AutoML Agent Interface

Select the dataset type

categorical

Choose a dataset

Breast Cancer

Select a GROQ LLM Model

gemini-2.0-flash

Run AutoML Agent

Instructor Response

Configuration Plan

{ "SMAC_scenario": { "facade_type": "HyperparameterOptimizationFacade", "min_budget": null, "max_budget": null, "n_workers": 4, "special_considerations": "For the Breast Cancer dataset, which is relatively small, multi-fidelity optimization might not be necessary. Standard hyperparameter optimization should be sufficient. The number of workers can be set based on available computational resources. Categorical features need to be properly encoded." }, "data_preprocessing": ["Encode categorical variables using one-hot encoding or label encoding.", "Handle missing values using imputation techniques or removal.", "Check for and address high cardinality of categorical variables."], "feature_engineering": ["Create interaction terms between relevant features.", "Generate polynomial features to capture non-linear relationships.", "Apply domain-specific feature engineering based on medical knowledge."], "common_challenges": ["Imbalanced class distribution.", "High dimensionality with a limited number of samples.", "Potential for overfitting."] }

Scenario Plan

Based on the dataset characteristics, a standard HyperparameterOptimizationFacade is appropriate. Multi-fidelity optimization is not needed due to the small size of the dataset. The number of workers is set to 4, assuming reasonable computational resources. Special considerations include handling categorical features and potential overfitting.

Train Function Plan

The train function should include data loading, preprocessing (encoding categorical features, handling missing values), model training, and validation. The function should return a performance metric (e.g., accuracy, F1-score) to be minimized by SMAC.

Generated Configuration Space Code

```
from ConfigSpace import ConfigurationSpace, UniformFloatHyperparameter, UniformInt
from ConfigSpace.conditions import InCondition

def get_configspace() -> ConfigurationSpace:
    """
    Returns a ConfigurationSpace object for hyperparameter optimization of a machi
    This configuration space is tailored for categorical datasets with a moderate
    and focuses on common hyperparameters for models that handle categorical featu
    """
    cs = ConfigurationSpace()

    # Define hyperparameters

    # Model type (Random Forest or Gradient Boosting)
    model_type = CategoricalHyperparameter(
        "model_type", choices=["random_forest", "gradient_boosting"], default_valu
    )
    cs.add_hyperparameter(model_type)

    # Random Forest hyperparameters
    rf_n_estimators = UniformIntegerHyperparameter(
        "rf_n_estimators", lower=50, upper=200, default_value=100,
    )
    rf_max_depth = UniformIntegerHyperparameter(
        "rf_max_depth", lower=2, upper=10, default_value=5,
    )
    rf_min_samples_split = UniformIntegerHyperparameter(
        "rf_min_samples_split", lower=2, upper=10, default_value=2,
    )
    rf_min_samples_leaf = UniformIntegerHyperparameter(
        "rf_min_samples_leaf", lower=1, upper=5, default_value=1,
    )

    # Gradient Boosting hyperparameters
    gb_n_estimators = UniformIntegerHyperparameter(
        "gb_n_estimators", lower=50, upper=200, default_value=100,
    )
    gb_learning_rate = UniformFloatHyperparameter(
        "gb_learning_rate", lower=1e-4, upper=1e-1, default_value=1e-2, log=True,
    )
    gb_max_depth = UniformIntegerHyperparameter(
        "gb_max_depth", lower=2, upper=10, default_value=3,
    )
    gb_min_samples_split = UniformIntegerHyperparameter(
        "gb_min_samples_split", lower=2, upper=10, default_value=2,
    )
    gb_min_samples_leaf = UniformIntegerHyperparameter(
        "gb_min_samples_leaf", lower=1, upper=5, default_value=1,
    )

    # Regularization hyperparameters applicable to both
    l1_ratio = UniformFloatHyperparameter(
        "l1_ratio", lower=0.0, upper=1.0, default_value=0.0,
    )

    # Add hyperparameters to the configuration space
    cs.add_hyperparameters([
        rf_n_estimators, rf_max_depth, rf_min_samples_split, rf_min_samples_leaf,
        gb_n_estimators, gb_learning_rate, gb_max_depth, gb_min_samples_split, gb_
        l1_ratio
    ])

    # Define conditional hyperparameters

    rf_condition = InCondition(child=rf_n_estimators, parent=model_type, values=["
    cs.add_condition(rf_condition)
    rf_condition = InCondition(child=rf_max_depth, parent=model_type, values=["ran
    cs.add_condition(rf_condition)
    rf_condition = InCondition(child=rf_min_samples_split, parent=model_type, valu
    cs.add_condition(rf_condition)
    rf_condition = InCondition(child=rf_min_samples_leaf, parent=model_type, value
    cs.add_condition(rf_condition)

    gb_condition = InCondition(child=gb_n_estimators, parent=model_type, values=["
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_learning_rate, parent=model_type, values=[
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_max_depth, parent=model_type, values=["gra
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_min_samples_split, parent=model_type, valu
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_min_samples_leaf, parent=model_type, value
    cs.add_condition(gb_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_Breast Cancer_20250617_174328",
        deterministic=False,
        n_trials=10,
        n_workers=1,
    )
    return scenario
```

Generated Training Function Code

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import accuracy_score, make_scorer, f1_score
from ConfigSpace import Configuration
from typing import Any
import warnings

def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    """
    Trains a machine learning model based on the provided configuration and datase

    Args:
        cfg (Configuration): A ConfigurationSpace object containing the hyperparam
        dataset (Any): A dictionary containing the training data ('X' for features
        seed (int): Random seed for reproducibility.

    Returns:
        float: The negative accuracy score on the validation set.
    """
    np.random.seed(seed)

    try:
        X = dataset['X']
        y = dataset['y']

        # Convert to DataFrame for easier preprocessing
        X = pd.DataFrame(X)
        y = pd.Series(y)

        # Identify categorical and numerical features (simplified)
        categorical_features = X.select_dtypes(include=['object', 'category']).col
        numerical_features = X.select_dtypes(include=['number']).columns.tolist()

        # Preprocessing pipelines
        numerical_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ])

        categorical_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ])

        # Column Transformer
        preprocessor = ColumnTransformer([
            ('numerical', numerical_pipeline, numerical_features),
            ('categorical', categorical_pipeline, categorical_features)
        ], remainder='passthrough') # or 'drop' if you don't want to keep the rest

        # Split data
        X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, ran

        X_train = preprocessor.fit_transform(X_train)
        X_val = preprocessor.transform(X_val)

        # Model selection and configuration
        model_type = cfg.get("model_type")

        if model_type == "random_forest":
            model = RandomForestClassifier(
                n_estimators=cfg.get("rf_n_estimators"),
                max_depth=cfg.get("rf_max_depth"),
                min_samples_split=cfg.get("rf_min_samples_split"),
                min_samples_leaf=cfg.get("rf_min_samples_leaf"),
                random_state=seed,
                n_jobs=-1,
                class_weight="balanced",
            )
        elif model_type == "gradient_boosting":
            model = GradientBoostingClassifier(
                n_estimators=cfg.get("gb_n_estimators"),
                learning_rate=cfg.get("gb_learning_rate"),
                max_depth=cfg.get("gb_max_depth"),
                min_samples_split=cfg.get("gb_min_samples_split"),
                min_samples_leaf=cfg.get("gb_min_samples_leaf"),
                random_state=seed,
            )
        else:
            raise ValueError(f"Unknown model type: {model_type}")

        # Train model
        with warnings.catch_warnings():
            warnings.filterwarnings("ignore", category=UserWarning) # Suppress som
            model.fit(X_train, y_train)

        # Evaluate model
        y_pred = model.predict(X_val)
        accuracy = accuracy_score(y_val, y_pred)
        print(f"Accuracy: {accuracy}")

        return -accuracy  # Return negative accuracy to be minimized by SMAC

    except Exception as e:
        print(f"Error during training: {e}")
        return -0.0  # Return a very low accuracy if training fails
```

AutoML Agent setup complete!

Loss Value

`-0.956140350877193`

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 categorical dataset.
It has 569 samples.
Categorical feature summary:

Target variable has 2 unique classes.

Categorical Data Handling Requirements:
1. Preprocessing Steps:
   - Encode categorical variables (one-hot or label encoding)
   - Handle missing values
   - Check for cardinality of categorical variables

2. Model Considerations:
   - Use appropriate encoding for model type
   - Handle high cardinality features appropriately
`

* Recommended Configuration based on the planner:
  * `{
  "SMAC_scenario": {
    "facade_type": "HyperparameterOptimizationFacade",
    "min_budget": null,
    "max_budget": null,
    "n_workers": 4,
    "special_considerations": "For the Breast Cancer dataset, which is
relatively small, multi-fidelity optimization might not be necessary. Standard
hyperparameter optimization should be sufficient. The number of workers can be
set based on available computational resources. Categorical features need to be
~~~

```
properly encoded."
  },
  "data_preprocessing": [
    "Encode categorical variables using one-hot encoding or label encoding.",
    "Handle missing values using imputation techniques or removal.",
    "Check for and address high cardinality of categorical variables."
  ],
  "feature_engineering": [
    "Create interaction terms between relevant features.",
    "Generate polynomial features to capture non-linear relationships.",
    "Apply domain-specific feature engineering based on medical knowledge."
  ],
  "common_challenges": [
    "Imbalanced class distribution.",
    "High dimensionality with a limited number of samples.",
    "Potential for overfitting."
  ]
}`

* 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 categorical dataset.
It has 569 samples.
Categorical feature summary:

Target variable has 2 unique classes.

Categorical Data Handling Requirements:
1. Preprocessing Steps:
   - Encode categorical variables (one-hot or label encoding)
   - Handle missing values
   - Check for cardinality of categorical variables

2. Model Considerations:
   - Use appropriate encoding for model type
   - Handle high cardinality features appropriately
`


* 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
```

~~~~
* 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_Breast
Cancer_20250617_174328" //this is important and should not be changed
        deterministic=True,
        //other parameters based on the information
    )
    return scenario
```

---

### **Suggested Scenario Plan:**

Based on the dataset characteristics, a standard
HyperparameterOptimizationFacade is appropriate. Multi-fidelity optimization is
not needed due to the small size of the dataset. The number of workers is set
to 4, assuming reasonable computational resources. Special considerations
include handling categorical features and potential overfitting.

---

**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
~~~~

```
* No additional code or explanations
* The output_directory should be "./logs/gemini-2.0-flash_Breast
Cancer_20250617_174328"
* Set the number of trials to 10 for sufficient exploration
* set the number of workers to 1
* do not set these parameters: walltime_limit, cputime_limit,
trial_walltime_limit ,trial_memory_limit="
```

```
"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 categorical dataset.
It has 569 samples.
Categorical feature summary:

Target variable has 2 unique classes.

Categorical Data Handling Requirements:
1. Preprocessing Steps:
   - Encode categorical variables (one-hot or label encoding)
   - Handle missing values
   - Check for cardinality of categorical variables

2. Model Considerations:
   - Use appropriate encoding for model type
   - Handle high cardinality features appropriately
`
  * ConfigSpace Definition: `from ConfigSpace import ConfigurationSpace,
UniformFloatHyperparameter, UniformIntegerHyperparameter,
CategoricalHyperparameter
from ConfigSpace.conditions import InCondition

def get_configspace() -> ConfigurationSpace:
    """
    Returns a ConfigurationSpace object for hyperparameter optimization of a
machine learning model.
    This configuration space is tailored for categorical datasets with a
moderate number of samples (around 500-600)
    and focuses on common hyperparameters for models that handle categorical
features well.
~~~

```
    """
    cs = ConfigurationSpace()

    # Define hyperparameters

    # Model type (Random Forest or Gradient Boosting)
    model_type = CategoricalHyperparameter(
        "model_type", choices=["random_forest", "gradient_boosting"],
default_value="random_forest"
    )
    cs.add_hyperparameter(model_type)

    # Random Forest hyperparameters
    rf_n_estimators = UniformIntegerHyperparameter(
        "rf_n_estimators", lower=50, upper=200, default_value=100,
    )
    rf_max_depth = UniformIntegerHyperparameter(
        "rf_max_depth", lower=2, upper=10, default_value=5,
    )
    rf_min_samples_split = UniformIntegerHyperparameter(
        "rf_min_samples_split", lower=2, upper=10, default_value=2,
    )
    rf_min_samples_leaf = UniformIntegerHyperparameter(
        "rf_min_samples_leaf", lower=1, upper=5, default_value=1,
    )

    # Gradient Boosting hyperparameters
    gb_n_estimators = UniformIntegerHyperparameter(
        "gb_n_estimators", lower=50, upper=200, default_value=100,
    )
    gb_learning_rate = UniformFloatHyperparameter(
        "gb_learning_rate", lower=1e-4, upper=1e-1, default_value=1e-2,
log=True,
    )
    gb_max_depth = UniformIntegerHyperparameter(
        "gb_max_depth", lower=2, upper=10, default_value=3,
    )
    gb_min_samples_split = UniformIntegerHyperparameter(
        "gb_min_samples_split", lower=2, upper=10, default_value=2,
    )
    gb_min_samples_leaf = UniformIntegerHyperparameter(
        "gb_min_samples_leaf", lower=1, upper=5, default_value=1,
    )

    # Regularization hyperparameters applicable to both
    l1_ratio = UniformFloatHyperparameter(
        "l1_ratio", lower=0.0, upper=1.0, default_value=0.0,
    )

    # Add hyperparameters to the configuration space
    cs.add_hyperparameters([
        rf_n_estimators, rf_max_depth, rf_min_samples_split,
rf_min_samples_leaf,
        gb_n_estimators, gb_learning_rate, gb_max_depth, gb_min_samples_split,
gb_min_samples_leaf,
        l1_ratio
    ])

    # Define conditional hyperparameters

    rf_condition = InCondition(child=rf_n_estimators, parent=model_type,
values=["random_forest"])
    cs.add_condition(rf_condition)
    rf_condition = InCondition(child=rf_max_depth, parent=model_type, values=
["random_forest"])
    cs.add_condition(rf_condition)
    rf_condition = InCondition(child=rf_min_samples_split, parent=model_type,
values=["random_forest"])
    cs.add_condition(rf_condition)
    rf_condition = InCondition(child=rf_min_samples_leaf, parent=model_type,
values=["random_forest"])
    cs.add_condition(rf_condition)

    gb_condition = InCondition(child=gb_n_estimators, parent=model_type,
values=["gradient_boosting"])
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_learning_rate, parent=model_type,
values=["gradient_boosting"])
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_max_depth, parent=model_type, values=
["gradient_boosting"])
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_min_samples_split, parent=model_type,
values=["gradient_boosting"])
    cs.add_condition(gb_condition)
    gb_condition = InCondition(child=gb_min_samples_leaf, parent=model_type,
values=["gradient_boosting"])
    cs.add_condition(gb_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_Breast
Cancer_20250617_174328",
        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
  * Minimize CPU/GPU synchronization
  * Implement efficient batch processing
  * Use appropriate device placement (CPU/GPU)
  * 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
  * The train function should be computational efficient

* **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
* 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
* For tracking the progress add prints
  The train function should include data loading, preprocessing (encoding
~~~

```
    categorical features, handling missing values), model training, and validation.
    The function should return a performance metric (e.g., accuracy, F1-score) to
    be minimized by SMAC."
}
```

Download Generated Code and Prompts