MOCKINGBIRD: PLATFORM FOR ADAPTING LLMS TO GENERAL MACHINE LEARNING TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are now being used with increasing frequency as chat bots, tasked with the summarizing information or generating text and code in accordance with user instructions. The rapid increase in reasoning capabilities and inference speed of LLMs has revealed their remarkable potential for applications extending beyond the domain of chat bots. However, there is a paucity of research exploring the integration of LLMs into a broader range of intelligent software systems. In this research, we propose a paradigm for leveraging LLMs as *mock functions* to adapt LLMs to general machine learning tasks. Furthermore, we present an implementation of this paradigm, entitled the *Mockingbird* platform. In this paradigm, users define *mock functions* which are defined solely by method signature and documentation. Unlike LLM-based code completion tools, this platform does not generate code at compile time; instead, it instructs the LLM to role-play these *mock functions* at runtime. Based on the feedback from users or error from software systems, this platform will instruct the LLM to conduct chains of thoughts to reflect on its previous output, thereby enabling it to perform reinforcement learning. This paradigm fully exploits the intrinsic knowledge and in-context learning ability of LLMs. In comparison to conventional machine learning methods, following distinctive advantages are offered: (a) Its intrinsic knowledge enables it to perform well in a wide range of zero-shot scenarios. (b) Its flexibility allows it to adapt to random increases or decreases of data fields. (c) It can utilize tools and extract information from sources that are inaccessible to conventional machine learning methods, such as the Internet. Finally, we evaluated its performance and demonstrated the previously mentioned benefits using several datasets from Kaggle. Our results indicate that this paradigm is highly competitive.

1 INTRODUCTION

Currently, large language models (LLM) are widely used in intelligent systems as a chat bot to assist users in summarizing, processing, and generating text. However, this only exploits the natural language processing capabilities of LLMs; there are many other capabilities are not fully utilized, such as the significantly improved reasoning capabilities of recent LLMs (Valmeekam et al., 2024).

A natural idea is to extend LLMs to non-linguistic tasks. In 2020, Brown et al. (2020) has found that language models are able to learn from their context, proving the basic feasibility of this idea; since then, many researchers have studied the properties of this ability in specific domains: (Kirsch et al. (2022); Chan et al. (2022); Jin et al. (2023); Bigelow et al. (2024) Kossen et al. (2024)). These work have well explained the nature of in-context learning ability, but provide little overall insight into the integration of LLMs into automatic intelligent systems with a wider range of tasks. In order to fill this paucity and to encourage more domains to benefit from the progress of LLM, we propose *Mockingbird*, a versatile and controllable paradigm for adapting LLMs to general machine tasks, and a corresponding open-source platform implementing this paradigm.

Inspired by the finding that the intelligence of LLMs is merely role-playing (Shanahan et al., 2023), we assume that LLMs can also acquire good performance in role-playing functions, and design our paradigm upon this assumption. The fundamental component of *Mockingbird* is *mock function*, a specific type of functions that do not have method bodies, but only have function declaration including documentation and method signatures (contracts on the function parameters and return values). In this paradigm, users are able to use *mock functions* as if they were ordinary functions with function bodies. In contrast to other LLM-drive code generators, *Mockingbird* does not implement these *mock functions* with code generated by LLMs at compile time; instead, it instructs LLMs to ’role-play’ them at runtime.

To elaborate further, the platform furnishes LLMs with program metadata including the method signature and accompanying documentation, retrieved from the program; then, function calls on *mock functions* are redirected to LLMs; parameters are packed into user messages, and return values are unpacked from assistant messages. In this manner, LLMs are employed as general-purpose functions. By leveraging the in-context learning capability of LLMs, users can influence the behavior of these *mock functions* with minimal effort by modifying the input-output message pairs presented within the chat histories of LLMs. Furthermore, we present several optional techniques that enhance practicality of this paradigm. For instance, we introduce the substitution-script acceleration, which replaces LLM-driven executors with substitution-scripts generated by LLMs, thereby reducing the time consumption significantly. Figure 1 shows a high-level overview of this paradigm.

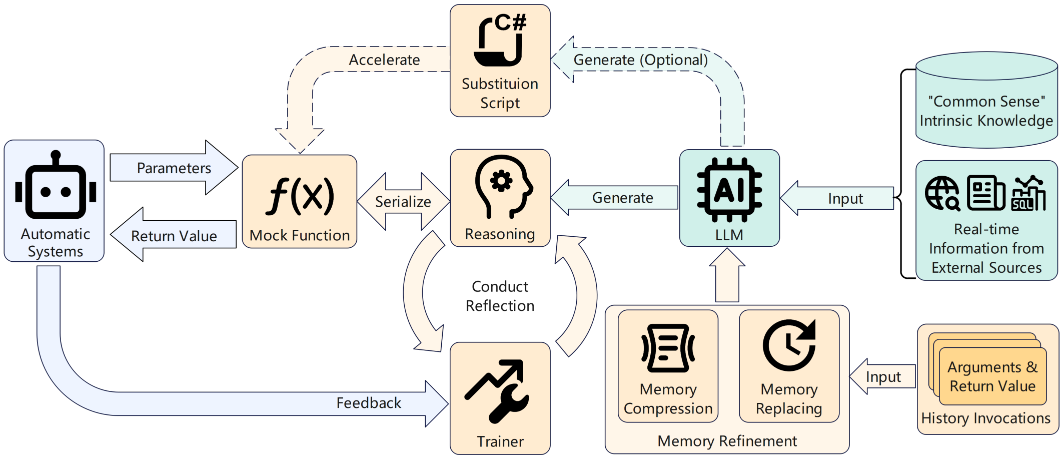


Figure 1: High-level overview of *Mockingbird*. Mock functions redirect ordinary function calls to the LLM, instructing the LLM to initiate reasoning and then generate the return value. Mock trainers use feedback to instruct the LLM to conduct reflection process on its previous errors. Optional modules such as substitution script, memory compression and memory replacing are introduced to further improve the practicality of this paradigm.

This paradigm offers an intuitive interface for systems that have difficulty adapting to chat-driven execution mode, thereby enabling them to leverage the capabilities of LLMs.

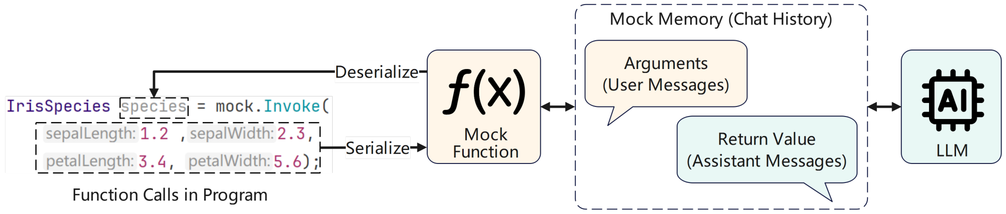
This work investigates the potential of applying this paradigm to general machine learning tasks. Furthermore, we implement an extensible machine learning framework for *mock functions*, which is capable of automatically conducting reflection on the incorrect output during the training process. In comparison to conventional machine learning methods, such as those based on statistics or artificial neural networks, *Mockingbird* has the following distinctive advantages (a) the intrinsic knowledge of LLMs acquired from the pre-training data enables this paradigm to perform well on few-shot and zero-shot tasks; (b) this paradigm is not constrained by a strict input data schema, allowing it to process incomplete data entries with missing fields. (c) this paradigm is readily capable of utilising tools and extracting information from non-structural sources, which are typically inaccessible to conventional machine learning techniques. The aforementioned advantages can serve to enhance the robustness and flexibility of automatic intelligent systems.

We evaluate the general applicability of this paradigm across a range of machine learning tasks from Kaggle. Overall, it achieves very competitive scores compared to conventional machine learning methods, and even outperforms many human competitors on several datasets.

2 MOCKINGBIRD

This paradigm is composed of following components:

Mock Function A mock function is a function that is defined solely in terms of its method signature (types and other restrictions on parameters and return value), with optional documentation provided to describe its purpose and any remarks pertinent to its use. In contrast to conventional functions, *mock functions* are not implemented by users or code generators at the compilation stage. Instead, they are role-played by LLMs at run-time. Except for this different behind the scene, the manner in which users interact with the system is identical to that of conventional functions. As for other necessary procedures, such as the JSON serialization, are automatically handled by *mock functions*. The fundamental responsibilities of *mock functions* can be summarized as follows: (a) providing LLMs with metadata to role-play the function; (b) handling conversions between program objects and chat messages in JSON format; (c) guaranteeing the formal correctness of responses generated by LLMs through the validation of JSON schema. Figure 2 illustrates the fundamental workflow of a mock function. Mock functions can be employed directly without training process; however, in such instances, it is advisable to provide comprehensive annotation regarding the anticipated behaviour of these functions as documentation comments within the code itself.



```
IrisSpecies species = mock.Invoke(
    sepalLength: 1.2 , sepalWidth: 2.3,
    petalLength: 3.4,  petalWidth: 5.6);
```

Deserialize
Serialize
Function Calls in Program
Mock Function
Mock Memory (Chat History)
Arguments (User Messages)
Return Value (Assistant Messages)
LLM

Figure 2: Basic workflow of a mock function. Mock functions automatically handle the conversion between program objects and chat messages, thus communicating between the program and LLMs.

Mock Invocation Mock invocations are designed to conveniently update history invocations in the context. They represent a mapping from invocations in program space to request-response message pairs in chat history. Mock invocations automatically update the content of request and response messages in the chat history when the their parameters or return values are changed. They serves as a simple interface to edit the context.

Mock Memory Mock memories are enhanced chat histories for this paradigm, whose elements are mock invocations instead of chat messages. Additionally, a branch control feature has been implemented for mock memories. This feature allows users to create sub-branches from a main branch, which will "inherit" the current invocation history of the main branch. Sub-branches can be dropped or committed back to the creation time-point in the main branch. Figure 3 depicts the process of creating sub-branches and committing them back to the main branch. This feature provides isolation among different branches, thereby enabling the execution of multiple LLM tasks in parallel; otherwise, no further requests can be added to the memory until the response is received, as LLMs may respond to requests from other tasks.

Mock Trainer The introduction of mock trainers serves to streamline the process of training and evaluating mock functions, while also assisting users of conventional machine learning frameworks in swiftly adapting to our paradigm. Once the data source, training policy, evaluation metrics and dataset separation ratio for training and evaluation have been set, the system will automatically handle the training and evaluation process. In the training stage, if LLMs output incorrect answers in a mock invocation, a reflection procedure will be conducted by the mock trainer to avoid making similar mistakes in future. The reflection procedure will be covered in detail in section 2.3.

Substitution script. A substitution script is a script generated by LLM to represent the behavior learned from the invocation history. This technique is introduced as an optional feature to reduce the time consumption, though this may result in a compromise of accuracy.

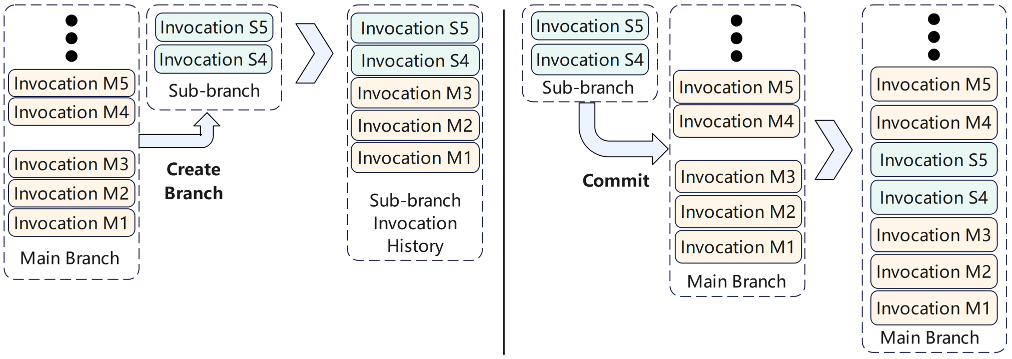


Figure 3: Branch control feature of mock memories. **Left**: when a branch is created, subsequent invocations added to the main branch will not affect the sub-branch. **Right**: when a sub-branch is committed, its invocations are merged into the main branch at the location where the sub-branch was created.

2.1 WORKFLOW

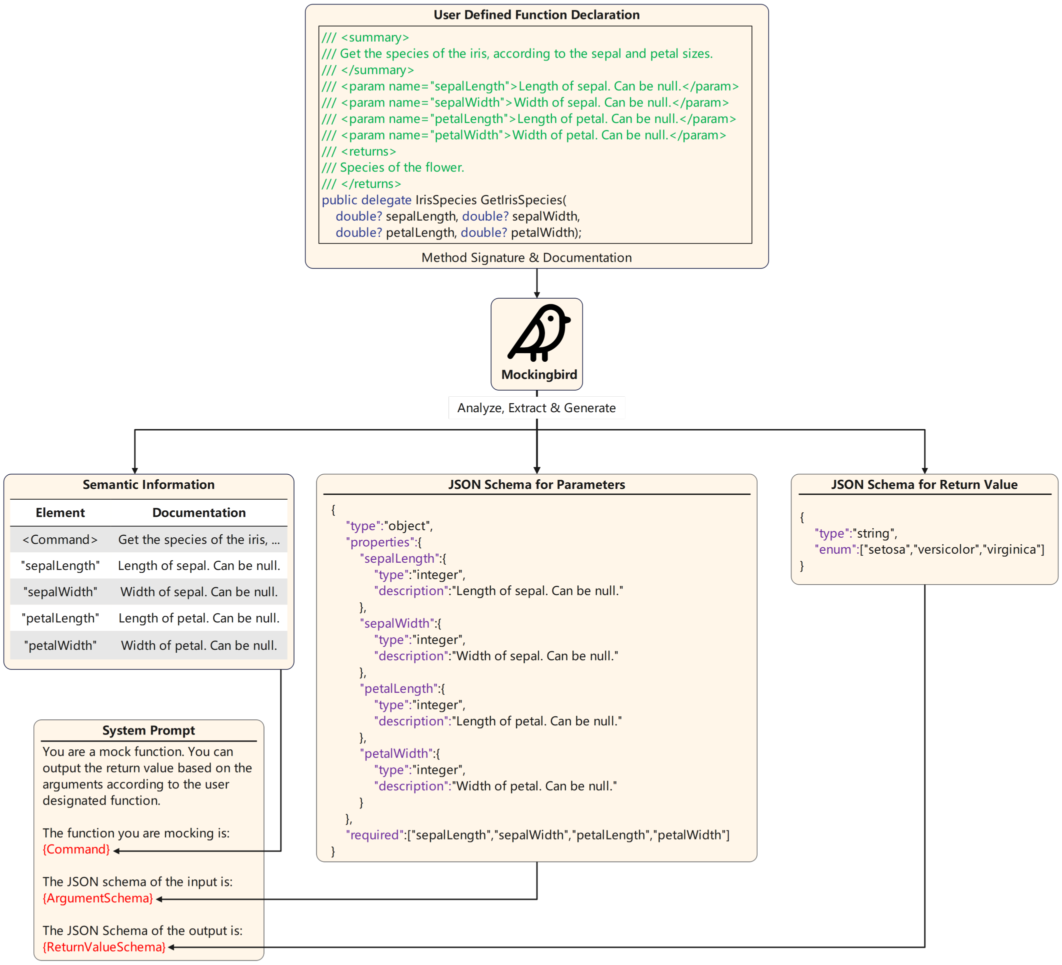
The following list delineates the machine learning workflow with *Mockingbird*:

1) **Setup mock function.** According to the metadata including method signature, documentation, and type information of parameters and return value, the mock function generate the system prompt consisting of directives and JSON schemas separately for parameters and return value. This prompt not only instructs the LLM to give a return value that matches the JSON schema, but also commands it to output the reasoning in the "remarks" field before outputting the return value in the "results" field. Mock functions add this system prompt into the message history during the setup phase. Details are discussed in section 2.2.
2) **Prepare serializers.** According to the type information collected in step 1, the platform generates and code of JSON serializers for parameters and return value, which are then loaded subsequent to their compilation.
3) **Perform an invocation with a LLM.** Once the parameters have been serialized into JSON data that adheres to the schema built in step 1, the JSON data should be appended to the message history as a user message. The message history should then be submitted to the LLM for a response. Upon receipt of the assistant message, mock functions decode return value from the JSON data contained within the response. Then a mock invocation is constructed and registered with the user message and the corresponding assistant message.
4) **Conduct reflection procedure.** In the training phase, the mock trainer compares the return value yielded by the assistant with the ground truth in the training data. If the difference exceeds the threshold preset by users, a reflection procedure will be initiated by the mock trainer. Parameters, return value, and the ground truth will be appended to a sub-branch of the main memory branch. Then the LLM will be instructed to reason why such mistakes are made and summarize notes to avoid making similar mistakes in the future. These notes are called "reflection notes" in this paradigm.
5) **Update invocations in the mock memory.** Once the reflection notes have been obtained, mock invocations which contain the wrong return values are updated. Their "results" fields are amended to the ground truth, while their "remarks" fields are replaced with reflection notes.
6) **Refine the mock memory.** The context length is typically smaller than the size of the training dataset, which means that not all data entries of the training dataset can be directly stored in the context. When the context of a mock function reaches the limitation of length, a memory refinement procedure will be initiated by the mock trainer. We provide a memory replacing policy and a memory compression policy (see section 2.4 for details) along with our implementation of this paradigm. Mock trainers are designed to be highly customizable, allowing users to configure them with their own memory refinement policies.
7) **Generate substitution script.** In contrast to other LLM-drive code generators which generate code at the outset, the substitution script is generated only after the LLM has acquired sufficient information about the "correct" behavior of the mock function. This distinguishes it from other

compile-time code generators. Once the substitution script has been generated and compiled, subsequent invocations will be redirected to the substitution script instead of the LLM. It should be noted that the content of the substitution script is not definitive, as the reflection process will invalidate the script, which always assuming the behavior of the mock function has been changed by the reflection process. This technique can reduce the time consumption to the level of native ordinary functions. However, it may result in a degree of compromise in terms of accuracy, and therefore we have made it an optional feature.

2.2 GUARANTEE OF FORMAL CORRECTNESS

Figure 4 shows the process of building system prompt for mock functions. The platform retrieves the documentation for the delegate (function declaration) from the program documentation file generated by the compiler, and extracts the semantic meanings for this delegate, parameters and return value. After that, JSON schemas for parameters and return values are generated separately with the type information acquired from the runtime. Semantic meanings are added into "description" fields of the corresponding elements in schemas. All these schemas are defined in the standard format of JSON schema, rather than expressed in natural languages.


User Defined Function Declaration
Method Signature & Documentation
Mockingbird
Analyze, Extract & Generate
Semantic Information
JSON Schema for Parameters
JSON Schema for Return Value
System Prompt

```
/// <summary>
/// Get the species of the iris, according to the sepal and petal sizes.
/// </summary>
/// <param name="sepalLength">Length of sepal. Can be null.</param>
/// <param name="sepalWidth">Width of sepal. Can be null.</param>
/// <param name="petalLength">Length of petal. Can be null.</param>
/// <param name="petalWidth">Width of petal. Can be null.</param>
/// <returns>
/// Species of the flower.
/// </returns>
public delegate IrisSpecies GetIrisSpecies(
    double? sepalLength, double? sepalWidth,
    double? petalLength, double? petalWidth);
```

Element	Documentation
<Command>	Get the species of the iris, ...
"sepalLength"	Length of sepal. Can be null.
"sepalWidth"	Width of sepal. Can be null.
"petalLength"	Length of petal. Can be null.
"petalWidth"	Width of petal. Can be null.

```
{
  "type":"object",
  "properties":{
    "sepalLength":{
      "type":"integer",
      "description":"Length of sepal. Can be null."
    },
    "sepalWidth":{
      "type":"integer",
      "description":"Width of sepal. Can be null."
    },
    "petalLength":{
      "type":"integer",
      "description":"Length of petal. Can be null."
    },
    "petalWidth":{
      "type":"integer",
      "description":"Width of petal. Can be null."
    }
  },
  "required":["sepalLength","sepalWidth","petalLength","petalWidth"]
}
```

```
{
  "type":"string",
  "enum":["setosa","versicolor","virginica"]
}
```

You are a mock function. You can output the return value based on the arguments according to the user designated function.

The function you are mocking is:
{Command}

The JSON schema of the input is:
{ArgumentSchema}

The JSON Schema of the output is:
{ReturnValueSchema}

Figure 4: The system prompt for mock functions are built according to the semantic information and JSON schema for parameters and return value. Semantic information explains the semantic meanings of parameters, which is crucial for LLMs to understand the task and parameters. JSON schemas for parameters and return values are used to constrain the layout and semantic meanings of data fields, ensuring the correct mutual understanding of data fields between program and LLMs.

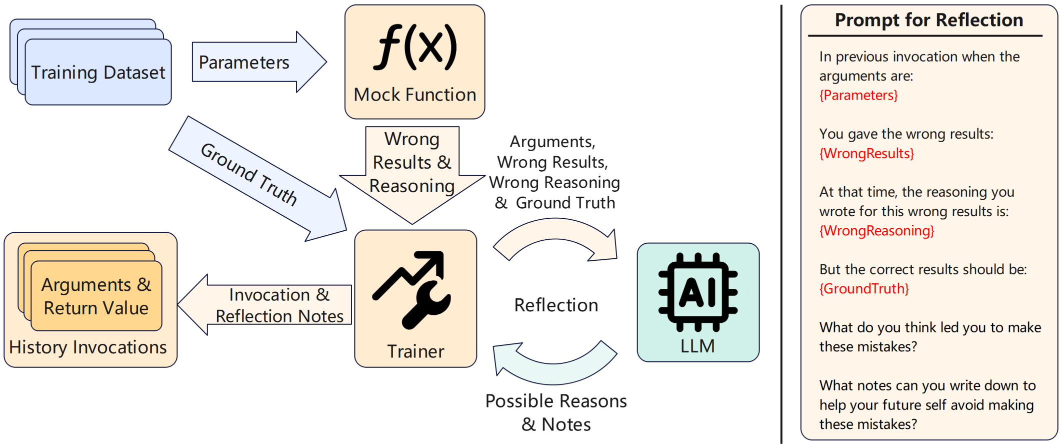
When instructions on the format of responses are described in natural languages, there is a possibility of a mismatch between the format of the responses and the expected format due to the ambiguity in the instructions. Even if instructions are clear without ambiguity, LLMs may still fail to obey these instructions due to the overlay of reasoning patterns (Jiang et al., 2024a). Solving this formal randomness is at vital importance: in contrast to user-oriented applications, formal correctness of responses are crucial for automatic systems. The return value cannot be correctly found and parsed in one request-response round if it is stored in field with random names. Furthermore, even if automatic systems discover an error with the schema, re-generation of responses will significantly increase the time consumption.

A number of studies have put forward the idea of fine-tuning-based solutions as a means of improving the ability of LLMs to follow instructions (Wallace et al. (2024);Jiang et al. (2024a)). But with an extra emphasis on general availability, we prefer to rely on non-fine-tuning solutions. Recent LLMs, such as *GPT-4o* has provided *structural output* feature that strictly limits the schema of the output. *Mockingbird* fully exploit this feature to guarantee the formal correctness of responses. For LLMs which currently do not support this feature, *Mockingbird* uses a JSON schema validator to verify the formal correctness of responses, and reject illegal responses with detailed error report to initiate another request-response round. For users using this paradigm with self-host LLMs, we highly recommend them to fine-tune their models to schema control instructions.

2.3 LEARNING THROUGH REFLECTIONS

Even though our experiments show that the intrinsic knowledge LLMs acquired from pre-training enables them to reach competitive scores on multiple tasks with zero-shot, the capacity for continuous improvement still remains a crucial consideration in practical applications.

Inspired by the workflow of neural network based machine learning frameworks, we introduce a reflection mechanism into *Mockingbird*. Figure 5 shows the workflow of reflection and the prompt used by mock trainers to instruct the LLM for reflection to perform reflections.



Prompt for Reflection

In previous invocation when the arguments are:
{Parameters}

You gave the wrong results:
{WrongResults}

At that time, the reasoning you wrote for this wrong results is:
{WrongReasoning}

But the correct results should be:
{GroundTruth}

What do you think led you to make these mistakes?

What notes can you write down to help your future self avoid making these mistakes?

Figure 5: **Left:** Workflow for reflection procedure. After acquiring the wrong results from the mock function, the mock trainer will instruct the LLM to reflect on the possible reasons for making such mistakes and summarize notes for not making similar mistakes in future invocations. **Right:** Prompt used by mock trainers to conduct reflections.

We implement mock trainers to automatically host and manage the learning procedure. In a training session, the mock trainer will iterate every data entry in the dataset, feeding parameters into the mock function, comparing the results obtained from the mock function with the ground truth. If the difference between the output results and ground truth exceeds the threshold preset by users, then a reflection procedure will be conducted: the mock trainer will generate a reflection instruction including the wrong results, wrong reasoning and the ground truth; thereafter, the LLM will be instructed to analyze the possible reasons for making these mistakes, and summarize notes to avoid making similar mistakes. The response from the LLM, consisting of possible reasons and notes for avoiding similar mistakes, is called “reflection notes” in this paradigm. Then, results field of the invocation to reflect will be corrected to the ground truth, and the reasoning field will be replaced with reflection notes.

In contrast to previous methods of in-context learning which focus more on the patterns of examples in the context (Liu et al. (2022);Bhattamishra et al. (2024)), we expect LLMs to reflect on their previous reasoning flows and learn by adapting to reasoning flows which are more likely to be correct, rather than relying solely on the in-context learning ability of LLMs.

2.4 MEMORY REPLACING AND COMPRESSION

Under a simplified assumption that all examples take equal tokens, denoted as m, and adding n examples into the context during the training procedure, then the total token consumption is $mn^2 + 0.5mn$, which shows it is economically impractical to have context of examples as long as sufficient. This leads to the urgent need for strategies to dynamically replace previous examples with examples that are more worth learning and remembering, or to compress the context as little loss of information as possible.

Many researches have proposed strategies for selecting examples (Zhang et al. (2022); Zhang et al. (2023); Lee et al. (2023); Qin et al. (2024); Nguyen & Wong (2023); Peng et al. (2024); S. et al. (2024)). These studies all show competitive improvement on specific domains, so from the perspective of paradigm design, instead of select one or more selection strategy, we implement the mock trainer in a highly customizable design. We provide a default implementation of the replacing policy by replacing correct invocations (without reflection notes) with the latest reflected invocations.

As for the context compression, current methods focus more on token level (Liskavets et al. (2024);Ge et al. (2024)) and even on the level of modifying the underlying implementation of transformers (Yu et al. (2024)). According to their evaluation results, these methods are effective enough, but they are too heavy for *Mockingbird*, since this paradigm is designed to work simultaneously with different LLMs at various price tiers. In addition, there is an undeniable difficulty in using these solutions with commercial close-source LLMs for the time being. Similar to methods proposed by Wang et al. (2024) and Jiang et al. (2024b), we provide a default implementation of semantic compression (“soft compression”), by instructing LLMs to summarize the reflection notes to compress previous invocations.

3 EVALUATION

We evaluate the performance of *Mockingbird* on several general machine learning tasks, covering the most common categories of classification and regression. It must be emphasised that our aim is to provide a more suitable paradigm for exploiting the capabilities of LLMs in automatic intelligent systems, rather than provide a *state-of-the-art* method that is comprehensively superior to conventional machine learning methods.

3.1 PERFORMANCE EVALUATION

In this section, we evaluate its performance separately on classification and regression datasets acquired from Kaggle; its performance is assessed by comparing with scores of human competitors in corresponding Kaggle leaderboard[1], whose scores can be considered as an estimated upper bound of the scores that non-professional users can achieve using conventional machine learning methods.

Table 1 shows its performance on classification tasks, and table 2 is for performance on regression tasks. Context length is the maximum number of invocations that the mock function can memorize through the training process; a context length of 0 means that the training process is skipped, which could represent the extent to which users can treat this paradigm as an “out-of-the-box” solution. Both evaluations are performed with *GPT-4o* as the underlying LLM.

---

[1] Raw leaderboard data can be found in supplementary materials. Data of leaderboard is updated to Oct 1st, 2024.

Table 1: Accuracy evaluation on classification tasks, compared with scores of human competitors from Kaggle leaderboard. Datasets are: *Titanic Survival Prediction* (Cukierski, 2012), *Poisonous Mushrooms Classification* (UCI, 1981), *Horse Colic* (McLeish & Cecile, 1989), *Obesity Risk Prediction* (Fabio Mendoza Palechor and, 2023) and *Insurance Cross-Selling*. **Humans' Best** is the best scores achieved by human competitors in the leaderboard; and **Outperformed** is the proportion of human competitors whose scores are outperformed by the best scores using this paradigm. The best accuracy is underlined.

Dataset	**Accuracy↑ with Context Length**					**Humans' Best**	**Outperformed**
	0	20	40	60	80		
Titanic	0.7879	0.7887	0.7743	0.7918	0.7866	1.0000	92.29%
Mushrooms	0.3720	0.5693	0.7302	0.9968	0.8359	0.9851	100%
Horses	0.5450	0.5531	0.5836	0.5532	0.5087	0.7818	6.42%
Insurances	0.5880	0.6730	0.7260	0.6989	0.6793	0.8975	18.35%

Table 2: Root mean square error (RMSE) and median absolute error (MedAE) evaluation on regression tasks, compared with scores of human competitors from Kaggle leaderboard. Datasets are: *Used Car Price Prediction*, *Abalone Dataset* (Nash et al., 1994), *Wild Blueberry Yield Prediction* and *Prediction of Mohs Hardness*. **Humans' Best** is the best scores achieved by human competitors in the leaderboard; and **Outperformed** is the proportion of human competitors whose scores are outperformed by the best scores using this paradigm. The best scores are underlined.

Dataset	**Error↓ with Context Length**					**Humans' Best**	**Outperformed**
	0	20	40	60	80		
Car Prices	22225	22523	28915	24534	22794	62917	100%
Mohs Hardness (MedAE)[†]	0.6000	0.5800	0.6000	-[‡]	-[‡]	0.2500	60.86%
Mohs Hardness (RMSE)	1.2803	1.1596	1.1314	-[‡]	-[‡]	-[†]	-[†]

[†] The leaderboard on Kaggle uses median absolute error to rank human competitors.
[‡] This dataset only contains 57 data entries.

3.2 DISCUSSION

Longer context is not all you need. Among these evaluations, we find that best performances are not guaranteed to be accompanied with the biggest context length. For instance, on *Poisonous Mushroom Classification* task, the performance peak appears around a context length of 40, and the performance decreases by 16.14% when the context length grows to 80. There is an extreme example on *Mohs Hardness Prediction* task. When the context length is 40, while 70.18% of the data entries are reflected and then stored in the context, the RMSE only reduces by 2.43%. This phenomenon is not as incomprehensible as it seems to be. Kossen et al. (2024) has pointed out that LLMs do not treat all data within the context equally, but tends to rely on examples semantically closer to the queries; also, they cannot fully resist their preference acquired through the pre-training, which is also shown in the performance plateau on *Titanic Survival Prediction Dataset*. In other words, intrinsic knowledge and the reasoning ability have made LLMs more than in-context learning machines; this leads us to reconsider the characteristics of LLMs machine learning, rather than continuing to view it simply as combining in-context learning of transformers with intrinsic knowledge.

Intrinsic knowledge does not always help. On *Poisonous Mushroom Classification* task, we observed a strange initial accuracy bellowing random guessing: 0.3720, which is 25.6% worse than random guessing. In this configuration, the context length is zero, which means there is no history invocations or reflection notes stored in the context which could influence its behavior, therefore its judgment is solely relying on its intrinsic knowledge. If we humans know the fact that our accuracy is significantly bellowing 0.5, then by simply reversing our final answers, we can reach a relatively high accuracy on such binary classification tasks. However, even when LLMs know they are outputting the wrong answer, they still need a sufficient examples and reflections in learning process to reduce the influence of their "harmful" intrinsic knowledge; in this task, that is the process of accuracy increasing from 0.3720, going through 0.5693, 0.7302, and finally reaches a high score outperforming the best score of human competitors.

Why cannot LLMs be "a good veterinarian"? In some cases LLMs struggle to learn. The performance of LLMs on *Horses Health Outcome Prediction* is especially bad. Similar to their performances on *Insurance Cross-selling Prediction*, they starts with an accuracy of nearly random guessing. Their accuracy improves as the size of training dataset grows on *Insurance Prediction* task until reaches the context length of 40, however, on *Horses Health Prediction* task, such improvement is not observed. An obvious difference between these tasks is the huge gap in the number of their data fields. There are 29 data fields in *Horses Health Prediction* task but only 12 data fields in *Insurance Prediction* task, in other words, the former task is inherently more complicated than the latter one. Despite this natural inference, by studying the operational log, we spotted that at the end of the training phase, 75.0% of invocations within the context on *Insurance Prediction* are initially correct invocations (LLMs gave the right answers for parameters contained in these invocations), while on *Insurance Prediction* task only 47.5% of the invocations are correct. This does not have to mean that have a context with a higher proportion of correct invocations can improve the performance. Based on the findings that current LLMs cannot discover the errors within their reasoning process in the absence of external feedback (Huang et al., 2024), we speculate that the reason it cannot learn on that task is that it is not really "learning" on that task. Being much more complicated is not the core problem behind that phenomenon. Its context length is equal to the size of training dataset, which means no invocations are replaced or dropped; then have a correct ratio of 47.5% means it outputs more wrong answers than the correct answers in the training stage. Ground truth from the training dataset can only indicates the existence of errors within the reasoning flows; what exactly these errors are requires LLMs to figure out through reasoning. However, there is no feedback for the reasoning process of looking for errors within their previous reasoning process. If they are initially bad at this kind of tasks, then chances are that they cannot reason out the errors within their previous reasoning process and consequently have no improvement through the whole training stage. During the training stage, they are guessing what their errors could be or what is the correct reasoning flow. The random guess within the previous reflection notes may also hinder their following reasoning and reflection. This finding and speculation warns us that we should not rely solely on the in-context learning ability of LLMs; also, having a wider intrinsic knowledge is beneficial because that may help LLMs avoiding these "guessing traps".

4 RELATED WORK

Adapting LLMs to Machine Learning Tasks Recently, Kashyap & Sinha (2024) also examined the feasibility of integrating LLMs into statistical learning workflows on "Titanic Survival Prediction" dataset; they claimed to reach a "significant improvement" with data preprocessing and thought refinement (chain of thought). However, our experiments show that the scores are relatively higher without data pre-processing and other additional steps introduced in their work, which means that their method is less effective than they claimed to be. Moreover, their method relies heavily on the user's knowledge of statistical learning, which is not friendly to the automatic intelligent systems.

In-Context Learning Brown et al. (2020) have discovered that LLMs are able to learn from analogies based on the context, and summarize this ability as "in-context learning". Based on solid experimental evidence, Kossen et al. (2024) draws the following important conclusions about in-context learning: (a) content in context can influence the behavior of LLMs, which means that in-context learning is indeed effective to some extent; (b) examples in the context that are semantically closer to the queries have more effect, and vice versa; this makes in-context learning different from conventional machine learning methods that treat all training data equally; (c) it is impossible to eliminate the preference that LLMs acquired through pre-training by in-context learning, but this preference can be reduced to some extent by prompting. These conclusions are consistent with our findings in experiments.

Code Generation and Self-Repair The delayed progressive generation of substitution scripts distinguished it from other code generation solutions, for it does not rely on self-repair, but incorporates information from the parameter-result pairs and corrects its behavior through reflections with external feedback. Olausson et al. (2023) have shown that LLMs struggle to repair the errors on their own, but that the effectiveness of repairs improves significantly when human feedback is provided. Also, Huang et al. (2024) have demonstrated that the reasoning performance cannot be improved without external feedback. Our design of deferring the substitution script generation is consistent with these findings. Meanwhile, compared to user assisted code generation solutions (Zhu-Tian et al. (2024); Fakhoury et al. (2024)), the feedback of substitution scripts can be provided by software systems rather than solely from humans users, which is more suitable to automatic intelligent systems.

5 CONCLUSION

In this work, we present the paradigm for adapting LLMs to general machine learning tasks based on *mock functions*, which instruct LLMs to role-play the function defined by users. We also propose a machine learning framework for mock functions, whose usage is similar to that of conventional neural network based machine learning frameworks. Optional techniques, including substitution script generation, memory refinement policies for replacing and compression are also introduced to address its shortcomings in inference cost and time consumption. This paradigm can fully exploit the reasoning ability, in-context learning feature, and intrinsic knowledge of LLMs, together with its dynamic nature, making it an out-of-the-box solution for automatic intelligent systems.

Finally, we evaluate its performance and scalability on several machine learning tasks from Kaggle, and it shows competitive results, even outperforming most the human competitors in some tasks. Meanwhile, we discuss several findings about LLM machine learning based on the analysis of the evaluation results and platform operational logs, we emphasis these findings: (a) longer context does not guarantee to yield better results, which indicates that characteristics of LLM machine learning is essentially different from Transformers in-context learning; (b) intrinsic knowledge is not always helpful, in some case they are harmful to the performance and LLMs need overcome them through learning and reflection; (c) chances are that LLM machine learning may encounters a “guessing trap”, where they struggle to learn through reflection due to failed to discover what their mistakes exactly are.

Limitation and Future Work Currently, it is not financially feasible to evaluate our paradigm with all possible types of datasets on all possible machine learning tasks to fully and specifically identify its suitable and unsuitable domains; therefore, we believe that validating this paradigm in many specific domains could still provide useful insights on characteristics of LLM machine learning. Moreover, the application potential of *Mockingbird* is limited by the current money cost and time consumption of LLM inference. However, there is a trend towards real-time LLM inference such as *Cerebras Inference Engine* (Zhang et al., 2024), we would like to explore the possibility of applying this paradigm to low-latency systems with real-time LLM inference. Also, the performance of this paradigm is currently only close to best scores of human competitors with conventional machine learning methods, but by adapting current automatic machine learning methods such as *AutoGluon* (Erickson et al., 2020), there could be a possibility for *Mockingbird* to reach the same or even higher scores than the top scores of human competitors.

REFERENCES

Satwik Bhattamishra, Arkil Patel, Phil Blunsom, and Varun Kanade. Understanding in-context learning in transformers and LLMs by learning to learn discrete functions. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=ekeyCgeRfC`.

Eric J Bigelow, Ekdeep Singh Lubana, Robert P. Dick, Hidenori Tanaka, and Tomer Ullman. In-context learning dynamics with random binary sequences. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=62K7mALO2q`.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL `https://arxiv.org/abs/2005.14165`.

Stephanie Chan, Adam Santoro, Andrew K. Lampinen, Jane Wang, Aaditya Singh, Pierre H. Richemond, James L. McClelland, and Felix Hill. Data distributional properties drive emergent in-context learning in transformers. In *NeurIPS*, 2022. URL `http://papers.nips.cc/paper_files/paper/2022/hash/77c6ccacfd9962e2307fc64680fc5ace-Abstract-Conference.html`.

Will Cukierski. Titanic - machine learning from disaster, 2012. URL `https://kaggle.com/competitions/titanic`.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data, 2020. URL `https://arxiv.org/abs/2003.06505`.

Alexis de la Hoz Manotas Fabio Mendoza Palechor and. Obesity or cvd risk (classify/regressor/cluster), 2023. URL `https://www.kaggle.com/dsv/7009925`.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation, 2024. URL `https://arxiv.org/abs/2404.10100`.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model, 2024. URL `https://arxiv.org/abs/2307.06945`.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2024. URL `https://arxiv.org/abs/2310.01798`.

Gangwei Jiang, Caigao Jiang, Zhaoyi Li, Siqiao Xue, Jun Zhou, Linqi Song, Defu Lian, and Ying Wei. Interpretable catastrophic forgetting of large language model fine-tuning via instruction vector, 2024a. URL `https://arxiv.org/abs/2406.12227`.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2024b. URL `https://arxiv.org/abs/2310.06839`.

Ming Jin, Shiyu Wang, Lintao Ma, Zhixuan Chu, James Y. Zhang, Xiaoming Shi, Pin-Yu Chen, Yuxuan Liang, Yuan-Fang Li, Shirui Pan, and Qingsong Wen. Time-llm: Time series forecasting by reprogramming large language models. *CoRR*, abs/2310.01728, 2023. URL `https://doi.org/10.48550/arXiv.2310.01728`.

Yuktesh Kashyap and Amrit Sinha. Llm is all you need : How do llms perform on prediction and classification using historical data. *International Journal For Multidisciplinary Research*, 2024. URL `https://doi.org/10.36948/ijfmr.2024.v06i03.23438`.

Louis Kirsch, James Harrison, Jascha Sohl-Dickstein, and Luke Metz. General-purpose in-context learning by meta-learning transformers. *CoRR*, abs/2212.04458, 2022. URL `https://doi.org/10.48550/arXiv.2212.04458`.

Jannik Kossen, Yarin Gal, and Tom Rainforth. In-context learning learns label relationships but is not conventional learning. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=YPIA7bgd5y`.

Yoonsang Lee, Pranav Atreya, Xi Ye, and Eunsol Choi. Crafting in-context examples according to lms' parametric knowledge. *CoRR*, abs/2311.09579, 2023. URL `https://doi.org/10.48550/arXiv.2311.09579`.

Barys Liskavets, Maxim Ushakov, Shuvendu Roy, Mark Klibanov, Ali Etemad, and Shane Luke. Prompt compression with context-aware sentence encoding for fast and improved llm inference, 2024. URL `https://arxiv.org/abs/2409.01227`.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for GPT-3? In Eneko Agirre, Marianna Apidianaki, and Ivan Vulić (eds.), *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pp. 100–114, Dublin, Ireland and Online, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.deelio-1.10. URL `https://aclanthology.org/2022.deelio-1.10`.

Mary McLeish and Matt Cecile. Horse Colic. UCI Machine Learning Repository, 1989. DOI: https://doi.org/10.24432/C58W23.

Warwick Nash, Tracy Sellers, Simon Talbot, Andrew Cawthorn, and Wes Ford. Abalone. UCI Machine Learning Repository, 1994. DOI: https://doi.org/10.24432/C55C7W.

Tai Nguyen and Eric Wong. In-context example selection with influences, 2023. URL `https://arxiv.org/abs/2302.11042`.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *ICLR 2024*, June 2023. URL `https://www.microsoft.com/en-us/research/publication/is-self-repair-a-silver-bullet-for-code-generation/`.

Keqin Peng, Liang Ding, Yancheng Yuan, Xuebo Liu, Min Zhang, Yuanxin Ouyang, and Dacheng Tao. Revisiting demonstration selection strategies in in-context learning, 2024. URL `https://arxiv.org/abs/2401.12087`.

Chengwei Qin, Aston Zhang, Chen Chen, Anirudh Dagar, and Wenming Ye. In-context learning with iterative demonstration selection, 2024. URL `https://arxiv.org/abs/2310.09881`.

Vinay M. S., Minh-Hao Van, and Xintao Wu. In-context learning demonstration selection via influence analysis, 2024. URL `https://arxiv.org/abs/2402.11750`.

Murray Shanahan, Kyle McDonell, and Laria Reynolds. Role play with large language models. *Nature*, 623(7987):493–498, 2023.

UCI. Mushroom. UCI Machine Learning Repository, 1981. https://doi.org/10.24432/C5959T.

Karthik Valmeekam, Kaya Stechly, and Subbarao Kambhampati. Llms still can't plan; can lrms? a preliminary evaluation of openai's o1 on planbench, 2024. URL `https://arxiv.org/abs/2409.13373`.

Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The instruction hierarchy: Training llms to prioritize privileged instructions, 2024. URL `https://arxiv.org/abs/2404.13208`.

Cangqing Wang, Yutian Yang, Ruisi Li, Dan Sun, Ruicong Cai, Yuzhu Zhang, and Chengqian Fu. Adapting llms for efficient context processing through soft prompt compression. In *Proceedings of the International Conference on Modeling, Natural Language Processing and Machine Learning*, CMNM '24, pp. 91–97, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400709760. doi: 10.1145/3677779.3677794. URL `https://doi.org/10.1145/3677779.3677794`.

Hao Yu, Zelan Yang, Shen Li, Yong Li, and Jianxin Wu. Effectively compress kv heads for llm, 2024. URL `https://arxiv.org/abs/2406.07056`.

Yiming Zhang, Shi Feng, and Chenhao Tan. Active example selection for in-context learning, 2022. URL `https://arxiv.org/abs/2211.04486`.

Yuanhan Zhang, Kaiyang Zhou, and Ziwei Liu. What makes good examples for visual in-context learning? In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 17773–17794. Curran Associates, Inc., 2023. URL `https://proceedings.neurips.cc/paper_files/paper/2023/file/398ae57ed4fda79d0781c65c926d667b-Paper-Conference.pdf`.

Zuoning Zhang, Dhruv Parikh, Youning Zhang, and Viktor Prasanna. Benchmarking the performance of large language models on the cerebras wafer scale engine, 2024. URL `https://arxiv.org/abs/2409.00287`.

Chen Zhu-Tian, Zeyu Xiong, Xiaoshuo Yao, and Elena Glassman. Sketch then generate: Providing incremental user feedback and guiding llm code generation through language-oriented code sketches, 2024. URL `https://arxiv.org/abs/2405.03998`.