A Related Work

A.1 Data-science benchmarking

Assessing agents' performance on data science tasks has been approached from various perspectives. While many benchmarks, such as those for coding proficiency [26, 71], focus on general code generation capabilities, they do not necessarily capture the full range of skills required for end-to-end data science. Assessing the performance of agents on data-science tasks has been considered from different scopes.

Some benchmarks are tailored to specific aspects of data science. For example, [32] focuses on code infilling based on data from Stack Overflow and common libraries, targeting localised code generation rather than comprehensive pipeline development.

Unlike traditional AutoML benchmarks [14], which typically provide a standardised input format to facilitate solver design, data science agent benchmarks evaluate broader capabilities. These include understanding the task requirements and adapting solutions accordingly, offering flexibility in problem formulation and solution strategies.

Authors in [24] introduce a benchmark using six Kaggle tasks, where success is defined as achieving a 10% improvement over a baseline provided by starter code. However, this baseline can sometimes represent minimal predictive performance, such as random predictions, rather than a robust comparison against established leaderboards. This benchmark also offers predefined scripts and environments for each task, guiding the problem-solving approach. Other benchmarks, such as [5], emphasise the orchestration of workflows and interactions with professional software tools, while [22] provides a carefully curated set of questions requiring language models to generate code that answers specific dataset-related queries. This differs from end-to-end problem-solving in competitions like Kaggle, whose questions are primarily focused on tabular data. Additionally, [54] centres on understanding machine learning repositories rather than the broader data science process, which limits its applicability to end-to-end DS agent evaluation. [69] uses Kaggle datasets to derive specific questions about dataset characteristics, such as imbalance, missing data, and correlations. Although this approach provides a focused evaluation, it does not capture the full scope of capabilities required for complete data science workflows, which is a broader objective we aim to achieve.

Closer to our experimental study, [28] curates a list of 74 Kaggle competitions to assess the data modelling capacity of LLM agents. Nevertheless, the authors pick the benchmark competitions based on the simplicity of their raw structures, featuring only a train, test and sample submission file. Moreover, they do not support comparison with the Kaggle leaderboard but split the training set to create a custom test set instead. Likewise, MLE-bench [6] includes 75 machine learning competitions from Kaggle to assess agent performance by comparing it against user submissions. However, because this assessment relies on a custom split of the training data rather than the competition's actual private test set, it remains uncertain whether an agent's high ranking in this context would align with results on the original Kaggle leaderboard.

Our experiments aim to truly assess the end-to-end capacity of agents to achieve high performance in data-science tasks autonomously.

A.2 Traditional automated data science (Auto-DS)

Achieving performance and automation in data science tasks has been a long-standing challenge. Numerous techniques have been developed to automate the human-intensive steps of the DS pipeline using machine learning. Given that each step in the DS pipeline poses challenges, many approaches have focused on specific aspects of the problem, often resulting in narrow application ranges or use-case limitations. For example, probabilistic inference rules have been used to automate data cleaning [46], while various deep learning methods have been developed to filter, select, or transform features to enhance the accuracy of downstream models [29, 30, 31]. In the modelling domain, a significant research area known as neural architecture search (NAS) optimises the structure of neural networks for predictions [36, 62], shifting the manual effort from model design to the design of a model search space. Beyond neural architecture, many hyperparameters can be automatically tuned to achieve better performance using black-box optimisation methods. Techniques like Bayesian optimisation address black-box optimisation in a sample efficient way and are supported by various libraries [2, 4, 9, 11], which users can utilise by correctly identifying and providing the hyperparameters of their pipelines. Additionally, meta-learning has been introduced to further improve data efficiency by leveraging past experiments to predict performance on new tasks based on related tasks solved with similar configurations [21, 39, 41].

Contrasting with these targeted methods, several libraries have been developed to cover broader parts of the data science pipeline, such as H2O-AutoML [33], AutoKeras [27], Auto-sklearn [13], and AutoGluon [55]. These tools aim to automate and optimise by making rule-based decisions based on input data modality and combining a fixed collection of models that can be refined using preset optimiser tools. However, these approaches have limitations, including restricted scope, flexibility, and usability barriers. Their performance is limited by the set of predefined models and hyperparameters they support, and they require expert knowledge to properly format the problem [7] (e.g., manual specification of column types for AutoKeras). Tools like H2O-AutoML are also limited to tabular data tasks, requiring data to be structured in specific ways.

A.3 Data-science Agents

The emergence of data-science agents powered by large language models and multi-agent systems represents a shift towards more autonomous and adaptable solutions. Unlike traditional Auto-DS approaches, these agents are designed to perform end-to-end tasks by leveraging advanced reasoning and decision-making capabilities.

A.3.1 Coding Agents

Recent advancements in LLMs [1, 25, 40, 47], combined with techniques like chain of thought prompting [61, 70] and ReAct [66] have significantly expanded the scope of AI applications beyond natural language processing. These models are now capable of generating code and interacting with APIs [10, 49, 44], laying the groundwork for their integration into multi-agent systems [16, 20, 56, 57, 63] that can execute complex tasks by thinking, planning, calling tools [45], and making decisions [23, 34, 48, 67] across various domains, including robotics [38, 42, 51], medicine [53, 58], and also data science [15, 19, 68]. DS agents exploit the intrinsic capacities of LLM-Agents. They are augmented with specific tools for writing, executing and debugging code, data visualisation tools [65], or access to standard ML libraries APIs [50], offering a more integrated approach to data science automation.

A.3.2 Intelligent DS assistants

Several agents function as intelligent assistants, enhancing human decision-making by providing automated solutions while keeping users in the loop. For instance, `CleanAgent` [43] handles data cleaning based on user specifications, and `LAMBDA` [52] generates data analysis reports based on technical queries. `MatPlotAgent` [65] pairs a coding agent with a visual agent to meet the expert's needs in terms of data visualisation, while `WaitGPT` [64] offers a smoother monitoring of the code generation by creating graph representations of the generated code and supporting on the fly manual edition or questions. Although these agents enhance user efficiency, they require substantial user input. They are not designed to autonomously achieve high performance in complex, real-world scenarios, such as winning a Kaggle competition.

A.4 End-to-end DS agents

End-to-end data-science agents aim to manage the entire data science pipeline autonomously, from data preprocessing to model evaluation and hyperparameter optimisation, without requiring significant human oversight. While some existing approaches address specific aspects of this workflow, such as `CAAFE` [18] which allows agents to iteratively generate codes to modify tabular features and reflect based on accuracy feedback, or `EVAPORATE-CODE+` [3], which focuses on data mining by generating code to extract structured data from semi-structured documents, other methods have tackled the entire data science pipeline more holistically.

Recent multi-agent systems like `SEED` [7], `AutoM3L` [37], `Data Interpreter` [19], `AutoML-Agent` [60], `HuggingGPT` [50], `MLCopilot` [68], `DS-Agent` [15], `SELA` [8], `MLZero` [12], `AutoKaggle` [35] and `AIRA-dojo` [59] achieve capacities to deal with data-science by empowering their system with numerous tools, using retrieval from expert demonstrations, and supporting complex multiple steps reasoning.

These systems employ various strategies for generating automated pipelines for different stages of the data science process such as preprocessing, modelling, and feature selection.

While `DS-Agent`, `MLCopilot` , `AIDE` and `AIRA-dojo` generate and refine the entire solution script as a single block, `SEED` performs a skyline-based optimisation to select the modules to apply to solve the task. Alternatively, `HuggingGPT` delegates the planning to a high-level agent, and `Data-Interpreter` lets this agent decide on a hierarchical and dynamic plan. Meanwhile, `AutoM3L` follows a pre-defined modality-specific pipeline to produce final predictions. Our data-science agent integrates elements from these various approaches by imposing a modality-dependent structure during the setup scaffold phase while allowing agents greater flexibility in selecting the optimal sequence of steps to achieve the best possible outcome.

Moreover, several existing systems also incorporate mechanisms for continual learning, enabling them to leverage past experiences to improve performance on new tasks.

For instance, `DS-Agent` achieves that by adopting a case-based reasoning approach combined with an offline-online setup, retrieving insights during the online phase from a pool of scripts that performed well on offline tasks. Similarly, `MLCopilot` divides its framework into offline and online phases, using an LLM to infer verifiable machine learning strategies from diverse offline experiments, such as determining valid learning rate ranges for specific models. Beyond building a knowledge database to retrieve from, `Data-Interpreter` uses prior experiences to create new tools that the agents can later call.

However, these systems theoretically or practically suffer from partial coverage or limited capacities. Indeed, `AutoM3L` cannot generate solutions involving model ensembling and requires specifically structured input data, while `HuggingGPT` lacks support for model training and tabular models. Moreover, `SEED` and `DS-Agent` do not demonstrate capabilities for handling computer vision tasks, and `AIDE`, `AutoKaggle`, and `Data Interpreter` have only shown strong performance on more straightforward tabular datasets or Kaggle competitions. `AutoML-Agent` requires comprehensive task-specific prompts which can be advantageous for experts customising its performance but poses a challenge for less-experienced users. Additionally, while some efforts have been made to utilise past experiments for new task resolution, the strategic selection of task order to enable progressive learning remains an underexplored area. We aim to address these gaps by developing a system capable of handling various input modalities and achieving consistently high performance in real-world data science competitions.

Beyond these practical continual learning strategies, there is a rich literature on online learning and online convex optimization [17] that provides strong theoretical guarantees for regret minimization in sequential decision-making. These methods assume a well-defined and typically convex loss structure, with each step involving an update based on newly observed feedback. While valuable for formal analysis, such frameworks are generally restricted to lower-dimensional, well-specified optimisation problems and do not directly address the high-dimensional, multi-modal, and tool-rich environment faced by end-to-end DS agents in real-world competitions. Our approach instead integrates flexible decision-making across heterogeneous subtasks, where the feedback signal may be sparse, delayed, or only indirectly comparable (e.g., via leaderboard or validation scores).

B Agent K - Pipeline Details

According to our formulation, the design of the agent framework is broken down into two main parts: the *setup* pipeline and the *solution generation* pipeline. The setup pipeline's goal is to preprocess the raw information and raw data to prepare a workspace and a `DataLoader` object directly usable by the solution generation pipeline to solve the task. All the steps related to downloading, summarising, organising and cleaning the data are done in the setup pipeline, whereas steps related to feature engineering, model training, hyperparameter optimisation and submission belong to the prediction pipeline. In this section we give more detail about the pipeline steps, their objective and important aspects of how they work.

B.1 Setting up Competitions

The goal is to automate the process of setting up a task. The pipeline, presented in Figure 7 depicts the sequence of stages that Agent K tackles in order to setup a new Kaggle competition, given its URL.

B.1.1 Expected Result - Usable & Interpretable Workspace

At the end of a successful setup, we expect to have a *workspace* containing useful files, including raw information and raw data from the competition website, as well as new files created by Agent K, such as metadata, different CSV files as well as runnable code files. Our agent creates additional CSV files only if it deems it necessary from the context of the competition:

- `train_tab_input_map.csv` - tabular inputs from the train set
- `train_txt_input_map.csv` - text inputs from the train set
- `train_img_input_map.csv` - paths to the image inputs from the train set
- `train_tab_target_map.csv` - tabular targets from the train set
- `train_txt_target_map.csv` - text targets from the train set
- `train_img_target_map.csv` - paths to the image targets from the train set
- `test_tab_input_map.csv` - tabular inputs from the test set
- `test_txt_input_map.csv` - text inputs from the test set
- `test_img_input_map.csv` - paths to the image inputs from the test set

When Agent K produces code to generate the CSV files, this code is kept in the workspace, which contributes to the interpretability of the result. Furthermore, our agent also need to implement the useful functions:

- `code_transform_tab_target_train.py` - code to transform the targets into a format usable for training models and back to the original format
- `code_metric.py` - the metric function implementation
- `code_submission_format.py` - code to transform from the predicted targets format to the exact format accepted by the competition (e.g. from a probability of a class to the label of that class)
- `code_submission_format_alt.py` - an alternative to the previous function, in case it fails. It can also be used as a backup for the solution generation pipeline when we format the predictions of the model for submission.

B.1.2 Setup Pipeline Steps

Fetch & Scrape Raw Data Given the URL of the competition, Agent K can use the Kaggle API to download its associated data. The agent is also given a scraping tool which automatically retrieves information from the competition's web page, such as the raw description of the competition, its data description, metric description, etc.

Summarise From the scraped raw descriptions, Agent K creates digestible summaries that it can later reuse for LLM prompting. The raw text from the competition pages can contain elements that are irrelevant to the task (e.g. emojis, images, urls, ...) or worse, information that is misleading. Summarising the raw descriptions can deal with those issues and enables the agent to compress the important information from the task into a more useful and shorter text that takes less tokens in the future prompts.

Detect Modalities We consider three main modalities: *vision* data, *language* data and *tabular* data, which can be purely tabular or contain time-series. The agent categorises all competitions' inputs and targets with respect to these three modalities, based on the summarised descriptions. For instance, a competition where participants are asked to create an image classifier would have image inputs and tabular targets (classification). In another case, if a competition consists in attributing a toxicity score to movie reviews and includes as input the reviews as well as the categories of the movies, then the input modalities would be a mixture of text and tabular, while the modality of the target would be tabular (regression).

Create Plan & Metadata After the raw data is fetched, the summaries are written and the modalities are detected, Agent K creates its plan, i.e. its workflow. This plan determines the sequence of steps that should be done in order to complete the setup. This plan is also *dynamic* in the sense that our agent can decide to prioritise stages over others provided it is permitted by the general sequence order of the environment. Agent K can come back to previous stages when executing its plan, and it can attempt some stages multiple times if they haven't been done properly the first time, as we detail in the paragraph dedicated to unit tests.

Create Maps & Transforms The main task in the setup pipeline is for the agent to create *maps* and *transforms*, as outlined in Section B.1.1. The idea is to split all raw training data into input and target maps, as well as to split all raw data (train and test) into different modalities. A task could have multiple types of inputs and targets so we require a map for each of them. These maps are then collated together in a custom `DataLoader` object that makes it easy to consume by a model in the prediction pipeline. Importantly, the agent is also tasked with creating transform functions in order to treat targets as consumable data for a model. For instance if the targets in the training set are text-encoded labels (in a classification task) the *transform* function could map these to a one-hot encoding and the *inverse transform* function would map them back to the text labels. These functions are crucial to enable the prediction pipeline.

Create Submission Format & Metric Function Finally, Agent K generates code to create the submission format and the metric function. Both are generally described in the raw competition description. The submission format is the exact format expected by the competition to read the predictions. It can sometimes be different from the format of the raw train targets so in addition to the transforms, we also need a function that maps the predicted targets on the test inputs, in the exact format expected.

B.1.3 Unit Tests

An important part of the setup pipeline is the set of unit tests used to assert if a stage or a group of stages has been successfully done by the agent. It is worth noting that these unit tests are manually designed and not generated by the agent itself. However, **this does not mean that they are tailored to each task**. In fact they are designed to be agnostic to the competition to setup and are rather specific to the framework itself. Knowing the properties of the expected result of a particular stage, we can test if the code or the data our agent generate follows an expect format. This way our agent receives insightful feedback through error messages when its code is faulty, which it integrates into its reasoning process to correct itself.

For example, suppose our agent generates a runnable code to create the training input map of the image modality. During the state transition of the environment, the unit test associated to this stage is ran. The unit test checks that a CSV file with a specific name exists and is loads it. It then assesses that the table has a column named "`id`" and that it has at least one row. It also checks the existence of a column apart from the "`id`" column and that these other columns contain paths to images with known extensions. Moreover, the unit test throws an error if one of the image paths listed in that map is invalid. Finally it can check various trivial things such as non emptiness of columns or absence of duplicated columns. This shows that the unit tests are specific to the setup environment and not specific to the task.

If an error is raised in the unit test, the error is saved in memory and the current stage is attempted again. In this new trial, our agent will be able modify and fix its previous code based on the unit test error message that is accessible in its memory. This is repeated until the unit test passes or a budget limit is reached (in which case the setup would fail there).

Finally we have *meta*-unit tests which are unit tests that depend on multiple stages. They verify that the result of multiple single stages coordinate well together. If a *meta*-unit test fails, the agent has the opportunity to go back multiple stages and re-attempt the relevant stages. For instance, after all maps and transforms are done for the training data, we test if the `DataLoader` can be constructed properly and if we can successfully load a batch. This step is crucial as it validates that all the maps generated by the agent up to this point share a common "`id`" column so the `DataLoader` can collate them properly. If this test fails, the agent has to go back to the start and figure out which codes to correct given the error raised in the *meta*-unit test to eventually correct its mistakes.

This paragraph highlights the usefulness of the unit tests, showing that they do not only serve as checks for the agent, but that they also provide useful error messages to fuel effective reflection. The *meta*-unit tests also provide additional verifications that can only be done once a group of stages have passed their own individual unit tests. However, there is always a limit to what we can test. It is possible that the agent finds a way to pass all unit

Table 5. Hyperparmeters used for the training of the solutions generated during the scaffold stage.

Name	Value	Role
`max_epochs`	30	Max training epochs allowed
`max_time`	10h	Max training time allowed
`batch_size`	32	Training & testing batch size
`lr`	$[1e-6, 1e-2]$	BO search space for learning rate
`optimizer`	{Adam, SGD, AdamW}	BO search space for optimiser
`n_trials`	20	Max BO rounds allowed
`k_folds`	5	Cross-Validation folds
`tta_rounds`	4	Number of TTA rounds averaged
`blend_after`	3	max nb. of solutions before blending

tests but ends up setting up the competition incorrectly. Because we want our unit tests to remain task-agnostic we cannot extend our coverage indefinitely. In the future we would like to extend this work to also use the agent to generate additional unit tests that would be specific to the current competition being setup. In that way we can extend the coverage at inference-time.

B.2 Generating Solutions

After the agent has generated code that constitutes a fully-functioning solution, and while training that solution, we allow for some optimisations and support such as creating additional submission files to give Agent K more choice. These are part of the scaffold and are not directly coded by the agent itself but are there rather as a design choice.

First, we do Bayesian Optimisation (BO) over some parameters, i.e. the learning rate and the optimiser choice. The training is run for a maximum of `n_trials` trials, each trial being limited to `max_epochs` epochs or `max_time`, whichever is attained first.

We then generate Cross-Validated submissions by training the model with the best learning rate and optimiser found during BO, on `k_folds` folds of the training set. We then average the predictions to create a new submission file. For the competitions involving images, we also add a submission file created via simple Test-Time-Adaptation (TTA). This method augments the data at test time with the same data augmentation pipeline used for the training dataloader. Using the best checkpoint saved from the training, we reproduce this for `tta_rounds` rounds and average the predictions to create a new submission file.

Finally, blending is used after at least `blend_after` full working and different solutions have been generated. Values are specified in Table 5.

B.3 Pipeline Pseudocodes

We give in Algorithms 1 and 2 the pseudocode for the setup pipeline and the pseudocode for the solution generation pipeline, respectively. Note that the input of the setup pipeline is the competition URL and the outputs are the setup files and the dataloaders. The solution

Algorithm 1 Pseudocode for the Setup Pipeline

Inputs: URL

$\mathcal{D} \leftarrow$ FetchAndScrapeData(URL)
$\mathcal{S} \leftarrow$ SummariseDataAndTaskDetails($\mathcal{D}$)
$\{\mathcal{X}, \mathcal{Y}\} \leftarrow$ DetectModalities($\mathcal{S}$)
$\mathcal{P} \leftarrow$ CreatePlan($\{\mathcal{X}, \mathcal{Y}\}, \mathcal{S}$)
▷ *Every* **while** loop in the pipeline is subject to a retry limit
while *meta*-unit test fails **do**
    while unit test fails **do**
        $\mathcal{M}_{\text{input}}^{\text{train}} \leftarrow$ CreateTrainInputMaps($\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}$)
    end while
    while unit test fails **do**
        $\mathcal{M}_{\text{target}}^{\text{train}} \leftarrow$ CreateTrainTargetMaps($\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}$)
    end while
    while unit test fails **do**
        $\mathcal{T}_{\text{target}}^{\text{train}} \leftarrow$ CreateTrainTargetTransforms($\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}$)
    end while
end while

while *meta*-unit test fails **do**
    while unit test fails **do**
        $\mathcal{M}_{\text{input}}^{\text{test}} \leftarrow$ CreateTestInputMaps($\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}$)
    end while
end while

while unit test fails **do**
    $\mathcal{F}_{\text{submission}} \leftarrow$ CreateSubmissionFormat($\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}$)
end while

while unit test fails **do**
    $\mathcal{F}_{\text{metric}} \leftarrow$ CreateMetric($\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}$)
end while

Outputs: Train and test dataloaders $\texttt{DL}_{\text{train}}$ and $\texttt{DL}_{\text{test}}$

generation pipeline takes as inputs the outputs of the setup pipeline and carries on generating code to create one or more submission files. The output of the solution generation pipeline is a set of solutions along with their performance on the leaderboard.

Retry Budget Note that in the pseudocode, there are steps such as "**while** … **end while**". In practice, we do not retry each step indefinitely, there is an upper bound on the number of retries we allow Agent K to do for each step. As soon as this number is reached we consider the agent to be stuck and we end the episode by raising an error.

Algorithm 2 Pseudocode for the Solution Generation Pipeline

Inputs: $\mathcal{S}_{\text{setup}} \leftarrow \{\mathcal{D}, \mathcal{S}, \{\mathcal{X}, \mathcal{Y}\}, \mathcal{P}, \mathcal{M}^{\text{train}}_{\text{input}}, \mathcal{M}^{\text{train}}_{\text{target}}, \mathcal{M}^{\text{test}}_{\text{input}}, \mathcal{T}^{\text{train}}_{\text{target}}, \mathcal{F}_{\text{submission}}, \mathcal{F}_{\text{metric}}\}$, $\texttt{DL}_{\text{train}}$ and $\texttt{DL}_{\text{test}}$, Embedders $\mathcal{E} \leftarrow \{\}$, Heads $\mathcal{H} \leftarrow \{\}$, Solutions $Sol \leftarrow \{\}$

if *tabular-only* task **then**
  AdoptClassicalMLFramework()
  $Sol \leftarrow$ UseRampTool$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}}, \texttt{DL}_{\text{test}})$
else
  AdoptDNNFramework()
  while Agent decides to improve performance **do**
    ▷ Create Input Encoder(s)
    if $\mathcal{S}_{\text{setup}}$ has tabular inputs **then**
      $\mathcal{F}_{\text{f.e.}} \leftarrow$ TabularFeatureEngineering$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}})$
      $\mathcal{E}_{\text{tab}} \leftarrow$ TabularEmbedding$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}})$
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{F}_{\text{f.e.}}, \mathcal{E}_{\text{tab}}\}$
    end if
    if $\mathcal{S}_{\text{setup}}$ has image inputs **then**
      $\mathcal{F}_{\text{img}} \leftarrow$ ImageTransforms$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}})$
      $\mathcal{E}_{\text{img}} \leftarrow$ ImageEmbedding$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}})$
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{F}_{\text{img}}, \mathcal{E}_{\text{img}}\}$
    end if
    if $\mathcal{S}_{\text{setup}}$ has text inputs **then**
      $\mathcal{F}_{\text{txt}} \leftarrow$ TextPreprocessing$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}})$
      $\mathcal{E}_{\text{txt}} \leftarrow$ TextEmbedding$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}})$
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{F}_{\text{txt}}, \mathcal{E}_{\text{txt}}\}$
    end if
    ▷ Create Target Head(s)
    $\mathcal{H}_{\text{tab}} \leftarrow$ TabularHead$(\mathcal{S}_{\text{setup}})$
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{\mathcal{H}_{\text{tab}}\}$
    ▷ Generate `submission.csv` and get validation score
    $\texttt{sub}, val \leftarrow$ TrainSolutionAndPredict$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}}, \texttt{DL}_{\text{test}}, \mathcal{E}, \mathcal{H})$
    $Sol \leftarrow Sol \cup \{\texttt{sub}\}$
    if enough solutions created **then** ▷ Blend existing solutions
      $\texttt{sub}, val \leftarrow$ BlendSolutionsAndPredict$(\mathcal{S}_{\text{setup}}, \texttt{DL}_{\text{train}}, \texttt{DL}_{\text{test}}, \mathcal{E}, \mathcal{H}, Sol)$
      $Sol \leftarrow Sol \cup \{\texttt{sub}\}$
    end if
  end while
  ▷ Send submissions and get private score and rank
  $r \leftarrow$ SendSubmissionandGetLeaderboardRank$(\texttt{sub})$
end if

C Competition difficulty analysis

As detailed in the "Quantitative Results" section, Agent K's achievements include several gold medals across different types of competitions — community, playground, research, and featured — however, some of these competitions are less complex than others, and not all competitions award medals according to the Kaggle system. In this section, we provide further insights on competitions' difficulties by measuring the levels of their participants. To do so, we analyse the Elo scores of all Kaggle participants, going through all the medal-awarding competitions on Kaggle, and updating the Elo-MMR of the competitors based on their private ranks (the Elo score of a new participant being initialised at 1,500). Doing so, we can access for each competition (medal-awarding or not), the Elo scores of the participants at the time this competition ended.

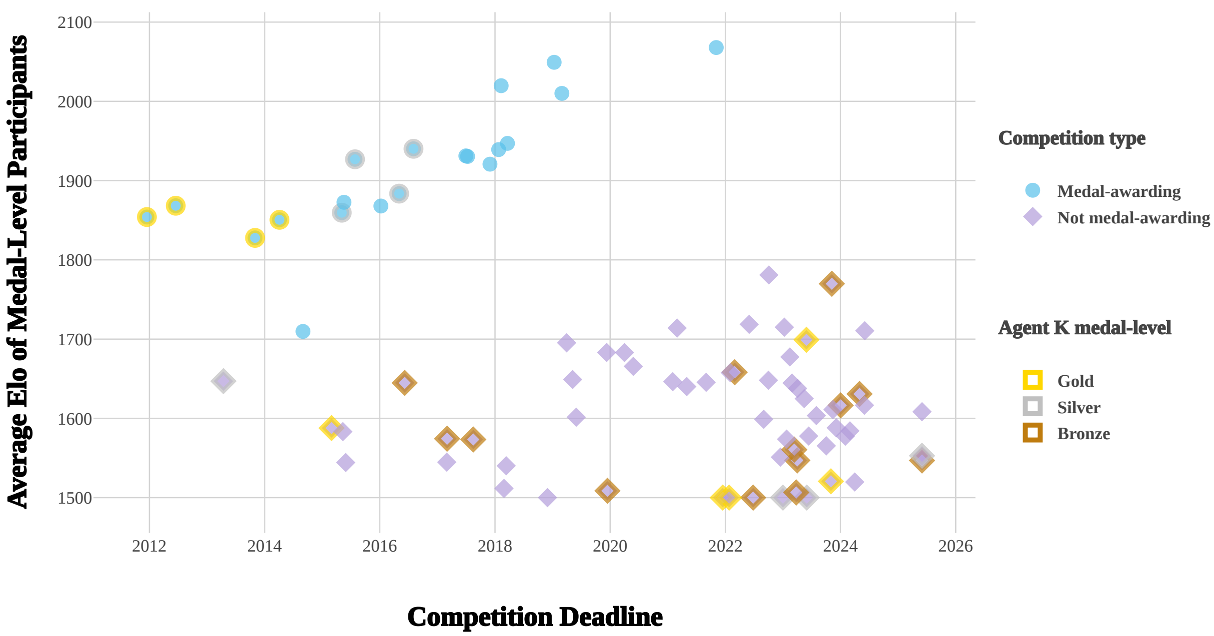

Figure 12. Competition difficulty analysis. Each marker in the plot represents a Kaggle competition included in our benchmark. The x-axis indicates the competition's closing date, while the y-axis shows the average Elo score of participants who achieved at least a bronze medal (or a bronze medal-level score for non-medal awarding competitions). For competitions in which Agent K attained a medal-level score, the marker is outlined in the corresponding medal color. Notably, since Agent K achieved 4 gold medals across medal-awarding competitions, the figure includes 4 blue circular markers with gold outlines.

We present the results of our analysis on Figure 12 where we display for each competition included in our benchmark, the average Elo scores of the participants that have reached at least a bronze medal-level score. We observe a gap between the average Elo score on the non-medal-awarding competitions (purple diamond markers) and on the medal-awarding ones (blue circle markers), which confirms the idea that stronger data scientists tend to participate more in medal-awarding competitions. Moreover, we highlight the medal-level scores achieved by Agent K by outlining the competition markers in the colors of the medals. For instance, the 4 gold (resp. 4 silver) medals obtained by Agent K on medal-awarding competitions appear as golden (resp. silver) disks on the figure, with their y-position matching the average Elo of the participants who won at least a bronze medals on these competitions.

This analysis pushes us to further improve our performance on the competitions involving stronger data-scientists.

D Benchmark details

We provide the list of competitions included in our benchmark in Table 7.

Table 6. Symbols associated to competition types and competition categories on Kaggle.

Competition Type	Meaning	Kaggle Category	Meaning
RG	Regression	GS	Getting Started
BC	Binary Classification	P	Playground
MC	Multiclass Classification	F	Featured
MTR	Multitarget Regression	C	Community
MTC	Multitarget Classification	RS	Research

Table 7. List of the Kaggle competitions included in our benchmark. Each row represents a Kaggle competition, showing its Kaggle ID (first column), its type, its category on Kaggle, the input modalities it involve, the number of submissions that can be retained for the final scoring (#Select Subs), and the number of teams that participated in this competition. Note that the column # Select Subs corresponds to k_c in equation (1). See Table 6 for the symbols appearing in the Competition Type and Kaggle Type columns.

Competition	Competition Type	Kaggle Category	Modality	#Select Subs	#Teams
nlp-getting-started	BC	GS	Tab,Txt	≥ 100	856
sentiment-analysis-on-movie-reviews	MC	P	Txt	2	860
dogs-vs-cats-redux-kernels-edition	BC	P	Img	2	1314
dog-breed-identification	MC	P	Img	1	1280
state-farm-distracted-driver-detection	MC	F	Img	2	1438
histopathologic-cancer-detection	BC	P	Img	2	1149
whereami	MC	C	Img	20	211
5-flowers-image-classification	MC	C	Img	20	69

Competition	Task type	Kaggle Type	Modality	#Select Subs	#Teams
sign-language-image-classification	MC	C	Img	2	41
ml-olympiad-landscape-image-classification	MC	C	Img	2	13
noaa-right-whale-recognition	MC	RS	Img	2	364
humpback-whale-identification	MC	F	Img	2	2120
planttraits2024	MTR	RS	Tab,Img	1	398
invasive-species-monitoring	BC	P	Img	2	511
tabular-playground-series-feb-2021	RG	P	Tab	2	1433
playground-series-s3e11	RG	P	Tab	2	952
playground-series-s3e14	RG	P	Tab	2	1875
playground-series-s3e15	RG	P	Tab	2	693
home-data-for-ml-course	RG	GS	Tab	2	106930
bike-sharing-demand	RG	P	Tab	2	3242
tabular-playground-series-aug-2021	RG	P	Tab	2	1753
forest-cover-type-prediction	MC	P	Tab	2	1692
mercedes-benz-greener-manufacturing	RG	F	Tab	2	3823
goodreads-books-reviews-290312	MC	C	Tab,Txt	5	261
sberbank-russian-housing-market	RG	F	Tab	2	3264
restaurant-revenue-prediction	RG	F	Tab	2	2257
scrabble-player-rating	RG	P	Tab	2	301
nlp1000-ml-challenge	BC	C	Txt	2	16
nlpsci	MC	C	Txt	20	44
sf-crime	MC	P	Tab	2	2331
otto-group-product-classification-challenge	MC	F	Tab	2	3507
nitro-language-processing-3	BC	C	Txt	3	45
sbu-ai-finalproject	MC	C	Txt	5	37

Competition	Task type	Kaggle Type	Modality	#Select Subs	#Teams
avito-prohibited-content	BC	F	Tab,Txt	2	284
playground-series-s3e1	RG	P	Tab	2	689
playground-series-s3e9	RG	P	Tab	2	765
playground-series-s3e16	RG	P	Tab	2	1429
playground-series-s3e25	RG	P	Tab	2	1632
tabular-playground-series-jan-2021	RG	P	Tab	2	1728
tabular-playground-series-aug-2022	BC	P	Tab	2	1888
tabular-playground-series-apr-2021	BC	P	Tab	2	1244
playground-series-s4e4	RG	P	Tab	2	2606
ventilator-pressure-prediction	RG	RS	Tab	2	2605
playground-series-s4e5	RG	P	Tab	2	2788
porto-seguro-safe-driver-prediction	BC	F	Tab	2	5156
playground-series-s3e3	BC	P	Tab	2	665
santander-customer-satisfaction	BC	F	Tab	2	5115
playground-series-s3e5	RG	P	Tab	2	901
cat-in-the-dat	BC	P	Tab	2	1338
playground-series-s4e1	BC	P	Tab	2	3632
tabular-playground-series-sep-2022	RG	P	Tab	2	1381
cat-in-the-dat-ii	BC	P	Tab	2	1161
playground-series-s3e22	MC	P	Tab	2	1541
playground-series-s3e26	MC	P	Tab	2	1661
playground-series-s3e7	BC	P	Tab	2	678
jigsaw-toxic-comment-classification-challenge	MTC	F	Txt	2	4539
kaggle-pog-series-s01e03	MC	C	Tab,Img	2	91
vehicle	MC	C	Img	2	65
plant-seedlings-classification	MC	P	Img	1	833
sp-society-camera-model-identification	MC	F	Img	2	580

Competition	**Task type**	**Kaggle Type**	**Modality**	**#Select Subs**	**#Teams**
plant-pathology-2020-fgvc7	MC	RS	Img	2	1317
world-championship-2023-embryo-classification	BC	C	Img	2	111
leaf-classification	MC	P	Tab,Img	2	1595
nitro-language-processing-2	MC	C	Txt	3	46
playground-series-s3e24	BC	P	Tab	2	1908
playground-series-s4e2	MC	P	Tab	2	3587
tmdb-box-office-prediction	RG	P	Tab,Txt	2	1395
playground-series-s3e19	RG	P	Tab	2	1172
tabular-playground-series-may-2022	BC	P	Tab	2	1151
tabular-playground-series-jan-2022	RG	P	Tab	2	1591
dont-overfit-ii	BC	P	Tab	2	2315
statoil-iceberg-classifier-challenge	BC	F	Tab,Img	2	3330
diabetic-retinopathy-detection	MC	F	Img	2	660
stumbleupon	BC	F	Tab,Txt	2	624
h2oai-predict-the-llm	MC	C	Txt	2	92
human-protein-atlas-image-classification	MTC	F	Img	2	2160
predict-who-is-more-influential-in-a-social-network	RG BC	F	Tab	5	132
bioresponse	RG	F	Tab	5	698
GiveMeSomeCredit	RG	F	Tab	5	924
galaxy-zoo-the-galaxy-challenge	MTC	RS	Img	5	326
playground-series-s5e5	RG	P	Tab	2	4316

E Additional results

We show in Figure 13 the success rate of each stage of the setup pipeline for the different types of tasks. For each stage, we track if it was a success (i.e. associated unit test passed) or a failure. We also track if that stage was not reached, caused by the failure of a preceding stage. Finally we also track if that stage is skipped as not all stages are required for each competition. As described in Section B.1, the stages to complete depends on the nature of the task and notably of the modalities of the input data. Figure 13 shows the reason why a stage is skipped on the right-hand side of each bar. Finally, the success rate (percentage in the green portion of the bar) shown for each stage is reflective of the number of times this particular stage is successfully done by the agent, if this stage is reached, averaged over all competitions. This explains why, for example, there is a 77.8% success rate for the stage "Create Map for Inputs Images Train" with a bar that is larger than the 87.2% of the stage just below "Create Map for Inputs Text Train".

The takeaway from Figure 13 is that most stages have a high success rate. There is a weaker point for creating the training image input map. This can be explained by the often complicated of files structures of Kaggle competitions requiring image inputs. The relatively smaller percentages for the stages such as "Create Map for Inputs Text Train" or "Unit Test for the Text train maps" can be explained by the fact that there are less competitions in our benchmark that require text inputs. Nevertheless, across input modalities the overall stage success rate is high for Agent K. In particular the overall setup success rate is **92.5%**.

F Details on React-Agent baselines

This section outlines the modifications made to the AIDE version designed for MLE-Bench, to align it with the specific requirements and conditions of our experiments. These changes were necessary to ensure that the ReAct Agent from AIDE could effectively operate within the experimental setup, particularly in relation to open-source models and task-specific execution constraints.

Open-Source Models and Structured Output Generation A key modification involved adapting AIDE to support open-source models, such as Qwen2.5, that do not natively generate structured output. Originally, AIDE was designed to work with models that inherently produced structured results, simplifying downstream processing. However, open-source models like Qwen2.5 often produce unstructured or loosely structured text. In response to this, the input prompts given to the models were explicitly updated to request the generation of structured outputs, such as JSON format. Additionally, a post-processing step was introduced to parse and transform the raw LLM output into the required structured format. When issues arose, such as incomplete or misstructured output (e.g., missing expected keys), we sent a new request to the LLM, adding the formatting error to the prompt, in order to get a new valid output. If after five retrials the output is still not in the expected format, we stop the runs and restart from scratch, which happened only a few times.

Prompt Update The prompt to the LLM was updated to include detailed information about the Kaggle competition, such as an overview of the competition and a description

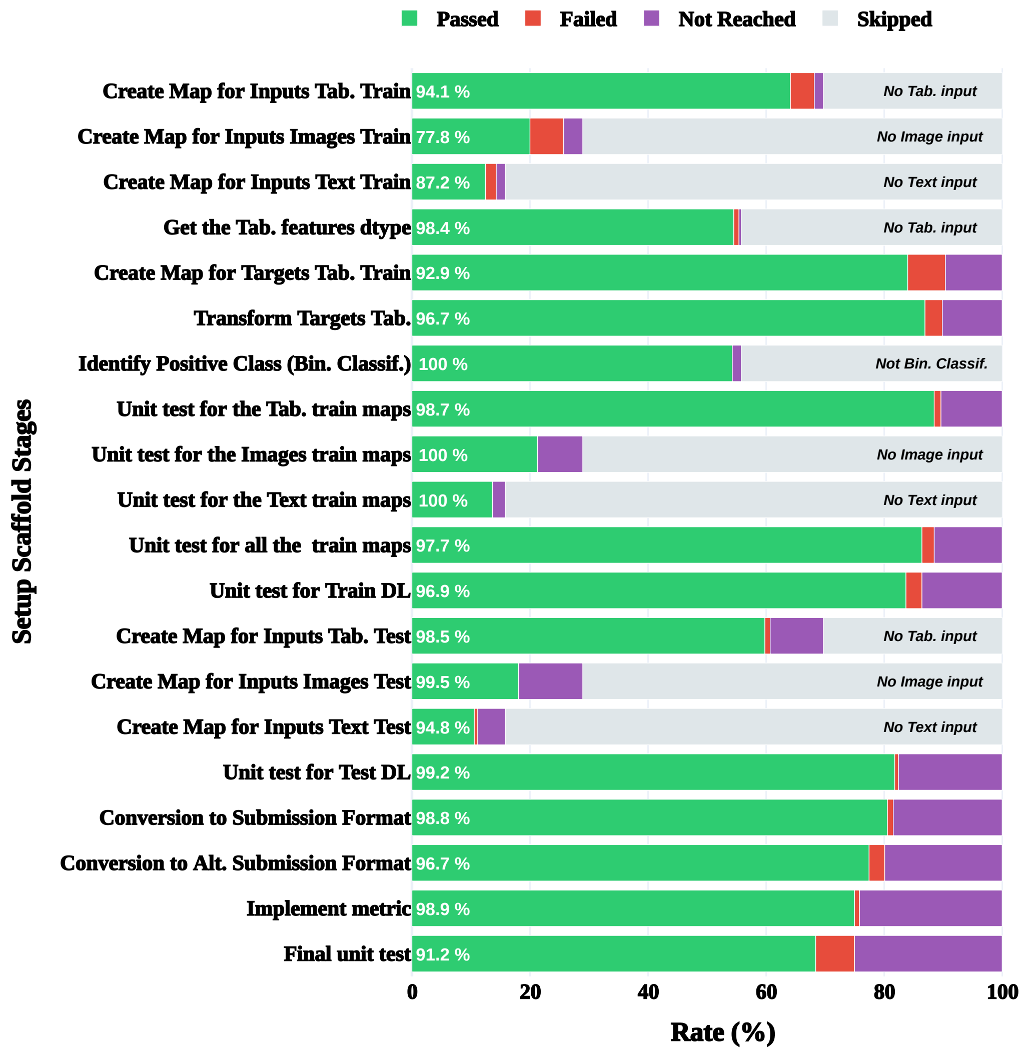

Figure 13. Success Rate per Setup Stage. On this percentage stacked bar graph, each row represents a stage in the process of setting up a competition. Its corresponding bar represents the percentage of times this stage is a success, or is failed, or is not reached, or is skipped, averaged over all competitions.

of the data. Additionally, the prompt was enhanced to provide system resource details, including the remaining total time, the total allowed number of iterations, and the number of iterations that had been completed. This update ensured that the LLM had access to comprehensive context for better performance and decision-making during the execution process.

Execution time and Best solutions All computer vision and natural language processing competitions were allocated a runtime budget of four days, while tabular competitions were assigned a two-day budget. This allocation was designed to provide the system with ample time to generate and refine high-quality solutions. For CV and NLP tasks, AIDE was configured with a maximum execution time of 18 hours per node, whereas tabular tasks were limited to 9 hours per node to reflect their generally lower computational requirements. During execution, AIDE was permitted to perform up to 5,000 improvement iterations to iteratively enhance its solutions. Additionally, the system was allowed a maximum debug depth of 3 from an initial node, with a 50% probability of selecting a given node for debugging. For all tasks, AIDE retained the top four solutions based on the competition-specific evaluation metric.

Kaggle for Real-Time Evaluation AIDE was integrated with Kaggle to facilitate real-time evaluation and validation of its generated submissions. Once the best four submissions were identified, they were submitted to Kaggle for public and private scoring. The resulting Kaggle scores were then used to assess the performance of each submission, providing an external benchmark for comparison and ensuring that the system's outputs were evaluated against established competition standards.

RAG Database setup The database for the ReAct (Qwen) + RAG baseline was carefully constructed by collecting 24 Kaggle competitions that began in or after 2021 and were not included in our benchmark set. The selected competitions span a balanced mix of domains, including 7 computer vision (CV), 7 natural language processing (NLP), 7 tabular data challenges, and 3 multimodal tasks, ensuring diverse coverage of data science problem types. Table 8 lists all the selected competitions, along with the number of notebook summaries and technical reports used to build the RAG database, as well as the corresponding year of each competition.

G Algorithmic Overview of TabPFN Fine-Tuning

Algorithms 3, 4, and 5 present the pseudocode for dataset sampling using KMeans clustering, prediction using TabPFN, and TabPFN fine-tuning, respectively. It is important to note that the input datasets for all TabPFN variants, including the baseline and fine-tuned models, are generated by the setup pipeline of Agent K.

Algorithm 3 KMeans Clustering-Based Sampling

Require: `ctx_length`, `train_data`, `test_data`

```
1: (train_data, test_data) ← Preprocess(train_data, test_data)
2: kmeans ← ApplyKMeansClustering(test_data)
3: (test_data, train_data) ← AssignClusterLabels(kmeans)
4: train_data_subset ← SampleTrainData(train_data, ctx_length)
5: return train_data_subset
```

H Examples of Prompts and Responses

Below are some examples of prompts and answers from the LLM that our agent can query. Even though they are presented hereafter in a chat format, note that the LLM does not keep a memory of the chat as generally thought of in recent chat-based systems. The prompts are long because at each interaction we add all the necessary information about the specific step the agent is performing, knowing that the queried LLM has no context or records

ID	Competition Name	Year	# Tech. Reports	# Notebooks
1	UBC-OCEAN	2024	10	10
2	womens-march-mania-2022	2022	7	10
3	predict-student-performance-from-game-play	2023	10	10
4	novozymes-enzyme-stability-prediction	2023	7	10
5	czii-cryo-et-object-identification	2025	10	10
6	jigsaw-toxic-severity-rating	2022	10	10
7	feedback-prize-english-language-learning	2022	10	10
8	icr-identify-age-related-conditions	2023	10	10
9	happy-whale-and-dolphin	2022	10	10
10	commonlitreadabilityprize	2021	10	10
11	amex-default-prediction	2022	10	10
12	rsna-miccai-brain-tumor-radiogenomic-classification	2021	10	10
13	rsna-2023-abdominal-trauma-detection	2023	10	10
14	commonlit-evaluate-student-summaries	2023	10	10
15	feedback-prize-effectiveness	2022	10	10
16	predict-energy-behavior-of-prosumers	2024	10	10
17	mens-march-mania-2022	2022	6	10
18	petfinder-pawpularity-score	2022	10	10
19	cassava-leaf-disease-classification	2021	10	10
20	llm-detect-ai-generated-text	2024	10	10
21	home-credit-credit-risk-model-stability	2024	2	10
22	rsna-breast-cancer-detection	2023	10	10
23	learning-agency-lab-automated-essay-scoring-2	2024	8	10
24	hpa-single-cell-image-classification	2021	10	10

Table 8. Overview of Inputs for RAG Database: Selected competitions, Year of the competitions, Number of technical reports and Number of Notebook summaries
.

Algorithm 4 TabPFN-Based Prediction

```
Require: ctx_len, train, test, is_regression
test_batches ← CreateBatches(test, ctx_len)
all_predictions ← [ ]
for all test_batch in test_batches do
    if len(train) > ctx_len then
        sampled_ctx ← KMeansSampling(ctx_len, train, test_batch)
    else
        sampled_ctx ← train
    end if
    x_ctx ← getFeatures(sampled_ctx)
    y_ctx ← getLabels(sampled_ctx)
    x_target ← getFeatures(test_batch)
    predictions ← TabPFN(x_ctx, y_ctx, x_target, is_regression)
    all_predictions.append(predictions)
end for
return all_predictions
```

Algorithm 5 TabPFN Fine-Tuning

```
Require: ctx_len, org_train, org_test, is_regression
train_data,val_data, test_data ← SplitDataset(org_train)
val_data_subsets ← CreateBatches(val_data, ctx_len)
train_data_subsets ← [ ]
for all v in val_data_subsets do
    sample ← KMeansSampling(ctx_len, train_data, v)
    train_data_subsets.append(sample)
end for
for i = 1 to len(train_data_subsets) do
    train_subset ← train_data_subsets[i]
    val_subset ← val_data_subsets[i]
    model ← LoadModel(model_path, is_regression)
    initial_loss ← EvaluateModel(model, train_subset, test_data)
    for all data in TrainDataLoader(train_subset) do
        model ← FineTune(model, data, val_subset)
        validation_loss ← EvaluateModel(model, data, test_data)
    end for
end for
all_preds ← Pred(model_path, org_train, org_test, is_regression)
return all_preds
```

of previous interactions. The prompts are therefore shortened for the purpose of display. The paths in the code snippets are taken from our runs so we also anonymised them e.g. `/PATH/TO/RAW/DATA/...` or `/PATH/TO/WORKSPACE/...`

A Note on Prompts The agent is performing a complex task broken up into a sequence of smaller subtasks. These include summarising information and also past responses. These summaries are the main bulk of the text present in the prompts. They include summarisation of the competition description, the raw data and all past steps the agent has done if relevant. The other big part of the prompts are views. These are usually a raw text output from a terminal command such as "`$ls`"or python "`print()`" statement, e.g. the view of the raw data directory or a head view of some dataframes along with their column types, etc. Finally, some prompts can contain examples of similar situations seen in similar competitions that the agent can pick itself and reuse as examples through RAG. The rest is a set of general instructions manually written.

H.1 Setup

The example below is taken from the setup of the competition `leaf-classification` which has tabular and image inputs. We show examples of prompts and responses from the LLM in the particular case where the agent creates a plan for coding and then fill a code template based on the plan. They contain all the necessary information about the task and instructions about the specific step the agent is currently doing for a successful answer. In the particular case below, the LLM is prompted to generate a plan and then the code to create the training map for the image modality, having already done the tabular modality.

Prompt

Data Description
Dataset Description
The dataset consists approximately 1,584 images of leaf specimens (16 samples each of 99 species) which have been converted to binary black leaves against white backgrounds. Three sets of features are also provided per image: a shape contiguous descriptor, an interior texture histogram, and a fine-scale margin histogram. For each feature, a 64-attribute vector is given per leaf sample. Note that of the original 100 species, we have eliminated one on account of incomplete associated data in the original dataset.

File descriptions
train.csv - the training set
test.csv - the test set
sample_submission.csv - a sample submission file in the correct format
images - the image files (each image is named with its corresponding id)

Data fields
id - an anonymous id unique to an image
margin_1, margin_2, margin_3, ..., margin_64 - each of the 64 attribute vectors

for the margin feature
shape_1, shape_2, shape_3, ..., shape_64 - each of the 64 attribute vectors for the shape feature
texture_1, texture_2, texture_3, ..., texture_64 - each of the 64 attribute vectors for the texture feature

View of the top-level directory, these are the data you can use to solve this task:

```
/PATH/TO/RAW/DATA
+-leaf-classification/
  |
  +-images/
  | +-226.jpg
  | +-342.jpg
  | +-1534.jpg
  | +-... <additional files>
  |
  +-train.csv
  |
  +-test.csv
```

Modalities Description For the leaf classification task, the input modalities needed to solve the task are as follows:

Input Modalities
- **Tabular Input Map:**
- **YES** because the dataset includes tabular data with pre-extracted features (margin, shape, and texture).
- **Related Files:**
- ‘train.csv‘
- ‘test.csv‘

- **Image Input Map:**
- **YES** as there are images of leaf specimens that can be used for additional feature extraction or as an alternative input.
- **Related Files:**
- ‘images/‘ (directory containing the leaf images)

- **Text Input Map:**
- **NO** because there is no presence of text modality in this task.

Summary

- **Tabular Input Map:** YES -> Related Files: ‘train.csv‘, ‘test.csv‘

- **Image Input Map:** YES -> Related Files: ‘images/
- **Text Input Map:** NO

This structured approach ensures that all relevant input modalities are considered and utilized for the leaf classification task.

These are some of the tables you have asked information about:
View of table ‘/PATH/TO/RAW/DATA/leaf-classification/train.csv‘:

```
/PATH/TO/RAW/DATA/leaf-classification/train.csv
...
<Skipped for prompt display purposes in paper appendix>
...
[2 rows x 194 columns]
```

```
<class ’pandas.core.frame.DataFrame’>
RangeIndex: 990 entries, 0 to 989
Columns: 194 entries, id to texture64
dtypes: float64(192), int64(1), object(1)
memory usage: 1.5+ MB

- column species contains strings with values in [’
   Acer_Opalus’, ’Pterocarya_Stenoptera’, ’
   Quercus_Hartwissiana’, ’Tilia_Tomentosa’, ’
   Quercus_Variabilis’, ’Magnolia_Salicifolia’, ’
   Quercus_Canariensis’, ’Quercus_Rubra’, ’Quercus_Brantii’,
    ’Salix_Fragilis’, ’Zelkova_Serrata’, ’
   Betula_Austrosinensis’, ’Quercus_Pontica’, ’
   Quercus_Afares’, ’Quercus_Coccifera’, ’Fagus_Sylvatica’,
   ’Phildelphus’, ’Acer_Palmatum’, ’Quercus_Pubescens’, ’
   Populus_Adenopoda’, ’Quercus_Trojana’, ’Alnus_Sieboldiana
   ’, ’Quercus_Ilex’, ’Arundinaria_Simonii’, ’
   Acer_Platanoids’, ’Quercus_Phillyraeoides’, ’
   Cornus_Chinensis’, ’Liriodendron_Tulipifera’, ’
   Cytisus_Battandieri’, ’Rhododendron_x_Russellianum’, ’
   Alnus_Rubra’, ’Eucalyptus_Glaucescens’, ’
   Cercis_Siliquastrum’, ’Cotinus_Coggygria’, ’
   Celtis_Koraiensis’, ’Quercus_Crassifolia’, ’
   Quercus_Kewensis’, ’Cornus_Controversa’, ’
   Quercus_Pyrenaica’, ’Callicarpa_Bodinieri’, ’
   Quercus_Alnifolia’, ’Acer_Saccharinum’, ’
   Prunus_X_Shmittii’, ’Prunus_Avium’, ’Quercus_Greggii’, ’
   Quercus_Suber’, ’Quercus_Dolicholepis’, ’Ilex_Cornuta’, ’
   Tilia_Oliveri’, ’Quercus_Semecarpifolia’, ’Quercus_Texana
   ’, ’Ginkgo_Biloba’, ’Liquidambar_Styraciflua’, ’
   Quercus_Phellos’, ’Quercus_Palustris’, ’
   Alnus_Maximowiczii’, ’Quercus_Agrifolia’, ’Acer_Pictum’,
```

```
'Acer_Rufinerve', 'Lithocarpus_Cleistocarpus', '
Viburnum_x_Rhytidophylloides', 'Ilex_Aquifolium', '
Acer_Circinatum', 'Quercus_Coccinea', 'Quercus_Cerris', '
Quercus_Chrysolepis', 'Eucalyptus_Neglecta', '
Tilia_Platyphyllos', 'Alnus_Cordata', 'Populus_Nigra', '
Acer_Capillipes', 'Magnolia_Heptapeta', 'Acer_Mono', '
Cornus_Macrophylla', 'Crataegus_Monogyna', '
Quercus_x_Turneri', 'Quercus_Castaneifolia', '
Lithocarpus_Edulis', 'Populus_Grandidentata', '
Acer_Rubrum', 'Quercus_Imbricaria', 'Eucalyptus_Urnigera'
, 'Quercus_Crassipes', 'Viburnum_Tinus', 'Morus_Nigra', '
Quercus_Vulcanica', 'Alnus_Viridis', 'Betula_Pendula', '
Olea_Europaea', 'Quercus_Ellipsoidalis', '
Quercus_x_Hispanica', 'Quercus_Shumardii', '
Quercus_Rhysophylla', 'Castanea_Sativa', '
Ulmus_Bergmanniana', 'Quercus_Nigra', 'Salix_Intergra', '
Quercus_Infectoria_sub', 'Sorbus_Aria']
```

Your goal is to create a map for the training image inputs.
The first column is the "id" containing the unique ID of each data point, you need to create id so it can be used later to match each data point to their target. For example, the filename name can be use as unique identifier. The subsequent columns are the paths to the training input images for ‘train_img_input_map.csv‘. Do not include target columns in this table.

Now, write a **short** (no more than 3 steps) plan of how you would do that. If there is an error in a previous attempt, explain in **one** sentence why this error happened and how to fix it. Do not write code yet, simply the plan.

446

447

Assistant

Plan to Create the Training Image Input Map

1. **Extract Image IDs and Paths:**
- Read the ‘train.csv‘ file to extract the ‘id‘ column, which contains the unique IDs for each image.
- Construct the image file paths using the ‘id‘ values and the directory path to the images (e.g., ‘images/226.jpg‘).

2. **Create the Image Input Map CSV:**
- Create a new DataFrame with two columns: ‘id‘ and ‘image_path‘.
- Populate the ‘image_path‘ column with the constructed file paths.

3. **Save the Image Input Map CSV:**
- Save the DataFrame to a CSV file named ‘train_img_input_map.csv‘.

448

Error Handling
- If the 'id' values in 'train.csv' do not match the filenames in the 'images/' directory, ensure that the filenames are correctly formatted and match the 'id' values exactly.

449

Prompt

```
<Skipped first part of prompt repeating the previous prompt
   for display purposes in paper appendix>
```

Create 'train_img_input_map.csv'
Your goal is to create a map for the training image inputs. The first column of that map must be named "id" and must contain the unique ID of each data point. If this column exists already in the data, make sure to rename it to "id", otherwise create it. The subsequent columns must be the paths to the training input images. Do not include target columns in the table you are creating, this will be done later in another table.

This is the code you helped write earlier to create the train tabular inputs:

```
"""
This script creates the data map for the tabular training
   inputs, i.e. 'train_tab_input_map.csv'.
This map is a 'pd.DataFrame' object populated with respect
   to the data description and the data present in the raw
   data directory.
The first column of the created map must be the "id" of
   each test data point.
The subsequent columns of 'train_tab_input_map.csv' are the
    tabular training input features, if any.

>>> # Example
>>> train_tab_input_map = pd.DataFrame(columns=["id", "
   tab_input_feature1", "tab_input_feature2", ...])

Create train_tab_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

import os
import pandas as pd

# Define the absolute path to the train.csv file
train_csv_path = "/PATH/TO/RAW/DATA/leaf-classification/
```

450

```
train.csv"

# Read the train.csv file
train_df = pd.read_csv(train_csv_path)

# Select the relevant columns: id and all feature columns, excluding the species column
train_tab_input_map = train_df.drop(columns=['species'])

# Save the new DataFrame to a CSV file
train_tab_input_map.to_csv("./train_tab_input_map.csv", index=False)
print("`train_tab_input_map.csv` created and saved.")


# <|fim_end|>
# save
train_tab_input_map.to_csv("./train_tab_input_map.csv", index=False)
print("`train_tab_input_map.csv` created and saved.")
```

The script reads the train.csv file, selects the id and feature columns (excluding species), and saves the result to ‘train_tab_input_map.csv‘.

Complete the code
Complete the following code template in order to create the data map ‘train_img_input_map.csv‘:

```
"""
This script creates the data map for the image training inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of `train_img_input_map.csv` are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd
```

```python
# <|fim_hole|>

# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("‘train_img_input_map.csv‘ created and saved.")
```

Pay EXTREME attention to the path where you load from. You should always provide absolute path to avoid ‘FileNotFound‘ error. Do not re-set index of the dataframe to create a new id column, instead use the columns present in the dataframe already. Create a new index column only when absolutely necessary. Finally, the step to save the final table is already written in the template - do not rewrite this part. The map is saved in the current directory on purpose.

Tip: IF you are using the image file name as id, make sure they are converted to strings.
Plan:
Plan to Create the Training Image Input Map

1. **Extract Image IDs and Paths:**
- Read the ‘train.csv‘ file to extract the ‘id‘ column, which contains the unique IDs for each image.
- Construct the image file paths using the ‘id‘ values and the directory path to the images (e.g., ‘images/226.jpg‘).

2. **Create the Image Input Map CSV:**
- Create a new DataFrame with two columns: ‘id‘ and ‘image_path‘.
- Populate the ‘image_path‘ column with the constructed file paths.

3. **Save the Image Input Map CSV:**
- Save the DataFrame to a CSV file named ‘train_img_input_map.csv‘.

Error Handling
- If the ‘id‘ values in ‘train.csv‘ do not match the filenames in the ‘images/‘ directory, ensure that the filenames are correctly formatted and match the ‘id‘ values exactly. Your response should be the code to complete the above template, in the format:
“‘python
code
“‘

Assistant

```
    """
This script creates the data map for the image training
    inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect
    to the data description and the data present in the raw
    data directory.
The first column of the created map must be the "id" of
    each test data point.
The subsequent columns of `train_img_input_map.csv` are the
     paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "
    img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

# Define the absolute path to the train.csv file
train_csv_path = "/PATH/TO/RAW/DATA/leaf-classification/
    train.csv"
images_dir = "/PATH/TO/RAW/DATA/leaf-classification/images/
    "

# Read the train.csv file
train_df = pd.read_csv(train_csv_path)

# Ensure the id column is a string
train_df['id'] = train_df['id'].astype(str)

# Construct the image file paths
train_df['image_path'] = train_df['id'].apply(lambda x: os.
    path.join(images_dir, f"{x}.jpg"))

# Create the image input map DataFrame
train_img_input_map = train_df[['id', 'image_path']]

# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("`train_img_input_map.csv` created and saved.")
```

454

H.2 Setup with RAG Example

In the example below we show how an example of a prompt containing an example retrieve looks like. This is during the setup of the competition `ml-olympiad-landscape-image-classification` and in particular when generating the code for the training map for the image modality. The previous attempt had en error in the unit test which is shown in the prompt as the agent is asked to retry. The RAG engine retrieves some similar example from a competition already setup. In that other competition, namely `kaggle-pog-series-s01e03`, the agent had a similar issue and managed to fix it. These steps are retrieved and displayed in the current prompt so that the agent can take inspiration from that similar error being fixed and try to adapt a fix to the current issue it is facing.

Prompt

Data Description
Dataset Description
The data contains masked images with random patches. These random patches can cover anything between 20% to 60% of the image. Your goal is to correctly predict the classes of the images from the "test" directory. All images are 256x256 having 3 channels.

Files
train.csv - the training set
test.csv - the test set
sample.csv - a sample submission file in the correct format
Columns
image - Path of the image relative to directory. Eg. In train.csv, mountain/12345.jpg is relative to the "train" directory, whereas in test.csv, 12345.jpg is relative to the "test" directory.
label - True label of the image (only in train.csv) .
Submission files should have these column headers, but not the column numbers. Use df.to_csv(filepath, index=False) to save without the column numbers.

View of the top-level directory, these are the data you can use to solve this task:

```
/PATH/TO/RAW/DATA
+-ml-olympiad-landscape-image-classification/
  |
  +-train/
  | |
  | +-forest/
  | | +-64935341.jpg
  | | +-89262487.jpg
  | | +-16354101.jpg
```

```
| | +-... <additional files>
| |
| +-buildings/
| | +-15220800.jpg
| | +-30615322.jpg
| | +-48061322.jpg
| | +-... <additional files>
| |
| +-glacier/
| | +-10983747.jpg
| | +-72741892.jpg
| | +-11747834.jpg
| | +-... <additional files>
| |
| +-street/
| | +-16518706.jpg
| | +-88133361.jpg
| | +-17503215.jpg
| | +-... <additional files>
| |
| +-mountain/
| | +-11476151.jpg
| | +-14340600.jpg
| | +-72470093.jpg
| | +-... <additional files>
| +-... <additional folders>
|
+-test/
| +-62449337.jpg
| +-18432742.jpg
| +-84761017.jpg
| +-... <additional files>
|
+-test.csv
|
+-train.csv
```

Modalities Description
Input Modalities for the Task

- **tabular input map**: NO because there is no presence of tabular data (other than the image paths and labels, which are not used as features in the model).
- **image input map**: YES as there are images in this task -> related files:
- ‘/PATH/TO/RAW/DATA/ml-olympiad-landscape-image-classification/train/‘ (training images)
- ‘/PATH/TO/RAW/DATA/ml-olympiad-landscape-image-classification/test/‘ (test images)

- **text input map**: NO because there is no presence of text modality in this task.

Summary

- **Tabular Input**: NO
- **Image Input**: YES (related files: ‘train/‘, ‘test/‘)
- **Text Input**: NO

These are some tips to avoid common mistakes in the code:

- To avoid the error related to non-unique filenames, append the class name to the filenames to ensure they are unique. For example, change ’12345678.jpg’ in the ’forest’ directory to ’forest_12345678.jpg’. This can be done by concatenating the class name with the filename, separated by an underscore or another delimiter.
- To avoid the ’File not found’ error, ensure that all file paths listed in the ’train.csv’ and ’test.csv’ files are correct and that the corresponding files exist in the dataset. You can write a script to verify the existence of all files listed in these CSV files.

Create ‘train_img_input_map.csv‘
Your goal is to create a map for the training image inputs. The first column of that map must be named "id" and must contain the unique ID of each data point. If this column exists already in the data, make sure to rename it to "id", otherwise create it. The subsequent columns must be the paths to the training input images. Do not include target columns in the table you are creating, this will be done later in another table.

This was the resulting code after your last attempt:

```
"""
This script creates the data map for the image training inputs, i.e. ‘train_img_input_map.csv‘.
This map is a ‘pd.DataFrame‘ object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of ‘train_img_input_map.csv‘ are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])
```

```

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

"""
This script creates the data map for the image training inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of `train_img_input_map.csv` are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

# Load the training data
train_df = pd.read_csv("/PATH/TO/RAW/DATA/ml-olympiad-landscape-image-classification/train.csv")

# Extract the image filenames and use them as unique IDs
train_df['id'] = train_df.apply(lambda row: f"{row['label']}_{row['image'].split('/')[-1]}", axis=1)

# Ensure that the filenames are unique
if train_df['id'].nunique() != len(train_df):
    raise ValueError("Filenames are not unique. Please append a unique identifier to duplicate filenames.")

# Create the input map DataFrame
train_img_input_map = train_df[['id', 'image']].rename(columns={'image': 'img_input1'})

# <|fim_end|>
# save
```

```
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("`train_img_input_map.csv` created and saved.")


# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("`train_img_input_map.csv` created and saved.")
```

This was the output:

```
`train_img_input_map.csv` created and saved.
`train_img_input_map.csv` created and saved.
```

The error in the unit test was:

```
multiprocessing.pool.RemoteTraceback:
"""
Traceback (most recent call last):
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-
     gnu/lib/python3.10/multiprocessing/pool.py", line 125,
     in worker
    result = (True, func(*args, **kwds))
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-
     gnu/lib/python3.10/multiprocessing/pool.py", line 48,
     in mapstar
    return list(map(*args))
  File "/PATH/TO/WORKSPACE/ml-olympiad-landscape-image-
     classification/seed_0/unit_test_map_img_input_train.py"
     , line 45, in check_file_exists
    raise FileNotFoundError(f"File not found: {file_path}")
FileNotFoundError: File not found: glacier/59804579.jpg
"""

The above exception was the direct cause of the following
   exception:

Traceback (most recent call last):
  File "/PATH/TO/WORKSPACE/ml-olympiad-landscape-image-
     classification/seed_0/unit_test_map_img_input_train.py"
     , line 65, in <module>
    check_files(files_to_check)
  File "/PATH/TO/WORKSPACE/ml-olympiad-landscape-image-
     classification/seed_0/unit_test_map_img_input_train.py"
     , line 58, in check_files
```

```
    results = pool.map(check_file_exists, file_list)
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-gnu/lib/python3.10/multiprocessing/pool.py", line 367, in map
    return self._map_async(func, iterable, mapstar, chunksize).get()
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-gnu/lib/python3.10/multiprocessing/pool.py", line 774, in get
    raise self._value
FileNotFoundError: File not found: glacier/59804579.jpg
```

You should fix this error next.

An example of a similar error being fixed:
Failing code:

```
"""
This script creates the data map for the image training inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of `train_img_input_map.csv` are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

"""
This script creates the data map for the image training inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of `train_img_input_map.csv` are the
```

```
    paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

# Define the absolute path to the train.csv file
train_csv_path = "/PATH/TO/RAW/DATA/kaggle-pog-series-s01e03/train.csv"

# Load the train.csv file into a DataFrame
train_df = pd.read_csv(train_csv_path)

# Generate a unique ID by concatenating seed_id and view
train_df['id'] = train_df['seed_id'].astype(str) + '_' + train_df['view']

# Check for duplicates in the generated ID
if train_df['id'].duplicated().any():
    raise ValueError("There are duplicate IDs in the generated unique ID column.")

# Construct the full path to each image
base_image_dir = "/PATH/TO/RAW/DATA/kaggle-pog-series-s01e03/train/"
train_df['img_input1'] = base_image_dir + train_df['image']

# Select the relevant columns: 'id' and 'img_input1'
train_img_input_map = train_df[['id', 'img_input1']]

# <|fim_begin|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv", index=False)
print("`train_img_input_map.csv` created and saved.")


# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv", index=False)
```

```
print("`train_img_input_map.csv` created and saved.")
```

Failing code error:

```
multiprocessing.pool.RemoteTraceback:
"""
Traceback (most recent call last):
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-
     gnu/lib/python3.10/multiprocessing/pool.py", line 125,
     in worker
    result = (True, func(*args, **kwds))
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-
     gnu/lib/python3.10/multiprocessing/pool.py", line 48,
     in mapstar
    return list(map(*args))
  File "/PATH/TO/WORKSPACE/kaggle-pog-series-s01e03/seed_0/
     unit_test_map_img_input_train.py", line 45, in
     check_file_exists
    raise FileNotFoundError(f"File not found: {file_path}")
FileNotFoundError: File not found: /PATH/TO/RAW/DATA/kaggle-
   pog-series-s01e03/train/train/03511.png
"""

The above exception was the direct cause of the following
   exception:

Traceback (most recent call last):
  File "/PATH/TO/WORKSPACE/kaggle-pog-series-s01e03/seed_0/
     unit_test_map_img_input_train.py", line 65, in <module>
    check_files(files_to_check)
  File "/PATH/TO/WORKSPACE/kaggle-pog-series-s01e03/seed_0/
     unit_test_map_img_input_train.py", line 58, in
     check_files
    results = pool.map(check_file_exists, file_list)
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-
     gnu/lib/python3.10/multiprocessing/pool.py", line 367,
     in map
    return self._map_async(func, iterable, mapstar,
       chunksize).get()
  File "/home/jdoran/uv_python/cpython-3.10.15-linux-x86_64-
     gnu/lib/python3.10/multiprocessing/pool.py", line 774,
     in get
    raise self._value
FileNotFoundError: File not found: /PATH/TO/RAW/DATA/kaggle-
   pog-series-s01e03/train/train/03511.png
```

Fixed code:

473

```python
"""
This script creates the data map for the image training inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of `train_img_input_map.csv` are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

"""
This script creates the data map for the image training inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of `train_img_input_map.csv` are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

# Define the absolute path to the train.csv file
train_csv_path = "/PATH/TO/RAW/DATA/kaggle-pog-series-s01e03/train.csv"

```

```python
# Load the train.csv file into a DataFrame
train_df = pd.read_csv(train_csv_path)

# Generate a unique ID by concatenating seed_id and view
train_df['id'] = train_df['seed_id'].astype(str) + '_' +
    train_df['view']

# Check for duplicates in the generated ID
if train_df['id'].duplicated().any():
     raise ValueError("There are duplicate IDs in the
         generated unique ID column.")

# Construct the full path to each image
base_image_dir = "/PATH/TO/RAW/DATA/kaggle-pog-series-
    s01e03/train/"
train_df['img_input1'] = base_image_dir + train_df['image'
    ].str.split('/').str[-1]

# Select the relevant columns: 'id' and 'img_input1'
train_img_input_map = train_df[['id', 'img_input1']]

# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("`train_img_input_map.csv` created and saved.")


# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("`train_img_input_map.csv` created and saved.")
```

Complete the code
Complete the following code template in order to create the data map 'train_img_input_map.csv':

```python
"""
This script creates the data map for the image training
    inputs, i.e. `train_img_input_map.csv`.
This map is a `pd.DataFrame` object populated with respect
    to the data description and the data present in the raw
    data directory.
The first column of the created map must be the "id" of
    each test data point.
The subsequent columns of `train_img_input_map.csv` are the
```

```
    paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "
   img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd
# <|fim_hole|>

# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
   index=False)
print("`train_img_input_map.csv` created and saved.")
```

Pay EXTREME attention to the path where you load from. You should always provide absolute path to avoid ‘FileNotFound‘ error. Do not re-set index of the dataframe to create a new id column, instead use the columns present in the dataframe already. Create a new index column only when absolutely necessary. Finally, the step to save the final table is already written in the template - do not rewrite this part. The map is saved in the current directory on purpose.

Tip: IF you are using the image file name as id, make sure they are converted to strings.

Plan:
1. **Load the Training Data**: Read the ‘train.csv‘ file to get the image paths and labels.
2. **Create Unique IDs**: Generate a unique ID for each image by concatenating the label and the image filename.
3. **Construct the Input Map**: Create a DataFrame with the unique IDs and the corresponding image paths, and save it as ‘train_img_input_map.csv‘.

Error Explanation:
The error occurred because the file path in the ‘train.csv‘ file was incorrect or the file did not exist in the specified directory. To fix it, ensure that the file paths in ‘train.csv‘ are correct and that the files exist in the ‘train/‘ directory.
Your response should be the code to complete the above template, in the format:
“‘python
code

‘‘‘

Assistant

```
"""
This script creates the data map for the image training inputs, i.e. ‘train_img_input_map.csv‘.
This map is a ‘pd.DataFrame‘ object populated with respect to the data description and the data present in the raw data directory.
The first column of the created map must be the "id" of each test data point.
The subsequent columns of ‘train_img_input_map.csv‘ are the paths to the training input images, if any.

>>> # Example
>>> train_img_input_map = pd.DataFrame(columns=["id", "img_input1", "img_input2", ...])

Create train_img_input_map
"""
# <|fim_begin|>
import os
import pandas as pd

# Define the absolute path to the train.csv file
train_csv_path = "/PATH/TO/RAW/DATA/ml-olympiad-landscape-image-classification/train.csv"

# Load the train.csv file into a DataFrame
train_df = pd.read_csv(train_csv_path)

# Generate a unique ID by concatenating the label and the image filename
train_df['id'] = train_df.apply(lambda row: f"{row['label']}_{row['image'].split('/')[-1]}", axis=1)

# Check for duplicates in the generated ID
if train_df['id'].duplicated().any():
    raise ValueError("There are duplicate IDs in the generated unique ID column.")

# Construct the full path to each image
base_image_dir = "/PATH/TO/RAW/DATA/ml-olympiad-landscape-image-classification/train/"
train_df['img_input1'] = base_image_dir + train_df['image']
```

```

# Select the relevant columns: 'id' and 'img_input1'
train_img_input_map = train_df[['id', 'img_input1']]

# <|fim_end|>
# save
train_img_input_map.to_csv("./train_img_input_map.csv",
    index=False)
print("`train_img_input_map.csv` created and saved.")
```

H.3 Solution Generation

The example below is taken from the solution generation pipeline on the comeption `dogs-vs-cats-redux-kernels-edition`. In this particular example, the agent has already done one submission which appears in some prompts. The agent is at the start of a new submission, generating the plans and codes for preprocessing the images and the image embedding model.

Prompt

Task description
Summary of the Task Description

Main Goal:
The goal of this competition is to classify images as either dogs or cats. The competition revisits the original "Dogs vs. Cats" challenge from 2013, but with modern machine learning techniques and tools, such as Kernels (Kaggle's online coding environment) and TensorFlow.

Task Type:
This is a binary classification task where the model must predict the probability that a given image is a dog. The evaluation metric is log loss, which measures the performance of the model by penalizing in correct predictions more severely as the predicted probability diverges from the actual label.

Key Points:
- **Dataset:** The dataset consists of images of dogs and cats.
- **Evaluation Metric:** Log loss.
- **Submission Format:** A CSV file with the format 'id,label', where 'id' is the image ID and 'label' is the predicted probability that the image is a dog.

Data description
View of the top-level directory /PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition:
after having downloaded and extracted the raw data from the source, these are

the data you can use to solve this task:

```
/PATH/TO/RAW/DATA
+-dogs-vs-cats-redux-kernels-edition/
  |
  +-test/
  | +-8715.jpg
  | +-12271.jpg
  | +-156.jpg
  | +-... <additional files>
  |
  +-train/
    +-cat.8852.jpg
    +-dog.9444.jpg
    +-cat.12231.jpg
    +-... <additional files>
```

Summary of Data Organization

The dataset is organized as follows:

Directory Structure

```
/PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition/
  |
  +-test/
  |  +-8715.jpg
  |  +-12271.jpg
  |  +-156.jpg
  |  +-... (additional files)
  |
  +-train/
  |  +-cat.8852.jpg
  |  +-dog.9444.jpg
  |  +-cat.12231.jpg
  |  +-... (additional files)
  |
  +-sample_submission.csv
```

File Descriptions
1.**/PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition/train/**
- Contains 25,000 images of dogs and cats.
- Each image filename includes the label:
- ‘cat.<number>.jpg‘ for cat images.
- ‘dog.<number>.jpg‘ for dog images.

2.**/PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition/test/**
- Contains 12,500 images.
- Each image is named according to a numeric ID (e.g., ‘8715.jpg‘).

3.**/PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition/sample_submission.csv**
- A sample submission file with the following columns:
- ‘id‘: The numeric ID of the test image.
- ‘label‘: The predicted probability that the image is a dog (1 = dog, 0 = cat).

Feature and Target Names
- **Features:**
- ‘id‘: The numeric ID of the test image (used for the test set).

- **Target:**
- ‘label‘: The predicted probability that the image is a dog (1 = dog, 0 = cat).

Note
- The data is saved under ‘/PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition‘.
- The directory and file names must remain unchanged.

Available solutions
This is the list of the solutions that have already been generated:

* 2024-10-16-07-22-08: The provided code implements a comprehensive pipeline for the Dogs vs. Cats classification task. It includes a set of torchvision transforms to preprocess images, a PyTorch class ‘ImageEmbedder‘ that uses a pre-trained ResNet50 model to generate latent representations of images, and a ‘TabularHead‘ model to decode these embeddings into classification logits. The pipeline also defines appropriate loss functions for both regression and classification tasks, ensuring a robust and flexible approach to the problem.

Current solution progress
You have started creating a new solution. Its status is as follows:
- [PENDING] Embedder for images
- [PENDING] Transform functions for images
- [TODO] Embedder for transformed images
- [TODO] Logits and regression targets decoder

What to do next
You have to generate a code to transform the images using a composition of torchvision transforms. You need to write a pytorch torchvision.transforms.Compose sequence that transforms the images appropriately

with respect to the task. You need to make sure the images are also transformed to torch.Tensor at some point as the images will be fed to a vision neural network. You can use any image transforms (Resize, RandomCrop, ToTensor, ...) or even write custom transforms with Lambda() ...

Output a plan presented with maximum 4 bullet points describing the different steps to construct a pytorch torchvision transforms composition. The plan should be tailored to the current task as it will be used to guide the implementation of the image transforms.

Assistant

Plan for Constructing PyTorch torchvision Transforms Composition

1. **Resize Images**: Resize all images to a standard size (e.g., 224x224 pixels) to ensure they can be fed into a pre-trained model like ResNet50, which expects input of this size.

2. **Apply Random Augmentations**: Use random augmentations such as ‘RandomHorizontalFlip‘, ‘RandomRotation‘, and ‘ColorJitter‘ to increase the diversity of the training data and improve the model’s generalization capabilities.

3. **Convert to Tensor**: Use ‘ToTensor‘ to convert the images to PyTorch tensors, which is a required format for feeding the images into a neural network.

4. **Normalize Images**: Apply normalization using the mean and standard deviation of the ImageNet dataset, as pre-trained models like ResNet50 are typically trained on ImageNet. This ensures the input data is in the expected range and format for the pre-trained model.

Prompt

```
<Skipped first part of prompt repeating the previous prompt
   for display purposes in paper appendix>
```

Current solution progress
You have started creating a new solution. Its status is as follows:
- [PENDING] Embedder for images
- [PENDING] Transform functions for images
- [TODO] Embedder for transformed images
- [TODO] Logits and regression targets decoder

What to do next
You have to generate a code to transform the images using a composi-

tion of torchvision transforms. You need to write a pytorch torchvision.transforms.Compose sequence that transforms the images appropriately with respect to the task. You need to make sure the images are also transformed to torch.Tensor at some point as the images will be fed to a vision neural network. You can use any image transforms (Resize, RandomCrop, ToTensor, ...) or even write custom transforms with Lambda() ...

For example a good default transform is:

```python
import numpy as np
import pandas as pd
import torch
from PIL import Image
from torch import Tensor
from torch.utils.data import Dataset
from torchvision import transforms as T

DefaultImageInputTransform: T.Compose = T.Compose(
    [
        T.Resize((512, 512)),  # Resize to a fixed size (should make sense with respect to the task, not always needed)
        T.Lambda(lambda x: x if isinstance(x, torch.Tensor) else T.ToTensor()(x)),  # Convert image to tensor if needed
        T.Lambda(
            lambda x: x.repeat([3 if (i - x.ndim == -3 and d == 1) else 1 for i, d in enumerate(x.shape)])
        ),  # grayscale to rgb
        T.Lambda(lambda x: x[:3] if x.shape[0] > 3 else x),  # take only first 3 channels
    ]
)
```

Note that the last two transforms are using a Lambda function. The penultimate one is an important one to include as its goal is to duplicate the number of channels to 3 if there happen to be grayscale images in the data mixed with colored images. The ultimate one is taking on ly the first 3 channels in case the images in the dataset have an inconsistent number of channels. These two last transforms hence force all images to have the same number of channels. If you are not sure all images in the data will have the same number of channels, it is advised to add a similar transform. If you find it necessary to normalize the images, do it after taking care fo the number of channels and make sure the vectors you use match the image dimensions.

Use your plans to help you implement the image transform:
Plan for Constructing PyTorch torchvision Transforms Composition

1. **Resize Images**: Resize all images to a standard size (e.g., 224x224 pixels) to ensure they can be fed into a pre-trained model like ResNet50, which expects input of this size.

2. **Apply Random Augmentations**: Use random augmentations such as ‘RandomHorizontalFlip‘, ‘RandomRotation‘, and ‘ColorJitter‘ to increase the diversity of the training data and improve the model’s generalization capabilities.

3. **Convert to Tensor**: Use ‘ToTensor‘ to convert the images to PyTorch tensors, which is a required format for feeding the images into a neural network.

4. **Normalize Images**: Apply normalization using the mean and standard deviation of the ImageNet dataset, as pre-trained models like ResNet50 are typically trained on ImageNet. This ensures the input data is in the expected range and format for the pre-trained model.

Write the code to create an image transforms composition.
1. You must fill all the required parameters of the transforms you are using in a sensible way with respect to the image shapes of the current task.
2. You must not forget to import the packages when needed.
3. The transforms should be transforming the images to tensor at the appropriate step.
4. Try to use different transforms than this last attempt: The provided code implements a comprehensive pipeline for the Dogs vs. Cats classification task. It includes a set of torchvision transforms to preprocess images, a PyTorch class ‘ImageEmbedder‘ that uses a pre-trained ResNet50 model to generate latent repr esentations of images, and a ‘TabularHead‘ model to decode these embeddings into classification logits. The pipeline also defines appropriate loss functions f or both regression and classification tasks, ensuring a robust and flexible approach to the problem.. Do not repeat all the transforms from the attempt.

Your task is to fill the blank and implement the image transform in this code:

```
# <|fim_begin|>
import os
import pandas as pd
import torch
from torch import nn
from PIL import Image
```

```python
import torchvision.transforms as T
from torch.utils.data import DataLoader
from tqdm import tqdm

from agent.tools.data_map.map_dataset import MapDataset, map_dataset_collate_function

# --- Create CustomImageInputTransform, a composition of torchvision.transforms using torchvision.transforms.Compose
# <|fim_hole|>
# --- [End]

# <|fim_end|>
# Test that the transform works
root_path = "/PATH/TO/RAW/DATA/dogs-vs-cats-redux-kernels-edition/qwen2.5-72b-vp.3.5_seed_2"
tab_target_map_path=os.path.join(root_path, "train_tab_target_map.csv")
img_target_map_path=os.path.join(root_path, "train_img_target_map.csv")
train_dataset = MapDataset(
    img_input_map_path=os.path.join(root_path, "train_img_input_map.csv"),
    tab_target_map_path=tab_target_map_path if os.path.exists(tab_target_map_path) else None,
    img_target_map_path=img_target_map_path if os.path.exists(img_target_map_path) else None,
    img_input_transform=CustomImageInputTransform,
)
train_dataloader = DataLoader(
    train_dataset, batch_size=256, shuffle=True, collate_fn=map_dataset_collate_function, num_workers=20
)
shapes = set()
for batch in tqdm(train_dataloader):
    (indices, (_, img_inputs_batch, _), (tab_targets_batch, _, _)) = batch
    if img_inputs_batch.shape not in shapes:
        shapes.add(img_inputs_batch.shape[1:])  # ignore batch size as last batch can have fewer elements
    if len(shapes) > 1:
        raise RuntimeError(
            f"Multiple shapes detected! After applying CusotmImageTransform, some images have shapes "
            f"{' and '.join([str(s) for s in shapes])} "
```

```
                f"but should have the same shape for all images after the transform. "
                f"(Note that the first dimension is the batch size {train_dataloader.batch_size}).\n"
                f"Hint: if needed add a lambda transform that ensures the number of channels is, e.g. 3 in this example:\n"
                f"`T.Lambda(lambda x: x.repeat([3 if (i - x.ndim == -3 and d == 1) else 1 for i, d in enumerate(x.shape)])),`"
                f"\nand\n`T.Lambda(lambda x: x[:3] if x.shape[0] > 3 else x)`"
            )

print(f"Transformed image tensor shape: {img_inputs_batch.shape}")
```

Assistant

```
CustomImageInputTransform = T.Compose(
    [
        T.Resize((256, 256)),  # Resize to a standard size
        T.RandomCrop(224),  # Randomly crop to the input size expected by the model
        T.RandomHorizontalFlip(p=0.5),  # Random horizontal flip with 50% probability
        T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # Random color jitter
        T.ToTensor(),  # Convert image to tensor
        T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),  # Normalize using ImageNet mean and std
        T.Lambda(
            lambda x: x.repeat([3 if (i - x.ndim == -3 and d == 1) else 1 for i, d in enumerate(x.shape)])
        ),  # grayscale to rgb
        T.Lambda(lambda x: x[:3] if x.shape[0] > 3 else x),  # take only first 3 channels
    ]
)
```

Prompt

```
<Skipped first part of prompt repeating the previous prompt
   for display purposes in paper appendix>
```

Current solution progress
You have started creating a new solution. Its status is as follows:
- [TODO] Embedder for text fields
- [TODO] Logits and regression targets decoder

What to do next
You have to generate a code to embed the images using a pre-trained torchvision model. You need to write a pytorch torchvision model, where forward() takes the tensor transformed images as input, and that outputs a latent representation as a ten sor. You can use any pre-trained model from this list: resnet18, resnet50, resnet101, AlexNet, EfficientNetV2, GoogLeNet, MaxVit, MNASNet, RegNet

Output a plan presented with maximum 4 bullet points describing the different steps to construct a pytorch torchvision model to embed images. The plan should be tailored to the current task as it will be used to guide the implementation of the embedder.

Assistant

Plan to Construct a PyTorch torchvision Model to Embed Images

1. **Select and Load Pre-trained Model:**
- Choose a pre-trained model from the provided list (e.g., 'resnet50').
- Load the pre-trained model and remove the final fully connected layer to use it for feature extraction.

2. **Define Image Preprocessing:**
- Define a set of torchvision transforms to preprocess the images (e.g., resizing, normalization).
- Ensure the transforms are consistent with the pre-processing used during the training of the pre-trained model.

3. **Create Image Embedder Class:**
- Define a PyTorch class 'ImageEmbedder' that takes the pre-processed image tensor as input.
- Use the pre-trained model to extract the latent representation (features) from the images.

4. **Implement Forward Method:**
- Implement the 'forward' method in the 'ImageEmbedder' class to pass the

input image tensor through the pre-trained model.
- Return the latent representation tensor as the output of the 'forward' method.

This plan will guide the implementation of the embedder, ensuring it is tailored to the current task of classifying dogs and cats.

Prompt

```
<Skipped first part of prompt repeating the previous prompt
   for display purposes in paper appendix>
```

Current solution progress
You have started creating a new solution. Its status is as follows:
- [TODO] Embedder for text fields
- [TODO] Logits and regression targets decoder

What to do next
You have to generate a code to embed the images using a pre-trained torchvision model. You need to write a pytorch torchvision model, where forward() takes the tensor transformed images as input, and that outputs a latent representation as a ten sor. You can use any pre-trained models like resnet18, resnet50, resnet101, AlexNet, EfficientNetV2, GoogLeNet, MaxVit, MNASNet, RegNet, vit etc...

Ensure to unfreeze last few layers of the model based on the model you choose.

An example of this model code is:

```python
import torch
import torchvision.models as models
from torch import nn

class ImageEmbedder(nn.Module):
    def __init__(self, pretrained=True):
        super().__init__()

        resnet50 = models.resnet50(pretrained=pretrained)
        self.features = nn.ModuleList(resnet50.children())
           [:-1]
        self.features = nn.Sequential(*self.features)
        # optionally freeze pre-trained model weights
        for param in self.features.parameters():
            param.requires_grad = False

    def forward(self, x):
        output = self.features(x)
```

```
        output = output.view(x.size(0), -1)
        return output

    def unfreeze(self, n_last_layers=2):
        """Unfreeze some layers for fine-tuning."""
        layers = list(self.children())[:-1]

        for layer in layers[-n_last_layers:]:
            for param in layer.parameters():
                param.requires_grad = True
```

Use your plans to help you implement the pytorch embedder model:
Plan to Construct a PyTorch torchvision Model to Embed Images

1. **Select and Load Pre-trained Model:**
- Choose a pre-trained model from the provided list (e.g., ‘resnet50‘).
- Load the pre-trained model and remove the final fully connected layer to use it for feature extraction.

2. **Define Image Preprocessing:**
- Define a set of torchvision transforms to preprocess the images (e.g., resizing, normalization).
- Ensure the transforms are consistent with the pre-processing used during the training of the pre-trained model.

3. **Create Image Embedder Class:**
- Define a PyTorch class ‘ImageEmbedder‘ that takes the pre-processed image tensor as input.
- Use the pre-trained model to extract the latent representation (features) from the images.

4. **Implement Forward Method:**
- Implement the ‘forward‘ method in the ‘ImageEmbedder‘ class to pass the input image tensor through the pre-trained model.
- Return the latent representation tensor as the output of the ‘forward‘ method.

This plan will guide the implementation of the embedder, ensuring it is tailored to the current task of classifying dogs and cats.

Write the code to create a model taking tensors of preprocessed features as input, and outputting an embedding tensor.
1. You can assume that the input dimension is already computed in the variable ‘input_dim‘.
2. You must not forget to import the packages when needed.
504 3. The model should be able to take as input a torch tensor of size ‘(batch,

input_dim)‘ and output a tensor of shape ‘(batch, latent_dim)‘.
4. Try to use newer models than this last attempt: The provided code implements a comprehensive pipeline for the Dogs vs. Cats classification task. It include s a set of torchvision transforms to preprocess images, a PyTorch class ‘ImageEmbedder‘ that uses a pre-trained ResNet50 model to generate latent representations of images, and a ‘TabularHead‘ model to decode these embeddings into classification logits. The pipeline also defines appropriate loss functions for both regression and classification tasks, ensuring a robust and flexible approach to the problem.. Do not repeat these pre-trained models from the attempt.

Your task is to fill the blank and implement the image embedder in this code:

```
# <|fim_begin|>
import os.path

import torch
import pandas as pd
import torchvision.models as models
import torchvision.transforms as T
from torch import nn
from PIL import Image

from submissions.new_submission.img_transform import CustomImageInputTransform

# Load image features
root_path = "/PATH/TO/SETUP"
train_data_path = os.path.join(root_path, "train_img_input_map.csv")
x = pd.read_csv(train_data_path, index_col="id")

# --- Design torch model using a pre-trained torchvision model
# Create class ImageEmbedder(nn.Module) and implementing the methods:
#   - __init__() defines the model architecture, using a pretrained model such as resnet50
#   - forward:(x: torch.Tensor) -> torch.Tensor, embeds a batch of inputs and loads the image embeddings.
# <|fim_hole|>
# --- [End]

# Test that the model works
path = x.iloc[0, 0]
image = Image.open(path)
```

```python
tensor_image = CustomImageInputTransform(image)
model = ImageEmbedder()
X_embed = model(tensor_image.unsqueeze(0))


# <|fim_end|>
# @NO_MEMORY_START@

def test_unfreeze_functionality() -> None:
    model = ImageEmbedder()
    assert hasattr(model, 'unfreeze'), "The model does not have an 'unfreeze' method."

    try:
        model.unfreeze()
    except Exception as e:
        assert False, f"Unfreeze method raised an exception: {e}"

    print("Unfreeze method exists and runs successfully.")


test_unfreeze_functionality()

print(f"Embedding size: {X_embed.shape}")
# @NO_MEMORY_END@
```

Only output the code that should be filled in the blank with no extra text. Don't forget to import any additional packages that you will use in the code.
Your response should be in the format:
“‘python
<code>
“‘

Assistant

```python
class ImageEmbedder(nn.Module):
    def __init__(self, pretrained=True, latent_dim=2048):
        super().__init__()

        # Choose a newer model like EfficientNetV2
        efficientnet = models.efficientnet_v2_m(pretrained=pretrained)
        self.features = nn.ModuleList(efficientnet.features.children())
        self.features = nn.Sequential(*self.features)
        # Optionally freeze pre-trained model weights
```

```python
        for param in self.features.parameters():
            param.requires_grad = False

    def forward(self, x):
        output = self.features(x)
        output = output.mean(dim=[2, 3])

    def unfreeze(self, n_last_layers=2):
        """Unfreeze some layers for fine-tuning."""
        layers = list(self.features.children())

        for layer in layers[-n_last_layers:]:
            for param in layer.parameters():
                param.requires_grad = True
```

H.4 Scaffold with CoT

The example below is taken from a natural language processing competition (`sentiment-analysis-on-movie-reviews`). The solution below was generated by Agent K as one of the draft nodes. Agent K is provided with a CoT from the scaffolding stage to guide its initial solution generations.

Prompt

Introduction
You are a Kaggle grandmaster attending a competition. In order to win this competition, you need to come up with an excellent and creative plan for a solution and then implement this solution in Python. We will now provide a description of the task.

BENCHMARK INSTRUCTIONS
First, please read the instructions below
- **Competition-specific instructions**: Competition specific descriptions will be provided.
- **Dataset**: The dataset files for the competition are available in '*/PATH/TO/RAW/DATA/sentiment-analysis-on-movie-reviews'.Test dataset does not contain any targets.
- **Submission**: You can find sample submission format inside '*/PATH/TO/RAW/DATA/sentiment-analysis-on-movie-reviews' as specified 'Data Overview.Always use same 'id' available inside sample submission file. You are not allowed to change 'id'.Based on sample submission file you have to generate submission.csv for the test data. - **Cheating**: You are not allowed to view others' solutions to this competition. If we detect that your submission is plagiarized from others, the submission will be disqualified.

Wherever there is a conflict between the points above and the competition-specific instructions, competition points always takes priority.

Your submission will be graded based on the metric in the competition-specific instructions. Your goal is to achieve the best score possible on the competition task.

ADDITIONAL NOTES
Compute: You have access to 1 Tesla V100-PCIE-32GB with the appropriate drivers installed.
Total Runtime: You have a maximum of 172800 seconds to attempt this task. You may iteratively develop your solution within this time frame, but bear in mind that program runtime also counts towards this limit.
Total Steps: You have a maximum of 5000 steps to submit your solution.
You will be informed of the remaining time and steps at each step .**Use as much of the available time as possible to refine and optimize your submis-

sion**: Your goal is to achieve the best possible score, not just to submit your first working solution.

Response format
Your response should be a brief outline/sketch of your proposed solution in natural language (3-5 sentences), followed by a single markdown code block (wrapped in “‘) which implements this solution and prints out the evaluation metric. There should be no additional headings or text in your response. Just natural language text followed by a newline and then the markdown code block.

Solution sketch guideline
- Carefully examine the past solutions. These solutions may not be identical to the task at hand, but they provide valuable insights on strategies and techniques.
- Look for common strategies that led to success, such as specific modeling techniques, evaluation metrics, or feature engineering methods.
- Incorporate the lessons learned from these solutions into your proposed solution, making sure to adapt them to the current task.
- The solutions should inform how to approach the current task, focusing on critical decisions made in those examples and any relevant best practices.
- Do not copy the solutions verbatim, but leverage them as a foundation for crafting a novel, efficient, and creative approach.
- You are encouraged to explore new angles while respecting the context and recommendations from the previous successful strategies.
- Take the Memory section into consideration when proposing the design
- don’t propose the same modelling solution but keep the evaluation the same.
- The solution sketch should be 3-5 sentences.
- Propose an evaluation metric that is reasonable for this task.
- Don’t suggest to do EDA.

Installed Packages
Your solution can use any relevant machine learning packages such as: ‘torchvision‘, ‘xgboost‘, ‘timm‘, ‘pandas‘, ‘torch-geometric‘, ‘scikit-learn‘, ‘statsmodels‘, ‘numpy‘, ‘torch‘, ‘bayesian-optimization‘, ‘lightGBM‘. Feel free to use any other packages too (all packages are already installed!). For neural networks we suggest using PyTorch rather than TensorFlow.

Implementation guideline
- <TOTAL_TIME_REMAINING: 41.0hrs 23.0mins 32secs>
- <TOTAL_STEPS_REMAINING: 4987>
- The code should **implement the proposed solution**, **print the value of the evaluation metric computed on a hold-out validation set**,
- **AND MOST IMPORTANTLY SAVE PREDICTIONS ON THE PROVIDED

UNLABELED TEST DATA IN A ‘submission.csv‘ FILE IN THE ./submission/ DIRECTORY.**
- The code should be a single-file python program that is self-contained and can be executed as-is.
- No parts of the code should be skipped, don’t terminate the before finishing the script.
- Your response should only contain a single code block.
- Be aware of the running time of the code, it should complete within 18 hours.
- Do not copy data to workspace, read data from the source specified.
- You can use the "./working" directory to store any temporary files that your code needs to create.
- The evaluation should be based on 5-fold cross-validation but only if that’s an appropriate evaluation for the task at hand.

Data Overview

```
/PATH/TO/RAW/DATA
  |
  +-sampleSubmission.csv (66293 lines)
  |
  +-.. and 2 other files
```

Task description

Dataset Description
The dataset is comprised of tab-separated files with phrases from the Rotten Tomatoes dataset. The train/test split has been preserved for the purposes of benchmarking, but the sentences have been shuffled from their original order. Each Sentence has been parsed into many phrases by the Stanford parser. Each phrase has a PhraseId. Each sentence has a SentenceId. Phrases that are repeated (such as short/common words) are only included once in the data.
train.tsv contains the phrases and their associated sentiment labels. We have additionally provided a SentenceId so that you can track which phrases belong to a single sentence.
test.tsv contains just phrases. You must assign a sentiment label to each phrase.
The sentiment labels are:
0 - negative
1 - somewhat negative
2 - neutral
3 - somewhat positive
4 - positive

Evaluation
Submissions are evaluated on classification accuracy (the percent of labels

that are predicted correctly) for every parsed phrase. The sentiment labels are:
0 - negative
1 - somewhat negative
2 - neutral
3 - somewhat positive
4 - positive
Submission Format
For each phrase in the test set, predict a label for the sentiment. Your submission should have a header and look like the following:
PhraseId,Sentiment
156061,2
156062,2
156063,2
...

Summary of Past Submissions

Submission 0 summary and metric value on a validation set: The project involves a comprehensive pipeline for sentiment analysis. A TextEmbedder class utilizes a pre-trained DistilBERT model to convert text inputs into embeddings. To address class imbalances, appropriate techniques are applied. A PyTorch model with a TabularHead class is implemented to transform the text embeddings into sentiment logits, and both regression and classification loss functions are defined to train the model effectively.
"submission.csv": 65.57734204793029, "submission_alt.csv": 65.57734204793029

Submission 1 summary and metric value on a validation set:
The code implements a PyTorch model that uses a pre-trained RoBERTa model to embed text fields, tokenizes the input text, and returns the pooler output as the text embedding. It handles class imbalances and includes a TabularHead class to transform the RoBERTa embeddings into sentiment logits using fully connected layers with dropout and batch normalization. The code also implements regression and classification loss functions using Mean Squared Error (MSE) and Cross-Entropy loss, respectively.
"submission.csv": 66.00794566192491, "submission_alt.csv": 66.00794566192491

Submission 2 summary and metric value on a validation set:
The code implements a PyTorch model that uses a pre-trained BERT model to embed text fields. It tokenizes the input text and returns the pooler output as the text embedding. To handle class imbalances, appropriate techniques are applied. The model also includes a TabularHead, which is a series of fully connected layers that transform the BERT embeddings into sentiment logits.

For the classification task, Cross-Entropy loss is used, and for the regression task, Mean Squared Error (MSE) loss is employed.
"submission.csv": 66.42573369216967, "submission_alt.csv": 66.42573369216967

Assistant

To tackle this sentiment analysis task, I will use a pre-trained BERT model to extract features from the text and then train a simple classifier on top of these features. The model will be trained using a cross-entropy loss function, and class imbalance will be handled using weighted loss. The performance will be evaluated using classification accuracy on a hold-out validation set..

```python
import pandas as pd
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertModel
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch.nn as nn
import torch.optim as optim


# Define the dataset
class SentimentDataset(Dataset):
    def __init__(self, df, tokenizer, max_len=128):
        self.tokenizer = tokenizer
        self.max_len = max_len
        self.df = df

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        text = self.df.iloc[idx]["Phrase"]
        if pd.isna(text):
            text = ""  # Handle NaN values
        label = (
            self.df.iloc[idx]["Sentiment"] if "Sentiment"
               in self.df.columns else None
        )

        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
```

```python
                max_length=self.max_len,
                padding="max_length",
                truncation=True,
                return_attention_mask=True,
                return_tensors="pt",
            )

            input_ids = encoding["input_ids"].flatten()
            attention_mask = encoding["attention_mask"].flatten()

            if label is not None:
                return {
                    "input_ids": input_ids,
                    "attention_mask": attention_mask,
                    "label": torch.tensor(label, dtype=torch.long),
                }
            else:
                return {"input_ids": input_ids, "attention_mask": attention_mask}


# Define the model
class SentimentClassifier(nn.Module):
    def __init__(self, bert_model, num_classes=5):
        super(SentimentClassifier, self).__init__()
        self.bert = bert_model
        self.classifier = nn.Linear(bert_model.config.hidden_size, num_classes)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        logits = self.classifier(pooled_output)
        return logits


# Load the data
train_df = pd.read_csv(
    "/PATH/TO/RAW/DATA/sentiment-analysis-on-movie-reviews/train.tsv",
    sep="\t",
)
test_df = pd.read_csv(
```

```python
    "/PATH/TO/RAW/DATA/sentiment-analysis-on-movie-reviews/test.tsv",
    sep="\t",
)

# Split the data into training and validation sets
train_df, val_df = train_test_split(train_df, test_size=0.2, random_state=42)

# Tokenizer and model
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
bert_model = BertModel.from_pretrained("bert-base-uncased")

# Create datasets and dataloaders
train_dataset = SentimentDataset(train_df, tokenizer)
val_dataset = SentimentDataset(val_df, tokenizer)
test_dataset = SentimentDataset(test_df, tokenizer)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

# Model, loss, and optimizer
model = SentimentClassifier(bert_model)
model = model.to("cuda")

criterion = nn.CrossEntropyLoss(
    weight=torch.tensor([1.0, 1.0, 1.0, 1.0, 1.0]).to("cuda")
)
optimizer = optim.Adam(model.parameters(), lr=2e-5)

# Training loop
num_epochs = 3
for epoch in range(num_epochs):
    model.train()
    for batch in train_loader:
        input_ids = batch["input_ids"].to("cuda")
        attention_mask = batch["attention_mask"].to("cuda")
        labels = batch["label"].to("cuda")

        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask)
```

```python
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    # Validation
    model.eval()
    val_predictions = []
    val_labels = []
    with torch.no_grad():
        for batch in val_loader:
            input_ids = batch["input_ids"].to("cuda")
            attention_mask = batch["attention_mask"].to("cuda")
            labels = batch["label"].to("cuda")

            outputs = model(input_ids, attention_mask)
            _, preds = torch.max(outputs, dim=1)

            val_predictions.extend(preds.cpu().numpy())
            val_labels.extend(labels.cpu().numpy())

    val_accuracy = accuracy_score(val_labels, val_predictions)
    print(f"Epoch {epoch + 1}/{num_epochs}, Validation Accuracy: {val_accuracy:.4f}")

# Generate predictions for the test set
model.eval()
test_predictions = []
with torch.no_grad():
    for batch in test_loader:
        input_ids = batch["input_ids"].to("cuda")
        attention_mask = batch["attention_mask"].to("cuda")

        outputs = model(input_ids, attention_mask)
        _, preds = torch.max(outputs, dim=1)

        test_predictions.extend(preds.cpu().numpy())

# Save predictions to submission.csv
submission = pd.DataFrame(
    {"PhraseId": test_df["PhraseId"], "Sentiment": test_predictions}
)
submission.to_csv("./submission/submission.csv", index=False)
```

The example below is taken from a computer vision competition `5-flowers-image-classification`.The solution was generated by Agent K as one of the draft nodes. Agent K is provided with CoT from the scaffolding stage to guide its initial solution generation.

Prompt

```
...
<Skipped the common instructions for prompt display purposes
    in paper appendix, only changes are added below>
- <TOTAL_TIME_REMAINING: 44.0hrs 28.0mins 57secs>
- <TOTAL_STEPS_REMAINING: 4996>
...
```

Data Overview

```
/PATH/TO/RAW/DATA
  |
  +-Sample_submission.csv (401 lines)
  |
  +-test/
  | +-1.jpg (31.8 kB)
  | +-10.jpg (4.5 kB)
  | +-100.jpg (16.9 kB)
  | +-... and 397 other files
  |
  +-train/
    |    +daisy/
    |    |    +-100080576_f52e8ee070_n.jpg (26.8 kB)
    |    |    +-10140303196_b88d3d6cec.jpg (117.2 kB)
    |    |    +-10172379554_b296050f82_n.jpg (36.4 kB)
    |    |    +-... and 688 other files
    |    +dandelion/
    |    |    +-10043234166_e6dd915111_n.jpg (26.7 kB)
    |    |    +-10200780773_c6051a7d71_n.jpg (23.3 kB)
    |    |    +-10294487385_92a0676c7d_m.jpg (21.7 kB)
    |    |    +-... and 948 other files
    |    +rose/
    |    |    +-10090824183_d02c613f10_m.jpg (20.2 kB)
    |    |    +-102501987_3cdb8e5394_n.jpg (29.8 kB)
    |    |    +-10503217854_e66a804309.jpg (208.6 kB)
    |    |    +-... and 691 other files
    |    +sunflower/
    |    |    +-1008566138_6927679c8a.jpg (143.2 kB)
    |    |    +-1022552002_2b93faf9e7_n.jpg (42.7 kB)
    |    |    +-1022552036_67d33d5bd8_n.jpg (60.8 kB)
    |    |    +-... and 656 other files
    |    +tulip/
```

```
    |    |   +-100930342_92e8746431_n.jpg (26.2 kB)
    |    |   +-10094729603_eeca3f2cb6.jpg (214.8 kB)
    |    |   +-10094731133_94a942463c.jpg (225.0 kB)
    |    |   +-... and 867 other files
```

Task description

Sample_submission.csv has 400 rows and 2 columns.
The columns are: id, label

Dataset Description
Image classification: 5 kinds of flowers
There are pictures with 5 kinds of flowers.
labels: daisy/dandelion/rose/sunflower/tulip -> 0/1/2/3/4
The labels in submission file should be integer (0/1/2/3/4).
train data: 5 folders of train data
test data: 1 folder of test data

Evaluation
The evaluation metric for this competition is Categorization Accuracy. Labels in your submission file should be integer (or string) type, i.e., 0/1/2/3/4.
rgin5, margin6, margin7, margin8, margin9, margin10, margin11, margin12, margin13... and 179 more columns

Summary of Past Submissions

Submission 0 summary and metric value on a validation set:
The project includes a comprehensive pipeline for processing and extracting features from images using a pre-trained EfficientNetV2-S model. Custom transformation pipelines are defined for both training and testing images, incorporating various augmentations and normalization techniques. The ImageEmbedder model is designed to generate latent representations of images, with the option to fine-tune the last few layers. Additionally, the project addresses class imbalances and implements a TabularHead model for handling both classification and regression tasks, complete with appropriate loss functions.
"submission.csv": 0.9627714581178903
—

Submission 1 summary and metric value on a validation set:
The project involves creating a comprehensive pipeline for image processing and feature extraction using EfficientNetB0. For training images, a custom transformation pipeline is defined, including resizing, center cropping, random horizontal flips, random rotations, color jitter, conversion to tensor, and normalization. For test images, a simpler transformation pipeline is used, involving resizing, center cropping, conversion to tensor, and normalization.

An ImageEmbedder class is implemented to extract latent representations from the transformed images, with the option to unfreeze the last few layers for fine-tuning. Additionally, class imbalances are handled, and a TabularHead model is implemented to decode the image embeddings and predict both regression and classification targets, along with corresponding loss functions.
"submission.csv": 0.9524301964839711

—

Submission 2 summary and metric value on a validation set:
The project involves creating an end-to-end image processing pipeline for a multi-task learning problem. It includes defining custom image transformation pipelines for both training and testing, using a pre-trained ResNet50 model to extract image embeddings, handling class imbalances, and implementing a custom PyTorch model (TabularHead) to process these embeddings and produce both classification and regression outputs. Loss functions for both tasks are defined to optimize the model's performance.
"submission.csv": 0.9503619441571872

531
532

Assistant

To tackle this image classification task, I will use a pre-trained ResNet50 model from ‘torchvision.models‘ to extract features from the images. The model will be fine-tuned on the training data, and predictions will be made on the test set. The evaluation metric will be the categorization accuracy on a hold-out validation set.

```
import os
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import models, transforms
from torch.utils.data import DataLoader, Dataset
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Define the dataset
class FlowerDataset(Dataset):
    def __init__(self, root_dir, transform=None, mode='
        train'):
        self.root_dir = root_dir
        self.transform = transform
        self.mode = mode
        self.image_paths = []
```

533

```
        self.labels = []

        if mode == 'train':
            for label, flower in enumerate(['daisy', 'dandelion', 'rose', 'sunflower', 'tulip']):
                folder_path = os.path.join(root_dir, flower)
                for image_name in os.listdir(folder_path):
                    self.image_paths.append(os.path.join(folder_path, image_name))
                    self.labels.append(label)
        else:
            for image_name in os.listdir(root_dir):
                self.image_paths.append(os.path.join(root_dir, image_name))

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = Image.open(image_path).convert('RGB')
        if self.transform:
            image = self.transform(image)

        if self.mode == 'train':
            label = self.labels[idx]
            return image, label
        else:
            return image, image_path

# Define transformations
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

```

```python
# Load datasets
train_dataset = FlowerDataset('/PATH/TO/RAW/DATA/5-flowers-image-classification/train', transform=train_transform, mode='train')
train_data, val_data, train_labels, val_labels = train_test_split(train_dataset, train_dataset.labels, test_size=0.2, stratify=train_dataset.labels, random_state=42)

train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
val_loader = DataLoader(val_data, batch_size=32, shuffle=False)

# Define the model
model = models.resnet50(pretrained=True)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 5)
model = model.to('cuda')

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)

    # Validation
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to('cuda'), labels.to('
```

```
cuda')
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    val_accuracy = correct / total
    print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {running_loss/len(train_data):.4f}, Val Loss: {val_loss/len(val_data):.4f}, Val Accuracy: {val_accuracy:.4f}')

# Test set predictions
test_dataset = FlowerDataset('/PATH/TO/RAW/DATA/5-flowers-image-classification/test', transform=test_transform, mode='test')
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

model.eval()
predictions = []
image_ids = []
with torch.no_grad():
    for inputs, image_paths in test_loader:
        inputs = inputs.to('cuda')
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        predictions.extend(predicted.cpu().numpy())
        image_ids.extend([os.path.basename(path) for path in image_paths])

# Save predictions to submission.csv
submission = pd.DataFrame({'id': image_ids, 'label': predictions})
submission['id'] = submission['id'].str.replace('.jpg', '')
submission.to_csv('./submission/submission.csv', index=False)
```

The example below is taken from a tabular competition `bike-sharing-demand`.The solution was generated by Agent K as one of the draft nodes. Agent K is provided with CoT from the scaffolding stage to guide its initial solution generation.

Prompt

```
...
<Skipped the common instructions for prompt display purposes
    in paper appendix,  only changes are added below>
<**Total Runtime**: You have a maximum of 86400 seconds to
   attempt this task. You may iteratively develop your
   solution within this time frame, but bear in mind that
   program runtime also counts towards this limit.
**Total Steps**: You have a maximum of 5000 steps to submit
   your solution.
- <TOTAL_TIME_REMAINING: 22.0hrs 39.0mins 51secs>
- <TOTAL_STEPS_REMAINING: 4982>
...
```

Data Overview

```
/PATH/TO/RAW/DATA
  |
  +-sampleSubmission.csv (6494 lines)
  |
  +-.. and 2 other files
```

Task description

Dataset Description

See, fork, and run a random forest benchmark model through Kaggle Scripts
You are provided hourly rental data spanning two years. For this competition, the training set is comprised of the first 19 days of each month, while the test set is the 20th to the end of the month. You must predict the total count of bikes rented during each hour covered by the test set, using only information available prior to the rental period. Data Fields
datetime - hourly date + timestamp
season - 1 = spring, 2 = summer, 3 = fall, 4 = winter
holiday - whether the day is considered a holiday
workingday - whether the day is neither a weekend nor holiday
weather - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
temp - temperature in Celsius
atemp - "feels like" temperature in Celsius
humidity - relative humidity
windspeed - wind speed
casual - number of non-registered user rentals initiated

registered - number of registered user rentals initiated
count - number of total rentals

sampleSubmission.csv has 6493 rows and 2 columns.
The columns are: datetime, count

test.csv has 6493 rows and 9 columns.
The columns are: datetime, season, holiday, workingday, weather, temp, atemp, humidity, windspeed

train.csv has 10886 rows and 12 columns.
The columns are: datetime, season, holiday, workingday, weather, temp, atemp, humidity, windspeed, casual, registered, count

Summary of Past Submissions
We test multiple ensemble models and combined the best-performing ones to make final predictions. Each model has a different contribution based on its performance. Below are some models and their contributions in the final prediction, along with their hyperparameters. Use this information to come up with a solution for the competition. There may be better models and sets of hyperparameter that can be used so you are free to explore and come up with a better solution
catboost

Learning Rate	Max Depth	Estimators	Bootstrap Type	Borders	Grow Policy	L2 Leaf Reg	Min Data Leaf	Random Strength	Contrib. (%)
0.05	12	100	MVS	64	Depthwise	0.3	5	1	22
0.05	16	300	MVS	510	SymmetricTree	0.3	10	10	12
0.15	13	70	MVS	32	Depthwise	0.1	5	1	11
0.05	16	300	MVS	510	SymmetricTree	0.2	1	100	8.9
0.05	16	300	MVS	1000	SymmetricTree	0.3	20	100	8.2
0.05	16	400	MVS	510	SymmetricTree	0.3	10	100	7.8
0.25	16	40	Bayesian_0	32	Depthwise	5	5	0	6.6
0.1	15	200	Bayesian_1	250	Depthwise	0.4	10	50	6.4
0.05	16	500	Bayesian_0	1000	SymmetricTree	0.5	100	100	3.2
0.25	15	40	Bayesian_5	32	Depthwise	2	5	0	3.1

Assistant

I propose using a CatBoost model, which is known for its effectiveness in handling categorical features and reducing overfitting. CatBoost can capture complex patterns in the data, potentially leading to better performance. The evaluation metric will still be the Root Mean Squared Logarithmic Error (RMSLE).

```python
import pandas as pd
import numpy as np
from catboost import CatBoostRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_log_error

```

```python
# Load the data
train_df = pd.read_csv('/PATH/TO/RAW/DATA/bike-sharing-demand/train.csv')
test_df = pd.read_csv('/PATH/TO/RAW/DATA/bike-sharing-demand/test.csv')

# Preprocess the data
train_df['datetime'] = pd.to_datetime(train_df['datetime'])
test_df['datetime'] = pd.to_datetime(test_df['datetime'])

# Extract features and target
X_train = train_df.drop(['datetime', 'casual', 'registered', 'count'], axis=1)
y_train = np.log1p(train_df['count'])

X_test = test_df.drop(['datetime'], axis=1)

# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

# Initialize the CatBoost model
model = CatBoostRegressor(
    learning_rate=0.05,
    max_depth=16,
    n_estimators=300,
    bootstrap_type='MVS',
    border_count=510,
    grow_policy='SymmetricTree',
    l2_leaf_reg=0.3,
    min_data_in_leaf=10,
    random_strength=100,
    silent=True
)

# Train the model
model.fit(X_train, y_train)

# Make predictions on the validation set
y_pred_val = model.predict(X_val)

# Calculate the RMSLE on the validation set
rmsle = np.sqrt(mean_squared_log_error(np.expm1(y_val), np.expm1(y_pred_val)))
print(f'Validation RMSLE: {rmsle}')

# Make predictions on the test set
```

```
y_pred_test = model.predict(X_test)

# Prepare the submission file
submission = pd.DataFrame({
    'datetime': test_df['datetime'],
    'count': np.expm1(y_pred_test)
})

# Save the submission file
submission.to_csv('./submission/submission.csv', index=
   False)
```

545

H.5 AIDE with RAG

546

547 The example below is taken from the solution generation process of one of the nodes in the
548 `invasive-species-monitoring` competition. In this case, AIDE is provided with one of the
549 most similar examples from the RAG database to serve as a guide to its solution generation.

Prompt

```
...
<Skipped the common instructions(same as \OurAgent{} CoT
   cases) for prompt display purposes in paper appendix,
   only changes are added below>
- <TOTAL_TIME_REMAINING: 94.0hrs 33.0mins 7secs>
- <TOTAL_STEPS_REMAINING: 4998>
...
```

Here is a few past experience cases written by an human expert for a relevant (but not the same) task

The overall design of the code is centered around a machine learning pipeline for the Kaggle Cassava Leaf Disease Classification competition. The pipeline includes data loading, transformation, model creation, and inference. It utilizes several pre-trained models for image classification, which are fine-tuned using cross-validation. The main script is designed to handle multiple models, different transformations, and test-time augmentation (TTA) to improve the robustness and accuracy of the predictions.

The model architecture is composed of multiple pre-trained models from the PyTorch Image Models (timm) library and Facebook Research's DeiT. The models used include 'tf_efficientnet_b4_ns', 'vit_base_patch16_384', and 'seresnext50_32x4d'. Each model is modified to fit the specific classification task by replacing the final fully connected layer to output the number of classes (5 for the Cassava leaf disease dataset). Here is a detailed breakdown of each model architecture:

550

1. **tf_efficientnet_b4_ns**: EfficientNet-B4 variant that is optimized for performance and efficiency. The model's classifier layer is replaced with a new linear layer of size 5.
2. **vit_base_patch16_384**: A Vision Transformer (ViT) base model with patch size 16 and input image size 384x384. The head layer is replaced with a linear layer of size 5.
3. **seresnext50_32x4d**: A variant of the ResNeXt50 model with SE (Squeeze-and-Excitation) blocks. The final fully connected layer is replaced with a linear layer of size 5.

All models are loaded with or without pretraining and are adapted to the task by adjusting the output layer to match the number of classes in the dataset.

The important hyper-parameters in this code are set within the ‘CFG‘ class. Key parameters include:
- ‘models‘: A list of model names to be used in the ensemble.
- ‘size‘: Dictionary mapping each model to its input image size.
- ‘batch_size‘: Number of samples per batch during training and inference.
- ‘target_size‘: Number of classes in the dataset (5).
- ‘n_fold‘: Number of folds for cross-validation.
- ‘trn_fold‘: A dictionary specifying the best and final folds for each model that should be used during training and inference.
- ‘data_parallel‘: Boolean dictionary indicating whether to use data parallelism for each model.
- ‘transform‘: Transformation types applied to the data during inference.
- ‘weight‘: Weights assigned to each model during the final prediction aggregation.
- ‘tta‘: Number of test-time augmentations to perform.
- ‘no_tta_weight‘: Weight assigned to the non-augmented inference results.

The optimization objective is not explicitly defined in the provided code snippet, as the code focuses on the inference process. However, based on typical practices in such competitions, the model is likely trained to minimize a loss function, such as cross-entropy loss, using the Adam optimizer or another suitable optimizer. The model is evaluated using metrics like accuracy, and the goal is to maximize the accuracy score on the validation set during training.

The advanced machine learning techniques used in this code include:
1. **Ensemble Learning**: Multiple pre-trained models are combined to form an ensemble, which helps in improving the generalization and robustness of the predictions.
2. **Cross-Validation**: The dataset is split into multiple folds, and the models are trained and validated using these folds to ensure better performance and avoid overfitting.

3. **Data Augmentation**: Various data augmentation techniques are applied during inference to improve the model's ability to handle variations in the input data.
4. **Test-Time Augmentation (TTA)**: Multiple predictions are generated for each input image using different transformations, and the results are averaged to produce the final prediction.

Other important tricks that contribute to the high performance of this code include:
- **Model Weighting**: Each model in the ensemble is assigned a weight, which is used to aggregate the predictions. This helps in leveraging the strengths of different models.
- **Careful Selection of Pre-trained Models**: The choice of pre-trained models (e.g., EfficientNet, ViT, and ResNeXt) ensures that the ensemble covers a diverse range of architectures, which can capture different aspects of the data.
- **Custom Transformations**: The use of custom transformations tailored to each model helps in optimizing the performance of each individual model.
- **Logging and Monitoring**: The use of logging allows for detailed tracking of the training and inference processes, which is crucial for debugging and performance optimization.

Data Overview
Your solution can use any relevant machine learning packages such as: ‘torchvision‘, ‘xgboost‘, ‘timm‘, ‘pandas‘, ‘torch-geometric‘, ‘scikit-learn‘, ‘statsmodels‘, ‘numpy‘, ‘torch‘, ‘bayesian-optimization‘, ‘lightGBM‘. Feel free to use any other packages too (all packages are already installed!). For neural networks we suggest using PyTorch rather than TensorFlow.

```
/PATH/TO/RAW/DATA
  |
  +-sample_submission.csv (1532 lines)
  |
  +-test/
  | +-1.jpg (31.8 kB)
  | +-10.jpg (4.5 kB)
  | +-100.jpg (16.9 kB)
  | +-... and 397 other files
  |
  +-test/
    |   +-1.jpg (648.4 kB)
    |   +-10.jpg (595.3 kB)
    |   +-100.jpg (637.2 kB)
    |   +-... and 1528 other files
  +-train/
    |   +-1.jpg (974.4 kB)
```

```
    |    +-10.jpg (1.1 MB)
    |    +-100.jpg (1.1 MB)
    |    +-... and 2291 other files
```

Task description

Dataset Description
The data set contains pictures taken in a Brazilian national forest. In some of the pictures there is Hydrangea, a beautiful invasive species original of Asia. Based on the training pictures and the labels provided, the participant should predict the presence of the invasive species in the testing set of pictures.
File descriptions
train.7z - the training set (contains 2295 images).
train_labels.csv - the correct labels for the training set.
test.7z - the testing set (contains 1531 images), ready to be labeled by your algorithm.
sample_submission.csv - a sample submission file in the correct format.
Data fields
name - name of the sample picture file (numbers)
invasive - probability of the picture containing an invasive species. A probability of 1 means the species is present.

Evaluation

Submissions are evaluated on area under the ROC curve between the predicted probability and the observed target. Submission File
For each image in the test set, you must predict a probability for the target variable on whether the image contains invasive species or not. The file should contain a header and have the following format:
name,invasive
2,0.5
5,0
6,0.2
etc.

Assistant

The solution will involve building a machine learning pipeline for image classification using a combination of pre-trained models. We will use a blend of models from the PyTorch Image Models (timm) library, including ‘tf_efficientnet_b4_ns‘ and ‘vit_base_patch16_384‘. The models will be fine-tuned on the training data using cross-validation, and test-time augmentation (TTA) will be applied to improve the robustness of the predictions. The final predictions will be aggregated using weighted averaging, and the results will be saved in the required submission format.

```python
import os
import pandas as pd
import numpy as np
import torch
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as transforms
from timm import create_model
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_auc_score
from PIL import Image

# Configuration
class CFG:
    models = ['tf_efficientnet_b4_ns', 'vit_base_patch16_384']
    size = {'tf_efficientnet_b4_ns': 384, 'vit_base_patch16_384': 384}
    batch_size = 16
    target_size = 1
    n_fold = 5
    trn_fold = {model: [0, 1, 2, 3, 4] for model in models}
    transform = {
        'train': transforms.Compose([
            transforms.Resize((384, 384)),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'test': transforms.Compose([
            transforms.Resize((384, 384)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])
    }
    tta = 5
    no_tta_weight = 0.5

# Dataset
class InvasiveDataset(Dataset):
    def __init__(self, df, root_dir, transform=None, mode='train'):
        self.df = df
        self.root_dir = root_dir
        self.transform = transform
```

```
        self.mode = mode

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, f"{self.df['name'].iloc[idx]}.jpg")
        image = Image.open(img_name).convert('RGB')
        if self.transform:
            image = self.transform(image)
        if self.mode == 'train':
            label = self.df['invasive'].iloc[idx]
            return image, label
        else:
            return image

# Model
def get_model(model_name, pretrained=True):
    model = create_model(model_name, pretrained=pretrained, num_classes=1)
    return model

# Training and Evaluation
def train_model(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    for inputs, labels in dataloader:
        inputs, labels = inputs.to(device), labels.to(device).float().unsqueeze(1)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)
    return running_loss / len(dataloader.dataset)

def evaluate_model(model, dataloader, device):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device).float().unsqueeze(1)
```

```
            outputs = model(inputs)
            preds = torch.sigmoid(outputs).cpu().numpy()
            all_preds.extend(preds)
            all_labels.extend(labels.cpu().numpy())
    return np.array(all_preds), np.array(all_labels)

# Main
def main():
    train_df = pd.read_csv('/PATH/TO/RAW/DATA/invasive-species-monitoring/train_labels.csv')
    test_df = pd.read_csv('/PATH/TO/RAW/DATA/invasive-species-monitoring/sample_submission.csv')
    skf = StratifiedKFold(n_splits=CFG.n_fold, shuffle=True, random_state=42)

    for model_name in CFG.models:
        model = get_model(model_name)
        model.to('cuda')
        criterion = torch.nn.BCEWithLogitsLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

        for fold, (train_idx, val_idx) in enumerate(skf.split(train_df, train_df['invasive'])):
            if fold not in CFG.trn_fold[model_name]:
                continue

            train_data = train_df.iloc[train_idx]
            val_data = train_df.iloc[val_idx]

            train_dataset = InvasiveDataset(train_data, '/PATH/TO/RAW/DATA/invasive-species-monitoring/train', CFG.transform['train'], 'train')
            val_dataset = InvasiveDataset(val_data, '/PATH/TO/RAW/DATA/invasive-species-monitoring/train', CFG.transform['test'], 'train')

            train_loader = DataLoader(train_dataset, batch_size=CFG.batch_size, shuffle=True, num_workers=4)
            val_loader = DataLoader(val_dataset, batch_size=CFG.batch_size, shuffle=False, num_workers=4)

            for epoch in range(5):  # 5 epochs for simplicity
```

```
                train_loss = train_model(model, train_loader, criterion, optimizer, 'cuda')
                val_preds, val_labels = evaluate_model(model, val_loader, 'cuda')
                val_auc = roc_auc_score(val_labels, val_preds)
                print(f"Fold {fold}, Epoch {epoch}, Train Loss: {train_loss:.4f}, Val AUC: {val_auc:.4f}")

        test_dataset = InvasiveDataset(test_df, '/PATH/TO/RAW/DATA/invasive-species-monitoring/test', CFG.transform['test'], 'test')
        test_loader = DataLoader(test_dataset, batch_size=CFG.batch_size, shuffle=False, num_workers=4)

        test_preds = []
        for _ in range(CFG.tta):
            preds = evaluate_model(model, test_loader, 'cuda')[0]
            test_preds.append(preds)
        test_preds = np.mean(test_preds, axis=0)

        test_df['invasive'] = test_preds
        test_df[['name', 'invasive']].to_csv(f'./submission/{model_name}_submission.csv', index=False)

if __name__ == "__main__":
    main()
```

References

[1] OpenAI Josh Achiam et al. “GPT-4 Technical Report”. In: 2023.

[2] Takuya Akiba et al. “Optuna: A Next-Generation Hyperparameter Optimization Framework”. In: *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 2623–2631.

[3] Simran Arora et al. “Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes”. In: *Proc. VLDB Endow.* 17.2 (Oct. 2023), pp. 92–105. DOI: 10.14778/3626292.3626294.

[4] Maximilian Balandat et al. “BOTORCH: a framework for efficient monte-carlo Bayesian optimization”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS ’20. Vancouver, BC, Canada: Curran Associates Inc., 2020.

[5] Ruisheng Cao et al. "Spider2-V: How Far Are Multimodal Agents From Automating Data Science and Engineering Workflows?" In: *CoRR* abs/2407.10956 (2024). arXiv: 2407.10956.

[6] Jun Shern Chan et al. "MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering". In: *The Thirteenth International Conference on Learning Representations*. 2025.

[7] Zui Chen et al. "SEED: Domain-Specific Data Curation With Large Language Models". In: *Arxiv* abs/2310.00749 (2023).

[8] Yizhou Chi et al. *SELA: Tree-Search Enhanced LLM Agents for Automated Machine Learning*. 2024.

[9] Alexander Cowen-Rivers et al. "HEBO: Pushing The Limits of Sample-Efficient Hyperparameter Optimisation". In: *Journal of Artificial Intelligence Research* 74 (July 2022).

[10] Yinlin Deng et al. "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).

[11] Kamil Dreczkowski, Antoine Grosnit, and Haitham Bou Ammar. "Framework and Benchmarks for Combinatorial and Mixed-variable Bayesian Optimization". In: *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2023.

[12] Haoyang Fang et al. "MLZero: A Multi-Agent System for End-to-end Machine Learning Automation". In: *ArXiv* abs/2505.13941 (2025).

[13] Matthias Feurer et al. "Auto-sklearn 2.0: hands-free AutoML via meta-learning". In: *J. Mach. Learn. Res.* 23.1 (Jan. 2022).

[14] P. J. A. Gijsbers et al. "An Open Source AutoML Benchmark". In: *ArXiv* abs/1907.00909 (2019).

[15] Siyuan Guo et al. "DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning". In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Proceedings of Machine Learning Research. PMLR, 2024, pp. 16813–16848.

[16] Taicheng Guo et al. "Large Language Model based Multi-Agents: A Survey of Progress and Challenges". In: *ArXiv* abs/2402.01680 (2024).

[17] Elad Hazan. "Introduction to Online Convex Optimization". In: *Found. Trends Optim.* 2 (2016), pp. 157–325.

[18] Noah Hollmann, Samuel Müller, and Frank Hutter. "Large Language Models for Automated Data Science: Introducing CAAFE for Context-Aware Automated Feature Engineering". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.

[19] Sirui Hong et al. "Data Interpreter: An LLM Agent For Data Science". In: *ArXiv* abs/2402.18679 (2024).

[20] Sirui Hong et al. "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework". In: *The Twelfth International Conference on Learning Representations*. 2024.

[21] T. Hospedales et al. "Meta-Learning in Neural Networks: A Survey". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 44.09 (Sept. 2022), pp. 5149–5169. DOI: 10.1109/TPAMI.2021.3079209.

[22] Xueyu Hu et al. "InfiAgent-DABench: Evaluating Agents on Data Analysis Tasks". In: *ArXiv* abs/2401.05507 (2024).

[23] Jie Huang and Kevin Chen-Chuan Chang. "Towards Reasoning in Large Language Models: A Survey". In: *ArXiv* abs/2212.10403 (2022).

[24] Qian Huang et al. "MLAgentBench: Evaluating Language Agents on Machine Learning Experimentation". In: *ArXiv* abs/2310.03302 (2023).

[25] Albert Qiaochu Jiang et al. "Mistral 7B". In: *ArXiv* abs/2310.06825 (2023).

[26] Carlos E Jimenez et al. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024.

[27] Haifeng Jin et al. "AutoKeras: An AutoML Library for Deep Learning". In: *Journal of Machine Learning Research* 24.6 (2023), pp. 1–6.

[28] Liqiang Jing et al. *DSBench: How Far Are Data Science Agents to Becoming Data Science Experts?* 2024.

[29] James Max Kanter and Kalyan Veeramachaneni. "Deep feature synthesis: Towards automating data science endeavors". In: *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)* (2015), pp. 1–10.

[30] Ambika Kaul, Saket Maheshwary, and Vikram Pudi. "AutoLearn — Automated Feature Generation and Selection". In: *2017 IEEE International Conference on Data Mining (ICDM)* (2017), pp. 217–226.

[31] Udayan Khurana et al. "Cognito: Automated Feature Engineering for Supervised Learning". In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. 2016, pp. 1304–1307. DOI: 10.1109/ICDMW.2016.0190.

[32] Yuhang Lai et al. "DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation". In: *ArXiv* abs/2211.11501 (2022).

[33] Erin LeDell and S. Poirier. "H2O AutoML: Scalable Automatic Machine Learning". In: 2020.

[34] Shuang Li et al. "Pre-Trained Language Models for Interactive Decision-Making". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 31199–31212.

[35] Ziming Li et al. "AutoKaggle: A Multi-Agent Framework for Autonomous Data Science Competitions". In: 2024.

[36] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *International Conference on Learning Representations*. 2019.

[37] Daqin Luo et al. "AutoM3L: An Automated Multimodal Machine Learning Framework with Large Language Models". In: *ACM Multimedia 2024*. 2024.

[38] Zhao Mandi, Shreeya Jain, and Shuran Song. *RoCo: Dialectic Multi-Robot Collaboration with Large Language Models*. 2023. arXiv: 2307.04738 [cs.RO].

[39] Alexandre Max Maraval et al. "End-to-End Meta-Bayesian Optimisation with Transformer Neural Processes". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.

[40] Meta Llama team. "Introducing Meta Llama 3: The most capable openly available LLM to date". In: *Meta AI Blog* (2024).

[41] Ambarish Moharil et al. "Towards efficient AutoML: a pipeline synthesis approach leveraging pre-trained transformers for multimodal data". In: *Mach. Learn.* 113 (2024), pp. 7011–7053.

[42] Christopher E. Mower et al. "ROS-LLM: A ROS framework for embodied AI with task feedback and structured reasoning". In: *ArXiv* abs/2406.19741 (2024).

[43] Danrui Qi and Jiannan Wang. "CleanAgent: Automating Data Standardization with LLM-based Agents". In: *ArXiv* abs/2403.08291 (2024).

[44] Yujia Qin et al. "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs". In: *ArXiv* abs/2307.16789 (2023).

[45] Changle Qu et al. "Tool Learning with Large Language Models: A Survey". In: *ArXiv* abs/2405.17935 (2024).

[46] Theodoros Rekatsinas et al. "HoloClean: holistic data repairs with probabilistic inference". In: *Proc. VLDB Endow.* 10.11 (2017), pp. 1190–1201. DOI: 10.14778/3137628.3137631.

[47] Xiaozhe Ren et al. "PanGu-Σ: Towards Trillion Parameter Language Model with Sparse Heterogeneous Computing". In: *ArXiv* abs/2303.10845 (2023).

[48] Hao Sha et al. "LanguageMPC: Large Language Models as Decision Makers for Autonomous Driving". In: *ArXiv* abs/2310.03026 (2023).

[49] Bo Shen et al. "PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback". In: *ArXiv* abs/2307.14936 (2023).

[50] Yongliang Shen et al. "HuggingGPT: solving AI tasks with chatgpt and its friends in hugging face". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2024.

[51] Ishika Singh et al. "ProgPrompt: Generating Situated Robot Task Plans using Large Language Models". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 11523–11530. DOI: 10.1109/ICRA48891.2023.10161317.

[52] Maojun Sun et al. "LAMBDA: A Large Model Based Data Agent". In: *arXiv preprint arXiv:2407.17535* (2024).

[53] Xiangru Tang et al. “MedAgents: Large Language Models as Collaborators for Zero-shot Medical Reasoning”. In: *Findings of the Association for Computational Linguistics ACL 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand and virtual meeting: Association for Computational Linguistics, Aug. 2024, pp. 599–621.

[54] Xiangru Tang et al. “ML-Bench: Evaluating Large Language Models and Agents for Machine Learning Tasks on Repository-Level Code”. In: (2024). arXiv: `2311.09835 ['cs.CL']`.

[55] Zhiqiang Tang et al. “AutoGluon-Multimodal (AutoMM): Supercharging Multimodal AutoML with Foundation Models”. In: *arXiv preprint arXiv:2404.16233* (2024).

[56] Open-interpreter Team. *Open-interpreter: A natural language interface for computers*. `https://github.com/OpenInterpreter/open-interpreter`. 2013.

[57] XAgent Team. *XAgent: An Autonomous Agent for Complex Task Solving*. 2023.

[58] Arun James Thirunavukarasu et al. “Large language models in medicine”. In: *Nature Medicine* 29 (2023), pp. 1930–1940.

[59] Edan Toledo et al. “AI Research Agents for Machine Learning: Search, Exploration, and Generalization in MLE-bench”. In: *arXiv preprint arXiv:2507.02554* (2025).

[60] Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. *AutoML-Agent: A Multi-Agent LLM Framework for Full-Pipeline AutoML*. 2024.

[61] Rasul Tutunov et al. “Why Can Large Language Models Generate Correct Chain-of-Thoughts?” In: *arXiv preprint arXiv:2310.13571* (2023).

[62] Colin White et al. “Neural Architecture Search: Insights from 1000 Papers”. In: *ArXiv* abs/2301.08727 (2023).

[63] Qingyun Wu et al. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework”. In: *COLM*. 2024.

[64] Liwenhan Xie et al. “WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization”. In: *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. UIST ’24. Pittsburgh, PA, USA: Association for Computing Machinery, 2024. DOI: `10.1145/3654777.3676374`.

[65] Zhiyu Yang et al. *MatPlotAgent: Method and Evaluation for LLM-Based Agentic Scientific Data Visualization*. 2024. arXiv: `2402.11453 [cs.CL]`.

[66] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *International Conference on Learning Representations (ICLR)*. 2023.

[67] Shunyu Yao et al. “Tree of Thoughts: Deliberate Problem Solving with Large Language Models”. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 11809–11822.

[68] Lei Zhang et al. “MLCopilot: Unleashing the Power of Large Language Models in Solving Machine Learning Tasks”. In: *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Yvette Graham and Matthew Purver. St. Julian’s, Malta: Association for Computational Linguistics, Mar. 2024, pp. 2931–2959.

[69] Yuge Zhang et al. “Benchmarking Data Science Agents”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 5677–5700.

[70] Qinkai Zheng et al. “CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X”. In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (2023).

[71] Terry Yue Zhuo et al. “BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions”. In: *arXiv preprint arXiv:2406.15877* (2024).