaMA 3 (Dubey et al., 2024), Qwen (Yang et al., 2024; Qwen Team, 2024), and DeepSeek (Liu et al., 2024a) exhibit remarkable capabilities across a wide range of tasks, including language understanding, reasoning, and code generation. On the other hand, *Code LLMs* are specialized models trained exclusively on large code corpora. Notable examples of *Code LLMs* include CodeBERT (Feng et al., 2020), Code-T5+ (Wang et al., 2023), CodeLLama (Roziere et al., 2023), Pangu-Coder (Shen

et al., 2023; Christopoulou et al., 2022), WizardCoder (Luo et al., 2023), and CodeGeex (Zheng et al., 2023). These models generally perform well on software engineering tasks but may not excel in understanding instructions as general LLMs. Our work primarily focuses on benchmarking general LLMs, as they are better suited to the multi-faceted skillset required for ML method implementation such as mathematical and language understanding, function calling, data specification understanding.

A.2 Benchmarks for Code Generation

Many NLP2Code benchmarks are introduced to assess the ability to generate function-level code (Chen et al., 2021; Austin et al., 2021), multi-language code (Athiwaratkun et al., 2022; Zheng et al., 2023; Cassano et al.), data processing code (Lai et al., 2023; Chandel et al., 2022; Jing et al., 2024), and class-level code (Du et al., 2024). In comparison, MLAlgo-Bench requires a broader set of skills, including data processing capabilities, class-level code generation, mathematical reasoning and ML function calling.

Recently, there has been growing interest in benchmarking the ability of LLMs on machine learning tasks, with notable efforts including MLAgentBench (Huang et al., 2024), MLGYM-Bench (Nathani et al., 2025), ML-Bench (Tang et al., 2023), and MLE-Bench (Chan et al., 2024). MLAgentBench and MLE-Bench focus on agentic frameworks that autonomously generate code from task descriptions, with MLE-Bench covering a broader set of tasks. MLGYM-Bench shares similarities with these benchmarks but additionally provides external tools to support task completion. However, none of these benchmarks evaluate LLMs' ability to implement novel ML methods or translate high-level, human-designed solutions into working code

B Detailed Information of Related ML Benchmarks

B.1 MLAgent-Bench

MLAgent-bench(Huang et al., 2024) primarily assesses whether LLMs can meet task requirements when given relevant tasks, dataset descriptions, and some starter code. MLAgent-bench includes a total of 13 tasks, consisting of 6 Kaggle competitions, several classic classification and regression tasks, as well as two tasks aimed at improving runtime efficiency. None of the tasks require implementing a model from scratch. Instead, they either involve modifying a given algorithm or directly utilizing existing libraries such as sklearn and huggingface for implementation.

Evaluation Metrics

- Success rate: Each task is run 8 times, and the success rate of these 8 solutions is calculated as the success rate for the current task. A successful implementation is defined as achieving a performance improvement of more than 10% compared to the baseline. Finally, the average success rate across all tasks is computed.
- Efficiency: Compare the average number of tokens and time spent by each agent.

B.2 ML-Bench

MLBench (Tang et al., 2023) primarily evaluates whether LLMs can successfully implement the given tasks based on relevant GitHub branches. MLBench consists of two settings: ML-LLM-Bench and ML-Agent-Bench. In ML-LLM-Bench, LLMs are provided with relevant GitHub branches and tasks, and are expected to directly generate the corresponding shell scripts. In the ML-Agent-Bench setting, the paper provides a sandbox interaction environment where, given a task and relevant GitHub branches, LLMs interact continuously with the environment for task completion. Both settings focus on evaluating the ability of LLMs to leverage existing code without involving the implementation of algorithms at the core level. ML-Bench uses Pass Rate and Pass@k as the evaluation metrics, where the later allows K times code generation.

B.3 MLE-Bench

MLE-Bench (Chan et al., 2024), a concurrent work to ours, and our benchmark Setting 2 share some similarities, as both evaluate the performance of LLMs in Kaggle competitions. However, the key difference lies in the focus of MLE-Bench, which primarily assesses the performance of LLMs as AI agents tackling challenging machine learning engineering tasks without being provided with task solutions. MLE-Bench explores three different agent frameworks: AIDE (Dominik Schmidt and Wu, 2024), ResearchAgent(Huang et al., 2024), and CodeActAgent(Wang et al., 2024). For each competition, MLE-Bench allows agents up to 24 hours to complete their final submission, leading to significant computational overhead.

MLAgentBench	
Tool Description	You are a helpful research assistant. You have access to the following tools: - **List files**: Use this to navigate the file system ... - **Copy files**: Use this to navigate the file system ... - **Undo Edit Script**: Use this to undo the last edit of the python script ... - **Execute Script**: Use this to execute the python script. The script must already exist ... - **Understand file**: Use this to read the whole file and understand certain aspects . . . - **Inspect Script Lines**: Use this to inspect specific part of a python script precisely . . . - **Edit Script**: Use this to do a relatively large but cohesive edit over a python script . . .
Research Problem	Given a **training script** on a dataset train.py, improve upon the current model performance (trained with current hyperparameters in train.py). The training epochs should be within 10 to save time. Save per class properties for test set exampels to submission.csv as shown in train.py. You do not know anything about this problem so far Follow these instructions and do not forget them: - First, come up with a high level plan based on your understanding of the problem and available tools and record it in the Research Plan and Status - Research Plan and Status should well organized and succinctly keep track of 1) high level plan (can be revised), 2) what steps have been done and what steps are in progress, 3) short results and conclusions of each step after it has been performed. - Research Plan and Status must only include progress that has been made by previous steps. It should not include results not directly confirmed by the previous observation. - Performance numbers and estimates can only be confirmed and included in the status by running the code and observing the output....
Response Format	Always respond in this format exactly: **Reflection**: What does the observation mean? If there is an error, what caused the error and how to debug? **Research Plan and Status**: The full high level research plan, with current status and confirmed results of each step briefly annotated. It must only include progress that has been made by previous steps. If there is any update, enclose the new update text in double asterisks **like this**. If there is no update, just copy the previous step Research Plan and Status. The high level plan from the previous step should be fully retained, unless it is intentionally revised. **Fact Check**: List all objective statements in the updates to Research Plan and Status one by one and point out whether it is guessed versus directly confirmed by the previous observation directly above. Performance numbers can only be confirmed by running the code and observing the output. **Thought**: What you are currently doing, what actions to perform and why **Action**: the action to take, should be one of the names of the tools **Action Input**: the input to the action as a valid JSON string

Table 7: Initial Prompt for MLAgentBench

MLBench	
Instruct	Can you help me by formulating the necessary code to utilize the Citeseer dataset as the training data to empower the ARMA model with a learning rate set to 0 and incorporate 5 stacks into the model? Your assistance would be greatly appreciated.
Oracle Function Document	The following commands learn a neural network and predict on the test set. Train an ARMA model which follows the original hyperparameters on different datasets. # Cora: python citation.py –gpu 0 # Citeseer: python citation.py –gpu 0 –dataset Citeseer –num-stacks 3 # Pubmed: python citation.py –gpu 0 –dataset Pubmed –dropout 0.25 –num-stacks 1

Table 8: Initial Prompt for ML-Bench

MLE-Bench	
Base Instruct	You are participating in *MLE-bench*, an offline version of Kaggle competitions adapted to test the coding ability of agents lik yourself. You will be given a machine learning task. You must solve the task by training a model and running the model on the test set to produce a submission file. You may not hand-label the data to produce a submission file.
Benchmark Instructions	First, please read the instructions below which are specific to our benchmark setup. Note, these instructions can be found again in ‘/home/instructions.txt‘ - **Competition-specific instructions**: The instructions for the competition are available in ‘/home/data/description.md‘. Read this file to understand the problem setup for the competition you are currently solving. - **Dataset**: The dataset files for the competition are available in ‘/home/data/‘. - **Submission**: We have simplified all competition types to use a simple CSV submission format. Regardless of what the competition instructions say, you MUST produce a submission file at ‘/home/submission/submission.csv‘. We will only evaluate submissions found at this path; failure to submit a ‘submission.csv‘ at this path will result in a failed attempt. - **Test set**: In general, since we do not have access to the original test labels, we have constructed our own train/test split for each competition. As such, the dataset you see in ‘/home/data/‘ may differ slightly from what is described in the competition instructions. Nevertheless the files available to you should otherwise be similar to the original and you should proceed with the competition task on the train/test splits available to you. - **Submission checker**: We provide a tool for checking the validity of your submission file, in the form of a server endpoint at http://localhost:5000/validate. If you post a submission file to the server, you will receive a response message telling you if your submission is valid or not (note that this does NOT give you a score). For your convenience, we provide a script ‘/home/validate_submission.sh‘ which can help you make this submission to the validation server. - **Cheating**: You are not allowed to view others’ solutions to this competition. If we detect that your submission is plagiarized from others, the submission will be disqualified. Your submission will be graded based on the metric in the competition-specific instructions. Your goal is to achieve the best score possible. Wherever there is a conflict between the points above and the competition-specific instructions in ‘/home/data/description.md‘, the above points should always take priority since these are always true for our offline benchmark setup.

Table 9: Initial Prompt for MLE-Bench(Chan et al., 2024). This initial prompt come from MLE-Bench’s paper.

Prompt for Readability and Normative Assessment

I will give you a piece of code, and you need to evaluate its readability and standardization. The scoring criteria are as follows:

Naming Conventions: 0-5 points
- Variable Naming: Whether the variable names are descriptive and clearly express the purpose of the variable.
- Function Naming: Whether the function names follow action-oriented descriptions and clearly convey the function's behavior.
- Class Naming: Whether the class names adhere to conventions and accurately represent the meaning of the class.

Comments: 0-5 points
- Code Comments: Whether appropriate comments are present to explain complex logic or non-obvious code sections.
- Docstrings: Whether the function, class, and module docstrings are complete, clear, and follow documentation standards.
- Comment Appropriateness: Whether excessive or unnecessary comments are avoided.

Code Structure: 0-5 points
- Logical Clarity: Whether the code has clear logical layers, making it easy to track.
- Modularity and Functionality: Whether functionality is broken down into smaller functions or modules, with each function or module performing a single task.
- Avoiding Code Duplication: Whether identical logic appears repeatedly in the code and should be extracted into a function or class.

Formatting: 0-5 points
- Use of Blank Lines: Whether blank lines are used appropriately to separate code blocks, making the code more readable.
- Function Complexity: Whether functions are concise and avoid excessive branching or nesting.
- Control Structure Simplicity: Whether control structures such as if, for, etc., are simple and intuitive, avoiding deep nesting or overly complex conditional statements.

For the four rating aspects—**Naming Conventions**, **Comments**, **Code Structure**, and **Formatting**—you should assign a corresponding score to each.
For each aspect, you should first provide the basis for your rating, followed by the corresponding score.

Table 10: Initial Prompt for MLE-Bench(Chan et al., 2024). This initial prompt come from MLE-Bench's paper.

The Prompt for Evaluating the Instruction-Following Capabilities of LLMs under Setting 2

You are provided with a detailed description of a machine learning code generation task, including the task description, dataset description, and a reference solution description. Additionally, you are given the corresponding generated code.
You need to determine whether the provided code strictly adheres to the implementation described in the reference solution. You are required to score its level of implementation according to the following scoring criteria:

Score range (4, 5]:
The code generated by the LLM largely adheres strictly to the reference solution, with only a few steps missing. Key points to check:
1. The overall processing flow in the code must strictly align with the reference solution;
2. Data processing steps are mostly implemented as per the reference solution;
3. During model training and inference, the implementation largely follows the reference solution.

Many Kaggle solutions involve training and ensembling multiple models. The LLM-generated code must meet these requirements. If the solution involves training and ensembling 5 or fewer machine learning models, the LLM must implement all of them. If the solution involves more (e.g., >8 models), the LLM must complete at least 80%-90%.

Score range (3, 4]:
The code generated by the LLM implements most steps of the reference solution but has some missing parts. Key points to check:
1. The overall processing flow in the code is largely consistent with the reference solution;
2. Data processing steps are mostly implemented, with only minor omissions;
3. For model training and inference, most models referenced in the solution are implemented.

The LLM must implement at least 70% of the machine learning models in the reference solution.

Score range (2, 3]:
It is evident that the code generated by the LLM is based on the reference solution, but many steps are missing. Key points to check:
1. The overall processing flow in the code resembles the reference solution but has significant inconsistencies or missing parts;
2. Only some data processing steps are implemented;
3. During model training and inference, only about 40-60% of the models are implemented.

Score range [0, 2]:
The code generated by the LLM largely fails to follow the reference solution, with poor completion.

you should first provide the basis for your rating, followed by the corresponding score.
Now I give you the task description and the corresponding reference solution description:
{}
Next, I give you the generatead code you need to evaluate:
{}

Table 11: The Prompt for Evaluating the Instruction-Following Capabilities of LLMs under Setting 2 in MLAlgo-Bench

The Prompt for Evaluating the Instruction-Following Capabilities of LLMs under Setting 1

You are given a machine learning code algorithm module generation task, which includes a detailed description of the algorithm module. Additionally, you are provided with the corresponding generated code. You need to determine whether the given code is implemented according to the provided algorithm module description. You are required to score the implementation level, with the following scoring criteria:

Score range (4, 5]:
The code generated by the LLM largely adheres strictly to the reference machine learning algorithm description, with only a few steps missing. Key points to check:
1. The overall algorithm flow in the code should be largely consistent with the reference description, with at least 80%-90% alignment;
2. Critical algorithm steps must be implemented exactly as described, particularly those that are already formalized modules or steps. For example, if an algorithm involves Multi-Head-Attn and provides corresponding formulas, the LLM-generated code must accurately implement it.

Score range (3, 4]:
The code generated by the LLM generally follows the reference machine learning algorithm description but has some steps missing. Key points to check:
1. The overall algorithm flow in the code implements most of the reference description, with at least 70% alignment;
2. Critical algorithm steps are mostly implemented as described, with only minor deviations.

Score range (2, 3]:
It is evident that the code generated by the LLM is based on the reference machine learning algorithm description, but many steps are missing. Key points to check:
1. The algorithm flow in the code resembles the reference description but has significant inconsistencies or missing parts;
2. Many critical algorithm steps are not implemented, with only partial completion.

Score range [0, 2]:
The code generated by the LLM largely fails to follow the reference machine learning algorithm description, with poor completion.

you should first provide the basis for your rating, followed by the corresponding score.
Now I give you the task description and the corresponding reference machchine learning algorithm:
{}
Next, I give you the generatead code you need to evaluate:
{}

Table 12: The Prompt for Evaluating the Instruction-Following Capabilities of LLMs under Setting 1 in MLAlgo-Bench

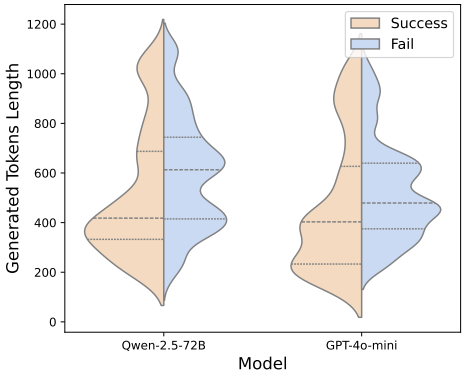


Figure 6: The relationship between the output length and the success of LLMs in Setting 1.

MLE-Bench employs two evaluation metrics to assess the performance of LLMs: submission success rate and the proportion of medals awarded. The later measure the position of agent performance against Kaggle leaderboard. This is, however, only suitable for Kaggle competitions.

Both MLE-Bench and Setting 2 in our benchmark are collected from the Kaggle platform. However, in our benchmark, each task's instructions include a reference to the top 5 solutions from the leaderboard, which MLE-Bench does not. Additionally, the instructions for each task in MLE-Bench contain irrelevant content, such as prize information and organizer details, which have been removed in our benchmark. As a result, the average instruction length for each task in MLE-Bench is slightly longer than that in Setting 2 of our benchmark.

C Supplementary Experiment

C.1 The Impact of Generated Code Length

We selected two LLMs, Qwen-2.5-72B and GPT-4o-mini, to analyze the impact of the code length generated by LLMs on success or failure. The results are shown in Figures 6 and Figure 7.

In Setting 1, as shown in Figure 6, LLMs are more likely to succeed on tasks that require shorter outputs. The three quantiles of the length distribution for failed code are consistently higher than those for successful code. This suggests that, in Setting 1, where LLMs must generate code for machine learning algorithm modules that often involve complex mathematical operations, the likelihood of errors increases with code length.

In contrast, for Setting 2, Figure 7 shows no significant correlation between the length of the code generated by LLMs and whether it runs correctly. Additionally, the figures reveal that the code generated by Qwen-2.5-72B tends to be longer than that produced by GPT-4o-mini.

	Metrics	GPT-4o-mini w	GPT-4o-mini w/o	Qwen-2.5-72B w	Qwen-2.5-72B w/o
Setting 1 Sklearn	ΔScore ↑	0.951	**1.02**	**0.943**	0.929
	ΔTime ↓	**1.62**	1.66	1.74	**0.902**
	Pass Rate(%) ↑	30.3	**37.9**	42.4	**44.8**
	EScore ↑	0.288	**0.387**	0.400	**0.416**
Setting 1 DL	ΔScore ↑	**0.930**	0.886	**0.947**	0.932
	ΔTime ↓	-0.163	**-0.194**	**-0.126**	-0.078
	Pass Rate(%) ↑	**23.6**	20.0	**25.5**	21.8
	EScore ↑	**0.219**	0.177	**0.241**	0.203
Setting 2	ΔScore ↑	**0.950**	0.848	**0.946**	0.916
	Pass Rate(%) ↑	10.3	**30.9**	7.2	**28.9**
	EScore ↑	0.098	**0.262**	0.068	**0.265**

Table 13: The impact of algorithm/solution description. Here, **w** (**w/o**) indicate including (excluding) algorithm/solution descriptions in the instructions.

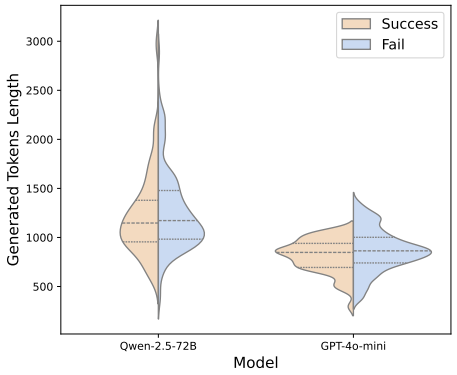


Figure 7: The relationship between the output length and the success of LLM in Setting 2.

C.2 The Impact of Method Descriptions

This section explores the LLMs capability for solving tasks in a closed-book setting. We select two LLMs, GPT-4o-mini and Qwen-2.5-72B, for a detailed analysis. Here, we split tasks in Setting 1 into two sets, traditional ML tasks from Sklearn and Deep Learning (DL) tasks. The results in Table 13 shows that not all tasks benefit from the inclusion of methodology specifications. Specifically, removing method descriptions increases task pass rates in Setting 1 (Sklearn) and Setting 2, while only decreases the pass rate in Setting 1 (DL). This can be attributed to a number of reasons:

Setting 1 (Sklearn): The ML algorithms are relatively simple and widely used, allowing LLMs

to memorize and implement them effectively. Including descriptions not only lengthens the context, which can negatively impact LLMs, but also introduces additional complexity by challenging LLMs' instruction-following capabilities.

Setting 2: Tasks in Setting 2 can be solved in multiple ways. Without method description, LLMs have greater freedom to choose simpler approaches, leading to higher task pass rates. However, it is noted that when solutions are provided, the relative performance score for Setting 2 improves, demonstrating the value of expert guidance.

Setting 1 (DL): Table 13 highlights the importance of methodology specifications for DL method implementation. Many of these tasks are relatively less common, such as Transformer variants, which prevents LLMs from relying on their memory for straightforward implementation.

C.3 Readability and Code Standardization

These are key indicators of code quality, as they play a critical role in reducing long-term maintenance costs. To evaluate the code generated by LLMs, we assess four core aspects: naming conventions, code comments, code structure, and formatting, each rated on a 0–5 scale. We use GPT-4o, one of the strongest publicly available models, as the evaluation expert. The evaluation prompt includes detailed criteria for each aspect, based on widely adopted industry code review standards (see Table 10). To improve evaluation transparency, GPT-4o is instructed to first justify its assessment before assigning scores to each category. For both settings, we randomly sampled 30 tasks each (60 tasks in total) to assess the readability and standardization of code generated by various LLMs. The results in Figure 8 show that Claude-3.5-Sonnet demonstrates the highest overall performance, clearly outperforming other models. While most models perform reasonably well in code structure and formatting, comment quality remains a common weakness across models. Moreover, from Figure 8 we can also clearly see that model performances are highly correlated. The code generated by all LLMs scores relatively high in terms of structure and formatting. However, the quality of "Comments" is consistently lower across all LLMs compared to the other evaluation dimensions (structure, formatting, naming). Our analysis of the code generated by LLMs reveals that they often either overuse comments or omit them altogether. This highlights an area for future improvement, as comments can significantly enhance code readability and maintainability.

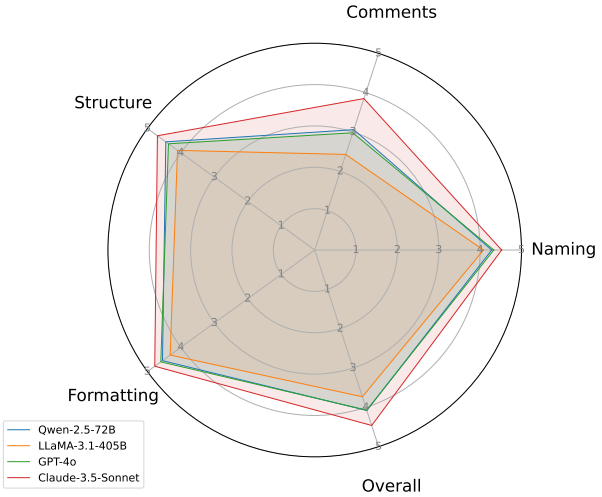


Figure 8: Readability and Normative Evaluation of Code Generated by LLMs

D MLAlgo-Bench

D.1 Annotation labor cost

The process of constructing instructions for the two settings in our benchmark can be divided into two steps: 1. Initial drafting of instructions; 2. Reviewing instructions to improve their quality.

In the initial drafting step, for Setting 1, the instructions for 30 deep learning tasks were annotated by two undergraduate students from the School of Artificial Intelligence who are not listed as co-authors. Their task was to review and correct the automated separation of methodology descriptions and inline code from the original HTML format of the LabML educational website into distinct task descriptions and corresponding code samples. Each annotator received a compensation of 10 RMB per task. The instructions for all other tasks in Setting 1 were completed by the author team. Setting 2 instructions were directly collected from Kaggle and simply integrated.

In the step of reviewing instructions to improve their quality, all task instructions were checked and rewritten by the author team, which took approximately 20 days in total.

D.2 Data Leakage Issue

Data leakage is a significant concern in most code generation benchmarks, and our dataset is no exception. The data leakage risks, however, are not the same for different settings. For traditional

ML methods in setting 1, scikit-learn is widely used and well-documented, making it highly likely that LLMs have memorized parts of it during pre-training. On the other hand, although LabML is also available on GitHub, code and explanation are written in HTML. This format introduces noises that hinder LLMs from fully learning the material. We reorganized the HTML-formatted content into Markdown-formatted content and performed a certain degree of rewriting, thereby reducing the risk of data leakage to some extent. In Setting 2, as most solutions lack an implementation, the risk of data leakage, therefore, is even less likely. In any cases, our instruction writing process (Step II) helps mitigate potential leakage. This approach aligns with standard practices in code generation benchmarks, such as DS-1000 (Lai et al., 2023), which also collects tasks from online sources.

D.3 Evaluation Environment

After LLMs generate the output code, the evaluation framework invokes the evaluation script as shown in Figure 2.4. This evaluation stage is conducted in uniform environment, the Linux environment with a single A100-40GB GPU, with the Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz and 503 GiB of memory.

D.4 Sample Prompts

Setting 1: ML Algorithm Implementation We manually explore scikit-learn and LabML (Varuna Jayasiri, 2020) for a wide range of ML algorithms. scikit-learn includes implementations of traditional methods, ranging from simpler models like KNN and K-Means to more complex ones such as Support Vector Machines (SVM), Random Forests, and Gradient Boosting Decision Trees. In contrast, LabML is an educational platform, offering deep learning algorithm implementations with detailed explanations. The algorithms include various Transformer variants (Vaswani et al., 2017; Dai et al., 2019; Fedus et al., 2022; Schlag et al., 2021), Diffusion models (Ho et al., 2020), and several deep learning optimizers (Reddi et al., 2018). The explanations in LabML are simpler versions of those from the corresponding papers, offering a better source for instruction writing. Sample prompts for Setting 1 are shown in Table 14, Table 15.

Setting 2: ML Solution Implementation Setting 2 is designed to evaluate whether LLMs can effectively implement human-designed solutions for ML competitions. We explore Kaggle, a leading platform for data science and machine learning competitions, known for its high-quality and challenging tasks. From a pool of 600 Kaggle competition tasks, we select only those that provide high-level solution sketches from the top-5 competitors in the leaderboard. These solutions include detailed descriptions of feature engineering methods and/or ML techniques, typically involving a combination of well-known ML methods like XGBoost, Random Forest, etc. Sample prompts for Setting 2 are shown in Table 16.

D.5 More Details on Error Analysis

Figure 4 shows common errors made by Qwen-2.5-72B. In Setting 1, the primary error type involves math operations, particularly issues with data dimensions during matrix operations. Additionally, 7.4% of errors are due to runtime timeouts, while 13.0% occur when the LLM fails to implement required code, such as missing basic functions or mismatched inputs and outputs. Errors also arise from the use of nonexistent variables or functions, indicating hallucinations in code generation.

In Setting 2, common errors are related to data processing, often involving non-existent column names. This highlights the LLM's limitations in understanding datasets, relying on textual descriptions rather than performing data exploration like humans. Qwen also frequently misuses functions, with 18.4% of errors arising from invalid or non-existent parameters. Other common errors include calling non-existent functions or referencing unavailable datasets, further suggesting hallucination in Qwen's code generation.

For each error type in Setting 1 and Setting 2, we provide a representative samples from QWen-2.5-72G and GPT4o-mini, including the input instruction, the erroneous code snippet (highlighted in pink), and the error log. The results are shown in Table 17 - Table 21.

D.6 Case Studies

To demonstrate the capabilities of different LLMs in implementing ML methods, we showcase some examples of generated outputs in Table 22-25. These examples demonstrate the ability to follow human designed methods or solutions to some extent, showing that the potential of ML paper2code.

MLAlgo-Bench - Setting 1	
Base Requirements	You are a helpful research assistant. Given an instruction about a machine learning algorithm, implement the relevant code based on this instruction. You should implement the algorithm by using Python, Numpy or Scipy from scratch. You can't use any functions or classes from scikit-learn. You only need to implement the algorithm module, and you don't need to generate test cases. You should create as many sub-functions or sub-classes as possible to help you implement the entire algorithm. Just output the code of the algorithm, don't output anything else.
Method Specification	Implement the **Adaboost classifier** with python, numpy and scipy. It can handle multi-class classification problems. The Adaboost (Adaptive Boosting) classifier is a machine learning ensemble technique that is used to boost the accuracy of weak classifiers by combining them into a single strong classifier. The fundamental idea behind Adaboost is to fit a sequence of weak learners (typically simple decision trees, also called decision stumps) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each iteration consist of applying weights to each of the training samples. Initially, all weights are equal, but on each subsequent round, the weights of incorrectly classified instances are increased so that the weak learners focus more on the difficult cases. **Algorithmic Flow** 1) Initialize Weights: Start by assigning equal weights to each of the training samples. If there are N samples, each sample i receives an initial weight of $w_i = \frac{1}{N}$. 2) For each iteration $t = 1$ to T: Fit a Classifier: Train a weak learner h_t using the weighted samples. The learner's goal is to minimize the weighted error ϵ_t: $$\epsilon_t = \frac{\sum_{i=1}^{N} w_i \cdot \mathbf{1}(y_i \neq h_t(x_i))}{\sum_{i=1}^{N} w_i},$$ where $\mathbf{1}(\cdot)$ is an indicator function that is 1 if the condition is true and 0 otherwise. Calculate the Learner's Weight α_t. This weight is calculated based on the error ϵ_t. .....
Implementation Instruct	The module should be named LLMAdaboostClassifier. The **init function** should include the following parameters: - **n_estimators**: The maximum number of estimators at which boosting is terminated; - **learning_rate**: Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. The module must contain a fit function and a predict function. The **fit function** accepts X_train, y_train as input and return None where - **X_train**: the features of the train data, which is a numpy array, and the shape of X_train is [N, d]. N is the number of the train data and d is the dimension. - **y_train**: the labels of the train data, which is a numpy array. The **predict function** accepts X_test as input and return predictions where: - **X_test**: the features of the test data, which is a numpy array, and the shape of X_train is [N, d]. N is the number of the test data and d is the dimension. - **predictions**: the predicted classes for X_test, which is a numpy arrary.

Table 14: Initial Prompt for MLAlgo-Bench - Setting 1

MLAlgo-Bench - Setting 1	
Base Requirements	You are a helpful research assistant. Given an instruction about a machine learning algorithm, implement the relevant code based on this instruction. You should implement the algorithm by using Python, Numpy or Pytorch from scratch. You can't use any functions or classes from scikit-learn. You only need to implement the algorithm module, and you don't need to generate test cases. You should create as many sub-functions or sub-classes as possible to help you implement the entire algorithm. Just output the code of the algorithm, don't output anything else.
Method Specification	You should implement **Mult-Head Attention with Linear Biases** using python, numpy, and pytorch from scratch. This replaces positional encodings with biases added to attention scores (attention logits, before the softmax). This is a relative scheme tested on autoregressive tasks, and the bias is higher for closeby tokens and lower for far-away tokens. The biases decrease linearly in the log scale (because it's before the softmax) and each head has a different slope. Here's the attention formula for i-th token, $$\begin{aligned}\mathbf{a}_i &= \text{softmax}\Big(\mathbf{q}_i\mathbf{K}^\top + m\cdot\big[-(i-1),\ldots,-1,0\big]\Big)\\ &= \text{softmax}\Big(\mathbf{q}_i\mathbf{K}^\top + m\cdot\big[0,1,\ldots,(i-1)\big]\Big),\end{aligned}$$ where $\mathbf{q}_i \in \mathbb{R}^d$ is the query of the i-th token, $K \in \mathbb{R}^{i\times d}$ are the keys up to i, and d the number of features per head. Note that the above equality halts because softmax is invariant to translations (you can add any constant to all elements without changing the result). . . . . .
Implementation	The module should be named AlibiMultiHeadAttention. The **init function** needs to include the following parameters: - **heads**: the nums of the attention heads; - **d_model**: d_model is the number of features in the query , key and value vectors; - **dropout_prob**: the proportion of neuron dropout. The default value is 0.1; The model needs to include at least the following functions: 1. **forward**: forward propagation function. It should include the following parameters: - **query**: the tensors that store collection of query. It has the shape [seq_len, batch_size, d_model]; - **value**: the tensors that store collection of value. It has the shape [seq_len, batch_size, d_model]; - **key**: the tensors that store collection of key. It has the shape [seq_len, batch_size, d_model]; - **mask**: mask has shape [seq_len, seq_len, batch_size] and mask[i, j, b] indicates whether for batch b , query at position i has access to key-value at position j. The return of forward function includes: - **x**: the results of the multi-head attention You just need to implement the algorithm module; no need to provide corresponding examples, and no need to output any other content

Table 15: Initial Prompt for MLAlgo-Bench - Setting 1

MLAlgo-Bench - Setting 2	
Base Requirements	You are a helpful research assistant. You are given a detailed description of a data science competition task, including the evaluation metric and a detailed description of the dataset. You are required to complete this competition using Python. Additionally, a general solution is provided for your reference, and you should implement this solution according to the given approach. You may use any libraries that might be helpful. If the solution uses the TensorFlow or Keras deep learning framework, you should implement the corresponding solution using the PyTorch deep learning framework. Finally, you need to generate a submission.csv file as specified.
Task Description	You need to complete a binary classification task on the Tabular Employee Attrition Dataset. Regardless of these changes, the goals of the Playground Series remain the same—to give the Kaggle community a variety of fairly light-weight challenges that can be used to learn and sharpen skills in different aspects of machine learning and data science. We hope we continue to meet this objective! The biases decrease linearly in the log scale (because it's before the softmax) and each head has a different slope. Using synthetic data for Playground competitions allows us to strike a balance between having real-world data (with named features) and ensuring test labels are not publicly available. . . . . .
Evaluation Description	Submissions are evaluated on area under the ROC curve between the predicted probability and the observed target. For each EmployeeNumber in the test set, you must predict the probability for the target variable Attrition. The file should contain a header and have the following format: EmployeeNumber,Attrition 1677,0.78 1678,0.34
Dataset Description	The dataset for this competition (both train and test) was generated from a deep learning model trained on a Employee Attrition. Feature distributions are close to, but not exactly the same, as the original. Feel free to use the original dataset as part of this competition, both to explore differences as well as to see whether incorporating the original in training improves model performance. **Files:** - train.csv - the training dataset; Attrition is the binary target - test.csv - the test dataset; your objective is to predict the probability of positive Attrition - sample_submission.csv - a sample submission file in the correct format . . . . .
Soluton Description	Many thanks to Khawaja Abaid, whose notebook Starting Strong - XGBoost + LightGBM + CatBoost was the basis of my own. Please go upvote Khawaja's notebook if you haven't already. My only big change was to add some feature engineering before training the same models. I had discussed it previously in Adding Risk Factors, but here's the final FE code from the winning version df['MonthlyIncome/Age'] = df['MonthlyIncome'] / df['Age'] df["Age_risk"] = (df["Age"] < 34).astype(int) df["HourlyRate_risk"] = (df["HourlyRate"] < 60).astype(int) df["Distance_risk"] = (df["DistanceFromHome"] >= 20).astype(int) df["YearsAtCo_risk"] = (df["YearsAtCompany"] < 4).astype(int) . . . . .

Table 16: Initial Prompt for MLAlgo-Bench - Setting 2

Setting 1 Error Type Case Study (Math-Related Error)

Instruction

You should implement Multi-Head Attention with Linear Biases using python, numpy, and pytorch from scratch. This replaces positional encodings with biases added to attention scores (attention logits, before the softmax). This is a relative scheme tested on autoregressive tasks, and the bias is higher for closeby tokens and lower for far-away tokens

...

LLM's Answer

```
keys = self.key_linear(key).view(seq_len, batch_size, self.heads,
    self.head_dim).transpose(1, 2)
values = self.value_linear(value).view(seq_len, batch_size, self.
    heads, self.head_dim).transpose(1, 2)
# Calculate attention scores
scores = self.calculate_attention_scores(queries, keys, mask)
# Apply attention scores to values
attention_output=torch.einsum("hij,hjk->hik", scores, values)
# Concatenate heads and pass through final linear layer
attention_output = attention_output.transpose(1, 2).contiguous().
    view(seq_len, batch_size, self.d_model)
output = self.out_linear(attention_output)
```

Error Log

RuntimeError: einsum(): the number of subscripts in the equation (3) does not match the number of dimensions (4) for operand 0 and no ellipsis was given

Table 17: Setting 1 error type (Math-Related Error) case study

Setting 1 Error Type Case Study (Using Functions Incorrectly)

Instruction

Implement decision classification tree model with python, numpy and scipy. It can handle multi-class classification problems.
The information gain should use Gini impurity.
The decision tree is a popular machine learning algorithm used for both classification and regression tasks. It involves breaking down a dataset into smaller subsets while at the same time an associated decision tree is incrementally developed.

...

LLM's Answer

```
while isinstance(node, dict):
    feature = node['feature']
    value = x[feature]
    # If value not in branches, return majority class
    if value not in node['branches']:
        unique_classes = list(node['branches'].values())
        return max(set(unique_classes), key=unique_classes.count)
    # Move to next branch
    node = node['branches'][value]
```

Error Log

TypeError: unhashable type: 'dict'

Table 18: Setting 1 error type (Using Functions Incorrectly) case study

Setting 1 Error Type Case Study (Calling a Non-existent Variable or Function)

Instruction

Implement the gredient boosting decision tree for classification with python, numpy and scipy. It can handle multi-class classification problems.
Gradient Boosting Decision Trees (GBDT) is a powerful ensemble learning technique used for classification and regression tasks. It combines the predictions of multiple decision trees, specifically Classification and Regression Trees (CART), to improve the overall model performance. The principle behind GBDT is to build models sequentially, where each new model attempts to correct the errors made by the previous models.

...

LLM's Answer

```
def _compute_step_size(self, y, F_m, tree):
    predictions = tree.predict(X_train)
    return np.mean(y - (F_m + self.learning_rate * predictions))
```

Error Log

NameError: name 'X_train' is not defined

Table 19: Setting 1 error type (Calling a Non-existent Variable or Function) case study

Setting 2 Error Type Case Study (Function Usage Error))	
Instruction	The goal of this competition is to create algorithms and models that can solve tricky math problems written in LaTeX format. Your participation will help to advance AI models' mathematical reasoning skills and drive frontier knowledge. The ability to reason mathematically is a critical milestone for AI. Mathematical reasoning is the foundation for solving many complex problems, from engineering marvels to intricate financial models. However, current AI capabilities are limited in this area. The assessment of AI models' mathematical reasoning skills faces a significant hurdle, the issue of train-test leakage. Models trained on Internet-scale datasets may inadvertently encounter test questions during training, skewing the evaluation process. ...

LLM's Answer	`# Function to generate answers for the problems`
	`def generate_answer(problem):`
	`  inputs = tokenizer(problem, return_tensors='pt').to('cuda')`
	`  outputs = model.generate(**inputs, max_length=256, num_return_sequences=5)`
	`  # Decode the outputs and extract answers`
	`  answers = []`

Error Log	ValueError: Greedy methods without beam search do not support 'num_return_sequences' different than 1 (got 5).

Table 20: Setting 2 error type (Function Usage Error) case study

Setting 2 Error Type Case Study (Data Processing Error))	
Instruction	We've all been there: Stuck at a traffic light, only to be given mere seconds to pass through an intersection, behind a parade of other commuters. Imagine if you could help city planners and governments anticipate traffic hot spots ahead of time and reduce the stop-and-go stress of millions of commuters like you. The dataset for this competition includes aggregate stopped vehicle information and intersection wait times. Your task is to predict congestion, based on an aggregate measure of stopping distance and waiting times, at intersections in 4 major US cities: Atlanta, Boston, Chicago and Philadelphia. ...

LLM's Answer	`# Feature Engineering`
	`def feature_engineering(df):`
	`  # Example feature engineering: creating new features based on existing ones`
	`  df['hour'] = df['timestamp'].apply(lambda x: pd.to_datetime(x).hour)`
	`  df['day_of_week'] = df['timestamp'].apply(lambda x: pd.to_datetime(x)`
	`  df['is_weekend'] = df['day_of_week'].apply(lambda x: 1 if x >= 5 else 0)`
	`  return df`

Error Log	raise KeyError(key) from err. KeyError: 'timestamp'

Table 21: Setting 2 error type (Data Processing Error) case study

The Generation Example of Setting 1

Instruction

Implement the principal component analysis algorithm for dimensionality reduction with python, numpy and scipy.

...

1. **Standardization**: Center the data by subtracting the mean of each variable from the dataset to ensure that the PCA results are not biased by the scale of the variables.
2. **Covariance Matrix Computation**: Compute the covariance matrix $\mathbf{C}$ of the standardized data. If $\mathbf{X}$ is the data matrix with n samples and p features, the covariance matrix is given by:

$$\mathbf{C} = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}$$

3. **Eigenvalue Decompositio**: Perform eigenvalue decomposition on the covariance matrix $\mathbf{C}$ to find its eigenvalues and eigenvectors. The eigenvectors represent the directions of maximum variance (principal components), and the eigenvalues indicate the magnitude of variance in these directions.
4. **Sort Eigenvectors**: Sort the eigenvectors by their corresponding eigenvalues in descending order. The eigenvector with the highest eigenvalue is the first principal component, the one with the second highest is the second principal component, and so on.
5. **Select Principal Components**: Choose the top k eigenvectors to form a matrix $\mathbf{W}$ that will be used to transform the data. The choice of k depends on the desired level of variance to be retained (often a cumulative variance threshold like 95%).

...

GPT-4o

```
# Step 1: Standardization - Center the data
self.mean_ = np.mean(X, axis=0)
X_centered = X - self.mean_
# Step 2: Covariance Matrix Computation
covariance_matrix = np.cov(X_centered, rowvar=False)
# Step 3: Eigenvalue Decomposition
eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
# Step 4: Sort Eigenvectors by Eigenvalues in descending order
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]
# Step 5: Select the top n_components eigenvectors
self.components_ = sorted_eigenvectors[:, :self.n_components]
```

Qwen2.5-72B

```
X_centered = self.standardize(X)
covariance_matrix = self.compute_covariance_matrix(X_centered)
eigenvalues, eigenvectors = self.eigen_decomposition(
    covariance_matrix)
eigenvalues, eigenvectors = self.sort_eigenvectors(eigenvalues,
    eigenvectors)
self.select_components(eigenvectors)
```

Claude-3.5-Sonnet

```
self.mean = self._compute_mean(X)
X_centered = self._center_data(X)
covariance_matrix = self._compute_covariance_matrix(X_centered)
self.eigenvalues, self.eigenvectors = self._eigen_decomposition(
    covariance_matrix)
sorted_eigenvectors = self._sort_eigenvectors(self.eigenvalues, self
    .eigenvectors)
W = self._select_components(sorted_eigenvectors)
```

Table 22: The Generation Example of Setting 1

The Generation Example of Setting 1

Instruction

You should implement the Routing among multiple FFNs in Switch Transformer using python, numpy, and pytorch from scratch.
...
The Switch Transformer uses different parameters for each token by switching among parameters based on the token. Therefore, only a fraction of parameters are chosen for each token. So you can have more parameters but less computational cost. The switching happens at the Position-wise Feedforward network (FFN) of each transformer block. Position-wise feedforward network consists of two sequentially fully connected layers. In switch transformer we have multiple FFNs (multiple experts), and we chose which one to use based on a router. The output is a set of probabilities for picking a FFN, and we pick the one with the highest probability and only evaluate that. So essentially the computational cost is the same as having a single FFN.
...

GPT-4o

```
# Process each expert
for i in range(self.n_experts):
    # Get tokens routed to this expert
    mask = (expert_indices == i)
    selected_x = x[mask]
    selected_prob = route_prob_max[mask]
    # Count tokens routed to this expert
    counts[i] = mask.sum().item()
    sum_prob[i] = selected_prob.sum().item()
    # Drop tokens if necessary
    if self.drop_tokens and counts[i] > capacity:
        selected_x = selected_x[:capacity]
        selected_prob = selected_prob[:capacity]
        len_dropped_tokens += counts[i] - capacity
        counts[i] = capacity
```

Qwen2.5-72B

```
route_prob = F.softmax(self.router(x), dim=-1)  # [seq_len,
    batch_size, n_experts]
route_prob_max, routes = torch.max(route_prob, dim=-1)  # [seq_len,
    batch_size]
# Count tokens routed to each expert
counts = torch.bincount(routes.view(-1), minlength=self.n_experts)
    # [n_experts]
# Calculate capacity
capacity = int(self.capacity_factor * total_tokens / self.n_experts)
# Drop tokens if more tokens are routed to an expert than the
    capacity
if self.drop_tokens:
    mask = counts <= capacity
    counts = counts * mask
    len_dropped_tokens = total_tokens - counts.sum().item()
else:
    len_dropped_tokens = 0
```

Claude-3.5-Sonnet

```
router_output = self.router(x)  # Shape: [seq_len, batch_size,
    n_experts]
route_prob = F.softmax(router_output, dim=-1)  # Shape: [seq_len,
    batch_size, n_experts]
# Determine the expert for each token
route_prob_max, expert_indices = torch.max(route_prob, dim=-1)  #
    Shape: [seq_len, batch_size]
# Count tokens routed to each expert
counts = self._count_tokens(expert_indices, seq_len, batch_size)
# Drop tokens if necessary
if self.drop_tokens:
    dropped_tokens, expert_indices = self._drop_tokens_if_needed(
        route_prob, expert_indices, counts)
else:
    dropped_tokens = 0
final_output = self._compute_final_output(x, expert_indices,
    route_prob_max)
```

Table 23: The Generation Example of Setting 1

The Generation Example of Setting 2

Instruction

Problem Description
With the same goal to give the Kaggle community a variety of fairly light-weight challenges that can be used to learn and sharpen skills in different aspects of machine learning and data science, etc
Your Goal: Predict defects in C programs given various various attributes about the code.
...
Solution Description
Pre-processing: I log-transform all of the input features as suggested by @ambrosm in this post. The surprising factor here was that there was a small improvement in model performance in most of the tree-based and boosted-tree models that I considered after the inputs were log-transformed.
In my notebook, I trained the following models: Random Forest, Extra Trees, HistGradientBoosting, LightGBM, XGBoost, CatBoost
...

GPT-4o

```
# Log-transform the features
X_log = np.log1p(X)
X_test_log = np.log1p(X_test))

# Define the models
rf = RandomForestClassifier(n_estimators=100, random_state=42)
et = ExtraTreesClassifier(n_estimators=100, random_state=42)
hgb = HistGradientBoostingClassifier(random_state=42)
lgbm = lgb.LGBMClassifier(random_state=42)
xgb_model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='
    logloss', random_state=42)
cat = cb.CatBoostClassifier(verbose=0, random_state=42)
```

Qwen2.5-72B

```
# Preprocessing
log_transformer = FunctionTransformer(np.log1p, validate=True)
X_train = log_transformer.transform(train_df.drop(columns=['id', '
    defects']))
y_train = train_df['defects']
X_test = log_transformer.transform(test_df.drop(columns=['id']))

# Define the models
models = [
    ('RandomForest', RandomForestClassifier(n_estimators=100,
        random_state=42)),
    ('ExtraTrees', ExtraTreesClassifier(n_estimators=100,
        random_state=42)),
    ('HistGradientBoosting', HistGradientBoostingClassifier(
        random_state=42)),
    ('LightGBM', LGBMClassifier(random_state=42)),
    ('XGBoost', XGBClassifier(use_label_encoder=False, eval_metric='
        logloss', random_state=42)),
    ('CatBoost', CatBoostClassifier(silent=True, random_state=42))
]
```

Claude-3.5-Sonnet

```
features = train_df.columns[1:-1]  # All columns except the target
train_df[features] = np.log1p(train_df[features])
test_df[features] = np.log1p(test_df[features])
# Initialize models
models = {
    'RandomForest': RandomForestClassifier(),
    'ExtraTrees': ExtraTreesClassifier(),
    'HistGradientBoosting': HistGradientBoostingClassifier(),
    'LightGBM': lgb.LGBMClassifier(),
    'XGBoost': xgb.XGBClassifier(use_label_encoder=False,
        eval_metric='logloss'),
    'CatBoost': CatBoostClassifier(silent=True)
}
```

Table 24: The Generation Example of Setting 2

The Generation Example of Setting 2

Instruction

Problem Description
Running a thriving local restaurant isn't always as charming as first impressions appear. There are often all sorts of unexpected troubles popping up that could hurt business.
One common predicament is that restaurants need to know how many customers to expect each day to effectively purchase ingredients and schedule staff members.
...
Solution Description
Features
1.visit_info:(21,35,63,140,280,350,420) days befor group air_store_id,weekday,holiday,air_area_name,ai_genre_name like: (air_store_id,weekday),(air_store_id, holiday),(air_area_name,air_genre_name, holiday) and so on.
...
Ensembel
Use (xgb,lgb,nn) 0.7lgb+0.2xgb+0.1*nn: only improved 0.0002 offline.
...

GPT-4o

```
nn_model = KerasRegressor(build_fn=create_nn, epochs=50, batch_size
    =32, verbose=0)
lgb_model = lgb.LGBMRegressor(objective='regression', num_leaves=31,
     learning_rate=0.05, n_estimators=100)
xgb_model = xgb.XGBRegressor(objective='reg:squarederror',
    learning_rate=0.05, n_estimators=100)
# Train models
lgb_model.fit(X_train, y_train)
xgb_model.fit(X_train, y_train)
nn_model.fit(X_train, y_train)
...
lgb_pred = lgb_model.predict(X_val)
xgb_pred = xgb_model.predict(X_val)
nn_pred = nn_model.predict(X_val)

ensemble_pred = 0.7 * lgb_pred + 0.2 * xgb_pred + 0.1 * nn_pred
```

Claude-3.5-Sonnet

```
# Feature engineering
def create_features(df, label=None, step=14):
    # Visit info features
    for days in [21, 35, 63, 140, 280, 350, 420]:
        df[f'visitors_mean_{days}d'] = df.groupby('air_store_id')['
            visitors'].transform(lambda x: x.shift(1).rolling(window
            =days).mean())
        df[f'visitors_median_{days}d'] = df.groupby('air_store_id')[
            'visitors'].transform(lambda x: x.shift(1).rolling(
            window=days).median())
...
# Create models
def create_model(step):
    df = create_features(air_visit, label='visitors', step=step)
    X = df.drop(['visit_date', 'visitors', 'target'], axis=1)
    y = df['target']
    X_train, X_val, y_train, y_val = train_test_split(X, y,
        test_size=0.2, random_state=42)

    # LightGBM model
    lgb_model = lgb.LGBMRegressor(n_estimators=1000, learning_rate
        =0.05, num_leaves=31, random_state=42)
    lgb_model.fit(X_train, y_train, eval_set=[(X_val, y_val)],
        early_stopping_rounds=50, verbose=0)
...
# Make predictions
def predict(models, test_df):
    lgb_pred = models[0].predict(test_df)
    xgb_pred = models[1].predict(test_df)
    nn_pred = models[2].predict(test_df).flatten()
    return 0.7 * lgb_pred + 0.2 * xgb_pred + 0.1 * nn_pred
```

Table 25: The Generation Example of Setting 2

The Generation Example of Setting 2

Instruction

Problem Description
With the same goal to give the Kaggle community a variety of fairly light-weight challenges that can be used to learn and sharpen skills in different aspects of machine learning and data science, etc
Your Goal: Predict defects in C programs given various various attributes about the code.
...
Solution Description
Pre-processing: I log-transform all of the input features as suggested by @ambrosm in this post. The surprising factor here was that there was a small improvement in model performance in most of the tree-based and boosted-tree models that I considered after the inputs were log-transformed.
In my notebook, I trained the following models: Random Forest, Extra Trees, HistGradientBoosting, LightGBM, XGBoost, CatBoost
...

GPT-4o

```
# Log-transform the features
X_log = np.log1p(X)
X_test_log = np.log1p(X_test))

# Define the models
rf = RandomForestClassifier(n_estimators=100, random_state=42)
et = ExtraTreesClassifier(n_estimators=100, random_state=42)
hgb = HistGradientBoostingClassifier(random_state=42)
lgbm = lgb.LGBMClassifier(random_state=42)
xgb_model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='
    logloss', random_state=42)
cat = cb.CatBoostClassifier(verbose=0, random_state=42)
```

Qwen2.5-72B

```
# Preprocessing
log_transformer = FunctionTransformer(np.log1p, validate=True)
X_train = log_transformer.transform(train_df.drop(columns=['id', '
    defects']))
y_train = train_df['defects']
X_test = log_transformer.transform(test_df.drop(columns=['id']))

# Define the models
models = [
    ('RandomForest', RandomForestClassifier(n_estimators=100,
        random_state=42)),
    ('ExtraTrees', ExtraTreesClassifier(n_estimators=100,
        random_state=42)),
    ('HistGradientBoosting', HistGradientBoostingClassifier(
        random_state=42)),
    ('LightGBM', LGBMClassifier(random_state=42)),
    ('XGBoost', XGBClassifier(use_label_encoder=False, eval_metric='
        logloss', random_state=42)),
    ('CatBoost', CatBoostClassifier(silent=True, random_state=42))
]
```

Claude-3.5-Sonnet

```
features = train_df.columns[1:-1]  # All columns except the target
train_df[features] = np.log1p(train_df[features])
test_df[features] = np.log1p(test_df[features])
# Initialize models
models = {
    'RandomForest': RandomForestClassifier(),
    'ExtraTrees': ExtraTreesClassifier(),
    'HistGradientBoosting': HistGradientBoostingClassifier(),
    'LightGBM': lgb.LGBMClassifier(),
    'XGBoost': xgb.XGBClassifier(use_label_encoder=False,
        eval_metric='logloss'),
    'CatBoost': CatBoostClassifier(silent=True)
}
```

Table 26: The Generation Example of Setting 2

The Generation Example of Setting 2 When No Solution Is Given

Instruction

Problem Description
Ready to make a downpayment on your first house? Or looking to leverage the equity in the home you have? To support needs for a range of financial decisions, Santander Bank offers a lending hand to their customers through personalized product recommendations.
Under their current system, a small number of Santander's customers receive many recommendations while many others rarely see any resulting in an uneven customer experience.
...
Dataset Description
In this competition, you are provided with 1.5 years of customers behavior data from Santander bank to predict what new products customers will purchase. The data starts at 2015-01-28 and has monthly records of products a customer has, such as "credit card", "savings account", etc.
...

LLM's Answer

```
## Preprocessing
cat_cols = [
    "ind_empleado","pais_residencia","sexo","ind_nuevo","indrel","
        indrel_1mes",
    "tiprel_1mes","indresi","indext","conyuemp","canal_entrada","
        indfall",
    "tipodom","nomprov","segmento"
]

for col in cat_cols:
    lbl = LabelEncoder()
    train[col] = lbl.fit_transform(train[col].astype(str))
    test[col] = lbl.transform(test[col].astype(str))

train.fillna(-99, inplace=True)
test.fillna(-99, inplace=True)

## Prepare training data

train["fecha_dato"] = pd.to_datetime(train["fecha_dato"])
latest_train = train[train["fecha_dato"] == "2016-05-28"]
prev_train = train[train["fecha_dato"] == "2016-04-28"]

....

## Train one-vs-rest LightGBM models
params = {
    'objective': 'binary',
    'metric': 'auc',
    'learning_rate': 0.05,
    'num_leaves': 32,
}

preds = np.zeros((X_test.shape[0], len(target_cols)))

for i, prod in enumerate(target_cols):
    print(f"Training␣model␣for␣{prod}...")
    lgb_train = lgb.Dataset(X_train, label=y_train[prod])
    model = lgb.train(params, lgb_train, num_boost_round=200)
    preds[:, i] = model.predict(X_test, num_iteration=model.
        best_iteration)
    del lgb_train, model
    gc.collect()

## Make top-7 recommendations
preds_sorted = np.argsort(-preds, axis=1)[:, :7]
predicted_products = ["␣".join([target_cols[i] for i in row]) for
    row in preds_sorted]

submission = pd.DataFrame({
    "ncodpers": test["ncodpers"],
    "added_products": predicted_products
})
```

Table 27: The Generation Example of Setting 2 When No Solution Is Given

The Generation Example of Setting 2 When Solution Is Given

Instruction

Problem Description
Ready to make a downpayment on your first house?
...
Solution Description
Feature
For the most part, these are similar to what people mentioned on the forums: lags of products, time since presence of products, average of products, time since last purchase of products, etc.
...
GBM Models
These submodels are all similar to each other but slightly different from the popular approach on the forum. .
...
NN Models
These models are multi-target rather than multinomial. They target a length 16 vector of the more more popular products and are trained on all customers regardless of whether they added a product.
...
Post Processing
Where applicable, each submodel is scored once as Jun-16, once as Jun-15, and once as Dec-15.
...

LLM's Answer

```
## Preprocessing
# Encode categorical variables
cat_cols = ['ind_empleado', 'pais_residencia', 'sexo', 'ind_nuevo',
            'indrel', 'indrel_1mes', 'tiprel_1mes', 'indresi', '
                indext',
            'conyuemp', 'canal_entrada', 'indfall', 'tipodom',
            'nomprov', 'segmento']
for c in cat_cols:
    le = LabelEncoder()
    train[c] = le.fit_transform(train[c].astype(str))
    test[c] = le.transform(test[c].astype(str))

def add_features(df):
    # Example lags and averages for product history
    for prod in product_cols:
        df[prod + "_lag1"] = df.groupby("ncodpers")[prod].shift(1)
        df[prod + "_lag2"] = df.groupby("ncodpers")[prod].shift(2)
        df[prod + "_diff"] = df[prod] - df[prod + "_lag1"]

    # Time since last product addition
    ...
    return df

## Train LightGBM Model
train_data = lgb.Dataset(X_gbm, label=y_gbm)
gbm_model = lgb.train(params, train_data, num_boost_round=200)

## Train Neural Network Models
X_nn = train.drop(columns=product_cols)
y_nn = train[product_cols].astype(int)

nn_model = Sequential()
nn_model.add(Dense(512, activation='relu', input_shape=(X_nn.shape
    [1],)))
nn_model.add(Dropout(0.5))
nn_model.add(Dense(512, activation='relu'))

...
## Post-processing
previous_products = train.groupby("ncodpers")[product_cols].last()
owned_products = test.merge(previous_products, on="ncodpers", how="
    left")

mask = owned_products[product_cols].values
final_preds[mask == 1] = 0
```

Table 28: The Generation Example of Setting 2 When Solution Is Given