OpenBox

Release beta

Thomas (Yang) Li

Jan 06, 2022

TABLE OF CONTENTS

OpenBox is an efficient open-source system designed for **solving generalized black-box optimization (BBO) problems**, such as automatic hyper-parameter tuning, automatic A/B testing, experimental design, database knob tuning, processor architecture and circuit design, resource allocation, automatic chemical design, etc.

The design of **OpenBox** follows the philosophy of providing **"BBO as a service"** - we opt to implement **OpenBox** as a distributed, fault-tolerant, scalable, and efficient service, with a wide range of application scope, stable performance across problems and advantages such as ease of use, portability, and zero maintenance.

There are two ways to use **OpenBox**: Standalone python package and Online BBO service.

CHAPTER

ONE

WHO SHOULD CONSIDER USING OPENBOX

- Those who want to **tune hyper-parameters** for their ML tasks automatically.
- Those who want to **find the optimal configuration** for their configuration search tasks (e.g., database knob tuning).
- Data platform owners who want to **provide BBO service in their platform**.
- Researchers and data scientists who want to **solve generalized BBO problems easily**.

CHAPTER

TWO

OPENBOX CAPABILITIES

OpenBox has a wide range of functionality scope, which includes:

1. BBO with multiple objectives and constraints.
2. BBO with transfer learning.
3. BBO with distributed parallelization.
4. BBO with multi-fidelity acceleration.
5. BBO with early stops.

In the following, we provide a taxonomy of existing BBO systems:

System/Package	Multi-obj.	FIOC	Constraint	History	Distributed
Hyperopt	×		×	×	
Spearmint	×	×		×	×
SMAC3	×		×	×	×
BoTorch		×		×	×
GPflowOPT		×		×	×
Vizier	×		×		
HyperMapper				×	×
HpBandSter	×		×	×	
OpenBox					

- **FIOC**: Support different input variable types, including Float, Integer, Ordinal and Categorical.
- **Multi-obj.**: Support optimizing multiple objectives.
- **Constraint**: Support inequality constraints.
- **History**: Support injecting prior knowledge from previous tasks into the current search. (means the system cannot support it for general cases)
- **Distributed**: Support parallel evaluations in a distributed environment.

CHAPTER

THREE

INSTALLATION

Please refer to our Installation Guide.

CHAPTER

FOUR

QUICK START

In the following, we provide an example of optimizing the Branin function. For more description of this example, please refer to Quick Start.

```
import numpy as np
from openbox import Optimizer, sp

# Define Search Space
space = sp.Space()
x1 = sp.Real("x1", -5, 10, default_value=0)
x2 = sp.Real("x2", 0, 15, default_value=0)
space.add_variables([x1, x2])

# Define Objective Function
def branin(config):
    x1, x2 = config['x1'], config['x2']
    y = (x2-5.1/(4*np.pi**2)*x1**2+5/np.pi*x1-6)**2+10*(1-1/(8*np.pi))*np.cos(x1)+10
    return y

# Run
if __name__ == '__main__':
    opt = Optimizer(branin, space, max_runs=50, task_id='quick_start')
    history = opt.run()
    print(history)
```

CHAPTER

FIVE

DOCUMENTATION

- To learn more about OpenBox, refer to OpenBox Overview.
- To install OpenBox, refer to OpenBox Installation Guide.
- To get started with OpenBox, refer to Quick Start Tutorial.

CHAPTER

SIX

RELATED ARTICLES

- Tuning LightGBM with OpenBox
- Tuning XGBoost using OpenBox

CHAPTER

SEVEN

RELEASES AND CONTRIBUTING

OpenBox has a frequent release cycle. Please let us know if you encounter a bug by filling an issue.

We appreciate all contributions. If you are planning to contribute any bug-fixes, please do so without further discussions.

If you plan to contribute new features, new modules, etc. please first open an issue or reuse an existing issue, and discuss the feature with us.

To learn more about making a contribution to OpenBox, please refer to our how-to-contribute page.

We appreciate all contributions and thank all the contributors!

CHAPTER

EIGHT

RELATED PUBLICATIONS

OpenBox: A Generalized Black-box Optimization Service Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, Ce Zhang, Bin Cui; ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2021).

MFES-HB: Efficient Hyperband with Multi-Fidelity Quality Measurements Yang Li, Yu Shen, Jiawei Jiang, Jinyang Gao, Ce Zhang, Bin Cui; The Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021).

CHAPTER

NINE

RELATED PROJECT

Targeting at openness and advancing the AutoML ecosystem, we have also released another open-source project.

- MindWare: an open-source system that provides end-to-end ML model training and inference capabilities.

CHAPTER

TEN

FEEDBACK

- File an issue on GitHub.
- Email us via liyang.cs@pku.edu.cn or shenyu@pku.edu.cn.

CHAPTER

ELEVEN

LICENSE

The entire codebase is under MIT license.

11.1 Overview

Black–box optimization (BBO) is the task of optimizing an objective function within a limited budget for function evaluations. ‘’Black-box” means that the objective function has no analytical form so that information such as the derivative of the objective function is unavailable. Since the evaluation of objective functions is often expensive, the goal of black-box optimization is to find a configuration that approaches the global optimum as rapidly as possible.

Traditional single-objective BBO has many applications, including:

- Automatic A/B testing.
- Experimental design.
- Database knob tuning.
- **Automatic hyperparameter tuning.**

Recently, generalized BBO emerges and has been applied to many areas:

- Processor architecture and circuit design.
- Resource allocation.
- Automatic chemical design.

Generalized BBO requires more general functionalities that may not be supported by traditional BBO, such as multiple objectives and constraints.

11.1.1 Design Principle

OpenBox is an efficient system designed for generalized Black-box Optimization (BBO). Its design satisfies the following desiderata:

- **Ease of use**: Minimal user effort, and user-friendly visualization for tracking and managing BBO tasks.
- **Consistent performance**: Host state-of-the-art optimization algorithms; Choose the proper algorithm automatically.
- **Resource-aware management**: Give cost-model-based advice to users, e.g., minimal workers or time-budget.
- **Scalability**: Scale to dimensions on the number of input variables, objectives, tasks, trials, and parallel evaluations.

- **High efficiency**: Effective use of parallel resources, system optimization with transfer-learning and multi-fidelities, etc.
- **Fault tolerance**, **extensibility**, and **data privacy protection**.

The figure below shows the high-level architecture of OpenBox service.

11.1.2 Main Components

- **Service Master** is responsible for node management, load balance, and fault tolerance.
- **Task Database** holds the history and states of all tasks.
- **Suggestion Server** generates new configurations for each task.
- **REST API** connects users/workers and suggestion service via RESTful APIs.
- **Evaluation workers** are provided and owned by the users.

11.1.3 Deployment Artifacts

Standalone Python package

Like other open-source packages, OpenBox has a frequent release cycle. Users can install the package via Pypi or source code on GitHub. For more installation details, refer to *Installation Guide*.

Distributed BBO service

We adopt the "BBO as a service" paradigm and implement OpenBox as a managed general service for black-box optimization. Users can access this service via RESTful API conveniently, regardless of other issues such as environment setups, software maintenance, and execution optimization. Moreover, OpenBox also provide Web UI for users to track and manage their running tasks. For deployment details, refer to *Deployment Guide*.

11.1.4 Performance Comparison

We compare OpenBox with six competitive open-source BBO systems on tuning LightGBM using 25 datasets. The performance rank (the lower, the better) is shown in the following figure. For dataset information and more experimental results, please refer to our published article.

11.2 Installation Guide

11.2.1 1 System Requirements

Installation Requirements:

- Python >= 3.6 (3.7 is recommended!)

Supported Systems:

- Linux (Ubuntu, . . .)
- macOS
- Windows

11.2.2 2 Preparations before Installation

We **STRONGLY** suggest you to create a Python environment via Anaconda:

```
conda create -n openbox3.7 python=3.7
conda activate openbox3.7
```

Then we recommend you to update your `pip` and `setuptools` as follows:

```
pip install pip setuptools --upgrade
```

11.2.3 3 Install OpenBox

3.1 Installation from PyPI

To install OpenBox from PyPI, simply run the following command:

```
pip install openbox
```

3.2 Manual Installation from Source

To install OpenBox using the source code, please run the following commands:

For Python >= 3.7:

```
git clone https://github.com/PKU-DAIR/open-box.git && cd open-box
cat requirements/main.txt | xargs -n 1 -L 1 pip install
python setup.py install
```

For Python == 3.6:

```
git clone https://github.com/PKU-DAIR/open-box.git && cd open-box
cat requirements/main_py36.txt | xargs -n 1 -L 1 pip install
python setup.py install
```

3.3 Test for Installation

You can run the following code to test your installation:

```
from openbox import run_test

if __name__ == '__main__':
    run_test()
```

If successful, you will receive the following message:

```
===== Congratulations! All trials succeeded. =====
```

If you encountered any problem during installation, please refer to the **Trouble Shooting** section.

11.2.4 4 Installation for Advanced Usage (Optional)

To use advanced features such as `pyrfr` (probabilistic random forest) surrogate and get hyper-parameter importance from history, please refer to Pyrfr Installation Guide to install `pyrfr`.

11.2.5 5 Trouble Shooting

If you encounter problems not listed below, please File an issue on GitHub or email us via *liyang.cs@pku.edu.cn*.

Windows

- 'Error: [WinError 5] Access denied'. Please open the command prompt with administrative privileges or append `--user` to the command line.
- 'ERROR: Failed building wheel for ConfigSpace'. Please refer to tips.
- For Windows users who have trouble installing lazy_import, please refer to tips. (Deprecated in 0.7.10)

macOS

- For macOS users who have trouble installing pyrfr, please refer to tips.
- For macOS users who have trouble building scikit-learn, this documentation might help.

11.3 Quick Start

This tutorial helps you run your first example with **OpenBox**.

11.3.1 Space Definition

First, define a search space.

```
from openbox import sp

# Define Search Space
space = sp.Space()
x1 = sp.Real("x1", -5, 10, default_value=0)
x2 = sp.Real("x2", 0, 15, default_value=0)
space.add_variables([x1, x2])
```

In this example, we create an empty search space, and then add two real (floating-point) variables into it. The first variable **x1** ranges from -5 to 10, and the second one **x2** ranges from 0 to 15.

OpenBox also supports other types of variables. Here are examples of how to define **Integer** and **Categorical** variables:

```
from openbox import sp

i = sp.Int("i", 0, 100)
kernel = sp.Categorical("kernel", ["rbf", "poly", "sigmoid"], default_value="rbf")
```

The **Space** in **OpenBox** is implemented based on **ConfigSpace** package. For advanced usage, please refer to ConfigSpace's documentation.

11.3.2 Objective Definition

Second, define the objective function to be optimized. Note that **OpenBox** aims to **minimize** the objective function. Here we provide an example of the **Branin** function.

```
import numpy as np

# Define Objective Function
def branin(config):
    x1, x2 = config['x1'], config['x2']
    y = (x2-5.1/(4*np.pi**2)*x1**2+5/np.pi*x1-6)**2+10*(1-1/(8*np.pi))*np.cos(x1)+10
    return y
```

The objective function takes as input a configuration sampled from **space** and outputs the objective value.

11.3.3 Optimization

After defining the search space and the objective function, we can run the optimization process as follows:

```
from openbox import Optimizer

# Run
opt = Optimizer(
    branin,
    space,
    max_runs=50,
    surrogate_type='gp',
    time_limit_per_trial=30,
    task_id='quick_start',
)
history = opt.run()
```

Here we create a **Optimizer** instance, and pass the objective function **branin** and the search space **space** to it. The other parameters are:

- **num_objs=1** and **num_constraints=0** indicates our branin function returns a single value with no constraint.
- **max_runs=50** means the optimization will take 50 rounds (optimizing the objective function 50 times).
- **surrogate_type='gp'**. For mathematical problems, we suggest using Gaussian Process (**'gp'**) as Bayesian surrogate model. For practical problems such as hyperparameter optimization (HPO), we suggest using Random Forest (**'prf'**).
- **time_limit_per_trial** sets the time budget (seconds) for each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

Then, **opt.run()** is called to start the optimization process.

11.3.4 Visualization

After the optimization, **opt.run()** returns the optimization history. Call **print(history)** to see the result:

```
print(history)
```

```
+-------------------------+--------------------+
| Parameters              | Optimal Value      |
+-------------------------+--------------------+
| x1                      | -3.138277          |
| x2                      | 12.254526          |
+-------------------------+--------------------+
| Optimal Objective Value | 0.398096578033325  |
+-------------------------+--------------------+
| Num Configs             | 50                 |
+-------------------------+--------------------+
```

Call **history.plot_convergence()** to visualize the optimization process:

```
history.plot_convergence(true_minimum=0.397887)
```

If you are using the Jupyter Notebook environment, call **history.visualize_jupyter()** for visualization of each trial:

```
history.visualize_jupyter()
```

Call **print(history.get_importance())** to print the parameter importance: (Note that you need to install the `pyrfr` package to use this function. Pyrfr Installation Guide)

```
print(history.get_importance())
```

```
+------------+------------+
| Parameters | Importance |
+------------+------------+
| x1         | 0.488244   |
| x2         | 0.327570   |
+------------+------------+
```

11.4 Examples

11.4.1 Single-Objective Black-box Optimization

In this tutorial, we will introduce how to tune hyperparameters of ML tasks with **OpenBox**.

Data Preparation

First, **prepare data** for your ML model. Here we use the digits dataset from sklearn as an example.

```
# prepare your data
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_digits

X, y = load_digits(return_X_y=True)
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,␣
↪random_state=1)
```

Problem Setup

Second, define the **configuration space** to search and the **objective function** to **minimize**. Here, we use LightGBM – a gradient boosting framework developed by Microsoft, as the classification model.

```
from openbox import sp
from sklearn.metrics import balanced_accuracy_score
from lightgbm import LGBMClassifier


def get_configspace():
    space = sp.Space()
    n_estimators = sp.Int("n_estimators", 100, 1000, default_value=500, q=50)
    num_leaves = sp.Int("num_leaves", 31, 2047, default_value=128)
    max_depth = sp.Constant('max_depth', 15)
    learning_rate = sp.Real("learning_rate", 1e-3, 0.3, default_value=0.1, log=True)
    min_child_samples = sp.Int("min_child_samples", 5, 30, default_value=20)
    subsample = sp.Real("subsample", 0.7, 1, default_value=1, q=0.1)
    colsample_bytree = sp.Real("colsample_bytree", 0.7, 1, default_value=1, q=0.1)
    space.add_variables([n_estimators, num_leaves, max_depth, learning_rate, min_child_
↪samples, subsample,
                         colsample_bytree])
    return space


def objective_function(config: sp.Configuration):
    params = config.get_dictionary()
    params['n_jobs'] = 2
    params['random_state'] = 47

    model = LGBMClassifier(**params)
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)
```

(continues on next page)

(continued from previous page)

```
    loss = 1 - balanced_accuracy_score(y_test, y_pred)  # minimize
    return dict(objs=(loss, ))
```

Here are some instructions on how to **define a configuration space**:

- When we define **n_estimators**, we set **q=50**, which means the values of the hyperparameter will be sampled at an interval of 50.
- When we define **learning_rate**, we set **log=True**, which means the values of the hyperparameter will be sampled on a logarithmic scale.

The input of the **objective function** is a **Configuration** instance sampled from the **space**. You can call **config.get_dictionary()** to convert **Configuration** into Python **dict**.

During this hyperparameter optimization task, once a new hyperparameter configuration is suggested, we rebuild the model based on the input configuration. Then, we fit the model, and evaluate the model's predictive performance. These steps are carried out in the objective function.

After evaluation, the objective function returns a **dict (Recommended)**. The result dictionary should contain:

- **'objs'**: A **list/tuple** of **objective values (to be minimized)**. In this example, we have only one objective so the tuple contains a single value.
- **'constraints'**: A **list/tuple** of **constraint values**. If the problem is not constrained, return **None** or do not include this key in the dictionary. Non-positive constraint values (**"<=0"**) imply feasibility.

In addition to returning a dictionary, for single-objective problems with no constraints, returning a single value is also supported.

Optimization

After defining the configuration space and the objective function, we can run the optimization process as follows:

```
from openbox import Optimizer

# Run
opt = Optimizer(
    objective_function,
    get_configspace(),
    num_objs=1,
    num_constraints=0,
    max_runs=100,
    surrogate_type='prf',
    time_limit_per_trial=180,
    task_id='so_hpo',
)
history = opt.run()
```

Here we create a **Optimizer** instance, and pass the objective function and the configuration space to it. The other parameters are:

- **num_objs=1** and **num_constraints=0** indicate that our function returns a single value with no constraint.
- **max_runs=100** means the optimization will take 100 rounds (optimizing the objective function 100 times).

- **surrogate_type='prf'**. For mathematical problem, we suggest using Gaussian Process (**'gp'**) as Bayesian surrogate model. For practical problems such as hyperparameter optimization (HPO), we suggest using Random Forest (**'prf'**).
- **time_limit_per_trial** sets the time budget (seconds) of each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

Then, **opt.run()** is called to start the optimization process.

Visualization

After the optimization, opt.run() returns the optimization history. Or you can call **opt.get_history()** to get the history. Then, call print(history) to see the result:

```
history = opt.get_history()
print(history)
```

```
+-------------------------+----------------------+
| Parameters              | Optimal Value        |
+-------------------------+----------------------+
| colsample_bytree        | 0.800000             |
| learning_rate           | 0.018402             |
| max_depth               | 15                   |
| min_child_samples       | 15                   |
| n_estimators            | 200                  |
| num_leaves              | 723                  |
| subsample               | 0.800000             |
+-------------------------+----------------------+
| Optimal Objective Value | 0.022305877305877297 |
+-------------------------+----------------------+
| Num Configs             | 100                  |
+-------------------------+----------------------+
```

Call **history.plot_convergence()** to visualize the optimization process:

```
history.plot_convergence()
```

If you are using the Jupyter Notebook environment, call history.visualize_jupyter() for visualization of each trial:

```
history.visualize_jupyter()
```

Call **print(history.get_importance())** print the hyperparameter importance: (Note that you need to install the `pyrfr` package to use this function. Pyrfr Installation Guide)

```
print(history.get_importance())
```

```
+-------------------+------------+
| Parameters        | Importance |
+-------------------+------------+
| learning_rate     | 0.293457   |
| min_child_samples | 0.101243   |
| n_estimators      | 0.076895   |
```

(continues on next page)

(continued from previous page)

```
| num_leaves        | 0.069107   |
| colsample_bytree  | 0.051856   |
| subsample         | 0.010067   |
| max_depth         | 0.000000   |
+-------------------+------------+
```

In this task, the top-3 influential hyperparameters are *learning_rate*, *min_child_samples*, and *n_estimators*.

11.4.2 Single-Objective with Constraints

In this tutorial, we will introduce how to optimize a constrained problem with **OpenBox**.

Problem Setup

First, **define search space** and **define objective function** to **minimize**. Here we use the constrained **Mishra** function.

```
import numpy as np
from openbox import sp

def mishra(config: sp.Configuration):
    config_dict = config.get_dictionary()
    X = np.array([config_dict['x%d' % i] for i in range(2)])
    x, y = X[0], X[1]
    t1 = np.sin(y) * np.exp((1 - np.cos(x))**2)
    t2 = np.cos(x) * np.exp((1 - np.sin(y))**2)
    t3 = (x - y)**2

    result = dict()
    result['objs'] = [t1 + t2 + t3, ]
    result['constraints'] = [np.sum((X + 5)**2) - 25, ]
    return result

params = {
    'float': {
        'x0': (-10, 0, -5),
        'x1': (-6.5, 0, -3.25)
    }
}
space = sp.Space()
space.add_variables([
    sp.Real(name, *para) for name, para in params['float'].items()
])
```

After evaluation, the objective function returns a **dict (Recommended)**. The result dictionary should contain:

- **'objs'**: A **list/tuple** of **objective values (to be minimized)**. In this example, we have only one objective so the tuple contains a single value.
- **'constraints'**: A **list/tuple** of **constraint values**. Non-positive constraint values (**"<=0"**) imply feasibility.

Optimization

After defining the search space and the objective function, we can run the optimization process as follows:

```
from openbox import Optimizer

opt = Optimizer(
    mishra,
    space,
    num_constraints=1,
    num_objs=1,
    surrogate_type='gp',
    acq_optimizer_type='random_scipy',
    max_runs=50,
    time_limit_per_trial=10,
    task_id='soc',
)
history = opt.run()
```

Here we create a **Optimizer** instance, and pass the objective function and the search space to it. The other parameters are:

- **num_objs=1** and **num_constraints=1** indicate that our function returns a single value with one constraint.
- **max_runs=50** means the optimization will take 50 rounds (optimizing the objective function 50 times).
- **time_limit_per_trial** sets the time budget (seconds) of each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

Then, **opt.run()** is called to start the optimization process.

Visualization

After the optimization, opt.run() returns the optimization history. Or you can call **opt.get_history()** to get the history. Then, call print(history) to see the result:

```
history = opt.get_history()
print(history)
```

```
+-------------------------+---------------------+
| Parameters              | Optimal Value       |
+-------------------------+---------------------+
| x0                      | -3.172421           |
| x1                      | -1.506397           |
+-------------------------+---------------------+
| Optimal Objective Value | -105.72769850551406 |
+-------------------------+---------------------+
| Num Configs             | 50                  |
+-------------------------+---------------------+
```

Call **history.plot_convergence()** to visualize the optimization process:

```
history.plot_convergence(true_minimum=-106.7645367)
```

11.4.3 Multi-Objective Black-box Optimization

In this tutorial, we will introduce how to optimize multi-objective problems with **OpenBox**.

Problem Setup

We use the multi-objective problem ZDT2 with three input dims in this example. As ZDT2 is a built-in function, its search space and objective function are wrapped as follows:

```
from openbox.benchmark.objective_functions.synthetic import ZDT2

dim = 3
prob = ZDT2(dim=dim)
```

```
import numpy as np
from openbox import sp
params = {'x%d' % i: (0, 1) for i in range(1, dim+1)}
space = sp.Space()
space.add_variables([sp.Real(k, *v) for k, v in params.items()])

def objective_function(config: sp.Configuration):
    X = np.array(list(config.get_dictionary().values()))
    f_0 = X[..., 0]
    g = 1 + 9 * X[..., 1:].mean(axis=-1)
    f_1 = g * (1 - (f_0 / g)**2)

    result = dict()
    result['objs'] = np.stack([f_0, f_1], axis=-1)
    return result
```

After evaluation, the objective function returns a **dict (Recommended)**. The result dictionary should contain:

- **'objs'**: A **list/tuple** of **objective values (to be minimized)**. In this example, we have two objectives so the tuple contains two values.
- **'constraints'**: A **list/tuple** of **constraint values**. If the problem is not constrained, return **None** or do not include this key in the dict. Non-positive constraint values (**"<=0"**) imply feasibility.

Optimization

```
from openbox import Optimizer
opt = Optimizer(
    prob.evaluate,
    prob.config_space,
    num_objs=prob.num_objs,
    num_constraints=0,
    max_runs=50,
    surrogate_type='gp',
    acq_type='ehvi',
    acq_optimizer_type='random_scipy',
    initial_runs=2*(dim+1),
    init_strategy='sobol',
```

(continues on next page)

(continued from previous page)

```
    ref_point=prob.ref_point,
    time_limit_per_trial=10,
    task_id='mo',
    random_state=1,
)
opt.run()
```

Here we create a **Optimizer** instance, and pass the objective function and the search space to it. The other parameters are:

- **num_objs** and **num_constraints** set how many objectives and constraints the objective function will return. In this example, **num_objs=2**.
- **max_runs=50** means the optimization will take 50 rounds (optimizing the objective function 50 times).
- **surrogate_type='gp'**. For mathematical problem, we suggest using Gaussian Process (**'gp'**) as Bayesian surrogate model. For practical problems such as hyperparameter optimization (HPO), we suggest using Random Forest (**'prf'**).
- **acq_type='ehvi'**. Use **EHVI(Expected Hypervolume Improvement)** as Bayesian acquisition function. For problems with more than 3 objectives, please use **MESMO('mesmo')** or **USEMO('usemo')**.
- **acq_optimizer_type='random_scipy'**. For mathematical problems, we suggest using **'random_scipy'** as acquisition function optimizer. For practical problems such as hyperparameter optimization (HPO), we suggest using **'local_random'**.
- **initial_runs** sets how many configurations are suggested by **init_strategy** before the optimization loop.
- **init_strategy='sobol'** sets the strategy to suggest the initial configurations.
- **ref_point** specifies the reference point, which is the upper bound on the objectives used for computing hypervolume. If using EHVI method, a reference point must be provided. In practice, the reference point can be set 1) using domain knowledge to be slightly worse than the upper bound of objective values, where the upper bound is the maximum acceptable value of interest for each objective, or 2) using a dynamic reference point selection strategy.
- **time_limit_per_trial** sets the time budget (seconds) of each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

Then, **opt.run()** is called to start the optimization process.

Visualization

Since we optimize both objectives at the same time, we get a pareto front as the result. Call **opt.get_history().get_pareto_front()** to get the pareto front.

```
import numpy as np
import matplotlib.pyplot as plt

# plot pareto front
pareto_front = np.asarray(opt.get_history().get_pareto_front())
if pareto_front.shape[-1] in (2, 3):
    if pareto_front.shape[-1] == 2:
        plt.scatter(pareto_front[:, 0], pareto_front[:, 1])
        plt.xlabel('Objective 1')
```

(continues on next page)

(continued from previous page)

```
        plt.ylabel('Objective 2')
    elif pareto_front.shape[-1] == 3:
        ax = plt.axes(projection='3d')
        ax.scatter3D(pareto_front[:, 0], pareto_front[:, 1], pareto_front[:, 2])
        ax.set_xlabel('Objective 1')
        ax.set_ylabel('Objective 2')
        ax.set_zlabel('Objective 3')
    plt.title('Pareto Front')
    plt.show()
```

Then plot the hypervolume difference during the optimization compared to the ideal pareto front.

```
# plot hypervolume
hypervolume = opt.get_history().hv_data
log_hv_diff = np.log10(prob.max_hv - np.asarray(hypervolume))
plt.plot(log_hv_diff)
plt.xlabel('Iteration')
plt.ylabel('Log Hypervolume Difference')
plt.show()
```

11.4.4 Multi-Objective with Constraints

In this tutorial, we will introduce how to optimize constrained multiple objectives problem with **OpenBox**.

Problem Setup

We use constrained multi-objective problem CONSTR in this example. As CONSTR is a built-in function, its search space and objective function are wrapped as follows:

```
from openbox.benchmark.objective_functions.synthetic import CONSTR

prob = CONSTR()
dim = 2
initial_runs = 2 * (dim + 1)
```

```
import numpy as np
from openbox import sp
params = {'x1': (0.1, 10.0),
          'x2': (0.0, 5.0)}
space = sp.Space()
space.add_variables([sp.Real(k, *v) for k, v in params.items()])

def objective_funtion(config: sp.Configuration):
    X = np.array(list(config.get_dictionary().values()))

    result = dict()
    obj1 = X[..., 0]
    obj2 = (1.0 + X[..., 1]) / X[..., 0]
    result['objs'] = np.stack([obj1, obj2], axis=-1)
```

(continues on next page)

(continued from previous page)

```
    c1 = 6.0 - 9.0 * X[..., 0] - X[..., 1]
    c2 = 1.0 - 9.0 * X[..., 0] + X[..., 1]
    result['constraints'] = np.stack([c1, c2], axis=-1)

    return result
```

After evaluation, the objective function returns a **dict (Recommended)**. The result dictionary should contain:

- **'objs'**: A **list/tuple** of **objective values (to be minimized)**. In this example, we have two objectives so the tuple contains two values.
- **'constraints'**: A **list/tuple** of **constraint values**. Non-positive constraint values (**"<=0"**) imply feasibility.

Optimization

```
from openbox import Optimizer
opt = Optimizer(
    prob.evaluate,
    prob.config_space,
    num_objs=prob.num_objs,
    num_constraints=prob.num_constraints,
    max_runs=100,
    surrogate_type='gp',
    acq_type='ehvic',
    acq_optimizer_type='random_scipy',
    initial_runs=initial_runs,
    init_strategy='sobol',
    ref_point=prob.ref_point,
    time_limit_per_trial=10,
    task_id='moc',
    random_state=1,
)
opt.run()
```

Here we create a **Optimizer** instance, and pass the objective function and the search space to it. The other parameters are:

- **num_objs** and **num_constraints** set how many objectives and constraints the objective function will return. In this example, **num_objs=2** and **num_constraints=2**.
- **max_runs=100** means the optimization will take 100 rounds (optimizing the objective function 100 times).
- **surrogate_type='gp'**. For mathematical problem, we suggest using Gaussian Process (**'gp'**) as Bayesian surrogate model. For practical problems such as hyperparameter optimization (HPO), we suggest using Random Forest (**'prf'**).
- **acq_type='ehvic'**. Use **EHVIC(Expected Hypervolume Improvement with Constraint)** as Bayesian acquisition function.
- **acq_optimizer_type='random_scipy'**. For mathematical problems, we suggest using **'random_scipy'** as acquisition function optimizer. For practical problems such as hyperparameter optimization (HPO), we suggest using **'local_random'**.
- **initial_runs** sets how many configurations are suggested by **init_strategy** before the optimization loop.
- **init_strategy='sobol'** sets the strategy to suggest the initial configurations.

- **ref_point** specifies the reference point, which is the upper bound on the objectives used for computing hypervolume. If using EHVI method, a reference point must be provided. In practice, the reference point can be set 1) using domain knowledge to be slightly worse than the upper bound of objective values, where the upper bound is the maximum acceptable value of interest for each objective, or 2) using a dynamic reference point selection strategy.
- **time_limit_per_trial** sets the time budget (seconds) of each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

Then, **opt.run()** is called to start the optimization process.

Visualization

Since we optimize both objectives at the same time, we get a pareto front as the result. Call **opt.get_history().get_pareto_front()** to get the pareto front.

```
import numpy as np
import matplotlib.pyplot as plt
# plot pareto front
pareto_front = np.asarray(opt.get_history().get_pareto_front())
if pareto_front.shape[-1] in (2, 3):
    if pareto_front.shape[-1] == 2:
        plt.scatter(pareto_front[:, 0], pareto_front[:, 1])
        plt.xlabel('Objective 1')
        plt.ylabel('Objective 2')
    elif pareto_front.shape[-1] == 3:
        ax = plt.axes(projection='3d')
        ax.scatter3D(pareto_front[:, 0], pareto_front[:, 1], pareto_front[:, 2])
        ax.set_xlabel('Objective 1')
        ax.set_ylabel('Objective 2')
        ax.set_zlabel('Objective 3')
    plt.title('Pareto Front')
    plt.show()
```

Then plot the hypervolume difference during the optimization compared to the ideal pareto front.

```
# plot hypervolume
hypervolume = opt.get_history().hv_data
max_hv = 92.02004226679216
log_hv_diff = np.log10(max_hv - np.asarray(hypervolume))
plt.plot(log_hv_diff)
plt.xlabel('Iteration')
plt.ylabel('Log Hypervolume Difference')
plt.show()
```

11.5 Advanced Usage

11.5.1 Parallel and Distributed Evaluation

Most proposed Bayesian optimization (BO) approaches only allow the exploration of the search space to occur sequentially. To fully utilize computing resources in a parallel infrastructure, **OpenBox** provides a mechanism for distributed parallelization, where multiple configurations can be evaluated concurrently across workers.

Two parallel settings are considered:

1. **Synchronous parallel setting (left)**. The worker pulls a new configuration from the suggestion server to evaluate until all the workers have finished their last evaluations.
2. **Asynchronous parallel setting (right)**. The worker pulls a new configuration when the previous evaluation is completed.

OpenBox proposes a local penalization based parallelization mechanism, the goal of which is to sample new configurations that are promising and far enough from the configurations being evaluated by other workers. This mechanism can handle the well-celebrated exploration vs. exploitation trade-off, and meanwhile prevent workers from exploring similar configurations.

In this tutorial, we illustrate how to optimize a problem in parallel manner on your local machine with **OpenBox**.

Problem Setup

First, **define configuration space** to search and **define objective function** to **minimize**. Here we use the **Branin** function.

```
import numpy as np
from openbox import sp

# Define Search Space
space = sp.Space()
x1 = sp.Real("x1", -5, 10, default_value=0)
x2 = sp.Real("x2", 0, 15, default_value=0)
space.add_variables([x1, x2])


# Define Objective Function
def branin(config):
    x1, x2 = config['x1'], config['x2']
    y = (x2 - 5.1 / (4 * np.pi ** 2) * x1 ** 2 + 5 / np.pi * x1 - 6) ** 2 \
        + 10 * (1 - 1 / (8 * np.pi)) * np.cos(x1) + 10
    return {'objs': (y,)}
```

If you are not familiar with the problem setup, please refer to *Quick Start Tutorial*.

Parallel Evaluation on Local

This time we use **ParallelOptimizer** to optimize the objective function in a parallel manner on your local machine.

```
from openbox import ParallelOptimizer

# Parallel Evaluation on Local Machine
opt = ParallelOptimizer(
    branin,
    space,
    parallel_strategy='async',
    batch_size=4,
    batch_strategy='default',
    num_objs=1,
    num_constraints=0,
    max_runs=50,
    surrogate_type='gp',
    time_limit_per_trial=180,
    task_id='parallel_sync',
)
history = opt.run()
```

In addition to **objective_function** and **space** being passed to **ParallelOptimizer**, the other parameters are as follows:

- **parallel_strategy='async' / 'sync'** sets whether the parallel evaluation is performed asynchronously or synchronously. We suggest using **'async'** because it makes better use of resources and achieves better performance than **'sync'**.
- **batch_size=4** sets the number of parallel workers.
- **batch_strategy='default'** sets the strategy on how to make multiple suggestions at the same time. We suggest using **'default'** for stable performance.
- **num_objs=1** and **num_constraints=0** indicates that our function returns a single objective value with no constraint.
- **max_runs=100** means the optimization will take 100 rounds (optimizing the objective function 100 times).
- **surrogate_type='gp'**. For mathematical problem, we suggest using Gaussian Process (**'gp'**) as Bayesian surrogate model. For practical problems such as hyperparameter optimization (HPO), we suggest using Random Forest (**'prf'**).
- **time_limit_per_trial** sets the time budget (seconds) of each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

After optimization, call **print(opt.get_history())** to see the result:

```
print(opt.get_history())
```

```
+-----------------------------------------------+
| Parameters              | Optimal Value       |
+-------------------------+---------------------+
| x1                      | -3.138286           |
| x2                      | 12.292733           |
+-------------------------+---------------------+
| Optimal Objective Value | 0.3985991718620365 |
```

(continues on next page)

(continued from previous page)

```
+-------------------------+--------------------+
| Num Configs             | 100                |
+-------------------------+--------------------+
```

Distributed Evaluation

OpenBox provides an efficient way to perform distributed optimization.

First, start the master node with the **DistributedOptimizer**. We use the branin objective function defined previously.

```
from openbox import DistributedOptimizer

# Distributed Evaluation
n_workers = 4
opt = DistributedOptimizer(
    branin,
    space,
    parallel_strategy='async',
    batch_size=n_workers,
    batch_strategy='default',
    num_objs=1,
    num_constraints=0,
    max_runs=50,
    surrogate_type='gp',
    time_limit_per_trial=180,
    task_id='distributed_opt',
    port=13579,
    authkey=b'abc',
)
history = opt.run()
```

In addition to **objective_function** and **space** being passed to **DistributedOptimizer**, the other parameters are as follows:

- **port**: network port of optimizer on master node.
- **authkey**: authorization key for worker to connect the master.
- **parallel_strategy='async' / 'sync'** sets whether the parallel evaluation is performed asynchronously or synchronously. We suggest using **'async'** because it makes better use of resources and achieves better performance than **'sync'**.
- **batch_size=4** sets the number of parallel workers.
- **batch_strategy='default'** sets the strategy on how to make multiple suggestions at the same time. We suggest using **'default'** for stable performance.
- **num_objs=1** and **num_constraints=0** indicates that our function returns a single objective value with no constraint.
- **max_runs=100** means the optimization will take 100 rounds (optimizing the objective function 100 times).
- **surrogate_type='gp'**. For mathematical problem, we suggest using Gaussian Process (**'gp'**) as Bayesian surrogate model. For practical problems such as hyperparameter optimization (HPO), we suggest using Random Forest (**'prf'**).

- **time_limit_per_trial** sets the time budget (seconds) of each objective function evaluation. Once the evaluation time exceeds this limit, objective function will return as a failed trial.
- **task_id** is set to identify the optimization process.

Next, start the worker nodes to receive jobs from master and evaluate configurations. In addition to the objective function, please specify **ip** of master node and **port**, **authkey** you set when starting the optimizer.

```
from openbox import DistributedWorker

worker = DistributedWorker(branin, ip="127.0.0.1", port=13579, authkey=b'abc')
worker.run()
```

After optimization, call **print(opt.get_history())** on master node to see the result:

```
print(opt.get_history())
```

```
+-----------------------------------------------+
| Parameters               | Optimal Value      |
+--------------------------+--------------------+
| x1                       | -3.138286          |
| x2                       | 12.292733          |
+--------------------------+--------------------+
| Optimal Objective Value | 0.3985991718620365 |
+--------------------------+--------------------+
| Num Configs              | 100                |
+--------------------------+--------------------+
```

11.5.2 Transfer Learning

When performing BBO, users often run tasks that are similar to previous ones. This observation can be used to speed up the current task. Compared with Vizier, which only provides limited transfer learning functionality for single-objective BBO problems, OpenBox employs a general transfer learning framework with the following advantages:

1. Support for generalized black-box optimization problems;
2. Compatibility with most Bayesian optimization methods.

OpenBox takes as input observations from + 1 tasks: D1, . . . , D for previous tasks and D for the current task. Each D = {(,)}, = 1, . . .,, includes a set of observations. Note that, is an array, including multiple objectives for configuration . For multi-objective problems with objectives, we propose to transfer the knowledge about objectives individually. Thus, the transfer learning of multiple objectives is turned into single-objective transfer learning processes. For each dimension of the objectives, we take the following transfer-learning technique:

1. We first train a surrogate model on for the -th prior task and on ;
2. Based on 1: and , we then build a transfer learning surrogate by combining all base surrogates: TL = agg({1, . . .,, };w);
3. The surrogate TL is used to guide the configuration search, instead of the original .

Concretely, we use gPoE to combine the multiple base surrogates (agg), and the parameters w are calculated based on the ranking of configurations, which reflects the similarity between the source tasks and the target task.

Performance Comparison

We compare OpenBox with a competitive transfer learning baseline Vizier and a non-transfer baseline SMAC3. The average performance rank (the lower, the better) of each algorithm is shown in the following figure. For experimental setups, dataset information and more experimental results, please refer to our published article.

11.6 OpenBox as Service

11.6.1 Introduction of OpenBox as Service

The design of **OpenBox** follows the paradigm of providing "BBO as a service".

The system architecture of **OpenBox** includes five main components:

- **Service Master** is responsible for node management, load balance, and fault tolerance.
- **Task Database** holds the history and states of all tasks.
- **Suggestion Server** generates new configurations for each task.
- **REST API** connects users/workers and suggestion service via RESTful APIs.
- **Evaluation workers** are provided and owned by the users.

Parallel Infrastructure

OpenBox is designed to generate suggestions for a large number of tasks concurrently, and a single machine would be insufficient to handle the workload. Our suggestion service is therefore deployed across several machines, called **suggestion servers**. Each **suggestion server** generates suggestions for several tasks in parallel, giving us a massively scalable suggestion infrastructure. Another main component is **service master**, which is responsible for managing the **suggestion servers** and balancing the workload. It serves as the unified endpoint, and accepts the requests from workers; in this way, each worker does not need to know the dispatching details. The worker requests new configurations from the **suggestion server** and the **suggestion server** generates these configurations based on an algorithm determined by the automatic algorithm selection module. Concretely, in this process, the **suggestion server** utilizes the local penalization based parallelization mechanism and transfer learning framework to improve the sample efficiency.

One main design consideration is to maintain a fault-tolerant production system, as machine crash happens inevitably. In **OpenBox**, the **service master** monitors the status of each server and preserves a table of active servers. When a new task comes, the **service master** will assign it to an active server and record this binding information. If one server is down, its tasks will be dispatched to a new server by the master, along with the related optimization history stored in the **task database**. Load balance is one of the most important guidelines to make such task assignments. In addition, the snapshot of **service master** is stored in the remote database service; if the master is down, we can recover it by restarting the node and fetching the snapshot from the database.

Service Interfaces

Task Description Language

For ease of usage, we design a Task Description Language (TDL) to define the optimization task. The essential part of TDL is to define the search space, which includes the type and bound for each parameter and the relationships among them. The parameter types — FLOAT, INTEGER, ORDINAL and CATEGORICAL are supported in **OpenBox**. In addition, users can add conditions of the parameters to further restrict the search space. Users can also specify the time budget, task type, number of workers, parallel strategy and use of history in TDL.

```
task_config = {
    "parameter": {
        "x1": {"type": "float", "default": 0,
            "bound": [-5, 10]} ,
        "x2": {"type": "int", "bound": [0, 15]} ,
        "x3": {"type": "cat", "default": "a1",
            "choice": ["a1", "a2", "a3"]} ,
        "x4": {"type": "ord", "default": 1,
            "choice": [1, 2, 3]}} ,
    "condition": {
        "cdn1": {"type": "equal", "parent": "x3",
            "child": "x1", "value": "a3"}} ,
    "number_of_trials": 200 ,
    "time_budget": 10800 ,
    "task_type": "soc",
    "parallel_strategy": "async",
    "worker_num": 10,
    "use_history": True
    }
```

Here's an example of TDL. It defines four parameters *x1-4* of different types and a condition *cdn1*, which indicates that *x1* is active only if *x3* = *"a3"*. The time budget is three hours, the parallel strategy is *async*, and transfer learning is enabled.

BasicWorkflow

Given the TDL for a task, the basic workflow of **OpenBox** is implemented as follows:

```
# Register the worker with a task .
global_task_id = worker.CreateTask(task_tdl)
worker.BindTask(global_task_id)
while not worker.TaskFinished():
    # Obtain a configuration to evaluate.
    config = worker.GetSuggestions()
    # Evaluate the objective function.
    result = Evaluate(config)
    # Report the evaluated results to the server.
    worker.UpdateObservations(config, result)
```

Here **Evaluate** is the evaluation procedure of objective function provided by users. By calling **CreateTask**, the worker obtains a globally unique identifier **global_task_id**. All workers registered with the same **global_task_id** are guaranteed to link with the same task, which enables parallel evaluations. While the task is not finished, the worker continues to call **GetSuggestions** and **UpdateObservations** to pull suggestions from the suggestion service and update their corresponding observations.

Interfaces

Users can interact with the **OpenBox** service via a **REST API**. We list the most important service calls as follows:

- **Register**: It takes as input the global_task_id, which is created when calling CreateTask from workers, and binds the current worker with the corresponding task. This allows for sharing the optimization history across multiple workers.
- **Suggest**: It suggests the next configurations to evaluate, given the historical observations of the current task.
- **Update**: This method updates the optimization history with the observations obtained from workers. The observations include three parts: the values of the objectives, the results of constraints, and the evaluation information.
- **StopEarly**: It returns a boolean value that indicates whether the current evaluation should be stopped early.
- **Extrapolate**: It uses performance-resource extrapolation, and interactively gives resource-aware advice to users.

11.6.2 OpenBox Service Deployment

This tutorial helps you deploy an **OpenBox** service. If you are an **OpenBox** service user, please refer to the Service User Tutorial.

1 Install OpenBox from Source

Installation Requirements:

- Python >= 3.6
- SWIG == 3.0

To install SWIG, please refer to SWIG Installation Guide

Make sure that SWIG is installed correctly installing OpenBox.

Then, clone the source code to the server where you want to deploy OpenBox service. The commands are as follows:

```
git clone https://github.com/PKU-DAIR/open-box.git
cd open-box
python setup.py install
```

2 Initialize MongoDB

OpenBox uses MongoDB to store user and task information.

2.1 Install and Run MongoDB

Please install and run MongoDB before running **OpenBox** service. For MongoDB installation guides, refer to the following links:

- https://docs.mongodb.com/guides/server/install/
- https://docs.mongodb.com/manual/installation/

You need to create a MongoDB user and set **auth=true** when starting MongoDB. Please record the **IP** and **port** of your database.

2.2 Modify service.conf File

After starting MongoDB, modify **"open-box/conf/service.conf"** to set database information. If this is your first time running the service, create the **service.conf** file by copying the template config file from **"open-box/conf/template/service.conf.template"** to **"open-box/conf/"** and rename it to **service.conf**.

The contents of **service.conf** are as follows:

```
[database]
database_address=127.0.0.1
database_port=27017
user=xxxx
password=xxxx
```

Please set database IP, port, user and password accordingly.

Caution: We have added **service.conf** to **.gitignore**. Do not push this file to **Github** to prevent the disclosure of private information.

3 Set up Email Registration Service

3.1 Prepare an Email for Registration Service

OpenBox requires an email address to send activation link when users register new accounts. Please enable SMTP authentication and then you may receive a secret key for authentication from the email provider.

3.2 Modify openbox/artifact/artifact/settings.py

Then, modify **"openbox/artifact/artifact/settings.py"** to set email information for registration service. Please fill in the following lines:

```
EMAIL_HOST = 'smtp.xxxx.com'
EMAIL_PORT = 465
EMAIL_HOST_USER = 'xxxx@xxxx.com'
EMAIL_HOST_PASSWORD = 'xxxx'
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
EMAIL_ACTIVE_ENABLE = False
```

- **EMAIL_HOST:** SMTP host of email registration service provider. E.g., 'smtp.gmail.com'.
- **EMAIL_PORT:** SMTP port of email registration service provider. Get the port from email service provider. You may try port 25,587 or other ports if port 465 doesn't work.
- **EMAIL_HOST_USER:** Your email address for registration service.
- **EMAIL_HOST_PASSWORD:** Your secret key for SMTP authentication.
- **EMAIL_ACTIVE_ENABLE:** Enable or disable the mail activation function.

Caution: Do not push the file with private information to **Github** to prevent leakage.

4 Migrate Database

```
cd <path to the source code>/open-box
./scripts/manage_service.sh migrate
```

5 Start/Stop OpenBox Service

Finally, after setting up the database and registration service, you can start up the **OpenBox** service.

To **start the service**, run the **manage_service.sh script** by the following commands:

```
cd <path to the source code>/open-box
./scripts/manage_service.sh start
```

The script will run **OpenBox** service in the background. The default service port is 11425. You can modify the script to change service port.

Then, visit http://127.0.0.1:11425/user_board/index/ (replace "127.0.0.1:11425" with your server ip:port) to see whether your service starts successfully. You may also try to create an account and run a task to test your **OpenBox** service. For more detailed guidance, please refer to the Service User Tutorial.

To **stop the service**, run the **manage_service.sh script** by the following commands:

```
cd <path to the source code>/open-box
./scripts/manage_service.sh stop
```

11.6.3 OpenBox Service Tutorial

In this tutorial, we will introduce how to use the remote **OpenBox** service.

Register an Account

Visit http://127.0.0.1:11425/user_board/index/ (replace "127.0.0.1:11425" with server ip:port) and you will see the homepage of **OpenBox** service. Register an account by email to use the service.

You need to activate your account by clicking on the link in the activation email.

Submit a Task

Here is an example of how to use **RemoteAdvisor** to interact with the **OpenBox** service.

```
import datetime
import time
import hashlib
import numpy as np

from openbox.artifact.remote_advisor import RemoteAdvisor
from openbox.utils.constants import SUCCESS, FAILED, TIMEOUT, MEMOUT
from openbox.utils.config_space import Configuration, ConfigurationSpace,␣
↪UniformFloatHyperparameter
```

(continues on next page)

(continued from previous page)

```
def townsend(config):
    X = np.array(list(config.get_dictionary().values()))
    res = dict()
    res['objs'] = [-(np.cos((X[0]-0.1)*X[1])**2 + X[0] * np.sin(3*X[0]+X[1]))]
    res['constraints'] = [-(-np.cos(1.5*X[0]+np.pi)*np.cos(1.5*X[1])+np.sin(1.5*X[0]+np.
↪pi)*np.sin(1.5*X[1]))]
    return res


# Send task id and config space at register
task_id = time.time()
townsend_params = {
    'float': {
        'x1': (-2.25, 2.5, 0),
        'x2': (-2.5, 1.75, 0)
    }
}
townsend_cs = ConfigurationSpace()
townsend_cs.add_hyperparameters([UniformFloatHyperparameter(e, *townsend_params['float
↪'][e])
                                 for e in townsend_params['float']])

password = 'your_password'
md5 = hashlib.md5()
md5.update(password.encode('utf-8'))
max_runs = 50
# Create remote advisor
config_advisor = RemoteAdvisor(config_space=townsend_cs,
                               server_ip='127.0.0.1',
                               port=11425,
                               email='your_email@xxxx.com',
                               password=md5.hexdigest(),
                               num_constraints=1,
                               max_runs=max_runs,
                               task_name="task_test",
                               task_id=task_id)

# Simulate max_runs iterations
for idx in range(max_runs):

    config_dict = config_advisor.get_suggestion()
    config = Configuration(config_advisor.config_space, config_dict)
    print('Get %d config: %s' % (idx+1, config))
    trial_info = {}
    start_time = datetime.datetime.now()
    obs = townsend(config)

    trial_info['cost'] = (datetime.datetime.now() - start_time).seconds
    trial_info['worker_id'] = 0
    trial_info['trial_info'] = 'None'
    print('Result %d is %s. Update observation to server.' % (idx+1, obs))
```

(continues on next page)

(continued from previous page)

```
    config_advisor.update_observation(config_dict, obs['objs'], obs['constraints'],
                                      trial_info=trial_info, trial_state=SUCCESS)

incumbents, history = config_advisor.get_result()
print(incumbents)
```

- Remember to set **server_ip, port** of the service and **email, password** of your account when creating **RemoteAdvisor**. A task is then registered to the service.
- Once you create a task, you can get configuration suggestions from the service by calling **RemoteAdvisor.get_suggestion()**.
- Run your job locally and send results back to the service by calling **RemoteAdvisor.update_observation()**.
- Repeat **get_suggestion** and **update_observation** to complete the optimization.

If you are not familiar with setting up a problem, please refer to *Quick Start Tutorial*.

Monitor a task on the Web Page

You can always monitor your task and watch the optimization results on **OpenBox** service web page.

Visit http://127.0.0.1:11425/user_board/index/ (replace "127.0.0.1:11425" by server ip:port) and login your account.

You will find all the tasks you created. Click the buttons to further observe the results and manage your tasks.

11.7 Research and Publications

11.7.1 Related Research

1. **OpenBox: A Generalized Black-box Optimization Service**; Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, Ce Zhang, Bin Cui; ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2021).
2. **MFES-HB: Efficient Hyperband with Multi-Fidelity Quality Measurements**; Yang Li, Shen Yu, Jiawei Jiang, Jinyang Gao, Ce Zhang, Bin Cui; The Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021).

11.8 Change Logs

Coming Soon!