AutoML Agent Interface

Select the dataset type

tabular

Choose a dataset

Iris

Select a GROQ LLM Model

gemini-2.0-flash

Run AutoML Agent

Generated Configuration Space Code

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

def get_configspace():
    cs = ConfigurationSpace()

    # Define hyperparameters for a RandomForestClassifier
    n_estimators = Integer("n_estimators", bounds=(10, 200), default=100)
    max_depth = Integer("max_depth", bounds=(2, 10), default=6)
    min_samples_split = Integer("min_samples_split", bounds=(2, 10), default=2)
    min_samples_leaf = Integer("min_samples_leaf", bounds=(1, 10), default=1)

    cs.add_hyperparameters([n_estimators, max_depth, min_samples_split, min_sample

    # Example of a forbidden clause
    forbidden_clause = ForbiddenAndConjunction(
        ForbiddenEqualsClause(max_depth, 2),
        ForbiddenEqualsClause(min_samples_split, 2)
    )

    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(
        configspace=cs,
        name="gemini-2.0-flashiris20250607_112729",
        output_directory="./automl_results",
        deterministic=False,
        n_workers=4,
        min_budget=1,
        max_budget=10,
        n_trials=10
    )
    return scenario
```

Generated Training Function Code

```
from typing import Any
from ConfigSpace import Configuration
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import warnings

def train(cfg: Configuration, dataset: Any, seed: int) -> float:
    """
    Trains a RandomForestClassifier on the given dataset using the provided config

    Args:
        cfg (Configuration): A ConfigSpace Configuration object containing hyperpa
        dataset (Any): A dictionary containing the training data, with 'X' for fea
        seed (int): Random seed for reproducibility.

    Returns:
        float: The average training loss (negative cross-validation score) over 10
    """
    X = dataset['X']
```

```
    y = dataset['y']

    # Data Preprocessing: StandardScaler
    scaler = StandardScaler()

    # Model: RandomForestClassifier
    model = RandomForestClassifier(
        n_estimators=cfg.get("n_estimators"),
        max_depth=cfg.get("max_depth"),
        min_samples_split=cfg.get("min_samples_split"),
        min_samples_leaf=cfg.get("min_samples_leaf"),
        random_state=seed,
        n_jobs=1  # Explicitly set n_jobs to 1 for consistency
    )
    pipeline = Pipeline([('scaler', scaler), ('model', model)])


    # Cross-validation (10-fold)
    with warnings.catch_warnings():
        warnings.filterwarnings("ignore")
        cv_scores = cross_val_score(pipeline, X, y, cv=10, scoring='neg_log_loss',


    # Return average loss
    loss = -np.mean(cv_scores)  # Convert negative log-loss to positive loss

    return loss
```

AutoML Agent setup complete!

Loss Value

`0.16374038282213835`

Prompts Used

▾ {

"config" :

```
"**TASK**

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

---

**STRICT OUTPUT RULES**

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

---

**ALLOWED CLASSES**

**Core**

* `ConfigurationSpace`
* `Categorical`
* `Float`
* `Integer`
* `Constant`

**Conditions**

* `EqualsCondition`
* `InCondition`
* `OrConjunction`

**Forbidden Clauses**

* `ForbiddenEqualsClause`
* `ForbiddenAndConjunction` *(must include at least one)*

**Distributions (only if needed)**

* `Beta`
* `Normal`

**Serialization (only if needed)**

* `to_yaml()`
```

```
* `from_yaml()`

---

**CONSTRAINTS**

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

---

**DATASET DESCRIPTION**

* Use the following information to design the configuration space:
  `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`
* Hyperparameter choices and model types must be suitable for this
classification dataset.

---

**SUGGESTED PARAMETERS From OpenML**

* Here are some parameter configurations for this dataset from OpenML (for
inspiration only, not mandatory):
```

```
`[{9551: OpenML Parameter
================
ID...........: 9551
Flow ID.......: 1068
Flow Name.....: weka.J48(28)_C
Flow URL......: https://www.openml.org/f/1068
Parameter Name: C
  |__Data Type: option
  |__Default..: 0.25
  |__Value....: 0.25, 9552: OpenML Parameter
================
ID...........: 9552
Flow ID.......: 1068
Flow Name.....: weka.J48(28)_M
Flow URL......: https://www.openml.org/f/1068
Parameter Name: M
  |__Data Type: option
  |__Default..: 2
  |__Value....: 2}, {9566: OpenML Parameter
================
ID...........: 9566
Flow ID.......: 1070
Flow Name.....: weka.Ridor(3)_F
Flow URL......: https://www.openml.org/f/1070
Parameter Name: F
  |__Data Type: option
  |__Default..: 3
  |__Value....: 3, 9567: OpenML Parameter
================
ID...........: 9567
Flow ID.......: 1070
Flow Name.....: weka.Ridor(3)_S
Flow URL......: https://www.openml.org/f/1070
Parameter Name: S
  |__Data Type: option
  |__Default..: 1
  |__Value....: 1, 9570: OpenML Parameter
================
ID...........: 9570
Flow ID.......: 1070
Flow Name.....: weka.Ridor(3)_N
Flow URL......: https://www.openml.org/f/1070
Parameter Name: N
  |__Data Type: option
  |__Default..: 2.0
  |__Value....: 2.0}]`
```

IMPORTANT RULE

* Do **not** use any class, function, method, or module outside the **ALLOWED CLASSES** list.

EXAMPLES

* See provided examples for valid usage of hyperparameters, conditions, forbidden clauses, and priors.

[EXAMPLES]

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

cs = ConfigurationSpace(
    space={
        "C": (-1.0, 1.0),
        "max_iter": (10, 100),
    },
)
```

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')
```

````
    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),
    },
)
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,
````

```
            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]
        ),
    },
)"
```

"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:
  * `name`: `"gemini-2.0-flashiris20250607_112729"`
  * `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)
  * `n_trials`: 10
---

**Other Parameters to Consider:**
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.

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

* Valid `import` statements
* A single `generate_scenario(cs)` function that returns a properly configured
`Scenario` object

---

            Based on the following SMAC documentation, analyze the dataset
characteristics and choose appropriate:
            1. Facade type (e.g., MultiFidelityFacade for multi-fidelity
optimization)
            2. Budget settings (min_budget and max_budget)
            3. Number of workers (n_workers)
            4. Other relevant scenario parameters

            SMAC Documentation:
            Getting Started
#
SMAC needs four core components (configuration space, target function, scenario
and a facade) to run an
optimization process, all of which are explained on this page.
They interact in the following way:
Interaction of SMAC's components
Configuration Space
#
The configuration space defines the search space of the hyperparameters and,
therefore, the tunable parameters' legal
```

```
therefore, the tunable parameters' legal
ranges and default values.
from
ConfigSpace
import
ConfigSpace
cs
=
ConfigurationSpace
({
"myfloat"
:
(
0.1
,
1.5
),
# Uniform Float
"myint"
:
(
2
,
10
),
# Uniform Integer
"species"
:
[
"mouse"
,
"cat"
,
"dog"
],
# Categorical
})
Please see the documentation of
ConfigurationSpace
for more details.
Target Function
#
The target function takes a configuration from the configuration space and
returns a performance value.
For example, you could use a Neural Network to predict on your data and get
some validation performance.
If, for instance, you would tune the learning rate of the Network's optimizer,
every learning rate will
```

```
every learning rate will
change the final validation performance of the network. This is the target
function.
SMAC tries to find the best performing learning rate by trying different values
and evaluating the target function -
in an efficient way.
def
train
(
self
,
config
:
Configuration
,
seed
:
int
)
->
float
:
model
=
MultiLayerPerceptron
(
learning_rate
=
config
[
"learning_rate"
])
model
.
fit
(
...
)
accuracy
=
model
.
validate
(
...
)
return
1
```

1
-
accuracy
SMAC always minimizes (the smaller the better)
Note
In general, the arguments of the target function depend on the intensifier. However,
in all cases, the first argument must be the configuration (arbitrary argument name is possible here) and a seed.
If you specified instances in the scenario, SMAC requires
instance
as argument additionally. If you use
SuccessiveHalving
or
Hyperband
as intensifier but you did not specify instances, SMAC passes
budget
as
argument to the target function. But don't worry: SMAC will tell you if something is missing or if something is not
used.
Warning
SMAC
always
minimizes the value returned from the target function.
Warning
SMAC passes either
instance
or
budget
to the target function but never both.
Scenario
#
The
Scenario
is used to provide environment variables. For example,
if you want to limit the optimization process by a time limit or want to specify where to save the results.
from
smac
import
Scenario
scenario
=
Scenario
(
configspace

```
=
cs
,
name
=
"experiment_name"
,
output_directory
=
Path
(
"your_output_directory"
)
walltime_limit
=
120
,
# Limit to two minutes
n_trials
=
500
,
# Evaluated max 500 trials
n_workers
=
8
,
# Use eight workers
...
)
```

Note

If no
name
is given, a hash of the experiment is used. Running the same experiment again at a later time will result in exactly the same hash. This is important, because the optimization will warmstart on the preexisting evaluations, if not otherwise specified in the
Facade
.
Facade
#

Warn

By default Facades will try to warmstart on preexisting logs. This behavior can be specified using the
overwrite
parameter.

A

facade
is the entry point to SMAC, which constructs a default optimization
pipeline for you. SMAC offers various facades, which satisfy many common use
cases and are crucial to
achieving peak performance. The idea behind the facades is to provide a simple
interface to all of SMAC's components,
which is easy to use and understand and without the need of deep diving into
the material. However, experts are
invited to change the components to their specific hyperparameter optimization
needs. The following
table (horizontally scrollable) shows you what is supported and reveals the
default
components
:
Black-Box
Hyperparameter Optimization
Multi-Fidelity
Algorithm Configuration
Random
Hyperband
#Parameters
low
low/medium/high
low/medium/high
low/medium/high
low/medium/high
low/medium/high
Supports Instances
❌
✅
✅
✅
✅
✅
Supports Multi-Fidelity
❌
❌
✅
✅
❌
✅
Initial Design
Sobol
Sobol
Random
Default
Default

Default
Surrogate Model
Gaussian Process
Random Forest
Random Forest
Random Forest
Not used
Not used
Acquisition Function
Expected Improvement
Log Expected Improvement
Log Expected Improvement
Expected Improvement
Not used
Not used
Acquisition Maximizer
Local and Sorted Random Search
Local and Sorted Random Search
Local and Sorted Random Search
Local and Sorted Random Search
Not Used
Not Used
Intensifier
Default
Default
Hyperband
Default
Default
Hyperband
Runhistory Encoder
Default
Log
Log
Default
Default
Default
Random Design Probability
8.5%
20%
20%
50%
Not used
Not used
Info
The multi-fidelity facade is the closest implementation to
BOHB
.

Note
We want to emphasize that SMAC is a highly modular optimization framework.
The facade accepts many arguments to specify components of the pipeline. Please also note, that in contrast
to previous versions, instantiated objects are passed instead of
kwargs
.
The facades can be imported directly from the
smac
module.
from
smac
import
BlackBoxFacade
as
BBFacade
from
smac
import
HyperparameterOptimizationFacade
as
HPOFacade
from
smac
import
MultiFidelityFacade
as
MFFacade
from
smac
import
AlgorithmConfigurationFacade
as
ACFacade
from
smac
import
RandomFacade
as
RFacade
from
smac
import
HyperbandFacade
as
HBFacade
smac

```
=
HPOFacade
(
scenario
=
scenario
,
target_function
=
train
)
smac
=
MFFacade
(
scenario
=
scenario
,
target_function
=
train
)
smac
=
ACFacade
(
scenario
=
scenario
,
target_function
=
train
)
smac
=
RFacade
(
scenario
=
scenario
,
target_function
=
train
)
```

smac
=
HBFacade
(
scenario
=
scenario
,
target_function
=
train
)

Multi-Fidelity Optimization
#
Multi-fidelity refers to running an algorithm on multiple budgets (such as
number of epochs or
subsets of data) and thereby evaluating the performance prematurely. You can
run a multi-fidelity optimization
when using
Successive Halving
or
Hyperband
.
Hyperband
is the default intensifier in the
multi-fidelity facade
and requires the arguments
min_budget
and
max_budget
in the scenario if no instances are used.
In general, multi-fidelity works for both real-valued and instance budgets. In
the real-valued case,
the budget is directly passed to the target function. In the instance case, the
budget is not passed to the
target function but
min_budget
and
max_budget
are used internally to determine the number of instances of
each stage. That's also the reason why
min_budget
and
max_budget
are
not required

when using instances:
The
max_budget
is simply the max number of instances, whereas the
min_budget
is simply 1.
Warning
smac.main.config_selector.ConfigSelector
contains the
min_trials
parameter. This parameter determines
how many samples are required to train the surrogate model. If budgets are
involved, the highest budgets
are checked first. For example, if min_trials is three, but we find only two
trials in the runhistory for
the highest budget, we will use trials of a lower budget instead.
Please have a look into our
multi-fidelity examples
to see how to use
multi-fidelity optimization in real-world applications.

Components
#
In addition to the basic components mentioned in
Getting Started
, all other components are
explained in the following paragraphs to give a better picture of SMAC. These
components are all used to guide
the optimization process and simple changes can influence the results
drastically.
Before diving into the components, we shortly want to explain the main Bayesian
optimization loop in SMAC.
The
SMBO
receives all instantiated components from the facade and the logic happens
here.
In general, a while loop is used to ask for the next trial, submit it to the
runner, and wait for the runner to
finish the evaluation. Since the runner and the
SMBO
object are decoupled, the while loop continues and asks for even
more trials (e.g., in case of multi-threading), which also can be submitted to
the runner. If all workers are
occupied, SMAC will wait until a new worker is available again. Moreover,
limitations like wallclock time and remaining
trials are checked in every iteration.
Surrogate Model

Surrogate model

#

The surrogate model is used to approximate the objective function of configurations. In previous versions, the model was referred to as the Empirical Performance Model (EPM). Mostly, Bayesian optimization is used/associated with Gaussian processes. However, SMAC also incorporates random forests as surrogate models, which makes it possible to optimize for higher dimensional and complex spaces.

The data used to train the surrogate model is collected by the runhistory encoder (receives data from the runhistory and transforms it). If budgets are involved, the highest budget which satisfies min_trials (defaults to 1) in smac.main.config_selector is used. If no budgets are used, all observations are used.

If you are using instances, it is recommended to use instance features. The model is trained on each instance associated with its features. Imagine you have two hyperparameters, two instances and no instance features, the model would be trained on:

HP 1
HP 2
Objective Value
0.1
0.8
0.5
0.1
0.8
0.75
505
7
2.4
505
7
1.3

You can see that the same inputs lead to different objective values because of two instances. If you associate each instance with a feature, you would end-up with the following data points:

HP 1
HP 2
Instance Feature
Objective Value
0.1
0.8
0

0
0.5
0.1
0.8
1
0.75
505
7
0
2.4
505
7
1
1.3

The steps to receiving data are as follows:

The intensifier requests new configurations via
next(self.config_generator)
.

The config selector collects the data via the runhistory encoder which iterates over the runhistory trials.

The runhistory encoder only collects trials which are in
considered_states
and timeout trials. Also, only the
highest budget is considered if budgets are used. In this step, multi-objective values are scalarized using the
normalize_costs
function (uses
objective_bounds
from the runhistory) and the multi-objective algorithm.
For example, when ParEGO is used, the scalarization would be different in each training.

The selected trial objectives are transformed (e.g., log-transformed, depending on the selected
encoder).

The hyperparameters might still have inactive values. The model takes care of that after the collected data
are passed to the model.

Acquisition Function
#

Acquisition functions are mathematical techniques that guide how the parameter space should be explored during Bayesian
optimization. They use the predicted mean and predicted variance generated by the surrogate model.

The acquisition function is used by the acquisition maximizer (see next section). Otherwise, SMAC provides
a bunch of different acquisition functions (Lower Confidence Bound, Expected Improvement, Probability Improvement,

Thompson, integrated acquisition functions and prior acquisition functions). We refer to literature
for more information about acquisition functions.
Note
The acquisition function calculates the acquisition value for each configuration. However, the configurations
are provided by the acquisition maximizer. Therefore, the acquisition maximizer is responsible for receiving
the next configurations.
Acquisition Maximize
#
The acquisition maximizer is a wrapper for the acquisition function. It returns the next configurations. SMAC
supports local search, (sorted) random search, local and (sorted) random search, and differential evolution.
While local search checks neighbours of the best configurations, random search makes sure to explore the configuration
space. When using sorted random search, random configurations are sorted by the value of the acquisition function.
Warning
Pay attention to the number of challengers: If you experience RAM issues or long computational times in the
acquisition function, you might lower the number of challengers.
The acquisition maximizer also incorporates the
Random Design
. Please see the
ChallengerList
for more information.
Initial Design
#
The surrogate model needs data to be trained. Therefore, the initial design is used to generate the initial data points.
We provide random, latin hypercube, sobol, factorial and default initial designs. The default initial design uses
the default configuration from the configuration space and with the factorial initial design, we generate corner
points of the configuration space. The sobol sequences are an example of quasi-random low-discrepancy sequences and
the latin hypercube design is a statistical method for generating a near-random sample of parameter values from
a multidimensional distribution.
The initial design configurations are yielded by the config selector first. Moreover, the config selector keeps
track of which configurations already have been returned to make sure a configuration is not returned twice.
Random Design
#

The random design is used in the acquisition maximizer to tell whether the next configuration should be
random or sampled from the acquisition function. For example, if we use a random design with a probability of
50%, we have a 50% chance to sample a random configuration and a 50% chance to sample a configuration from the
acquisition function (although the acquisition function includes exploration and exploitation trade-off already).
This design makes sure that the optimization process is not stuck in a local optimum and we
are
guaranteed
to find the best configuration over time.
In addition to simple probability random design, we also provide annealing and modulus random design.
Intensifier
#
The intensifier compares different configurations based on evaluated :term: trial<Trial>
so far. It decides
which configuration should be
intensified
or, in other words, if a configuration is worth to spend more time on (e.g., evaluate another seed pair, evaluate on another instance, or evaluate on a higher budget).
Warning
Always pay attention to
max_config_calls
or
n_seeds
: If this argument is set high, the intensifier might
spend a lot of time on a single configuration.
Depending on the components and arguments, the intensifier tells you which seeds, budgets, and/or instances
are used throughout the optimization process. You can use the methods
uses_seeds
,
uses_budgets
, and
uses_instances
(directly callable via the facade) to (sanity-)check whether the intensifier uses these arguments.
Another important fact is that the intensifier keeps track of the current incumbent (a.k.a. the best configuration
found so far). In case of multi-objective, multiple incumbents could be found.
All intensifiers support multi-objective, multi-fidelity, and multi-threading:
Multi-Objective: Keeping track of multiple incumbents at once.

Multi-Fidelity: Incorporating instances or budgets.
Multi-Threading: Intensifier are implemented as generators so that calling
next
on the intensifier can be
repeated as often as needed. Intensifier are not required to receive results as the results are directly taken from
the runhistory.
Note
All intensifiers are working on the runhistory and recognize previous logged trials (e.g., if the user already
evaluated something beforehand). Previous configurations (in the best case, also complete trials) are added to the
queue/tracker again so that they are integrated into the intensification process.
That means continuing a run as well as incorporating user inputs are natively supported.
Configuration Selector
#
The configuration selector uses the initial design, surrogate model, acquisition maximizer/function, runhistory,
runhistory encoder, and random design to select the next configuration. The configuration selector is directly
used by the intensifier and is called everytime a new configuration is requested.
The idea behind the configuration selector is straight forward:
Yield the initial design configurations.
Train the surrogate model with the data from the runhistory encoder.
Get the next
retrain_after
configurations from the acquisition function/maximizer and yield them.
After all
retrain_after
configurations were yield, go back to step 2.
Note
The configuration selector is a generator and yields configurations. Therefore, the current state of the
selector is saved and when the intensifier calls
next
, the selector continues there where it stopped.
Note
Everytime the surrogate model is trained, the multi-objective algorithm is updated via
update_on_iteration_start
.
Multi-Objective Algorithm
#
The multi-objective algorithm is used to scalarize multi-objective values. The

multi-objective algorithm
gets normalized objective values passed and returns a single value. The resulting value (called by the
runhistory encoder) is then used to train the surrogate model.
Warning
Depending on the multi-objective algorithm, the values for the runhistory encoder might differ each time
the surrogate model is trained. Let's take ParEGO for example:
Everytime a new configuration is sampled (see ConfigSelector), the objective weights are updated. Therefore,
the scalarized values are different and the acquisition maximizer might return completely different configurations.
RunHistory
#
The runhistory holds all (un-)evaluated trials of the optimization run. You can use the runhistory to
get (running) configs, (running) trials, trials of a specific config, and more.
The runhistory encoder iterates over the runhistory to receive data for the surrogate model. The following
code shows how to iterate over the runhistory:

```
smac
=
HPOFacade
(
...
)
# Iterate over all trials
for
trial_info
,
trial_value
in
smac
.
runhistory
.
items
():
# Trial info
config
=
trial_info
.
config
instance
=
trial_info
```

```
.
instance
budget
=
trial_info
.
budget
seed
=
trial_info
.
seed
# Trial value
cost
=
trial_value
.
cost
time
=
trial_value
.
time
status
=
trial_value
.
status
starttime
=
trial_value
.
starttime
endtime
=
trial_value
.
endtime
additional_info
=
trial_value
.
additional_info
# Iterate over all configs
for
config
in
```

```
smac
.
runhistory
.
get_configs
():
# Get the cost of all trials of this config
average_cost
=
smac
.
runhistory
.
average_cost
(
config
)
```

Warning

The intensifier uses a callback to update the incumbent everytime a new trial is added to the runhistory.

RunHistory Encoder

#

The runhistory encoder is used to encode the runhistory data into a format that can be used by the surrogate model.

Only trials with the status

considered_states

and timeout trials are considered. Multi-objective values are scalarized using the

normalize_costs

function (uses

objective_bounds

from the runhistory). Afterwards, the normalized value is processed by the multi-objective algorithm.

Callback

#

Callbacks provide the ability to easily execute code before, inside, and after the Bayesian optimization loop.

To add a callback, you have to inherit from

smac.Callback

and overwrite the methods (if needed).

Afterwards, you can pass the callbacks to any facade.

```
from
smac
import
MultiFidelityFacade
,
Callback
```

```
class
CustomCallback
(
Callback
):
def
on_start
(
self
,
smbo
:
SMBO
)
->
None
:
pass
def
on_end
(
self
,
smbo
:
SMBO
)
->
None
:
pass
def
on_iteration_start
(
self
,
smbo
:
SMBO
)
->
None
:
pass
def
on_iteration_end
(
```

```
`
self
,
smbo
:
SMBO
,
info
:
RunInfo
,
value
:
RunValue
)
->
bool
|
None
:
# We just do a simple printing here
print
(
info
,
value
)
smac
=
MultiFidelityFacade
(
...
callbacks
=
[
CustomCallback
()]
)
smac
.
optimize
()
            Please analyze the dataset and documentation to determine:
            1. Should multi-fidelity optimization be used? (Consider dataset
size and training time)
            2. What budget range is appropriate? (Consider training epochs or
data subsets)
            3. How many workers should be used? (Consider available resources)
```

```
            3. How many workers should be used? (consider available resources)
            4. Are there any special considerations for this dataset type?

            Then generate a scenario configuration that best suits this
dataset.
            "
```

"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, dataset: Any, seed: int) -> 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**.

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

* You must carefully read and follow the dataset description provided, which
includes:
  * Data format and dimensions
  * Required preprocessing steps
  * Special handling requirements
  * Framework-specific considerations

```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
* Follow the data format requirements specified in the dataset description
* Handle any necessary data transformations as described in the dataset
description

---

### **Supporting Code Provided:**

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

def get_configspace():
    cs = ConfigurationSpace()

    # Define hyperparameters for a RandomForestClassifier
    n_estimators = Integer("n_estimators", bounds=(10, 200), default=100)
    max_depth = Integer("max_depth", bounds=(2, 10), default=6)
    min_samples_split = Integer("min_samples_split", bounds=(2, 10), default=2)
    min_samples_leaf = Integer("min_samples_leaf", bounds=(1, 10), default=1)

    cs.add_hyperparameters([n_estimators, max_depth, min_samples_split,
min_samples_leaf])

    # Example of a forbidden clause
    forbidden_clause = ForbiddenAndConjunction(
        ForbiddenEqualsClause(max_depth, 2),
        ForbiddenEqualsClause(min_samples_split, 2)
    )

    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(
        configspace=cs,
```

```
        name="gemini-2.0-flashiris20250607_112729",
        output_directory="./automl_results",
        deterministic=False,
        n_workers=4,
        min_budget=1,
        max_budget=10,
        n_trials=10
    )
    return scenario
`

* 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`

---

### **Additional Instructions**

* The code must not hardcode dataset dimensions
* The function must be runnable and not assume unavailable classes or modules
* You must only output the `def train(...)` function and nothing else
* Always check dataset description for format hints and requirements before
processing
Data preprocessing involves scaling the features since they have different
ranges. Consider StandardScaler or MinMaxScaler.
```

```
Feature engineering could involve creating polynomial features or feature
interactions, but given the small number of features, this might lead to
overfitting.

A common challenge is that the dataset is small, which can lead to overfitting.
Regularization techniques are important.

The dataset is balanced, so accuracy is an appropriate metric. However, it's
still useful to look at precision, recall, and F1-score, especially if you want
to understand class-specific performance.

Consider using cross-validation to evaluate the model performance robustly due
to the limited size of the dataset.

When interpreting the model, be mindful of the curse of dimensionality if
feature engineering is performed aggressively.

The Iris dataset is linearly separable; therefore, simpler models can achieve
high accuracy.

Apply dimensionality reduction techniques such as PCA or LDA to identify the
most important features and potentially improve model performance and
interpretability.

Investigate non-linear relationships between features and the target variable
using techniques like kernel methods or decision tree-based models.

Be cautious about outliers, as they can significantly impact model performance.
Consider using robust scaling techniques or outlier removal methods if
necessary.

When building classification models, evaluate and compare the performance of
different algorithms, such as logistic regression, support vector machines, and
decision trees.

Visualize the data using scatter plots or pair plots to gain insights into the
relationships between features and the target variable.

Address multicollinearity among features by using techniques like variance
inflation factor (VIF) or principal component analysis (PCA)."
}
```