

Appendix

A Prompts for MLE-STAR

A.1 Retriever agent

```
# Competition
{task description}

# Your task
- List {M} recent effective models and their example codes to win the above competition.

# Requirement
- The example code should be concise and simple.
- You must provide an example code, i.e., do not just mention GitHubs or papers.

Use this JSON schema:
Model = {'model_name': str, 'example_code': str}
Return: list[Model]
```

Figure 9: Prompt used for retrieving task-specific models using web search.

MLE-STAR starts by generating an initial solution. Here, we propose using web search as a tool for MLE-STAR first to retrieve M state-of-the-art models for the given task. Specifically, MLE-STAR leverages a retriever agent $\mathcal{A}_{\text{retriever}}$ with the above prompt (Figure 9). $\mathcal{A}_{\text{retriever}}$ takes task description $\mathcal{T}_{\text{task}}$ as input and retrieves M pairs of $\{\mathcal{T}_{\text{model}}, \mathcal{T}_{\text{code}}\}$. Here, we guide MLE-STAR to generate the retrieved result as structured output (*i.e.*, JSON). After we obtain JSON file, we parse them into separate model cards. See the `example_outputs/retriever_output.txt` in supplementary material.

A.2 Candidate evaluation agent

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- We will now provide a task description and a model description.
- You need to implement your Python solution using the provided model.

# Task description
{task description}

# Model description
## Model name
{model description}

## Example Python code
{example code}

# Your task
- Implement the solution in Python.
- You must use the model as described in the model description.
- This first solution design should be relatively simple, without ensembling or
hyper-parameter optimization.
- Propose an evaluation metric that is reasonable for this task.
- All the provided data is already prepared and available in the `./input` directory. There
is no need to unzip any files.
- Do not include other models that are not directly related to the model described.
- Use PyTorch rather than TensorFlow. Use CUDA if you need. All the necessary libraries are
installed.
- The code should implement the proposed solution and print the value of the evaluation
metric computed on a hold-out validation set.
- Only use the provided train data in the `./input` directory. Do not load test data.
- If there are more than 30,000 training samples, you must subsample to 30,000 for a faster
run.

# Required
- There should be no additional headings or text in your response.
- Print out or return a final performance metric in your answer in a clear format with the
exact words: 'Final Validation Performance: {final_validation_score}'.
- The code should be a single-file Python program that is self-contained and can be
executed as-is.
- Your response should only contain a single code block.
- Do not use exit() function in the Python code.
- Do not use try: and except: or if else to ignore unintended behavior.
```

Figure 10: Prompt used for evaluating retrieved models.

MLE-STAR uses candidate evaluation agent $\mathcal{A}_{\text{init}}$ to evaluate the performance of the retrieved model. As shown in Figure 10, by taking task description ($\mathcal{T}_{\text{task}}$), model description ($\mathcal{T}_{\text{model}}$), and corresponding code example ($\mathcal{T}_{\text{code}}$), $\mathcal{A}_{\text{init}}$ generates a Python script. See `example_outputs/candidate_evaluation.py` in the supplementary materials for an example. The Python script for the retrieved model evaluation is guided to be relatively simple, and to contain the evaluation result computed on a hold-out validation set. In addition, if there are too many training samples, $\mathcal{A}_{\text{init}}$ uses the subset of training sample for faster execution.

A.3 Merging agent

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- We will now provide a base solution and an additional reference solution.
- You need to implement your Python solution by integrating reference solution to the base solution.

# Base solution
{base code}

# Reference solution
{reference code}

# Your task
- Implement the solution in Python.
- You have to integrate the reference solution to the base solution.
- Your code base should be the base solution.
- Try to train additional model of the reference solution.
- When integrating, try to keep code with similar functionality in the same place (e.g., all preprocessing should be done and then all training).
- When integrating, ensemble the models.
- The solution design should be relatively simple.
- The code should implement the proposed solution and print the value of the evaluation metric computed on a hold-out validation set.
- Only use the provided train data in the `./input` directory.
- If there are more than 30,000 training samples, you must subsample to 30,000 for a faster run.

# Required
- There should be no additional headings or text in your response.
- Print out or return a final performance metric in your answer in a clear format with the exact words: 'Final Validation Performance: {final_validation_score}'.
- The code should be a single-file Python program that is self-contained and can be executed as-is.
- Your response should only contain a single code block.
- Do not use exit() function in the Python code.
- Do not use try: and except: or if else to ignore unintended behavior
```

Figure 11: Prompt used for merging the candidate models for generating initial solution.

MLE-STAR leverages an agent $\mathcal{A}_{\text{merger}}$ to merge the retrieved models into a consolidated initial solution. As shown in Figure 11, this process is done sequentially, where the prompt guides the agent to integrate the reference code (*i.e.*, the best candidate model code among the models that are not merged yet) into the base code (*i.e.*, the current candidate merged script). The output of $\mathcal{A}_{\text{merger}}$ is a Python script (see `example_outputs/merged_candidate.py` in supplementary materials for an example), which will be the next candidate merged script. See Appendix B for the sequential procedure of MLE-STAR when generating the initial solution.

A.4 Ablation study agent

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to perform an ablation study on the current
Python solution to know which parts of the code contribute the most to the overall
performance.
- We will now provide a current Python solution.
- We will also provide the summaries of previous ablation studies.

# Python solution
{solution script}

## Previous ablation study result {0}
{previous_ablations[0]}

## Previous ablation study result {1}
{previous_ablations[1]}

...

## Previous ablation study result {t-1}
{previous_ablations[t-1]}

# Instructions
- You need you to generate a simple Python code that performs an ablation study on the
train.py script.
- The generated code should create variations by modifying or disabling parts (2-3 parts)
of the training process.
- Your ablation study should concentrate on the other parts that have not been previously
considered.
- For each ablation, print out how the modification affects the model's performance.

# Response format
- There should be no additional headings or text in your response.
- The Python code for the ablation study should not load test data. It should only focus on
training and evaluating the model on the validation set.
- The code should include a printing statement that shows the performance of each ablation.
- The code should consequently print out what part of the code contributes the most to the
overall performance.
```

Figure 12: Prompt used for generating a Python script for ablation studies.

To effectively explore specialized improvement strategies, MLE-STAR identifies and targets specific code blocks. This code block selection is guided by an ablation study performed by an agent \mathcal{A}_{abl} . As shown in Figure 12, \mathcal{A}_{abl} generates a Python code designed to perform an ablation study on current solution. The prompt guides the agent to modify or disable specific component. We provide an example generated code for ablation study (which is generated by \mathcal{A}_{abl}) in the supplementary materials (see `example_outputs/ablation.py`). Moreover, to encourage exploration of different pipeline parts, the summaries of previous ablation studies are also used as valuable feedback. See Appendix C for the example output of the agent \mathcal{A}_{abl} .

A.5 Ablation study summarization agent

```
# Your code for ablation study was:
{code for ablation study}

# Ablation study results after running the above code:
{raw result}

# Your task
- Summarize the result of ablation study based on the code and printed output.
```

Figure 13: Prompt used for summarizing the result of the ablation study.

After executing the code for an ablation study, denoted as a_t , the output result r_t is produced. Since r_t often contains content unrelated to the ablation (for example, printing the loss value across training epochs), a summarization module $\mathcal{A}_{\text{summarize}}$ is utilized with the prompt mentioned above (Figure 13). This module takes a_t and r_t as input to summarize and parse the ablation study results. Here, a_t is also used because it provides information about the modification. See Appendix C for the examples of r_t and the summarization result.

A.6 Extractor

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to extract a code block from the current Python solution and improve the extracted block for better performance.
- Your suggestion should be based on the ablation study results of the current Python solution.
- We will now provide the current Python solution and the ablation study results.
- We also provide code blocks which you have tried to improve previously.

# Python solution
{solution script}

# Ablation study results
{summary of ablation study}

## Code block {0}
{prev_code_blocks[0]}
## Code block {1}
{prev_code_blocks[1]}
...
## Code block {t-1}
{prev_code_blocks[t-1]}

# Your task
- Given the ablation study results, suggest an effective next plan to improve the above Python script.
- The plan should be a brief outline/sketch of your proposed solution in natural language (3-5 sentences).
- Please avoid plan which can make the solution's running time too long (e.g., searching hyperparameters in a very large search space).
- Try to improve the other part which was not considered before.
- Also extract the code block from the above Python script that need to be improved according to the proposed plan. You should try to extract the code block which was not improved before.

# Response format
- Your response should be a brief outline/sketch of your proposed solution in natural language (3-5 sentences) and a single markdown code block which is the code block that need to be improved.
- The code block can be long but should be exactly extracted from the Python script provided above.

Use this JSON schema:

Refine_Plan = {'code_block': str, 'plan': str}
Return: list[Refine_Plan]
```

Figure 14: Prompt used for extracting the code block that has the most significant impact on overall pipeline.

MLE-STAR uses an extractor module $\mathcal{A}_{\text{extractor}}$ to analyze the \mathcal{T}_{abl} and then identify the code block c_t . As shown in Figure 14, $\mathcal{A}_{\text{extractor}}$ takes the summary of the ablation study, current solution code, and the previously refined code blocks as input, and is guided to output the code block which has the most significant impact on performance. Here, the initial plan for refining the extracted code block is also generated. See `example_outputs/code_block.txt` for an example of extracted code block.

A.7 Coder

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need refine the code block for better performance based on the improvement plan.
- We will now provide the code block and the improvement plan.

# Code block
{code_block}

# Improvement plan
{plan}

# Your task
- Implement the improvement plan on the above code block. But do not remove subsampling if exists.
- The code block should be improved according to the proposed plan.
- Note that all the variable including actual data is defined earlier (since you are just seeing a code block), therefore do not introduce dummy variables.

# Response format
- Your response should be a single markdown code block (wrapped in ```) which is the improved code block.
- There should be no additional headings or text in your response.
```

Figure 15: Prompt used for implementing refinement plan on the extracted code block.

The implementation of code block refinement is done by $\mathcal{A}_{\text{coder}}$, which takes the extracted code block and the refinement plan as input, and outputs the refined code block. We provide an example of the target code block, proposed plan, and the output of refined code block by $\mathcal{A}_{\text{coder}}$ in the supplementary materials (see `example_outputs/coder_outputs` directory).

A.8 Planner

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you have to improve the code block for better performance.
- We will provide the code block you are improving and the improvement plans you have tried.

# Code block
{code block}

# Improvement plans you have tried

## Plan: {plans[0]}
## Score: {scores[0]}

## Plan: {plans[1]}
## Score: {scores[1]}

...

## Plan: {plans[k-1]}
## Score: {scores[k-1]}

# Your task
- Suggest a better plan to improve the above code block.
- The suggested plan must be novel and effective.
- Please avoid plans which can make the solution's running time too long (e.g., searching hyperparameters in a very large search space).
- The suggested plan should differ from the previous plans you have tried and should receive a higher score.

# Response format
- Your response should be a brief outline/sketch of your proposed solution in natural language (3-5 sentences).
- There should be no additional headings or text in your response.
```

Figure 16: Prompt used for generating the next refinement plan which targets the extracted code block.

To discover potentially more effective or novel refinement strategies (targeting the extracted code block), MLE-STAR iteratively generates further plans through a planning agent $\mathcal{A}_{\text{planner}}$. As shown in Figure 16, $\mathcal{A}_{\text{planner}}$ takes the extracted code block and the previous attempts as input and proposes the next plan. These are examples of proposed plans:

```
f'''Since feature engineering had the biggest impact, I will focus on improving
→ the cabin feature extraction. Instead of simply splitting the Cabin string, I
→ will create dummy variables for each unique Deck and Side. Also, the Cabin_num
→ will be kept as numerical, imputing missing values using a median strategy to
→ handle potential outliers. This approach should provide more granular
→ information to the models.'''
```



```
#'''Instead of one-hot encoding 'Deck' and 'Side' directly, I will explore
↳ interaction features between 'Deck', 'Side', and potentially 'Cabin_num'.
↳ Specifically, I'll create combined features like 'Deck_Side' and
↳ 'Deck_Cabin_num' to capture potential dependencies. Furthermore, I will impute
↳ missing 'Cabin_num' values using a more sophisticated method like k-NN
↳ imputation, considering other features like 'Deck', 'Side', and 'RoomService'
↳ to improve imputation accuracy. This should capture more complex relationships
↳ within the cabin data and lead to better model performance.'''
```

```
#'''I propose a plan that focuses on a more nuanced approach to 'Cabin_num' and
↳ interaction terms. First, I'll bin 'Cabin_num' into ordinal categories (e.g.,
↳ low, medium, high) based on quantiles, as the absolute number might not be as
↳ important as its relative position. Then, I'll create interaction features
↳ between the binned 'Cabin_num', 'Deck', and 'Side' using one-hot encoding.
↳ This will allow the model to learn specific combinations of cabin location and
↳ number range that might be predictive. Finally, I will use a simple imputer
↳ for the missing values in 'Cabin_num' before binning.'''
```

A.9 Ensemble strategy planner

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you have to ensemble {L} Python Solutions for better performance.
- We will provide the Python Solutions and the ensemble plans you have tried.

# 1st Python Solution
{solution1}
# 2nd Python Solution
{solution2}
...
# {L}th Python Solution
{solutionL}

# Ensemble plans you have tried

## Plan: {plans[0]}
## Score: {scores[0]}
## Plan: {plans[1]}
## Score: {scores[1]}
...
## Plan: {plans[r-1]}
## Score: {scores[r-1]}

# Your task
- Suggest a better plan to ensemble the {L} solutions. You should concentrate how to merge, not the other parts like hyperparameters.
- The suggested plan must be easy to implement, novel, and effective.
- The suggested plan should be differ from the previous plans you have tried and should receive a higher (or lower) score.

# Response format
- Your response should be an outline/sketch of your proposed solution in natural language.
- There should be no additional headings or text in your response.
- Plan should not modify the original solutions too much since execution error can occur.
```

Figure 17: Prompt used for generating the next ensemble plan.

As shown in Figure 17, similar to $\mathcal{A}_{\text{planner}}$, $\mathcal{A}_{\text{ens_planner}}$ proposes an effective ensemble plan based on the history of previously attempted ensemble plans and their resulting performance as feedback. These are examples of attempted ensemble plans.

```
#!/usr/bin/env python
'''Averaging the predicted probabilities from both models is a straightforward
→ and effective ensembling technique. First, modify the AutoGluon solution to
→ output probabilities instead of hard predictions using
→ `predictor.predict_proba(test_data)`. Then, obtain the predicted probabilities
→ from the LightGBM model using
→ `lgbm_classifier.predict_proba(X_test_processed)`. Average these probabilities
→ for each class. Finally, generate the final predictions by thresholding the
→ averaged probability of the 'Transported' class at 0.5. Create the submission
→ file based on these averaged and thresholded predictions.'''
```

```

#'''Here's an ensembling plan leveraging stacking with a simple meta-learner:

1.  **Generate Predictions:** Use both the AutoGluon model and the LightGBM model
    → to generate predictions on the original training data (train.csv). This is
    → crucial for training the meta-learner. For AutoGluon, use
    → `predictor.predict_proba(train_data)` and extract the probabilities for the
    → 'Transported' class. For LightGBM, preprocess the training data using the
    → same pipeline as the test data and get probabilities with
    → `lgbm_classifier.predict_proba(X_processed)` and again, extract the
    → probabilities for the 'Transported' class.

2.  **Create Meta-Features:** Combine the predicted probabilities from AutoGluon
    → and LightGBM for the training data into a new dataframe. This dataframe will
    → have two columns: 'AutoGluon_Prob' and 'LGBM_Prob', and the 'Transported'
    → column from the original training data as the target variable for the
    → meta-learner.

3.  **Train Meta-Learner:** Use a simple model like Logistic Regression as the
    → meta-learner. Train this Logistic Regression model using the meta-features
    → (AutoGluon_Prob, LGBM_Prob) to predict the 'Transported' column. This step
    → aims to learn how to best combine the predictions of the base models.

4.  **Generate Test Predictions:** Get the predicted probabilities from AutoGluon
    → and LightGBM on the test set, as in the averaging approach.

5.  **Create Meta-Features for Test Data:** Create a dataframe for the test data,
    → with the same structure as the training meta-features (AutoGluon_Prob,
    → LGBM_Prob) from the test set.

6.  **Meta-Learner Prediction:** Use the trained Logistic Regression model to
    → predict the final 'Transported' probabilities on the test meta-features.

7.  **Threshold and Submit:** Threshold the predicted probabilities from the
    → meta-learner at 0.5 to get the final predictions (True/False) and create the
    → submission file.'''

```

```

#'''Here's an ensembling plan that focuses on weighted averaging with optimized
→ weights determined by a simple grid search on a validation set:

1.  **Validation Split:** Split the original training data into two parts: a
→ training set (e.g., 80% of the data) and a validation set (e.g., 20% of the
→ data). Crucially, perform the preprocessing steps (OneHotEncoding, Scaling,
→ etc.) separately on the training and validation sets to avoid data leakage.

2.  **Generate Validation Predictions:** Use both the AutoGluon model and the
→ LightGBM model to generate predictions on the validation set. For AutoGluon,
→ obtain probabilities using `predictor.predict_proba(validation_data)`. For
→ LightGBM, preprocess the validation data using the same pipeline trained on
→ the training split and get probabilities using
→ `lgbm_classifier.predict_proba(X_validation_processed)`.

3.  **Grid Search for Optimal Weights:** Define a grid of weights for AutoGluon
→ and LightGBM. For instance, iterate through weights from 0.0 to 1.0 in
→ increments of 0.1 for AutoGluon, with the LightGBM weight being (1 - AutoGluon
→ weight). For each weight combination:
    * Calculate the weighted average of the predicted probabilities from
→ AutoGluon and LightGBM on the validation set.
    * Threshold the averaged probabilities at 0.5 to obtain binary predictions.
    * Calculate the accuracy of these predictions against the true labels in the
→ validation set.

4.  **Select Best Weights:** Choose the weight combination that yields the highest
→ accuracy on the validation set.

5.  **Generate Test Predictions:** Obtain the predicted probabilities from
→ AutoGluon and LightGBM on the test set, as before.

6.  **Weighted Averaging on Test Set:** Use the optimal weights determined in step
→ 4 to calculate the weighted average of the predicted probabilities from
→ AutoGluon and LightGBM on the test set.

7.  **Threshold and Submit:** Threshold the weighted average probabilities at 0.5
→ to obtain the final predictions and create the submission file.

This plan is easy to implement, avoids complex meta-learners that can overfit, and
→ focuses on finding the best combination of the two models based on a
→ validation set. It adapts to the strengths of each model by giving them
→ different weights.'''

```

A.10 Ensembler

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to ensemble {L} Python Solutions for better performance based on the ensemble plan.
- We will now provide the Python Solutions and the ensemble plan.

# 1st Python Solution
{solution1}
# 2nd Python Solution
{solution2}
...
# {L}th Python Solution
{solutionL}

# Ensemble Plan
{plan}

# Your task
- Implement the ensemble plan with the provided solutions.
- Unless mentioned in the ensemble plan, do not modify the original Python Solutions too much."
- All the provided data (except previous submissions; do not load submissions) is already prepared and available in the `.\input` directory. There is no need to unzip any files.
- The code should implement the proposed solution and print the value of the evaluation metric computed on a hold-out validation set.

# Response format required
- Your response should be a single markdown code block (wrapped in ```) which is the ensemble of {L} Python Solutions.
- There should be no additional headings or text in your response.
- Do not subsample or introduce dummy variables. You have to provide full new Python Solution using the {L} provided solutions.
- Do not forget the `./final/submission.csv` file.
- Print out or return a final performance metric in your answer in a clear format with the exact words: 'Final Validation Performance: {final_validation_score}'.
- The code should be a single-file Python program that is self-contained and can be executed as-is.
```

Figure 18: Prompt used for implementing ensemble plan on the solutions generated by MLE-STAR in parallel.

The proposed ensemble plan is implemented by $\mathcal{A}_{\text{ensembler}}$. This agent takes the two final solutions which is generated in parallel by MLE-STAR, and the ensemble plan as input, and outputs the Python script, *i.e.*, the merged code solution (see Appendix C for examples since the final solution is selected among the merged code solution).

A.11 Debugging agent

```
# Code with an error:
{code}

# Error:
{bug}

# Your task
- Please revise the code to fix the error.
- Do not remove subsampling if exists.
- Provide the improved, self-contained Python script again.
- There should be no additional headings or text in your response.
- All the provided input data is stored in "./input" directory.
- Remember to print a line in the code with 'Final Validation Performance:
{final_validation_score}' so we can parse performance.
- The code should be a single-file python program that is self-contained and can be
executed as-is.
- Your response should only contain a single code block.
- Do not use exit() function in the refined Python code.
```

Figure 19: Prompt used for debugging.

If the execution of a Python script triggers an error, MLE-STAR employs a debugging module $\mathcal{A}_{\text{debugger}}$ to attempt correction using the above prompt (Figure 19).

A.12 Data leakage checker

```
# Python code
{code}

# Your task
- Extract the code block where the validation and test samples are preprocessed using training samples.
- Check that the model is trained with only training samples.
- Check that before printing the final validation score, the model is not trained the validation samples.
- Also check whether the validation and test samples are preprocessed correctly, preventing information from the validation or test samples from influencing the training process (i.e., preventing data leakage).

# Requirement
- Extract a code block and also check the data leakage.
- The code block should be an exact subset of the above Python code.
- Your response for a code block should be a single markdown code block.
- If data leakage is present on validation and test samples, answer 'Yes Data Leakage'.
- If data leakage is not present on validation and test samples, answer 'No Data Leakage'.

Use this JSON schema:
Answer = {'leakage_status': str, 'code_block': str}
Return: list[Answer]
```

Figure 20: Prompt used for extract the code block whether data preprocessing is done.

```
# Python code
{code}

# Your task
- In the above Python code, the validation and test samples are influencing the training process, i.e., not correctly preprocessed.
- Ensure that the model is trained with only training samples.
- Ensure that before printing the final validation score, the model is not trained on the validation samples.
- Refine the code to prevent such data leakage problem.

# Requirement
- Your response should be a single markdown code block.
- Note that all the variables are defined earlier. Just modify it with the above code.
```

Figure 21: Prompt used for correcting the code block with a risk of data leakage.

To mitigate the risk of introducing data leakage, MLE-STAR first extract the code block where preprocessing is done. This is achieved by using the above prompt in Figure 20, which takes the current solution script as input, and then generates (1) the code block and (2) whether the extracted code block has a risk of data leakage. If leakage is detected, the code block is corrected with the prompt in Figure 21, and MLE-STAR replaces the original code block to the corrected version.

A.13 Data usage checker

```
I have provided Python code for a machine learning task (attached below):

# Solution Code
{initial solution}

Does above solution code uses all the information provided for training? Here is task
description and some guide to handle:

# Task description
{task description}

# Your task
- If the above solution code does not use the information provided, try to incorporate all.
Do not bypass using try-except.
- DO NOT USE TRY and EXCEPT; just occur error so we can debug it!
- See the task description carefully, to know how to extract unused information
effectively.
- When improving the solution code by incorporating unused information, DO NOT FORGET to
print out 'Final Validation Performance: {final_validation_score}' as in original solution
code.

# Response format:
Option 1: If the code did not use all the provided information, your response should be a
single markdown code block (wrapped in ```) which is the improved code block. There should
be no additional headings or text in your response
Option 2: If the code used all the provided information, simply state that "All the
provided information is used."
```

Figure 22: Prompt used for data usage checker.

To ensure the utilization of all relevant provided data, MLE-STAR utilizes a data usage checker agent $\mathcal{A}_{\text{data}}$. This agent checks the initial solution with the task description, and revise the initial script using the prompt in Figure 22.

B Algorithms

B.1 Algorithm for generating an initial solution

Algorithm 1 Generating an initial solution

```
1: Input: task description  $\mathcal{T}_{\text{task}}$ , datasets  $\mathcal{D}$ , score function  $h$ , number of retrieved models  $M$ ,  
2:  $\{\mathcal{T}_{\text{model}}^i, \mathcal{T}_{\text{code}}^i\}_{i=1}^M = \mathcal{A}_{\text{retriever}}(\mathcal{T}_{\text{task}})$   
3: for  $i = 1$  to  $M$  do  
4:    $s_{\text{init}}^i = \mathcal{A}_{\text{init}}(\mathcal{T}_{\text{task}}, \mathcal{T}_{\text{model}}^i, \mathcal{T}_{\text{code}}^i)$   
5:   Evaluate  $h(s_{\text{init}}^i)$  using  $\mathcal{D}$   
6: end for  
7:  $s_0 \leftarrow s_{\text{init}}^{\pi(1)}$   
8:  $h_{\text{best}} \leftarrow h(s_0)$   
9: for  $i = 2$  to  $M$  do  
10:   $s_{\text{candidate}} \leftarrow \mathcal{A}_{\text{merger}}(s_0, s_{\text{init}}^{\pi(i)})$   
11:  Evaluate  $h(s_{\text{candidate}})$  using  $\mathcal{D}$   
12:  if  $h(s_{\text{candidate}}) \geq h_{\text{best}}$  then  
13:     $s_0 \leftarrow s_{\text{candidate}}$   
14:     $h_{\text{best}} \leftarrow h(s_0)$   
15:  else  
16:    break  
17:  end if  
18: end for  
19: Output: initial solution  $s_0$ 
```

B.2 Algorithm for refining a code block for solution improvement

Algorithm 2 Refining solution

```

1: Input: initial solution  $s_0$ , outer loop steps  $T$ , inner loop steps  $K$ 
2:  $s_{\text{final}} \leftarrow s_0$ 
3:  $h_{\text{best}} \leftarrow h(s_0)$ 
4:  $\mathcal{T}_{\text{abl}}, \mathcal{C} = \{\}, \{\}$ 
5: for  $t = 0$  to  $T - 1$  do
6:    $a_t = \mathcal{A}_{\text{abl}}(s_t, \mathcal{T}_{\text{abl}})$ 
7:    $r_t = \text{exec}(a_t)$ 
8:    $\mathcal{T}_{\text{abl}}^t = \mathcal{A}_{\text{summarize}}(a_t, r_t)$ 
9:    $c_t, p_0 = \mathcal{A}_{\text{extractor}}(\mathcal{T}_{\text{abl}}^t, s_t, \mathcal{C})$ 
10:   $c_t^0 = \mathcal{A}_{\text{coder}}(c_t, p_0)$ 
11:   $s_t = s_t.\text{replace}(c_t, c_t^0)$ 
12:  Evaluate  $h(s_t^0)$  using  $\mathcal{D}$ 
13:  if  $h(s_t^0) \geq h_{\text{best}}$  then
14:     $s_{\text{final}} \leftarrow s_t^0$ 
15:     $h_{\text{best}} \leftarrow h(s_t^0)$ 
16:  end if
17:  for  $k = 1$  to  $K - 1$  do
18:     $p_k = \mathcal{A}_{\text{planner}}(c_t, \{p_j, h(s_t^j)\}_{j=0}^{k-1})$ 
19:     $c_t^k = \mathcal{A}_{\text{coder}}(c_t, p_k)$ 
20:     $s_t^k = s_t.\text{replace}(c_t, c_t^k)$ 
21:    Evaluate  $h(s_t^k)$  using  $\mathcal{D}$ 
22:    if  $h(s_t^k) \geq h_{\text{best}}$  then
23:       $s_{\text{final}} \leftarrow s_t^k$ 
24:       $h_{\text{best}} \leftarrow h(s_t^k)$ 
25:    end if
26:  end for
27:   $\mathcal{T}_{\text{abl}} \leftarrow \mathcal{T}_{\text{abl}} + \mathcal{T}_{\text{abl}}^t$ 
28:   $\mathcal{C} \leftarrow \mathcal{C} + c_t$ 
29: end for
30: Output: final solution  $s_{\text{final}}$ 

```

B.3 Algorithm for further improvement by exploring ensemble strategies

Algorithm 3 Ensembling final solutions

```

1: Input: candidate final solutions  $s_{\text{final}}^1, \dots, s_{\text{final}}^L$ , ensemble loop steps  $R$ 
2:  $e_0 = \mathcal{A}_{\text{ens\_planner}}(\{s_{\text{final}}^l\}_{l=1}^L)$ 
3:  $s_{\text{ens}}^0 = \mathcal{A}_{\text{ensembler}}(e_0, \{s_{\text{final}}^l\}_{l=1}^L)$ 
4: Evaluate  $h(s_{\text{ens}}^0)$  using  $\mathcal{D}$ 
5: for  $r = 1$  to  $R - 1$  do
6:    $e_r = \mathcal{A}_{\text{ens\_planner}}(\{s_{\text{final}}^l\}_{l=1}^L, \{e_j, h(s_{\text{ens}}^j)\}_{j=0}^{r-1})$ 
7:    $s_{\text{ens}}^r = \mathcal{A}_{\text{ensembler}}(e_r, \{s_{\text{final}}^l\}_{l=1}^L)$ 
8:   Evaluate  $h(s_{\text{ens}}^r)$  using  $\mathcal{D}$ 
9: end for
10:  $s_{\text{ens}}^* = s_{\text{ens}}^{r^*}$  where  $r^* = \arg \max_{r \in \{0, \dots, R-1\}} h(s_{\text{ens}}^r)$ 
11: Output:  $s_{\text{ens}}^*$ 

```

C Qualitative examples

C.1 Generated code for ablation study

We provide an example generated code for ablation study (which is generated by \mathcal{A}_{abl}) in the supplementary material (see `example_outputs/ablation.py`).

C.2 Raw output of ablation study after execution

```
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.001167 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1647
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 26
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Baseline Validation Performance: 0.8195542774982028
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.002991 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1647
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 26
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Ablation 1 (No StandardScaler) Validation Performance: 0.8102084831056794
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000367 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1609
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 7
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Ablation 2 (No OneHotEncoder) Validation Performance: 0.7886412652767792
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.001942 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1647
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 26
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Ablation 3 (No Imputation) Validation Performance: 0.8195542774982028
Final Validation Performance: 0.8195542774982028
```

Figure 23: Example output after running the code for ablation study.

We provide an example output after running a code for ablation study using `spaceship-titanic` competition. As shown in Figure 23, the execution result often contains content unrelated to the ablation (e.g., training information of LightGBM). Therefore, MLE-STAR utilizes $\mathcal{A}_{\text{summarize}}$ to parse the ablation study results, which will be illustrated in the following Appendix C.3.

C.3 Summary of ablation study

The ablation study investigated the impact of three preprocessing steps on the performance of a LightGBM classifier: StandardScaler, OneHotEncoder, and Imputation. The baseline model, which included all three preprocessing steps, achieved a validation accuracy of 0.8196.

* **Ablation 1 (No StandardScaler):** Removing the StandardScaler resulted in a slightly lower validation accuracy of 0.8102. This suggests that scaling the numerical features does contribute positively to the model's performance, although the impact is relatively small.

* **Ablation 2 (No OneHotEncoder):** Removing the OneHotEncoder led to a more significant drop in validation accuracy to 0.7886. This indicates that encoding the categorical features using OneHotEncoder is important for the model's ability to learn from the data.

* **Ablation 3 (No Imputation):** Removing the Imputation did not change the validation accuracy, which remained at 0.8196. This suggests that the imputation strategy used (mean for numerical and mode for categorical) does not hurt the model performance.

In summary, the OneHotEncoder has the most significant positive impact on the model's performance, followed by the StandardScaler. The Imputation does not seem to have a significant impact on the model's performance.

Figure 24: Example of summarized result of ablation study.

To parse the information only about the impact of each ML components, MLE-STAR leverages $\mathcal{A}_{\text{summarize}}$ to summarize the raw output of ablation study such as Figure 23. As a result, we obtain the well-organized summary of the ablation study as shown in Figure 24. Note that such summarization is used as input of $\mathcal{A}_{\text{extractor}}$ to extract the code block which has most significant impact on performance.

D Qualitative comparison

We provide qualitative comparison results (*i.e.*, the final solution code) in supplementary material. Ours are denoted as `mle_star.py` and solution generated by AIDE [12] is denoted as `aide.py` in folder name with competition ID. Note that both agent used Gemini-2.0-Flash as a base LLM.

E Benchmark

E.1 MLE-bench Lite

Table 9: Competitions contained in MLE-bench Lite [16].

Competition ID	Category	Dataset Size (GB)
aerial-cactus-identification	Image Classification	0.0254
aptos2019-blindness-detection	Image Classification	10.22
denoising-dirty-documents	Image To Image	0.06
detecting-insults-in-social-commentary	Text Classification	0.002
dog-breed-identification	Image Classification	0.75
dogs-vs-cats-redux-kernels-edition	Image Classification	0.85
histopathologic-cancer-detection	Image Regression	7.76
jigsaw-toxic-comment-classification-challenge	Text Classification	0.06
leaf-classification	Image Classification	0.036
mlsp-2013-birds	Audio Classification	0.5851
new-york-city-taxi-fare-prediction	Tabular	5.7
nomad2018-predict-transparent-conductors	Tabular	0.00624
plant-pathology-2020-fgvc7	Image Classification	0.8
random-acts-of-pizza	Text Classification	0.003
ranzcr-clip-catheter-line-classification	Image Classification	13.13
siim-isic-melanoma-classification	Image Classification	116.16
spooky-author-identification	Text Classification	0.0019
tabular-playground-series-dec-2021	Tabular	0.7
tabular-playground-series-may-2022	Tabular	0.57
text-normalization-challenge-english-language	Seq->Seq	0.01
text-normalization-challenge-russian-language	Seq->Seq	0.01
the-icml-2013-whale-challenge-right-whale-redux	Audio Classification	0.29314

In this paper, we utilize MLE-bench (especially Lite version) [16] as our main benchmark to verify MLE-STAR’s effectiveness compared to the alternatives. In a nutshell, MLE-bench consists of 75 offline Kaggle competitions. Each competition has an associated description, dataset, and grading code. Additionally, MLE-bench consists of various problem types, such as tabular prediction, text classification, image classification, etc. However, since utilizing full 75 competitions is expensive, we use the Lite version, which is the low complexity split of MLE-bench (*i.e.*, MLE-bench Lite). MLE-bench Lite consists of 22 competitions, and the description of competitions is provided in Table 9.

E.2 Tabular tasks from DS-Agent

Table 10: Tabular competitions used in DS-Agent [10]

Competition ID	Category	Evaluation Metrics
media-campaign-cost	Tabular Regression	RMLSE
wild-blueberry-yield	Tabular Regression	MAE
spaceship-titanic	Tabular Classification	Accuracy
enzyme-substrate	Tabular Classification	AUROC

We also provide the descriptions of tabular competitions used in DS-Agent’s development phase [10] in Table 10.

E.3 Generating submission file

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to come up with an excellent solution in Python.
- We will now provide a task description and a Python solution.
- What you have to do on the solution is just loading test samples and create a submission file.

# Task description
{task description}

# Python solution
{final solution}

# Your task
- Load the test samples and create a submission file.
- All the provided data is already prepared and available in the `./input` directory. There is no need to unzip any files.
- Test data is available in the `./input` directory.
- Save the test predictions in a `submission.csv` file. Put the `submission.csv` into `./final` directory.
- You should not drop any test samples. Predict the target value for all test samples.
- This is a very easy task because the only thing to do is to load test samples and then replace the validation samples with the test samples. Then you can even use the full training set!

# Required
- Do not modify the given Python solution code too much. Try to integrate test submission with minimal changes.
- There should be no additional headings or text in your response.
- The code should be a single-file Python program that is self-contained and can be executed as-is.
- Your response should only contain a single code block.
- Do not forget the `./final/submission.csv` file.
- Do not use exit() function in the Python code.
- Do not use try: and except: or if else to ignore unintended behavior.
```

Figure 25: Prompt used for incorporating loading test sample and generating a submission file.

In order to evaluate on MLE-bench Lite, one should create a submission file about prediction results on test samples with required format. To achieve this, MLE-STAR uses an agent $\mathcal{A}_{\text{test}}$, which takes the task description and the final solution as input, and outputs the code that incorporates loading test sample and creating a submission file. This is done by using a prompt in Figure 25.

```

# Introduction
- From the give Python solution, you need to extract a code block where subsampling of training samples is used. We will now provide the current Python solution."

# Current Python solution
{final solution}

# Your task
- Extract a code block where subsampling of training samples is used.

# Response format
- Your response should be a single markdown code block (wrapped in ```) which is the code block.
- The code block should be exactly extracted from the Python script provided above.

```

Figure 26: Prompt used for extracting the code block which performs subsampling.

```

# Introduction
- From the give Python code block, remove the subsampling and make it to use full training samples. We will now provide the current Python code block.

# Current Python code block
{code block with subsampling}

# Your task
- Remove the subsampling and make it to use full training samples.
- Note that all the variable including actual data is defined earlier (since you are just seeing a code block), therefore do not introduce dummy variables.

# Response format
- Your response should be a single markdown code block (wrapped in ```) which is the code block.

```

Figure 27: Prompt used for guiding MLE-STAR to utilize full training samples.

Removing subsampling. As shown in Figure 10, MLE-STAR uses the subset of training sample for faster refinement (since evaluating the solution candidate can take a lot of time). However, in order to get a better performance, when generating a submission file MLE-STAR removes such subsampling code. Specifically, this is done by first extracting the code block which performs subsampling (using prompt in Figure 26), and then modify the extracted code block to utilize all the provided samples, using prompt in Figure 27.

F Experimental setup

We conducted our experiments mainly using 96 vCPUs with 360 GB Memory (Intel(R) Xeon(R) CPU), and 8 NVIDIA V100 GPUs with 16 GB Memory.

Required time to generate a single solution using MLE-STAR. With the configuration of four retrieved models, four inner loops, four outer loops, and five rounds for exploring the ensemble strategy, MLE-STAR requires 14.1 hours to generate a single final solution, on average across 22 tasks and all three random trials (*i.e.*, total 66 experiments). On the other hand, we found that AIDE [12] requires 15.4 hours. This indicates that our method does not require more time to run compare to the best alternative. Note that a maximum time limit of 24 hours was set for both methods, following the MLE-bench’s experimental setup.

G Additional quantitative results

Table 11: Additional comparisons with AutoGluon and DS-Agent in four tabular tasks.

Model	media-campaign-cost	wild-blueberry-yield	spaceship-titanic	enzyme-substrate
Evaluation Metrics	RMLSE (\downarrow)	MAE (\downarrow)	Accuracy (\uparrow)	AUROC (\uparrow)
AutoGluon [26]	0.2707	305	0.8044	0.8683
DS-Agent [10]				
gpt-3.5	0.2702	291	/	0.5534
gpt-4	0.2947	267	0.7977	0.8322
gemini-2.0-flash	0.2964	213	0.7982	0.8727
MLE-STAR (Ours)				
gemini-2.0-flash	0.2911	163	0.8091	0.9101

This section provides detailed results for the comparison with DS-Agent [10]. In particular, we provide additional comparisons with AutoGluon [26] and DS-Agent using other LLMs (*i.e.*, GPT-3.5 and GPT-4). Except for DS-Agent with Gemini-2.0-Flash and MLE-STAR, all experimental results are taken from the original paper [10]. As shown in Table 11, MLE-STAR consistently outperforms DS-Agent with Gemini-2.0-Flash, while also outperforms AutoGluon with high margin on three tabular tasks.

It is worth to note that AutoGluon is restricted to task types, *i.e.*, specially designed for tabular data. In contrast, MLE-STAR is a general framework for any kinds of tasks, where well-written task description, containing the task information, is the only requirement to work on the given tasks. Therefore, while AutoGluon is not a direct competitor in this regard, MLE-STAR shows improved performance even when compared to AutoGluon.

H Analysis on data contamination

```
Your task is to check whether the python solution is similar to the reference discussion.
Now we will give you reference discussion and our python solution.

# Reference discussion
{reference discussion}

# Python solution
{final solution}

# Your task
- Check whether the python solution just copy and pastes the reference discussion.
- If it is sufficiently novel and different, please answer 'Novel'.
- Otherwise, if you think it is too similar, please answer 'Same'.
- Your answer should be only one of 'Novel' or 'Same'.
```

Figure 28: Prompt used for identifying whether the final solution generated by MLE-STAR is novel.

Since Kaggle competitions in MLE-bench are publicly accessible, there is a potential risk that LLMs might have been trained with the relevant discussions about the challenge. For example, if an LLM has memorized a discussion of the best performing solution, one easy way for the MLE agent to follow that discussion during the refinement phase.

However, to alleviate such potential issue, we show that MLE-STAR’s solution is sufficiently novel compared to the discussions on Kaggle. Here, we use discussions collected in GitHub repository of MLE-bench [16] are collected by the authors of MLE-bench [16]. To be specific, these discussions are top discussion posts of each competition. As a result, we collected a total of 25 discussions from 7 competitions, resulting in 75 discussion-solution pairs, where solution represents the final solution obtained by MLE-STAR. Using LLM as a judge with the prompt in Figure 28, we found that all the final solutions generated by MLE-STAR with Gemini-2.0-Flash were judged to be sufficiently novel compared to the top discussions. Note that we use Gemini-2.5-Pro to judge the novelty of MLE-STAR’s solutions.

I Broader impacts

By automating complex ML tasks, MLE-STAR could lower the barrier to entry for individuals and organizations looking to leverage ML, potentially fostering innovation across various sectors. In addition, as state-of-the-art models are updated and improved over time, the performance of solutions generated by MLE-STAR is expected to be automatically boosted. This is because our framework leverages a search engine to retrieve effective models from the web to form its solutions. This inherent adaptability ensures that MLE-STAR continues to provide increasingly better solutions as the field of ML advances.

J Related works on data science agents

While our work focuses on LLM-based agents tailored for machine learning engineering, other research explores agents for general data science tasks [49, 50, 51], including data analysis and visualization. Among these, Data Interpreter [11] employs a graph-based approach, dividing tasks into subtasks and refining the task graph based on successful completion. DatawiseAgent [52] proposes a two-stage process: initially generating a tree-structured plan, followed by an exploration of the solution space. Although these methods exhibit generalizability to various data science tasks, including aspects of machine learning engineering, their evaluation prioritizes overall task completion rates rather than performance on specific engineering challenges.

K Cost analysis

Table 12: **Cost analysis on token usage.** Experiment results on 4 tasks from MLE-bench Lite, repeated three seeds using Gemini-2.0-Flash. We report the mean token usage.

Method	# Input tokens	# Output tokens
AIDE [12]	194K	38K
MLE-STAR (Ours)	874K	175K

We acknowledge that MLE-STAR requires higher cost due to increased token usage. As shown in the Table 12, MLE-STAR requires 4.5 times more input and output tokens compared to AIDE. This is primarily because of our method generates longer scripts during the targeted refinement and ensembling stages (*i.e.*, average line of the solution code for MLE-STAR in MLE-bench Lite is 414, while that for AIDE is 147). Still with Gemini-2.0-Flash (cost is \$0.15/0.60 per 1M input/output tokens), the cost of MLE-STAR is only about \$0.24 per each ML challenge. Despite the higher token cost, it is important to note that MLE-STAR consistently and significantly achieves better results than AIDE.

L Discussion on potential plagiarism.

Table 13: Similarity of MLE-STAR’s solution code and associated notebooks from each competition.

Competition ID	Submission #1	Submission #2	Submission #3
aerial-cactus-identification	0.35	0.42	0.39
aptos2019-blindness-detection	0.27	0.29	0.27
dog-breed-identification	0.32	0.30	0.32
dogs-vs-cats-redux-kernels-edition	0.33	0.27	0.31
histopathologic-cancer-detection	0.32	0.36	0.35
histopathologic-cancer-detection	0.32	0.36	0.35
leaf-classification	0.27	0.31	0.30
plant-pathology-2020-fgvc7	0.31	0.35	0.39
ranzcr-clip-catheter-line-classification	0.30	0.33	0.34
siim-isic-melanoma-classification	0.24	0.27	0.24
denoising-dirty-documents	0.25	0.27	0.27
detecting-insults-in-social-commentary	0.34	0.16	0.18
jigsaw-toxic-comment-classification-challenge	0.30	0.27	0.30
random-acts-of-pizza	0.37	0.39	0.37
spooky-author-identification	0.20	0.21	0.28
new-york-city-taxi-fare-prediction	0.35	0.33	0.34
nomad2018-predict-transparent-conductors	0.24	0.27	0.23
tabular-playground-series-dec-2021	0.26	0.19	0.29
tabular-playground-series-may-2022	0.24	0.32	0.20
text-normalization-challenge-english-language	0.22	0.19	0.20
text-normalization-challenge-russian-language	0.18	0.23	0.22

In Table 13, we report a similarity score compared against the top associated notebooks from each Kaggle competition, measured by Dolos [48], which is a source code plagiarism detection tool. Note that the results of the mlsp-2013-birds competitions have been omitted because MLE-STAR was unable to generate submissions using Gemini-2.0-Flash. Additionally, the results of the icml-2013-whale-challenge-right-whale-redux competition have also been omitted because the authors of MLE-bench did not provide the relevant notebooks.