AutoML Agent Interface

Choose a dataset

MNIST Dataset

Select the dataset type

image

Select a GROQ LLM Model

llama-3.3-70b-versatile

Run AutoML Agent

AutoML Agent setup complete!

Generated Configuration Space Code

```
from ConfigSpace import ConfigurationSpace, Categorical, Float, Integer, EqualsCon

def get_configspace():
    cs = ConfigurationSpace(seed=1234)

    learning_rate = Categorical("learning_rate", ["adaptive", "constant"])
    alpha = Float("alpha", [1e-6, 1e-1], log=True)
    max_iter = Integer("max_iter", [100, 1000])
    eta0 = Float("eta0", [1e-4, 1.0], log=True)
    early_stopping = Categorical("early_stopping", [True, False], default=True)

    cs.add_hyperparameters([learning_rate, alpha, max_iter, eta0, early_stopping])

    cond_eta0 = EqualsCondition(eta0, learning_rate, "constant")
    cs.add_condition(EqualsCondition(eta0, learning_rate, "constant"))

    forbidden_clause = ForbiddenAndConjunction(
        ForbiddenEqualsClause(learning_rate, "constant"),
        ForbiddenEqualsClause(early_stopping, False)
    )
```

```
    cs.add_forbidden_clause(forbidden_clause)

    return cs
```

Generated Scenario Code

```
from smac.scenario import Scenario
from ConfigSpace import ConfigurationSpace

def generate_scenario(cs):
    scenario = Scenario({
        'run_obj': 'quality',
        'runcount-limit': 100,
        'wallclock-limit': 3600,
        'output_dir': "./automl_results",
        'shared_model': False,
        'cs': cs
    })
    return scenario
```

Generated Training Function Code

```
from typing import Any
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import SGDClassifier
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

def train(cfg: Any, seed: int, dataset: Any) -> float:
    X: np.ndarray = dataset['X']
    y: np.ndarray = dataset['y']

    learning_rate: str = cfg.get('learning_rate')
    alpha: float = cfg.get('alpha')
    max_iter: int = cfg.get('max_iter')
    eta0: float = cfg.get('eta0', 1.0)  # default to 1.0 if not provided

    class_weights = compute_class_weight(class_weight='balanced', classes=np.uniqu
    class_weights_dict = dict(zip(np.unique(y), class_weights))

    model = SGDClassifier(
        loss='log_loss',
```

```
        penalty='l2',
        alpha=alpha,
        max_iter=max_iter,
        learning_rate=learning_rate,
        eta0=eta0 if learning_rate == 'constant' else 1.0,
        warm_start=True,
        random_state=seed,
        class_weight=class_weights_dict
    )

    scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
    loss = 1.0 - np.mean(scores)

    return loss
```

Loss Value

`0.11991428571428586`

Prompts Used

▼ {

```
"config" :
```

```
"[TASK]
Goal: Write a Python function called get_configspace() that returns a
ConfigurationSpace for a classification task.

Strict Output Rules:
- Output only the get_configspace() function and necessary imports.
- No extra text, no explanation, no comments.
- Code must be syntactically correct, executable, and SMAC-compatible.

[ALLOWED_CLASSES]
# Core Classes
- ConfigurationSpace
- Categorical
- Float
- Integer
- Constant

# Conditions
- EqualsCondition
- InCondition
- OrConjunction

# Forbidden Clauses
- ForbiddenEqualsClause
- ForbiddenAndConjunction

# Distributions (only if needed)
- Beta
- Normal

# Serialization (only if needed)
- to_yaml()
- from_yaml()

[ALLOWED_OPTIONS]
- default
- log
- distribution
- seed

[REQUIRED_HYPERPARAMETERS]
- learning_rate: Categorical(["adaptive", "constant"])
- alpha: Float([1e-6, 1e-1], log=True)
- max_iter: Integer([100, 1000])
- eta0: Float([1e-4, 1.0], log=True) [Condition: active only if learning_rate
== "constant"]
- early_stopping: Categorical([True, False], default=True)
```

```
[CONDITIONS]
- eta0 must be active only when learning_rate == "constant" (use
EqualsCondition).

[CONSTRAINTS]
- Must use at least one ForbiddenAndConjunction to block invalid combinations.

[CONFIGURATION_SPACE]
- seed = 1234

[DATASET_DESCRIPTION]
- This is an image dataset.
Number of images: 70000
Labels available: 70000


[IMPORTANT_RULE]
- Do NOT use any classes, functions, methods, or modules outside of
[ALLOWED_CLASSES].

[EXAMPLES]

# Example 1: Basic ConfigurationSpace
```python
from ConfigSpace import ConfigurationSpace

cs = ConfigurationSpace(
    space={
        "C": (-1.0, 1.0),
        "max_iter": (10, 100),
    },
    seed=1234,
)
```
Example 2: Adding Hyperparameters
```python
from ConfigSpace import ConfigurationSpace, Categorical, Float, Integer

kernel_type = Categorical('kernel_type', ['linear', 'poly', 'rbf', 'sigmoid'])
degree = Integer('degree', bounds=(2, 4), default=2)
coef0 = Float('coef0', bounds=(0, 1), default=0.0)
gamma = Float('gamma', bounds=(1e-5, 1e2), default=1, log=True)

cs = ConfigurationSpace()
cs.add([kernel_type, degree, coef0, gamma])
```
```

Example 3: Adding Conditions

```python
from ConfigSpace import EqualsCondition, InCondition, OrConjunction

cond_1 = EqualsCondition(degree, kernel_type, 'poly')
cond_2 = OrConjunction(
    EqualsCondition(coef0, kernel_type, 'poly'),
    EqualsCondition(coef0, kernel_type, 'sigmoid')
)
cond_3 = InCondition(gamma, kernel_type, ['rbf', 'poly', 'sigmoid'])
```

Example 4: Adding Forbidden Clauses

```pyhon
from ConfigSpace import ForbiddenEqualsClause, ForbiddenAndConjunction

penalty_and_loss = ForbiddenAndConjunction(
    ForbiddenEqualsClause(penalty, "l1"),
    ForbiddenEqualsClause(loss, "hinge")
)
constant_penalty_and_loss = ForbiddenAndConjunction(
    ForbiddenEqualsClause(dual, "False"),
    ForbiddenEqualsClause(penalty, "l2"),
    ForbiddenEqualsClause(loss, "hinge")
)
penalty_and_dual = ForbiddenAndConjunction(
    ForbiddenEqualsClause(dual, "False"),
    ForbiddenEqualsClause(penalty, "l1")
)
```

Example 5: Serialization

```python
from pathlib import Path
from ConfigSpace import ConfigurationSpace

path = Path("configspace.yaml")
cs = ConfigurationSpace(
    space={
        "C": (-1.0, 1.0),
        "max_iter": (10, 100),
    },
    seed=1234,
)
cs.to_yaml(path)
loaded_cs = ConfigurationSpace.from_yaml(path)
```

Example 6: Priors

```python
```

```
import numpy as np
from ConfigSpace import ConfigurationSpace, Float, Categorical, Beta, Normal

cs = ConfigurationSpace(
    space={
        "lr": Float(
            'lr',
            bounds=(1e-5, 1e-1),
            default=1e-3,
            log=True,
            distribution=Normal(1e-3, 1e-1)
        ),
        "dropout": Float(
            'dropout',
            bounds=(0, 0.99),
            default=0.25,
            distribution=Beta(alpha=2, beta=4)
        ),
        "activation": Categorical(
            'activation',
            items=['tanh', 'relu'],
            weights=[0.2, 0.8]
        ),
    },
    seed=1234,
)
```

"

```
"scenario" :
```

```
"**Generate a Valid `generate_scenario(cs)` Function for SMAC (Python Only, Strict Output)**

**Goal:**
Write a **Python function** named `generate_scenario(cs)` that creates and returns a `Scenario` object configured for use with SMAC.

---

**Strict Rules:**

- Only output the function `generate_scenario(cs)` and necessary import statements.
- Use Python 3.10 type annotations.
- Use **only valid parameters** supported by SMAC's `Scenario` class.
- Code must be fully executable with the latest **SMAC v2.0+**.
- Do **not** include any explanation, usage examples, comments, or extra output—**only the function and imports**.
- no types needed for the function dont use type declaration

---

### Functional Requirements:
- The input `cs` is a object.
- Set the optimization objective to minimize **validation loss**.
- Set the `output_dir` to `"./automl_results"`
- Enable `shared_model`: False (for parallel optimization)
- Support **multi-fidelity tuning** suitable for **neural networks**.
- Allow **cloud-compatible parallel execution**.

---

### Output Rules:
- Only valid Python function and necessary imports.
- Use only allowed parameters listed below.
- Ensure compatibility with the `Scenario` class from SMAC.

---

### Allowed Scenario Parameters (for LLM Reference Only - Do Not Output):

- **algo_runs_timelimit**: Max CPU time for optimization (float)
- **always_race_default**: Race new configs against default (bool)
- **cost_for_crash**: Cost assigned to crashes in quality-based runs (float or list)
- **cutoff**: Max runtime per run (float, needed if `run_obj` = 'runtime')
- **deterministic**: Whether target function is deterministic (bool)
```

```
- **execdir**: Execution directory (str)
- **feature_fn**: Path to instance feature file (str)
- **initial_incumbent**: Initial config, e.g. 'DEFAULT' (str)
- **memory_limit**: Max memory in MB (float)
- **multi_objectives**: List of objectives to optimize (list[str])
- **overall_obj**: PARX for runtime penalty (str)
- **pcs_fn**: Path to PCS file (str)
- **run_obj**: Optimization metric: 'runtime' or 'quality' (str)
- **save_results_instantly**: Save after each update (bool)
- **ta**: Target algorithm call (str)
- **ta_run_limit**: Max algorithm runs (int)
- **test_inst_fn**, **train_inst_fn**: Files with test/train instances (str)
- **wallclock_limit**: Max wall-clock time (float)
- **abort_on_first_run_crash**: Abort if first run crashes (bool)
- **acq_opt_challengers**: Number of challengers for acquisition (int)
- **hydra_iterations**: Number of Hydra iterations (int)
- **input_psmac_dirs**: For parallel runs (list)
- **intens_adaptive_capping_slackfactor**: Slack factor for adaptive capping
(float)
- **intens_min_chall**: Min challengers per intensification (int)
- **intensification_percentage**: Fraction of time for intensification (float)
- **limit_resources**: Limit time/memory using pynisher (bool)
- **maxR**, **minR**: Max/min calls per config (int)
- **output_dir**: Output directory (str)
- **rand_prob**: Probability of running a random config (float)
- **random_configuration_chooser**: Path to custom random chooser (str)
- **rf_do_bootstrapping**: Use bootstrapping in RF (bool)
- **rf_max_depth**, **rf_min_samples_leaf**, **rf_min_samples_split**: RF
params (int)
- **rf_num_trees**: Number of RF trees (int)
- **rf_ratio_features**: Ratio of features per split (float)
- **shared_model**: Enable parallel shared model (bool)
- **sls_max_steps**, **sls_n_steps_plateau_walk**: Local search params (int)
- **transform_y**: Transform cost values (str)
- **use_ta_time**: Use target algorithm time (bool)

---

### Example:
from smac import Scenario

def generate_scenario(cs):
    scenario = Scenario({
        'option': 'value',
        ...
    })
```

```
True format: def generate_scenario(cs):


**Reminder:** Only output the function `generate_scenario(cs)` and required
imports. No extra text.
"
```

```
"train_function" :
```

```
"**Generate production-grade Python code for a machine learning training
function with the following STRICT requirements:**

---

### **Function signature** must be:

```python
def train(cfg: Configuration, seed: int, dataset: Any) -> float:
```

Function Behavior Requirements:

* The function **must accept** a `dataset` dictionary with:

  * `dataset['X']`: feature matrix
  * `dataset['y']`: label vector

* Assume `cfg` is a sampled configuration object:

  * Use `cfg.get('key')` to access **primitive values** only (`int`, `float`,
`str`, etc.).
  * **Do not** access or manipulate hyperparameter objects directly.

* Use **stratified k-fold cross-validation** via
`sklearn.model_selection.cross_val_score`.

  * Set `random_state=seed` to ensure reproducibility.

* You **must train a classification model** that satisfies:

  * Supports **early stopping** if `max_iter` is given.
  * Has `warm_start=True` or similar functionality.
  * Accepts `random_state=seed`.
  * Accepts `learning_rate` (`'constant'` or `'adaptive'`).
  * Accepts `C = 1.0 / alpha` or equivalent regularization strength.
  * If `learning_rate == 'constant'`, use `eta0`; otherwise, ignore `eta0`.

* You **may use any suitable model or framework** (e.g. `scikit-learn`,
`TensorFlow`, `PyTorch`) as long as it meets these requirements.

* Return a **loss value**:
  `loss = 1.0 - mean cross-validation accuracy`
  (lower loss = better model)
```

```
---

### **Only use the following hyperparameters:**

* `learning_rate`: str (`'constant'` or `'adaptive'`)
* `alpha`: float (log-scaled)
* `max_iter`: int
* `eta0`: float (only used if `learning_rate == 'constant'`)

Use them like this:

```python
value = cfg.get('hyperparameter_name')
```

Additional Constraints:

* Include all necessary imports.
* Use full type annotations for all arguments and variables.
* Output **only** the function definition and required imports — no extra text or example calls.

Supporting Code Provided:

* ConfigSpace definition: `from ConfigSpace import ConfigurationSpace, Categorical, Float, Integer, EqualsCondition, ForbiddenAndConjunction, ForbiddenEqualsClause

def get_configspace():
    cs = ConfigurationSpace(seed=1234)

    learning_rate = Categorical("learning_rate", ["adaptive", "constant"])
    alpha = Float("alpha", [1e-6, 1e-1], log=True)
    max_iter = Integer("max_iter", [100, 1000])
    eta0 = Float("eta0", [1e-4, 1.0], log=True)
    early_stopping = Categorical("early_stopping", [True, False], default=True)

    cs.add_hyperparameters([learning_rate, alpha, max_iter, eta0, early_stopping])

    cond_eta0 = EqualsCondition(eta0, learning_rate, "constant")
    cs.add_condition(EqualsCondition(eta0, learning_rate, "constant"))
```

```
    forbidden_clause = ForbiddenAndConjunction(
        ForbiddenEqualsClause(learning_rate, "constant"),
        ForbiddenEqualsClause(early_stopping, False)
    )
    cs.add_forbidden_clause(forbidden_clause)

    return cs
`
* SMAC scenario: `from smac.scenario import Scenario
from ConfigSpace import ConfigurationSpace

def generate_scenario(cs):
    scenario = Scenario({
        'run_obj': 'quality',
        'runcount-limit': 100,
        'wallclock-limit': 3600,
        'output_dir': "./automl_results",
        'shared_model': False,
        'cs': cs
    })
    return scenario
`
* Dataset description: `This is an image dataset.
Number of images: 70000
Labels available: 70000
`

"
}
```