AutoML Agent Interface

Select the dataset type

tabular

Choose a dataset

Iris

Select a GROQ LLM Model

gemini-2.0-flash

Run AutoML Agent

Instructor Response

Configuration Plan

For the Iris dataset, consider these configurations:

1. **k-NN:** Tune `n_neighbors` (3-7), `weights` ('uniform', 'distance'), and `metric` ('euclidean', 'manhattan', 'minkowski').
2. **SVM:** Optimize `C` (0.01-10), `kernel` ('linear', 'rbf', 'poly'), and `gamma` ('scale', 'auto', 0.01-1).

Reasoning:

- The dataset is small, so complex models may overfit. Simple models like k-NN and SVM with linear kernels are good starting points.
- k-NN benefits from tuning the number of neighbors, weighting scheme, and distance metric.
- SVM performance depends on the regularization strength (`C`), kernel type, and kernel coefficient (`gamma`).

Scenario Plan

Given the Iris dataset's small size and relatively quick training time, multi-fidelity optimization may not be essential but can be explored. However, to start simple:

1. **Facade Type:** `HyperparameterOptimizationFacade` (HPOFacade) is suitable for basic hyperparameter optimization.
2. **Budget Settings:** Not applicable for HPOFacade.
3. **Number of Workers:** `n_workers` can be set based on available resources. A small number like 2-4 might be sufficient for local experimentation.
4. **Scenario Parameters:**
   - `walltime_limit` : Set a reasonable time limit (e.g., 600 seconds).
   - `n_trials` : Set a number of trials (e.g., 100-200).

If Multi-Fidelity is desired:

1. **Facade Type:** Use `MultiFidelityFacade` .
2. **Budget Settings:** Define `min_budget` (e.g., 10% of the data) and `max_budget` (100% of the data, or all the data).
3. **Number of Workers:** Same as above.
4. **Scenario Parameters:** Same as above, plus `min_budget` and `max_budget` .

Train Function Plan

1. **Data Loading:** Load the Iris dataset using a library like scikit-learn.
2. **Model Initialization:** Initialize the model (e.g., k-NN, SVM) with the hyperparameters provided by SMAC.
3. **Training:** Train the model on the Iris dataset.
4. **Validation:** Evaluate the model's performance using cross-validation (e.g., 5-fold cross-validation) to get a robust estimate of the accuracy.
5. **Return Value:** Return the validation error (1 - accuracy) as the target function value for SMAC to minimize.

Generated Configuration Space Code

```
from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformInt

def get_configspace() -> ConfigurationSpace:
    cs = ConfigurationSpace()

    # k-NN Hyperparameters
    n_neighbors = UniformIntegerHyperparameter(
        "knn__n_neighbors", lower=3, upper=7, default_value=5
    )
    weights = CategoricalHyperparameter(
```

```
        "knn__weights", choices=['uniform', 'distance'], default_value='uniform'
    )
    metric = CategoricalHyperparameter(
        "knn__metric", choices=['euclidean', 'manhattan', 'minkowski'], default_va
    )

    # SVM Hyperparameters
    C = UniformFloatHyperparameter(
        "svm__C", lower=0.01, upper=10, default_value=1.0, log=True
    )
    kernel = CategoricalHyperparameter(
        "svm__kernel", choices=['linear', 'rbf', 'poly'], default_value='rbf'
    )
    gamma = CategoricalHyperparameter(
        "svm__gamma", choices=['scale', 'auto'], default_value='scale'
    )
    gamma_numeric = UniformFloatHyperparameter(
        "svm__gamma_numeric", lower=0.01, upper=1, default_value=0.1, log=True
    )


    # Add Hyperparameters to Configuration Space
    cs.add_hyperparameters([n_neighbors, weights, metric, C, kernel, gamma, gamma_


    # Add conditions
    gamma_numeric_condition = InCondition(child=gamma_numeric, parent=gamma, value
    cs.add_condition(gamma_numeric_condition)


    return cs
```

Generated Scenario Code

```
from smac import Scenario
from ConfigSpace import ConfigurationSpace

def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_Iris_20250617_171257",
        deterministic=False,
        n_trials=10,
```

```
        n_workers=1
    )
    return scenario
```

Generated Training Function Code

```
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from ConfigSpace import Configuration
from typing import Any
import warnings
warnings.filterwarnings("ignore")

def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    """
    Trains a scikit-learn model (k-NN or SVM) based on the provided configuration,
    dataset, and seed.  Returns the negative mean cross-validation accuracy.
    """
    np.random.seed(seed)

    X = dataset['X']
    y = dataset['y']

    try:
        if 'knn__n_neighbors' in cfg:
            knn = KNeighborsClassifier(
                n_neighbors=cfg.get('knn__n_neighbors'),
                weights=cfg.get('knn__weights'),
                metric=cfg.get('knn__metric')
            )
            scores = cross_val_score(knn, X, y, cv=5, scoring='accuracy')
            return -np.mean(scores)
        elif 'svm__C' in cfg:
            gamma = cfg.get('svm__gamma')
            gamma_val = cfg.get('svm__gamma_numeric') if gamma in ['scale', 'auto'

            svm = SVC(
                C=cfg.get('svm__C'),
                kernel=cfg.get('svm__kernel'),
                gamma=gamma_val,
                random_state=seed
            )
            scores = cross_val_score(svm, X, y, cv=5, scoring='accuracy')
```

```
            return -np.mean(scores)
        else:
            return -0.0  # Dummy value
    except Exception as e:
        print(f"Error during training: {e}")
        return -0.0 # Return dummy value on error
```

AutoML Agent setup complete!

Loss Value

`-0.9600000000000002`

Starting Optimization Process

Starting Optimization Process

Prompts Used

▼ {

```
"config" :
```

"**Generate a production-grade Python configuration space for machine learning hyperparameter optimization with the following STRICT requirements:**

Function signature must be:

```python
from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter,
UniformIntegerHyperparameter, CategoricalHyperparameter
def get_configspace() -> ConfigurationSpace:
```

Configuration Space Requirements:

* The configuration space **must** be appropriate for the dataset type and characteristics:
  * Dataset Description: `This is a tabular dataset.
It has 150 samples and 4 features.
Feature columns and types:
- 0: float64
- 1: float64
- 2: float64
- 3: float64

Feature statistical summary:

	0	1	2	3
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Label distribution:
0   50
1   50
2   50
Name: count, dtype: int64`

* Recommended Configuration based on the planner:
  * `For the Iris dataset, consider these configurations:

```
1.  **k-NN:** Tune `n_neighbors` (3-7), `weights` ('uniform', 'distance'), and `metric` ('euclidean', 'manhattan', 'minkowski').
2.  **SVM:** Optimize `C` (0.01-10), `kernel` ('linear', 'rbf', 'poly'), and `gamma` ('scale', 'auto', 0.01-1).

**Reasoning:**
*   The dataset is small, so complex models may overfit. Simple models like k-NN and SVM with linear kernels are good starting points.
*   k-NN benefits from tuning the number of neighbors, weighting scheme, and distance metric.
*   SVM performance depends on the regularization strength (`C`), kernel type, and kernel coefficient (`gamma`).`
* The configuration space **must** include:
  * Appropriate hyperparameter ranges based on the dataset characteristics
  * Reasonable default values
  * Proper hyperparameter types (continuous, discrete, categorical)
  * Conditional hyperparameters if needed
  * Proper bounds and constraints

* **Best Practices:**
  * Use meaningful hyperparameter names
  * Include proper documentation for each hyperparameter
  * Consider dataset size and complexity when setting ranges
  * Ensure ranges are not too narrow or too wide
  * Add proper conditions between dependent hyperparameters

* **Common Hyperparameters to Consider:**
  * Learning rate (if applicable)
  * Model-specific hyperparameters
  * Regularization parameters
  * Architecture parameters
  * Optimization parameters

---

### **Output Format:**

* Return **only** the `get_configspace()` function
* Include necessary imports
* No example usage or additional code
* The function must be self-contained and executable

---

### **Error Prevention:**

* Ensure all hyperparameter names are valid Python identifiers
```

```
* Verify that all ranges and bounds are valid
* Check that conditional hyperparameters are properly defined
* Validate that default values are within the specified ranges

---

### **Example Structure:**

```python
def get_configspace() -> ConfigurationSpace:
    cs = ConfigurationSpace()

    # Add hyperparameters
    learning_rate = UniformFloatHyperparameter(
        "learning_rate", lower=1e-4, upper=1e-1, default_value=1e-2, log=True
    )
    cs.add_hyperparameter(learning_rate)

    # Add more hyperparameters...

    return cs
```

Reminder: The output must be limited to:
* Valid `import` statements
* A single `get_configspace()` function that returns a properly configured
`ConfigurationSpace` object
* No additional code or explanations"
```

```
"scenario" :
```

"**Generate a production-grade Python scenario configuration for SMAC hyperparameter optimization with the following STRICT requirements:**

Function signature must be:

```python
from smac import Scenario
from ConfigSpace import ConfigurationSpace
def generate_scenario(cs: ConfigurationSpace) -> Scenario:
```

Scenario Configuration Requirements:

* The scenario **must** be optimized for the dataset characteristics:
  * Dataset Description: `This is a tabular dataset.
It has 150 samples and 4 features.
Feature columns and types:
- 0: float64
- 1: float64
- 2: float64
- 3: float64

Feature statistical summary:

	0	1	2	3
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Label distribution:

0	50
1	50
2	50

Name: count, dtype: int64`

* The scenario **must** include:
  * Appropriate budget settings (min_budget, max_budget)
  * Optimal number of workers for parallelization
  * Reasonable walltime and CPU time limits

```
  * Proper trial resource constraints
  * Appropriate number of trials

* **Best Practices:**
  * Set deterministic=False for better generalization
  * Use multi-fidelity optimization when appropriate
  * Configure proper output directory structure
  * Set appropriate trial resource limits
  * Enable parallel optimization when possible

* **Resource Management:**
  * Set appropriate memory limits for trials
  * Configure proper walltime limits
  * Enable parallel processing when beneficial
  * Consider dataset size for budget settings

---

### **Available Parameters:**
    configspace : ConfigurationSpace
        The configuration space from which to sample the configurations.
    name : str | None, defaults to None
        The name of the run. If no name is passed, SMAC generates a hash from
the meta data.
        Specify this argument to identify your run easily.
    output_directory : Path, defaults to Path("smac3_output")
        The directory in which to save the output. The files are saved in
`./output_directory/name/seed`.
    deterministic : bool, defaults to False
        If deterministic is set to true, only one seed is passed to the target
function.
        Otherwise, multiple seeds (if n_seeds of the intensifier is greater
than 1) are passed
        to the target function to ensure generalization.
    objectives : str | list[str] | None, defaults to "cost"
        The objective(s) to optimize. This argument is required for multi-
objective optimization.
    crash_cost : float | list[float], defaults to np.inf
        Defines the cost for a failed trial. In case of multi-objective, each
objective can be associated with
        a different cost.
    termination_cost_threshold : float | list[float], defaults to np.inf
        Defines a cost threshold when the optimization should stop. In case of
multi-objective, each objective *must* be
        associated with a cost. The optimization stops when all objectives
crossed the threshold.
    walltime_limit : float, defaults to np.inf
```

```
        The maximum time in seconds that SMAC is allowed to run.
    cputime_limit : float, defaults to np.inf
        The maximum CPU time in seconds that SMAC is allowed to run.
    trial_walltime_limit : float | None, defaults to None
        The maximum time in seconds that a trial is allowed to run. If not
specified,
        no constraints are enforced. Otherwise, the process will be spawned by
pynisher.
    trial_memory_limit : int | None, defaults to None
        The maximum memory in MB that a trial is allowed to use. If not
specified,
        no constraints are enforced. Otherwise, the process will be spawned by
pynisher.
    n_trials : int, defaults to 100
        The maximum number of trials (combination of configuration, seed,
budget, and instance, depending on the task)
        to run.
    use_default_config: bool, defaults to False.
        If True, the configspace's default configuration is evaluated in the
initial design.
        For historic benchmark reasons, this is False by default.
        Notice, that this will result in n_configs + 1 for the initial design.
Respecting n_trials,
        this will result in one fewer evaluated configuration in the
optimization.
    instances : list[str] | None, defaults to None
        Names of the instances to use. If None, no instances are used.
        Instances could be dataset names, seeds, subsets, etc.
    instance_features : dict[str, list[float]] | None, defaults to None
        Instances can be associated with features. For example, meta data of
the dataset (mean, var, ...) can be
        incorporated which are then further used to expand the training data of
the surrogate model.
    min_budget : float | int | None, defaults to None
        The minimum budget (epochs, subset size, number of instances, ...) that
is used for the optimization.
        Use this argument if you use multi-fidelity or instance optimization.
    max_budget : float | int | None, defaults to None
        The maximum budget (epochs, subset size, number of instances, ...) that
is used for the optimization.
        Use this argument if you use multi-fidelity or instance optimization.
    seed : int, defaults to 0
        The seed is used to make results reproducible. If seed is -1, SMAC will
generate a random seed.
    n_workers : int, defaults to 1
        The number of workers to use for parallelization. If `n_workers` is
greather than 1, SMAC will use
```

```
        Dask to parallelize the optimization.

---

### **Output Format:**

* Return **only** the `generate_scenario(cs)` function
* Include necessary imports
* No example usage or additional code
* The function must be self-contained and executable

---

### **Error Prevention:**

* Ensure all parameters are within valid ranges
* Verify that resource limits are reasonable
* Check that budget settings are appropriate
* Validate that parallelization settings are correct
* Ensure the training function can be pickled for parallel processing

---

### **Example Structure:**

```python
def generate_scenario(cs: ConfigurationSpace) -> Scenario:
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_Iris_20250617_171257" //this
is important and should not be changed
        deterministic=True,
        //other parameters based on the information
    )
    return scenario
```

Suggested Scenario Plan:

Given the Iris dataset's small size and relatively quick training time, multi-
fidelity optimization may not be essential but can be explored. However, to
start simple:

1. **Facade Type:** `HyperparameterOptimizationFacade` (HPOFacade) is suitable
```

1. **Facade Type:** `HyperparameterOptimizationFacade` (HPOFacade) is suitable for basic hyperparameter optimization.
2. **Budget Settings:** Not applicable for HPOFacade.
3. **Number of Workers:** `n_workers` can be set based on available resources. A small number like 2-4 might be sufficient for local experimentation.
4. **Scenario Parameters:**
    * `walltime_limit`: Set a reasonable time limit (e.g., 600 seconds).
    * `n_trials`: Set a number of trials (e.g., 100-200).

If Multi-Fidelity is desired:

1. **Facade Type:** Use `MultiFidelityFacade`.
2. **Budget Settings:** Define `min_budget` (e.g., 10% of the data) and `max_budget` (100% of the data, or all the data).
3. **Number of Workers:** Same as above.
4. **Scenario Parameters:** Same as above, plus `min_budget` and `max_budget`.

Reminder: The output must be limited to:
* Valid `import` statements
* A single `generate_scenario(cs)` function that returns a properly configured `Scenario` object
* No additional code or explanations
* The output_directory should be "./logs/gemini-2.0-flash_Iris_20250617_171257"
* Set the number of trials to 10 for sufficient exploration
* Set not resource limits
* set the number of workers to 1
"

```
"train_function" :
```

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

Function signature must be:

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

Function Behavior Requirements:

* The function **must** handle the dataset properly:
  * Dataset Description: `This is a tabular dataset.
It has 150 samples and 4 features.
Feature columns and types:
- 0: float64
- 1: float64
- 2: float64
- 3: float64

Feature statistical summary:

	0	1	2	3
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Label distribution:

0	50
1	50
2	50

Name: count, dtype: int64`
  * ConfigSpace Definition: `from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformIntegerHyperparameter, CategoricalHyperparameter, InCondition

def get_configspace() -> ConfigurationSpace:

```
    cs = ConfigurationSpace()

    # k-NN Hyperparameters
    n_neighbors = UniformIntegerHyperparameter(
        "knn__n_neighbors", lower=3, upper=7, default_value=5
    )
    weights = CategoricalHyperparameter(
        "knn__weights", choices=['uniform', 'distance'],
default_value='uniform'
    )
    metric = CategoricalHyperparameter(
        "knn__metric", choices=['euclidean', 'manhattan', 'minkowski'],
default_value='euclidean'
    )

    # SVM Hyperparameters
    C = UniformFloatHyperparameter(
        "svm__C", lower=0.01, upper=10, default_value=1.0, log=True
    )
    kernel = CategoricalHyperparameter(
        "svm__kernel", choices=['linear', 'rbf', 'poly'], default_value='rbf'
    )
    gamma = CategoricalHyperparameter(
        "svm__gamma", choices=['scale', 'auto'], default_value='scale'
    )
    gamma_numeric = UniformFloatHyperparameter(
        "svm__gamma_numeric", lower=0.01, upper=1, default_value=0.1, log=True
    )


    # Add Hyperparameters to Configuration Space
    cs.add_hyperparameters([n_neighbors, weights, metric, C, kernel, gamma,
gamma_numeric])


    # Add conditions
    gamma_numeric_condition = InCondition(child=gamma_numeric, parent=gamma,
values=['scale', 'auto'])
    cs.add_condition(gamma_numeric_condition)


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

def generate_scenario(cs: ConfigurationSpace) -> Scenario:
```

```
    scenario = Scenario(
        configspace=cs,
        name="HyperparameterOptimization",
        output_directory="./logs/gemini-2.0-flash_Iris_20250617_171257",
        deterministic=False,
        n_trials=10,
        n_workers=1
    )
    return scenario
```

* The function **must** accept a `dataset` dictionary with:
  * `dataset['X']`: feature matrix or input tensor
  * `dataset['y']`: label vector or label tensor

* The function **must** handle the configuration properly:
  * Access primitive values using `cfg.get('key')`
  * Handle all hyperparameters defined in the configuration space
  * Apply proper type conversion and validation
  * Handle conditional hyperparameters correctly

* **Model Requirements:**
  * Infer input and output dimensions dynamically
  * Follow data format requirements
  * Handle necessary data transformations
  * Implement proper model initialization
  * Use appropriate loss functions
  * Apply proper regularization
  * Handle model-specific requirements

* **Training Requirements:**
  * Implement proper training loop
  * Handle batch processing
  * Apply proper optimization
  * Implement early stopping if needed
  * Handle validation if required
  * Return appropriate loss value

* **Performance Optimization Requirements:**
  * Minimize memory usage and allocations
  * Use vectorized operations where possible
  * Avoid unnecessary data copying
  * Optimize data loading and preprocessing
  * Use efficient data structures
  * Minimize CPU/GPU synchronization
  * Implement efficient batch processing
  * Use appropriate device placement (CPU/GPU)

```
    * Use appropriate device placement (cpu/cuda)
    * Optimize model forward/backward passes
    * Minimize Python overhead

* **Code Optimization Requirements:**
    * Keep code minimal and focused
    * Avoid redundant computations
    * Use efficient algorithms
    * Minimize function calls
    * Optimize loops and iterations
    * Use appropriate data types
    * Avoid unnecessary object creation
    * Implement efficient error handling
    * Use appropriate caching strategies

* **Best Practices:**
    * Use proper error handling
    * Implement proper logging
    * Handle edge cases
    * Ensure reproducibility
    * Optimize performance
    * Follow framework best practices

---

### **Frameworks:**

Choose **one** of the following frameworks based on the dataset and
requirements:
* **PyTorch**: For deep learning tasks
* **TensorFlow**: For deep learning tasks
* **scikit-learn**: For traditional ML tasks

---

### **Output Format:**

* Return **only** the `train()` function
* Include necessary imports
* No example usage or additional code
* The function must be self-contained and executable
* Code must be minimal and optimized for performance

---

### **Error Prevention:**

* Validate all inputs
```

* Validate all inputs
* Handle missing or invalid hyperparameters
* Check data types and shapes
* Handle edge cases
* Implement proper error messages

Example Structure:

```python
def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    # Set random seed for reproducibility
    torch.manual_seed(seed)

    # Extract hyperparameters efficiently
    lr, bs = cfg.get('learning_rate'), cfg.get('batch_size')

    # Prepare data efficiently
    X, y = dataset['X'], dataset['y']

    # Initialize model with optimized parameters
    model = Model(X.shape[1], **cfg).to(device)

    # Optimized training loop
    for epoch in range(10):
        loss = train_epoch(model, X, y, lr, bs)

    return loss
```

Reminder: The output must be limited to:
* Valid `import` statements
* A single `train()` function that returns a float loss value
* No additional code or explanations
* Code must be optimized for performance and minimal in size
* Return negative loss/error since SMAC minimizes the objective
* For accuracy metrics, return negative accuracy (e.g. -accuracy)
* For error metrics, return the raw error value (e.g. mse, rmse)
* Ensure consistent sign convention across all metrics
  1. **Data Loading:** Load the Iris dataset using a library like scikit-learn.
2. **Model Initialization:** Initialize the model (e.g., k-NN, SVM) with the hyperparameters provided by SMAC.
3. **Training:** Train the model on the Iris dataset.
4. **Validation:** Evaluate the model's performance using cross-validation.

```
    4.  **Validation:** Evaluate the model's performance using cross-validation
    (e.g., 5-fold cross-validation) to get a robust estimate of the accuracy.
    5.  **Return Value:** Return the validation error (1 - accuracy) as the target
    function value for SMAC to minimize."
}
```

Download Generated Code and Prompts