A Preliminary Evaluation of Large Language Models for Data Science Code Generation

Farshad Ghorbanishovaneh[1] **Lars Kotthoff**[1]

[1]University of Wyoming, USA

Abstract AutoML has made substantial strides in enabling domain scientists with little or no expertise in Machine Learning to develop state-of-the-art Data Science solutions. One of the main remaining challenges is to write the code to do so, especially in nontechnical disciplines where scientists are unlikely to have programming expertise. Large Language Models are increasingly adept at generating code for complex tasks. In this paper, we ask the question to what extent they can solve basic Data Science tasks. We survey a range of commercial and open-weight models on a basic Data Science task. Our results indicate that while many LLMs still struggle, acceptable results are achieved by some newer models, including small open-weight and certain commercial versions.

1 Introduction

Large Language Models (LLMs) have shown great promise in the generation of code to solve complex tasks and to prototype complex systems (Austin et al., 2021; Chen et al., 2021; Lyu et al., 2024; OpenAI et al., 2024; Balog et al., 2017). They have reduced the barrier for domain scientists to produce custom code even without a background in programming (Xu et al., 2024; Tayebi Arasteh et al., 2024). This raises the question we investigate in this paper: *Can LLMs reliably generate complete, executable ML pipelines for basic Data Science tasks?*

Recent advances in LLMs have made it increasingly feasible to generate executable code for data science workflows, lowering the entry barrier for non-programmers across disciplines (Xu et al., 2021; Zhang et al., 2024). This has led to a growing interest in evaluating the reliability and limitations of LLM-generated code, particularly in the context of machine learning pipelines (Abbassi et al., 2025; Chen et al., 2023). Previous studies have explored model reasoning ability, hallucination frequency, and task-specific performance, especially within programming-intensive domains (OpenAI et al., 2024; Chen et al., 2021; DeepSeek-AI et al., 2025; Abdin et al., 2025). Building on this foundation, our work investigates how effectively a diverse set of LLMs, both commercial and open weight, can produce usable ML code for standard data science tasks. We evaluated LLMs on their ability to autonomously generate complete, executable pipeline scripts for real-world data science tasks, moving beyond fragment-level code (Abbassi et al., 2025; Zhang et al., 2024).

The goal is to benchmark LLM effectiveness in AutoML-like workflows and to understand the reliability, adaptability, and practical viability of LLMs in data science environments. Although some models demonstrated robust zero-shot performance (Anthropic, 2025; Gemini Team et al., 2024), others required iterative correction or failed entirely, highlighting the importance of prompt design and feedback mechanisms (Chen et al., 2023; Xu et al., 2024; Hu et al., 2021).

2 Experimental Setup

We selected a diverse set of 35 LLMs, including commercial models such as Claude 3.5 Sonnet, Claude 3.7 Sonnet, Claude 3.7 Sonnet (Thinking), Claude Sonnet 4, Gemini 2.0 Flash, Gemini 2.5 Pro, GPT 4.1, GPT 4o, GPT o1, O3 Mini, and O4 Mini, as well as open-weight models including Codecetral, Codellama 70B, CodeQwen, Cogito 70B, Command R 7B, DeepSeek Coder V2 236B,

DeepSeek R1 671B, Gemma 3 27B, Gemma 7B, LLaMA 3.3, LLaMA 4 16×17B, LLaMA 4 Maverick, LLaMA 4 Scout, Mistra Large, Mistral Small 3.1, Phi 4, Phi 4 Reasoning, Qwen 2.5 72B, Qwen 3, Qwen 3 235B, Qwq, and Wizard Coder. Model selection was guided by each model's stated or demonstrated ability to perform code generation. Some models were specifically trained for programming tasks, while others were general-purpose LLMs cited in prior work for their coding capabilities.

Commercial models were accessed via their official APIs. Open-weight models were executed locally using the Ollama framework on a machine running Red Hat Enterprise Linux 9.4 (Plow) and 2 AMD EPYC 9454 CPUs (96 cores), 1.2 TB of RAM, and 8 NVIDIA H100 GPUs. The environment had Python 3.11 on Red Hat Enterprise Linux 9.

2.1 Task Definition

All models were prompted to build a complete classification pipeline using the Wine Quality dataset, including data loading, exploratory analysis, preprocessing, model training (Random Forest, Decision Tree, and SVM), hyperparameter tuning, evaluation (F1-score, accuracy, confusion matrix), and visualization of both data characteristics and model performance. The expected output was a self-contained Python script, which was to assess correctness, completeness, and functionality. Specifically, each script was required to:

- Load the dataset and provide a descriptive summary, including shape, feature types, basic statistics, and class distribution.
- Perform exploratory data analysis (e.g., checking missing values, data types, outliers) and basic preprocessing such as encoding and scaling.
- Train Decision Tree, Random Forest, and SVM classifiers and evaluate them using cross-validation.
- Perform hyperparameter tuning (e.g., Random Search or Bayesian Optimization).
- Evaluate results using accuracy, F1-score, and confusion matrices.
- Visualize model performance using confusion matrices, metric comparisons (e.g., F1-score, accuracy), and hyperparameter optimization results.

2.2 Prompting Framework

All prompts were issued in a controlled environment using Visual Studio Code, with GitHub Copilot Chat in Ask mode as the interface. Although Copilot facilitated the interaction, all responses were generated solely by the selected underlying models. Each model received up to three chances to revise its code. If execution failed, the error message was returned as feedback for the next attempt. A model was marked as failed if no valid code was produced after three iterations.

2.3 Evaluation and Grading Criteria

Each model's output was graded both for visual output and code quality. Two separate rubrics were used:

Effective Plot Grade (A–F): Assigned based on the presence, clarity, and usefulness of the plots. An A indicates clear and well-labeled visualizations that cover both the dataset and model performance. B reflects mostly complete plots with minor omissions or clarity issues. C corresponds to basic visualizations with some missing key results. D indicates minimal plotting with limited interpretability. E reflects poor or irrelevant plots, and F is assigned when no usable plots were produced, either because the plotting code was missing, not executed, or the code did not run at all. Full details are provided in Table 2.

Code Grade (A–F): Assigned based on the completeness, clarity, and correctness of the code. An A denotes complete, correct, and fully functional code that does not require feedback. B reflects mostly correct code with minor issues. C indicates partial completeness or flawed implementation. D represents minimal, underdeveloped code, and E corresponds to fragmentary or poorly structured code that runs only in part. F is reserved for code that failed to run after all correction attempts. For models that required feedback to fix errors, the grade was reduced by one level from what the output would have otherwise earned. See Table 3 for detailed criteria.

3 Results and Discussion

The evaluation demonstrated that both commercial and open-weight LLMs are capable of generating executable machine learning pipelines with varying degrees of success. Performance varied not only by model family but also by parameter count, prompt alignment, and feedback responsiveness.

Among commercial models, Claude Sonnet 4 and GPT-4.1 produced consistently accurate and well-structured code on the first attempt with minimal or no feedback. Gemini 2.5 Pro also performed well, generating correct and tuned pipelines with competitive performance.

In the open-weight category, Phi-4, Gemma 3, and Mistral-Large emerged as strong performers. These models often compiled successfully on the first attempt and delivered competitive metrics. Their performance suggests that recent open-weight advancements have narrowed the gap with proprietary systems in specific AutoML contexts.

The effectiveness of model-generated plots varied significantly across models. High-performing models such as Gemma 2.5 Pro, Claude Sonnet 4 produced visualizations that generally supported interpretation, although even among these, the quality of visual output was inconsistent. Interestingly, some models with smaller parameter sizes, such as Phi-4 Reasoning, demonstrated strong reasoning capabilities and produced reasonably effective plots—ranking better than several larger models. Furthermore, not all models produced valid outputs or complete plots, as indicated by the missing entries in the 'Effective Plot' column. This suggests that neither model size nor tuning alone guarantees useful visual outputs.

Some large-scale models—such as LLaMA 4 variants and DeepSeek R1—did not consistently outperform smaller or more optimized alternatives. In several cases, their code was more error-prone or required more correction attempts, contradicting expectations based on scale alone. This highlights that model size is not always a reliable predictor of execution robustness in code synthesis tasks. Other models, such as O3 Mini, CodeLlama 70B, and WizardCoder, failed to generate working code after three attempts. Gemma 7B evaluated only on the training set.

3.1 Key Patterns and Observations

- **Large models do not always perform better**. Smaller or open-weight models like Phi-4 and Gemma 3 outperformed larger counterparts like LLaMA 4 in several cases.
- **Error handling was a strong differentiator**. Models such as Claude 3.7, Gemini 2.0 Flash, and GPT 4o successfully incorporated feedback by correcting errors over multiple attempts, showing greater adaptability.
- **Confabulations still happen**. Some models fabricated function calls.

4 Conclusion

This study evaluated the ability of LLMs to generate executable machine learning pipelines for structured classification tasks. The results showed that both the commercial models Claude Sonnet 4 and Gemini 2.5 Pro, and the open-weight model Phi-4 Reasoning, were able to produce high-performance code with minimal repeated prompting.

LLM	Tuned	F1-Score	Attempt(s)	Feedback	Effective Plot Grade	Code Grade
Claude Sonnet 4	Yes	0.670	**1**	**No**	**A**	**A**
Gemini 2.5 Pro	Yes	0.667	**1**	**No**	**A**	**A**
Phi 4 Reasoning	Yes	0.661	**1**	**No**	B	**A**
Claude 3.7 Sonnet	Yes	0.660	3	Yes	**A**	B
Claude 3.7 Sonnet (Thinking)	Yes	0.675	3	Yes	B	B
GPT 4.1	Yes	0.664	**1**	**No**	B	B
LLaMA 4 16×17B	Yes	0.662	**1**	**No**	B	B
LLaMA 4 Maverick	Yes	0.651	2	**No**	B	B
Qwen 3 235B	Yes	**0.709**	**1**	**No**	B	C
Gemma 3 27B	Yes	0.540	2	**No**	B	C
Gemini 2.0 Flash	Yes	0.391	3	Yes	B	C
Qwq	Yes	0.640	**1**	**No**	C	C
Phi 4	Yes	0.638	**1**	**No**	C	C
GPT 4o	Yes	0.664	3	Yes	D	C
DeepSeek R1 671B	Yes	0.642	3	**No**	D	C
GPT o1	No	0.644	3	Yes	E	C
Mistra Large	Yes	0.640	**1**	**No**	C	D
DeepSeek Coder V2 236B	Yes	0.326	**1**	**No**	D	D
LLaMA 3.3	Yes	0.645	3	Yes	D	E
Gemma 7B*	Yes	1.000*	3	Yes	F	F
Claude 3.5 Sonnet	NA	–	3	Yes	F	F
CodeQwen	NA	–	3	Yes	F	F
Cogito 70B	NA	–	3	Yes	F	F
Command R 7B	NA	–	3	Yes	F	F
LLaMA 4 Scout	NA	–	3	Yes	F	F
O3 Mini	NA	–	3	Yes	F	F
O4 Mini	NA	–	3	Yes	F	F
Qwen 2.5 72B	NA	–	3	Yes	F	F
Qwen 3	NA	–	3	Yes	F	F
Wizard Coder	NA	–	3	Yes	F	F
Codecetral	NA	–	3	No	F	F
Codellama 70B	NA	–	3	No	F	F
Mistral Small 3.1	NA	–	3	No	F	F

Table 1: Best reported run for each LLM (highest F1-score). "–" = No valid code produced. "NA" = Not applicable. "Yes" = Feedback was used to correct the code. "No" = No feedback was needed. *Gemma 7B scores are on the training set only. **"Effective Plot Grade"**: Grade assigned based on the presence, clarity, and usefulness of the plots in supporting model interpretation. For grading criteria and visual examples, refer to the Appendix.

Although top-tier models often succeeded on the first attempt and required little to no feedback, others, particularly some large-scale models such as LLaMA 4 and DeepSeek R1, produced inconsistent or suboptimal results, despite their size. The integration of a feedback loop proved essential for evaluating model adaptability, highlighting that many models can recover from initial failures when provided with clear error messages. However, this did not occur in all cases, and confabulations remain a problem. Overall, our findings indicate that LLMs are approaching practical usability in automating parts of the data science workflow, especially in generating baseline ML pipelines.

References

Abbassi, A. A., Silva, L. D., Nikanjam, A., and Khomh, F. (2025). Unveiling inefficiencies in llm-generated code: Toward a comprehensive taxonomy.

Abdin, M., Agarwal, S., Awadallah, A., Balachandran, V., Behl, H., Chen, L., de Rosa, G., Gunasekar, S., Javaheripi, M., Joshi, N., Kauffmann, P., Lara, Y., Mendes, C. C. T., Mitra, A., Nushi, B., Papailiopoulos, D., Saarikivi, O., Shah, S., Shrivastava, V., Vineet, V., Wu, Y., Yousefi, S., and Zheng, G. (2025). Phi-4-reasoning technical report.

Anthropic (2025). Claude opus 4 & claude sonnet 4 system card. Technical report, Anthropic. Accessed: 2025-06-24.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2017). Deepcoder: Learning to write programs.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., and et al., J. K. (2021). Evaluating large language models trained on code.

Chen, X., Lin, M., Schärli, N., and Zhou, D. (2023). Teaching large language models to self-debug.

DeepSeek-AI et al. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Gemini Team et al. (2024). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2021). Lora: Low-rank adaptation of large language models.

Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., and Thongtanunam, P. (2024). Automatic programming: Large language models and beyond.

OpenAI et al. (2024). Gpt-4 technical report.

Tayebi Arasteh, S., Han, T., Lotfinia, M., Kuhl, C., Kather, J. N., Truhn, D., and Nebelung, S. (2024). Large language models streamline automated machine learning for clinical studies. *Nature Communications*, 15(1).

Xu, F. F., Vasilescu, B., and Neubig, G. (2021). In-ide code generation from natural language: Promise and challenges.

Xu, J., Li, J., Liu, Z., Suryanarayanan, N. A. V., Zhou, G., Guo, J., Iba, H., and Tei, K. (2024). Large language models synergize with automated machine learning.

Zhang, L., Zhang, Y., Ren, K., Li, D., and Yang, Y. (2024). Mlcopilot: Unleashing the power of large language models in solving machine learning tasks.

A Appendix: Prompts Used for Model Training

```
I want to build a machine learning model using the dataset at "./dataset.csv".
This is a classification problem. The dataset has features like fixed acidity,
volatile acidity,citric acid,residual sugar,chlorides,free sulfur dioxide
total sulfur dioxide,density,pH,sulphates,alcohol,quality, and the target
column is "quality". Here's a sample of the data:

7.4,0.7,0,1.9,0.076,11,34,0.9978,3.51,0.56,9.4,5
7.8,0.88,0,2.6,0.098,25,67,0.9968,3.2,0.68,9.8,5
7.8,0.76,0.04,2.3,0.092,15,54,0.997,3.26,0.65,9.8,5

Follow these steps using Python code:
Single python script to implement the entire process.
Add comments to explain each step.
Define if this is classification or regression.
Load the dataset and describe it: shape, features, target.
Check for missing values, data types, and outliers.
Visualize feature distributions, correlations, and class balance.
Train these models: Random Forest, Decision Tree, SVM.
Choose hyperparameters and search ranges for each model.
Use Random Search or Bayesian Optimization for tuning.
Use cross-validation during hyperparameter search.
Evaluate models using F1-score, accuracy, and confusion matrix.
Compare model performance before and after tuning.
Show plots of evaluation metrics, search results, and best settings.
```

B Grading Criteria for Generated Plots (A–F)

Grade	Plotting Performance Criteria
A	**All required plots are present, clear, and well-labeled.** Plots visualize the data (feature distributions, correlations, class balance) *and* the modeling results (e.g., confusion matrix, model comparison, hyperparameter search results, feature importances). All plots help the user understand the process and outcome.
B	**Most plots included; minor omissions or lack of clarity.** All major results are visualized, but may be missing one of: feature importance, search results, or before/after comparisons. Plots are generally clear, with minor issues (labels, redundancy, or polish).
C	**Some major plots missing; model evaluation only partly visualized.** Most data visualizations are present, but at least one critical result plot is missing (e.g., confusion matrix only printed, or no model comparison). Results are not fully interpretable from plots alone.
D	**Only basic data plots included, almost all result/model visualizations missing.** Only the most basic data plots (e.g., a histogram or countplot) are included. Nearly all model evaluation and result plots are absent, and results are primarily printed.
E	**Minimal plotting; little to no data or result visualization.** May include a single basic plot (e.g., one histogram), or plots are present but unreadable or irrelevant. No meaningful visualization of model results or evaluation. Most information is printed as text.
F	**No relevant plots at all, or only code for plotting but nothing executed, or code did not compile at all.** No plots of data or results; everything is printed or missing entirely. Fails the requirement that plots are used for both data and model results.

Table 2: Grading Rubric for Plotting Quality

C Grading Criteria for Generated Code (A–F)

Grade	Code Performance Criteria
A	**Excellent, clean, complete, and error-free code**. Full pipeline: data handling, preprocessing, model training, evaluation, and optional interpretation. Modular, well-commented, with reproducible results.
B	**Good and mostly complete code with minor issues**. Includes all major parts of the pipeline but may lack refinement (e.g., limited tuning or basic structure). Code is readable and logical, with few issues.
C	**Partially complete or inconsistent code**. Code runs but may omit important steps (e.g., preprocessing or evaluation). Structure or clarity may be weak. Suboptimal modeling or redundant code is common.
D	**Minimal viable code with limited usefulness**. Includes only basic model training or evaluation. Major pipeline steps are missing. Code may run but lacks clarity and completeness.
E	**Non-functional or highly fragmentary code**. Code runs only in part or is too minimal to be useful. Many critical components are missing, with poor structure and explanation.
F	**Code does not run or is completely missing/incoherent**. Contains unrecoverable errors, undefined variables, or is unrelated to the task. Fails to compile or produce results.

Table 3: Code Grading Rubric. If the model used feedback, the assigned grade is reduced by one letter. Only models that failed to run entirely receive a grade of F.