AutoML Agent Interface

Select the dataset type

image

Choose a dataset

Fashion-MNIST (Keras)

Select a GROQ LLM Model

llama-3.3-70b-versatile

Run AutoML Agent

Generated Configuration Space Code

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

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

    learning_rate = Categorical('learning_rate', ['constant', 'invscaling', 'adapt
    eta0 = Float('eta0', bounds=(0.01, 1.0), default=0.1, log=True)
    max_iter = Integer('max_iter', bounds=(100, 1000), default=200)
    tol = Float('tol', bounds=(1e-5, 1e-1), default=1e-3, log=True)
    early_stopping = Categorical('early_stopping', ['True', 'False'], default='Fal
    validation_fraction = Float('validation_fraction', bounds=(0.01, 0.5), default
    n_jobs = Integer('n_jobs', bounds=(1, 10), default=1)
    random_state = Integer('random_state', bounds=(0, 100), default=42)

    cs.add_hyperparameters([learning_rate, eta0, max_iter, tol, early_stopping, va

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

    forbidden_eta0_and_early_stopping = ForbiddenAndConjunction(
        ForbiddenEqualsClause(eta0, 0.1),
        ForbiddenEqualsClause(early_stopping, 'True')
    )
    cs.add_forbidden_clause(forbidden_eta0_and_early_stopping)
    return cs
```

Generated Scenario Code

```
from smac.scenario import Scenario

def generate_scenario(cs):
    scenario = Scenario(
        configspace=cs,
        output_directory="./automl_results",
        deterministic=False,
        n_workers=4,
        min_budget=1,
        max_budget=100
    )
    return scenario
```

Generated Training Function Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import numpy as np
from ConfigSpace import Configuration

def train(cfg: Configuration, seed: int, dataset: dict) -> float:
    """
    Train a neural network model on the given dataset.

    Args:
    - cfg (Configuration): A Configuration object containing hyperparameters.
    - seed (int): The random seed for reproducibility.
    - dataset (dict): A dictionary containing the feature matrix 'X' and label vec

    Returns:
    - loss (float): The average training loss over 10 epochs.
    """

    # Set the random seed for reproducibility
    torch.manual_seed(seed)
    np.random.seed(seed)

    # Get the input and output dimensions dynamically from the dataset
    input_size = dataset['X'].values.shape[1]
    num_classes = len(np.unique(dataset['y'].values))

    # Check if the input data is already image-shaped
    if len(dataset['X'].values.shape) == 4:
        # If it's already image-shaped, use it as is
        X = dataset['X'].values
    else:
        # If not, reshape it to be image-shaped
        # Assuming the input size is a perfect square
        side_length = int(np.sqrt(input_size))
        if side_length ** 2 != input_size:
            raise ValueError("Input size is not a perfect square")
        X = dataset['X'].values.reshape(-1, 1, side_length, side_length)

    # Create a PyTorch dataset and data loader
    tensor_X = torch.from_numpy(X).float()
    tensor_y = torch.from_numpy(dataset['y'].values).long()
    dataset = TensorDataset(tensor_X, tensor_y)
    data_loader = DataLoader(dataset, batch_size=32, shuffle=True)

    # Create a simple neural network model
    model = nn.Sequential(
        nn.Conv2d(1, 10, kernel_size=5),
        nn.ReLU(),
        nn.Flatten(),
        nn.Linear(10 * (side_length - 4) ** 2, num_classes)
    )

    # Get the learning rate and optimizer from the configuration
    learning_rate = cfg.get('learning_rate')
    eta0 = cfg.get('eta0')

    if learning_rate == 'constant':
        optimizer = optim.SGD(model.parameters(), lr=eta0)
    elif learning_rate == 'invscaling':
        optimizer = optim.SGD(model.parameters(), lr=eta0, momentum=0.9)
    elif learning_rate == 'adaptive':
        optimizer = optim.Adam(model.parameters(), lr=eta0)

    # Train the model for 10 epochs
    loss_fn = nn.CrossEntropyLoss()
    total_loss = 0
    for epoch in range(10):
        for batch_X, batch_y in data_loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = loss_fn(outputs, batch_y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

    # Return the average training loss
    return total_loss / (10 * len(data_loader.dataset))
```

AutoML Agent setup complete!

Loss Value

`0.059312928309887644`

Prompts Used

▼ {

```
"config" :
```

```
"**TASK**

Goal: Write a Python function called `get_configspace()` that returns a valid
`ConfigurationSpace` for a classification task.

---

**STRICT OUTPUT RULES**

* Output only the `get_configspace()` function and necessary imports.
* Do not include any extra text, explanations, or comments.
* Code must be syntactically correct, executable, and compatible with SMAC.

---

**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`

---

**CONDITIONS**

* `eta0` must be active **only when** `learning_rate == "constant"` (use `EqualsCondition`).

---

**CONSTRAINTS**

* Must include **at least one** `ForbiddenAndConjunction` to block invalid combinations.

---

**CONFIGURATION SPACE REQUIREMENTS**

* Initialize `ConfigurationSpace` with `seed=1234`.

---

**DATASET DESCRIPTION**

* The configuration space must be based on the following information
This is an image dataset.
Number of images: 60000
Labels available: 60000
Raw feature shape: (60000, 784)
.
* Hyperparameters and model choices must reflect what is appropriate for that dataset type.

---

**IMPORTANT RULE**

* Do **not** use any classes, functions, methods, or modules outside of the **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(
~~~

~~~
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',
~~~

```
            'activation',
            items=['tanh', 'relu'],
            weights=[0.2, 0.8]
        ),
    },
    seed=1234,
)
```
"
```
```

```
"scenario" :
```

~~~
"---

**Objective:**
Generate a **Python function** named `generate_scenario(cs)` that returns a
valid `Scenario` object configured for SMAC (v2.0+), strictly following the
rules below.

---

**Output Format Rules (Strict):**

* Output **only** the function `generate_scenario(cs)` and the **necessary
import statements**.
* Use **Python 3.10 syntax** but **do not** include type annotations for the
function or parameters.
* The code must be **fully executable** with the latest **SMAC v2.0+** version.
* Output **only valid Python code** - **no comments**, **no explanations**,
**no extra text**, and **no example usage**.
* The function must be **self-contained**.

---

**Functional Requirements:**

* The input `cs` is a `ConfigurationSpace` object.
* Return a `Scenario` configured with the following:
  * `output_directory`: `"./automl_results"`
  * `deterministic`: `False` (enable variability)
  * `n_workers`: greater than 1 (to enable parallel optimization)
  * `min_budget` and `max_budget`: set appropriately for multi-fidelity tuning
(e.g., training epochs)
---

**Reminder:** The output must be limited to:

* Valid `import` statements
* A single `generate_scenario(cs)` function that returns a properly configured
`Scenario` object
* Do not use any parameters other than the ones explicitly listed in this
prompt.

---

**Example (Correct Output Format):**

```python
from smac import Scenario
~~~

~~~
from ConfigSpace import Configuration

def generate_scenario(cs: Configuration):
    scenario = Scenario(
        configspace=cs,
        objectives="validation_loss",
        output_directory="./automl_results",
        deterministic=False,
        min_budget=1,
        max_budget=100,
        n_workers=4
    )
    return scenario
```
"
~~~

"train_function" :

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

---

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

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

---

### **Function Behavior Requirements:**

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

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

* Assume `cfg` is a sampled configuration object:

  * Access primitive values using `cfg.get('key')` (only `int`, `float`, `str`, etc.).
  * **Do not access or manipulate non-primitive hyperparameter objects**.
  * Set `random_state=seed` or equivalent to ensure reproducibility in your chosen framework.

* The function must return the **average training loss** over 10 epochs.

* You must check whether dataset['X'] is already image-shaped (e.g., len(X.shape) == 4). If not, and CNN is used, reshape carefully and raise a ValueError if the input size is not a perfect square.

* Do not assume dataset['X'] has a specific shape. Always verify input dimensions before reshaping.

* If using a CNN model, you must validate that reshaping is safe and explain your assumption.

```python
return loss  # float
```

* Lower `loss` means a better model.
~~~

~~~
---

### **Frameworks**

You may choose **PyTorch**, **TensorFlow**, or **scikit-learn**, depending on the dataset and supporting code provided.

---

### **Model Requirements**

* Infer input and output dimensions dynamically from the dataset:

  ```python
  input_size = dataset['X'].shape[1]
  num_classes = len(np.unique(dataset['y']))
  ```

---

### **Optimizer Logic**

If `learning_rate` is specified in `cfg`, use:

* `'constant'`:

  * Use SGD with `lr=eta0` (supported in all frameworks)
* `'invscaling'`:

  * Use SGD with `lr=eta0` and `momentum=power_t` (if supported, otherwise fall back gracefully)
* `'adaptive'`:

  * Use Adam or equivalent with `lr=eta0`

- Only use valid parameters for each optimizer. Do **not** use unsupported arguments (e.g., `eta0` in PyTorch ASGD or `AdaptiveASGD`).

---

### **Supporting Code Provided:**

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

def get_configspace():
~~~

```
    cs = ConfigurationSpace(seed=1234)

    learning_rate = Categorical('learning_rate', ['constant', 'invscaling',
'adaptive'])
    eta0 = Float('eta0', bounds=(0.01, 1.0), default=0.1, log=True)
    max_iter = Integer('max_iter', bounds=(100, 1000), default=200)
    tol = Float('tol', bounds=(1e-5, 1e-1), default=1e-3, log=True)
    early_stopping = Categorical('early_stopping', ['True', 'False'],
default='False')
    validation_fraction = Float('validation_fraction', bounds=(0.01, 0.5),
default=0.1)
    n_jobs = Integer('n_jobs', bounds=(1, 10), default=1)
    random_state = Integer('random_state', bounds=(0, 100), default=42)

    cs.add_hyperparameters([learning_rate, eta0, max_iter, tol, early_stopping,
validation_fraction, n_jobs, random_state])

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

    forbidden_eta0_and_early_stopping = ForbiddenAndConjunction(
        ForbiddenEqualsClause(eta0, 0.1),
        ForbiddenEqualsClause(early_stopping, 'True')
    )
    cs.add_forbidden_clause(forbidden_eta0_and_early_stopping)

    return cs
`

* SMAC scenario: `from smac.scenario import Scenario

def generate_scenario(cs):
    scenario = Scenario(
        configspace=cs,
        output_directory="./automl_results",
        deterministic=False,
        n_workers=4,
        min_budget=1,
        max_budget=100
    )
    return scenario
`

* Dataset description: `This is an image dataset.
Number of images: 60000
Labels available: 60000
Raw feature shape: (60000, 784)
`
```

```
    ---

    ### **Additional Instructions**

    * The code must not hardcode dataset dimensions like `784` or class count `10`.
    * The function must be runnable and not assume unavailable classes or modules.
    * You must only output the `def train(...)` function and nothing else.
    "
}
```