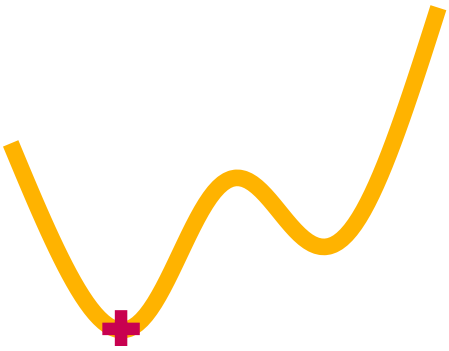


BayesO Documentation

Release 0.4.3 alpha

Jungtaek Kim and Seungjin Choi

Dec 03, 2020

Contents

BayesO (pronounced "bayes-o") is a simple, but essential Bayesian optimization package, written in Python. It is developed by machine learning group at POSTECH. This project is licensed under the MIT license.

This documentation describes the details of implementation, getting started guides, some examples with BayesO, and Python API specifications. The code can be found in our GitHub repository.

CHAPTER 1

About BayesO

Simple, but essential Bayesian optimization package. It is designed to run advanced Bayesian optimization with implementation-specific and application-specific modifications as well as to run Bayesian optimization in various applications simply. This package contains the codes for Gaussian process regression and Gaussian process-based Bayesian optimization. Some famous benchmark and custom benchmark functions for Bayesian optimization are included in bayeso-benchmarks, which can be used to test the Bayesian optimization strategy. If you are interested in this package, please refer to that repository.

1.1 Supported Python Version

We test our package in the following versions.

- Python 3.6
- Python 3.7
- Python 3.8

1.2 Related Package for Benchmark Functions

The related package **bayeso-benchmarks**, which contains some famous benchmark functions and custom benchmark functions is hosted in this repository. It can be used to test a Bayesian optimization strategy.

The details of benchmark functions implemented in **bayeso-benchmarks** are described in these notes.

1.3 Contributor

- Jungtaek Kim (POSTECH)

1.4 Citation

```
@misc{KimJ2017bayeso,
    author={Kim, Jungtaek and Choi, Seungjin},
    title={{BayesO}: A {Bayesian} optimization framework in {Python}},
    howpublished={\url{http://bayeso.org}},
    year={2017}
}
```

1.5 Contact

- Jungtaek Kim: jtkim@postech.ac.kr

1.6 License

CHAPTER 2

About Bayesian Optimization

Bayesian optimization is a **global optimization** strategy for **black-box** and **expensive-to-evaluate** functions. Generic Bayesian optimization follows these steps:

1. Build a **surrogate function** with historical inputs and their observations.
2. Compute and maximize an **acquisition function**, defined by the outputs of surrogate function.
3. Observe the **maximizer** of acquisition function from a true objective function.
4. Accumulate the maximizer and its observation.

This project helps us to execute this Bayesian optimization procedure. In particular, **Gaussian process regression** is used as a surrogate function, and various acquisition functions such as **probability improvement**, **expected improvement**, and **Gaussian process upper confidence bound** are included in this project.

CHAPTER 3

Installing BayesO

We recommend it should be installed in **virtualenv**. You can choose one of three installation options.

3.1 Installing from PyPI

It is for user installation. To install the released version from PyPI repository, command it.

```
$ pip install bayeso
```

3.2 Compiling from Source

It is for developer installation. To install **bayeso** from source code, command

```
$ pip install .
```

in the **bayeso** root.

3.3 Compiling from Source (Editable)

It is for editable development mode. To use editable development mode, command

```
$ pip install -r requirements.txt
$ python setup.py develop
```

in the **bayeso** root.

3.4 Uninstalling

If you would like to uninstall **bayeso**, command it.

```
$ pip uninstall bayeso
```

3.5 Required Packages

Mandatory pacakges are inlcuded in **requirements.txt**. The following **requirements** files include the package list, the purpose of which is described as follows.

- **requirements-optional.txt**: It is an optional package list, but it needs to be installed to execute some features of **bayeso**.
- **requirements-dev.txt**: It is for developing the **bayeso** package.
- **requirements-examples.txt**: It needs to be installed to execute the examples included in the **bayeso** repository.

CHAPTER 4

Building Gaussian Process Regression

This example is for building Gaussian process regression models. First of all, import the packages we need and **bayeso**.

```
import numpy as np

from bayeso import covariance
from bayeso.gp import gp
from bayeso.utils import utils_covariance
from bayeso.utils import utils_plotting
```

Declare some parameters to control this example.

```
use_tex = False
num_test = 200
str_cov = 'matern52'
```

Make a simple synthetic dataset, which produces with cosine functions.

```
X_train = np.array([
    [-3.0],
    [-2.0],
    [-1.0],
    [2.0],
    [1.2],
    [1.1],
])
Y_train = np.cos(X_train) + 10.0
X_test = np.linspace(-3, 3, num_test)
X_test = X_test.reshape((num_test, 1))
Y_test = np.cos(X_test) + 10.0
```

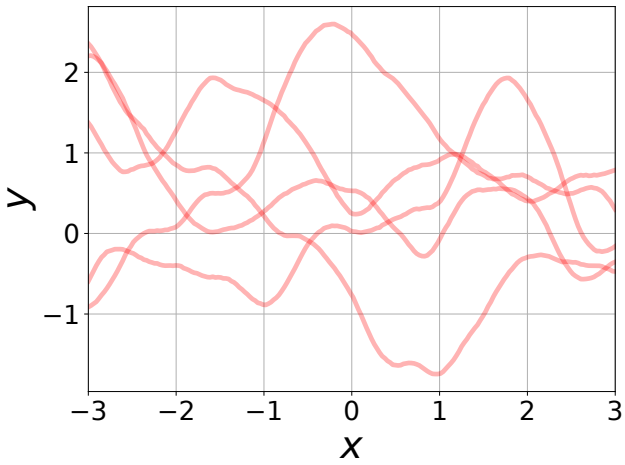
Sample functions from a prior distribution, which is zero mean.

```
mu = np.zeros(num_test)
hyps = utils_covariance.get_hyps(str_cov, 1)
```

(continues on next page)

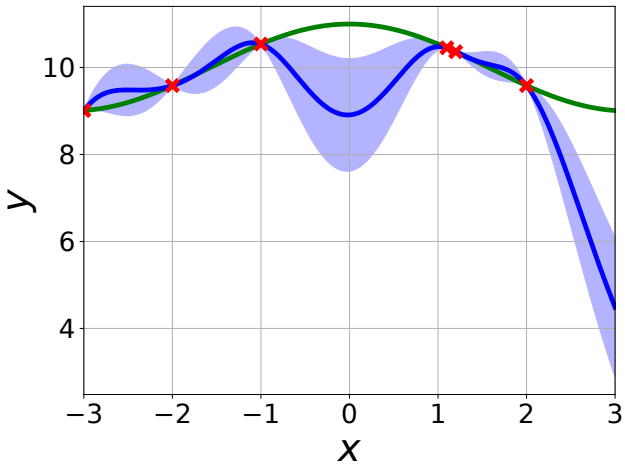
(continued from previous page)

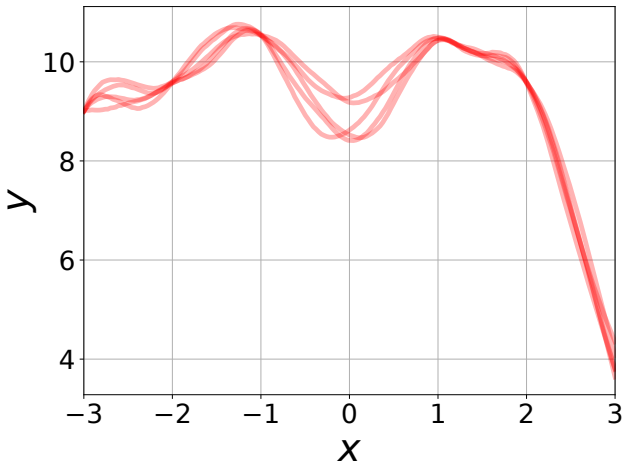
```
Sigma = covariance.cov_main(str_cov, X_test, X_test, hyps, True)

Ys = gp.sample_functions(mu, Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')
```



Build a Gaussian process regression model with fixed hyperparameters. Then, plot the result.

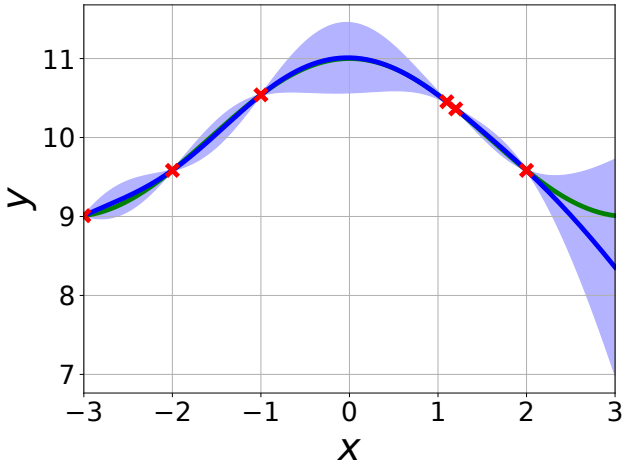
```
hyps = utils_covariance.get_hyps(str_cov, 1)
mu, sigma, Sigma = gp.predict_with_hyps(X_train, Y_train, X_test, hyps, str_cov=str_
↪cov)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')
```

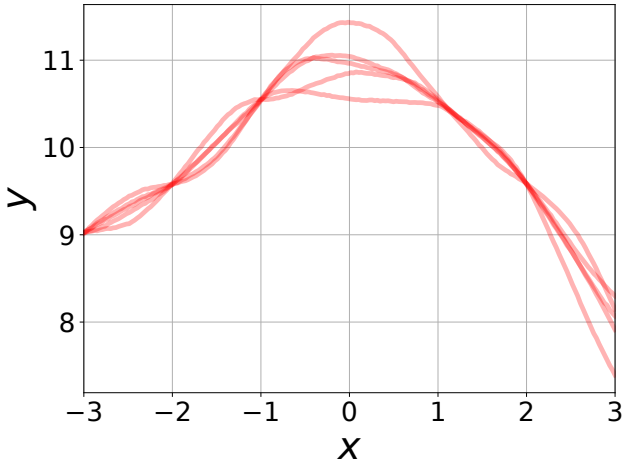




Build a Gaussian process regression model with the hyperparameters optimized by marginal likelihood maximization, and plot the result.

```
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test, str_
↪cov=str_cov)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')
```





Declare some functions that would be employed as prior functions.

```
def cosine(X):
    return np.cos(X)

def linear_down(X):
    list_up = []
    for elem_X in X:
        list_up.append([-0.5 * np.sum(elem_X)])
    return np.array(list_up)

def linear_up(X):
    list_up = []
    for elem_X in X:
        list_up.append([0.5 * np.sum(elem_X)])
    return np.array(list_up)
```

Make an another synthetic dataset using a cosine function.

```
X_train = np.array([
    [-3.0],
    [-2.0],
    [-1.0],
])
Y_train = np.cos(X_train) + 2.0
X_test = np.linspace(-3, 6, num_test)
X_test = X_test.reshape((num_test, 1))
Y_test = np.cos(X_test) + 2.0
```

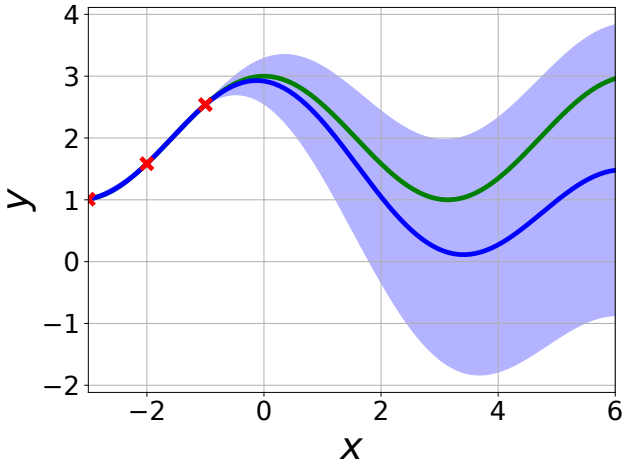
Build Gaussian process regression models with the prior functions we declare above and the hyperparameters optimized by marginal likelihood maximization, and plot the result.

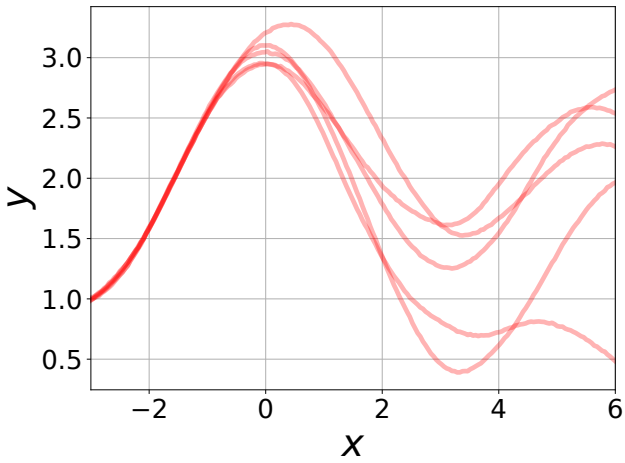
```
def cosine(X):
    return np.cos(X)

def linear_down(X):
    list_up = []
    for elem_X in X:
        list_up.append([-0.5 * np.sum(elem_X)])
    return np.array(list_up)
```

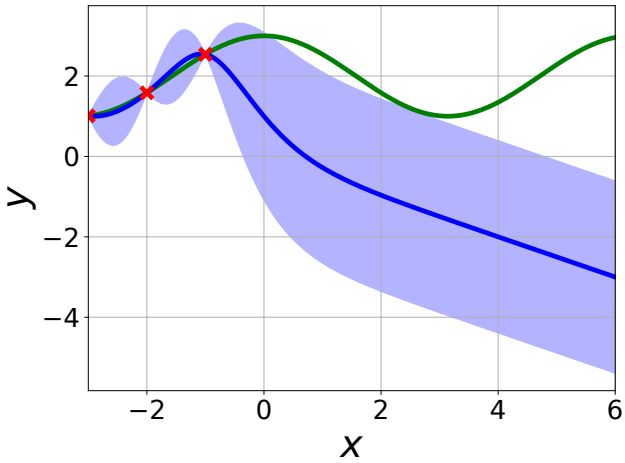
(continues on next page)

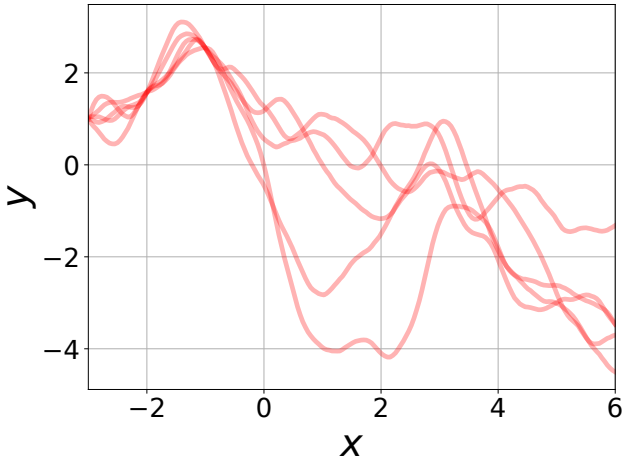
(continued from previous page)

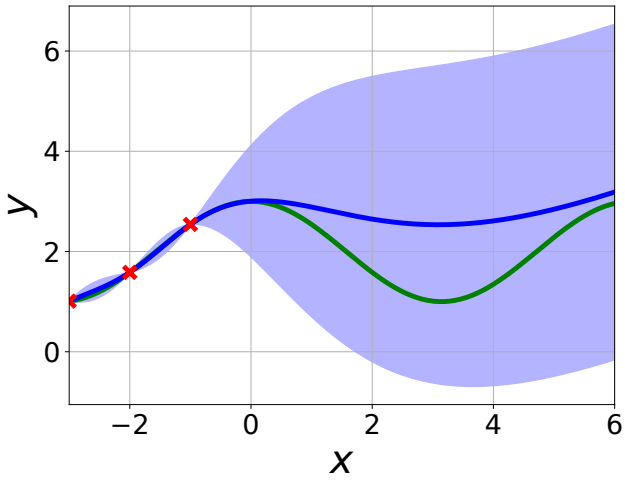
```
def linear_up(X):
    list_up = []
    for elem_X in X:
        list_up.append([0.5 * np.sum(elem_X)])
    return np.array(list_up)

X_train = np.array([
    [-3.0],
    [-2.0],
    [-1.0],
])
Y_train = np.cos(X_train) + 2.0
X_test = np.linspace(-3, 6, num_test)
X_test = X_test.reshape((num_test, 1))
Y_test = np.cos(X_test) + 2.0

prior_mu = cosine
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test,
                                                  str_cov=str_cov, prior_mu=prior_mu)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

prior_mu = linear_down
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test,
                                                  str_cov=str_cov, prior_mu=prior_mu)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

prior_mu = linear_up
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test,
                                                  str_cov=str_cov, prior_mu=prior_mu)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')
```

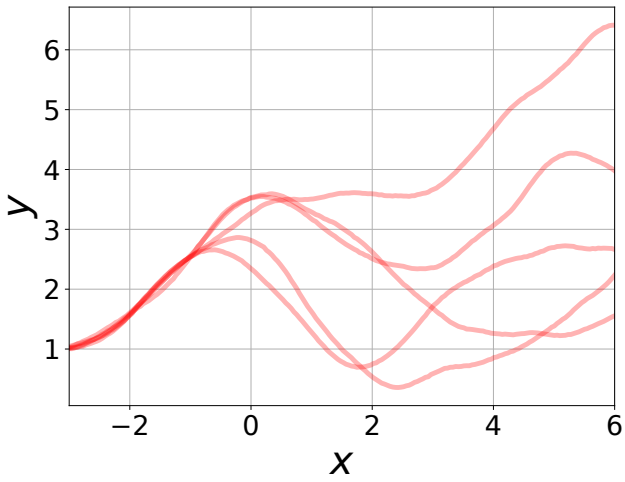

4
3
2
1
0
−1
−2
y
−2
0
2
4
6
x


3.0
2.5
2.0
1.5
1.0
0.5
y
−2
0
2
4
6
x


2
0
−2
−4
y
−2
0
2
4
6
x







Full code:

```
import numpy as np

from bayeso import covariance
from bayeso.gp import gp
```

(continues on next page)

(continued from previous page)

```
from bayeso.utils import utils_covariance
from bayeso.utils import utils_plotting

use_tex = False
num_test = 200
str_cov = 'matern52'

X_train = np.array([
    [-3.0],
    [-2.0],
    [-1.0],
    [2.0],
    [1.2],
    [1.1],
])
Y_train = np.cos(X_train) + 10.0
X_test = np.linspace(-3, 3, num_test)
X_test = X_test.reshape((num_test, 1))
Y_test = np.cos(X_test) + 10.0

mu = np.zeros(num_test)
hyps = utils_covariance.get_hyps(str_cov, 1)
Sigma = covariance.cov_main(str_cov, X_test, X_test, hyps, True)

Ys = gp.sample_functions(mu, Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

hyps = utils_covariance.get_hyps(str_cov, 1)
mu, sigma, Sigma = gp.predict_with_hyps(X_train, Y_train, X_test, hyps, str_cov=str_
↪cov)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test, str_
↪cov=str_cov)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

def cosine(X):
    return np.cos(X)

def linear_down(X):
```

(continues on next page)

(continued from previous page)

```
    list_up = []
    for elem_X in X:
        list_up.append([-0.5 * np.sum(elem_X)])
    return np.array(list_up)

def linear_up(X):
    list_up = []
    for elem_X in X:
        list_up.append([0.5 * np.sum(elem_X)])
    return np.array(list_up)

X_train = np.array([
    [-3.0],
    [-2.0],
    [-1.0],
])
Y_train = np.cos(X_train) + 2.0
X_test = np.linspace(-3, 6, num_test)
X_test = X_test.reshape((num_test, 1))
Y_test = np.cos(X_test) + 2.0

prior_mu = cosine
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test,
                                                  str_cov=str_cov, prior_mu=prior_mu)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

prior_mu = linear_down
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test,
                                                  str_cov=str_cov, prior_mu=prior_mu)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')

prior_mu = linear_up
mu, sigma, Sigma = gp.predict_with_optimized_hyps(X_train, Y_train, X_test,
                                                  str_cov=str_cov, prior_mu=prior_mu)
utils_plotting.plot_gp_via_distribution(
    X_train, Y_train, X_test, mu, sigma,
    Y_test=Y_test, use_tex=use_tex,
    str_x_axis='$x$', str_y_axis='$y$'
)

Ys = gp.sample_functions(mu.flatten(), Sigma, num_samples=5)
```

(continues on next page)

(continued from previous page)

```
utils_plotting.plot_gp_via_sample(X_test, Ys, use_tex=use_tex,
                                  str_x_axis='$x$', str_y_axis='$y$')
```

CHAPTER 5

Optimizing Sampled Function via Thompson Sampling

This example is to optimize a function sampled from a Gaussian process prior via Thompson sampling. First of all, import the packages we need and **bayeso**.

```
import numpy as np

from bayeso import covariance
from bayeso.gp import gp
from bayeso.utils import utils_covariance
from bayeso.utils import utils_plotting
```

Declare some parameters to control this example, including zero-mean prior, and compute a covariance matrix.

```
num_points = 1000
str_cov = 'se'
num_iter = 50
num_ts = 10

list_Y_min = []

X = np.expand_dims(np.linspace(-5, 5, num_points), axis=1)
mu = np.zeros(num_points)
hyps = utils_covariance.get_hyps(str_cov, 1)
Sigma = covariance.cov_main(str_cov, X, X, hyps, True)
```

Optimize a function sampled from a Gaussian process prior. At each iteration, we sample a query point that outputs the mininum value of the function sampled from a Gaussian process posterior.

```
for ind_ts in range(0, num_ts):
    print('TS:', ind_ts + 1, 'round')
    Y = gp.sample_functions(mu, Sigma, num_samples=1)[0]

    ind_init = np.argmin(Y)
    bx_min = X[ind_init]
    y_min = Y[ind_init]
```

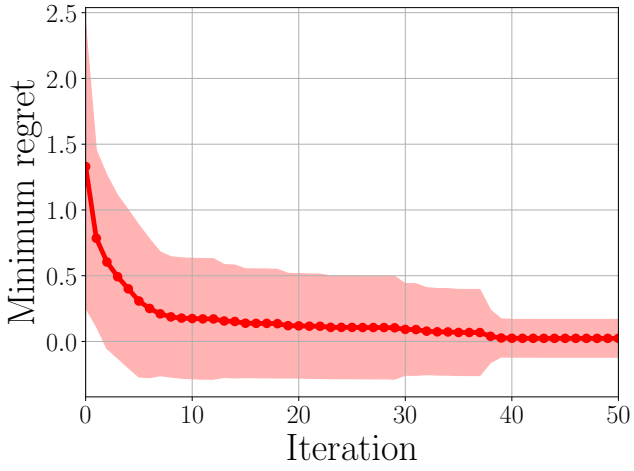
(continues on next page)

(continued from previous page)

```
    ind_random = np.random.choice(num_points)

    X_ = np.expand_dims(X[ind_random], axis=0)
    Y_ = np.expand_dims(np.expand_dims(Y[ind_random], axis=0), axis=1)

    for ind_iter in range(0, num_iter):
        print(ind_iter + 1, 'iteration')

        mu_, sigma_, Sigma_ = gp.predict_with_optimized_hyps(X_, Y_, X, str_cov=str_
↪cov)
        ind_ = np.argmin(gp.sample_functions(np.squeeze(mu_, axis=1), Sigma_, num_
↪samples=1)[0])

        X_ = np.concatenate([X_, [X[ind_]]], axis=0)
        Y_ = np.concatenate([Y_, [[Y[ind_]]]], axis=0)

    list_Y_min.append(Y_ - y_min)

Ys = np.array(list_Y_min)
Ys = np.squeeze(Ys, axis=2)
print(Ys.shape)
```

Plot the result obtained from the code block above.

```
utils_plotting.plot_minimum_vs_iter(
    np.array([Ys]), ['TS'], 1, True,
    use_tex=True, range_shade=1.0,
    str_x_axis=r'\textrm{Iteration}',
    str_y_axis=r'\textrm{Minimum regret}'
)
```



Full code:

```
import numpy as np

from bayeso import covariance
from bayeso.gp import gp
from bayeso.utils import utils_covariance
```

(continues on next page)

(continued from previous page)

```
from bayeso.utils import utils_plotting

num_points = 1000
str_cov = 'se'
num_iter = 50
num_ts = 10

list_Y_min = []

X = np.expand_dims(np.linspace(-5, 5, num_points), axis=1)
mu = np.zeros(num_points)
hyps = utils_covariance.get_hyps(str_cov, 1)
Sigma = covariance.cov_main(str_cov, X, X, hyps, True)

for ind_ts in range(0, num_ts):
    print('TS:', ind_ts + 1, 'round')
    Y = gp.sample_functions(mu, Sigma, num_samples=1)[0]

    ind_init = np.argmin(Y)
    bx_min = X[ind_init]
    y_min = Y[ind_init]

    ind_random = np.random.choice(num_points)

    X_ = np.expand_dims(X[ind_random], axis=0)
    Y_ = np.expand_dims(np.expand_dims(Y[ind_random], axis=0), axis=1)

    for ind_iter in range(0, num_iter):
        print(ind_iter + 1, 'iteration')

        mu_, sigma_, Sigma_ = gp.predict_with_optimized_hyps(X_, Y_, X, str_cov=str_
↪cov)
        ind_ = np.argmin(gp.sample_functions(np.squeeze(mu_, axis=1), Sigma_, num_
↪samples=1)[0])

        X_ = np.concatenate([X_, [X[ind_]]], axis=0)
        Y_ = np.concatenate([Y_, [[Y[ind_]]]], axis=0)

    list_Y_min.append(Y_ - y_min)

Ys = np.array(list_Y_min)
Ys = np.squeeze(Ys, axis=2)
print(Ys.shape)

utils_plotting.plot_minimum_vs_iter(
    np.array([Ys]), ['TS'], 1, True,
    use_tex=True, range_shade=1.0,
    str_x_axis=r'\textrm{Iteration}',
    str_y_axis=r'\textrm{Minimum regret}'
)
```

CHAPTER 6

Optimizing Branin Function

This example is for optimizing Branin function. It needs to install **bayeso-benchmarks**, which is included in **requirements-optional.txt**. First, import some packages we need.

```
import numpy as np

from bayeso import bo
from benchmarks.two_dim_branin import Branin
from bayeso.wrappers import wrappers_bo
from bayeso.utils import utils_plotting
```

Then, declare Branin function we will optimize and a search space for the function.

```
obj_fun = Branin()
bounds = obj_fun.get_bounds()

def fun_target(X):
    return obj_fun.output(X)
```

We optimize the objective function with 10 Bayesian optimization rounds and 50 iterations per round with 3 initial random evaluations.

```
str_fun = 'branin'

num_bo = 10
num_iter = 50
num_init = 3
```

With BO class in *bayeso.bo*, optimize the objective function.

```
model_bo = bo.BO(bounds, debug=False)
list_Y = []
list_time = []

for ind_bo in range(0, num_bo):
```

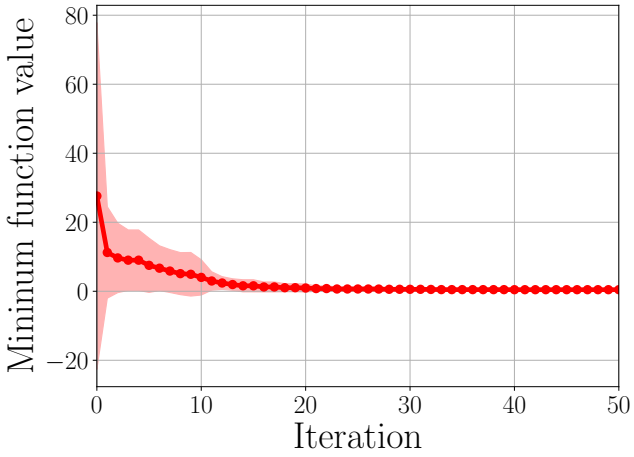
(continues on next page)

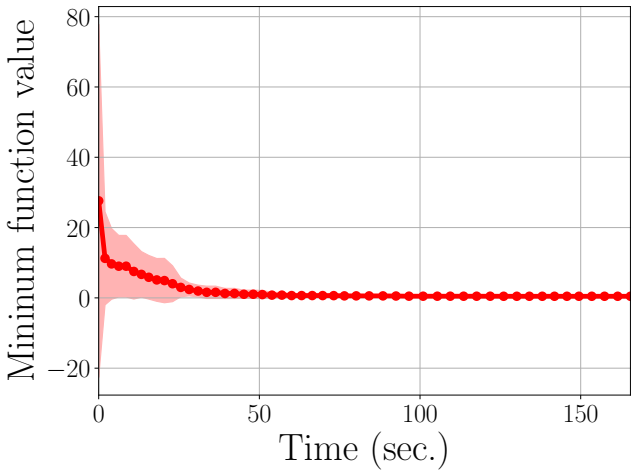
(continued from previous page)

```
    print('BO Round', ind_bo + 1)
    X_final, Y_final, time_final, _, _ = wrappers_bo.run_single_round(
        model_bo, fun_target, num_init, num_iter,
        str_initial_method_bo='uniform', str_sampling_method_ao='uniform', num_
↪samples_ao=100,
        seed=42 * ind_bo
    )
    list_Y.append(Y_final)
    list_time.append(time_final)

arr_Y = np.array(list_Y)
arr_time = np.array(list_time)

arr_Y = np.expand_dims(np.squeeze(arr_Y), axis=0)
arr_time = np.expand_dims(arr_time, axis=0)
```

Plot the results in terms of the number of iterations and time.

```
utils_plotting.plot_minimum_vs_iter(arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Iteration}',
    str_y_axis=r'\textrm{Mininum function value}')
utils_plotting.plot_minimum_vs_time(arr_time, arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Time (sec.)}',
    str_y_axis=r'\textrm{Mininum function value}')
```





Full code:

```
import numpy as np

from bayeso import bo
from benchmarks.two_dim_branin import Branin
from bayeso.wrappers import wrappers_bo
from bayeso.utils import utils_plotting

obj_fun = Branin()
bounds = obj_fun.get_bounds()

def fun_target(X):
    return obj_fun.output(X)

str_fun = 'branin'

num_bo = 10
num_iter = 50
num_init = 3

model_bo = bo.BO(bounds, debug=False)
list_Y = []
list_time = []

for ind_bo in range(0, num_bo):
    print('BO Round', ind_bo + 1)
    X_final, Y_final, time_final, _, _ = wrappers_bo.run_single_round(
        model_bo, fun_target, num_init, num_iter,
        str_initial_method_bo='uniform', str_sampling_method_ao='uniform', num_
↪samples_ao=100,
        seed=42 * ind_bo
    )
    list_Y.append(Y_final)
    list_time.append(time_final)

arr_Y = np.array(list_Y)
arr_time = np.array(list_time)

arr_Y = np.expand_dims(np.squeeze(arr_Y), axis=0)
arr_time = np.expand_dims(arr_time, axis=0)
```

(continues on next page)

(continued from previous page)

```
utils_plotting.plot_minimum_vs_iter(arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Iteration}',
    str_y_axis=r'\textrm{Mininum function value}')
utils_plotting.plot_minimum_vs_time(arr_time, arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Time (sec.)}',
    str_y_axis=r'\textrm{Mininum function value}')
```

CHAPTER 7

Constructing xgboost Classifier with Hyperparameter Optimization

This example is for optimizing hyperparameters for **xgboost** classifier. In this example, we optimize *max_depth* and *n_estimators* for *xgboost.XGBClassifier*. It needs to install **xgboost**, which is included in **requirements-examples.txt**. First, import some packages we need.

```
import numpy as np
import xgboost as xgb
import sklearn.datasets
import sklearn.metrics
import sklearn.model_selection

from bayeso import bo
from bayeso.wrappers import wrappers_bo
from bayeso.utils import utils_plotting
```

Get handwritten digits dataset, which contains digit images of 0 to 9, and split the dataset to training and test datasets.

```
digits = sklearn.datasets.load_digits()
data_digits = digits.images
data_digits = np.reshape(data_digits,
    (data_digits.shape[0], data_digits.shape[1] * data_digits.shape[2]))
labels_digits = digits.target

data_train, data_test, labels_train, labels_test = sklearn.model_selection.train_test_
↪split(
    data_digits, labels_digits, test_size=0.3, stratify=labels_digits)
```

Declare an objective function we would like to optimize. This function trains *xgboost.XGBClassifier* with the training dataset and given hyerparameter vector *bx* and returns (1 - accuracy), which computed by the test dataset.

```
def fun_target(bx):
    model_xgb = xgb.XGBClassifier(
        max_depth=int(bx[0]),
        n_estimators=int(bx[1])
    )
```

(continues on next page)

(continued from previous page)

```
    model_xgb.fit(data_train, labels_train)
    preds_test = model_xgb.predict(data_test)
    return 1.0 - sklearn.metrics.accuracy_score(labels_test, preds_test)
```

We optimize the objective function with our *bayeso.bo.BO* for 50 iterations. 5 initial points would be given and 10 rounds would be run.

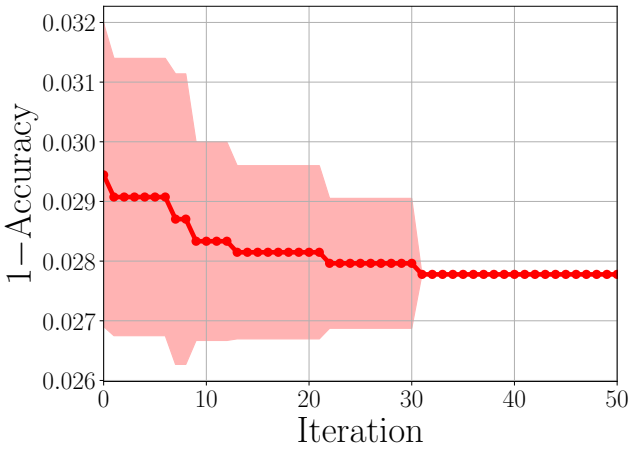
```
str_fun = 'xgboost'

# (max_depth, n_estimators)
bounds = np.array([[1, 10], [100, 500]])
num_bo = 10
num_iter = 50
num_init = 5
```

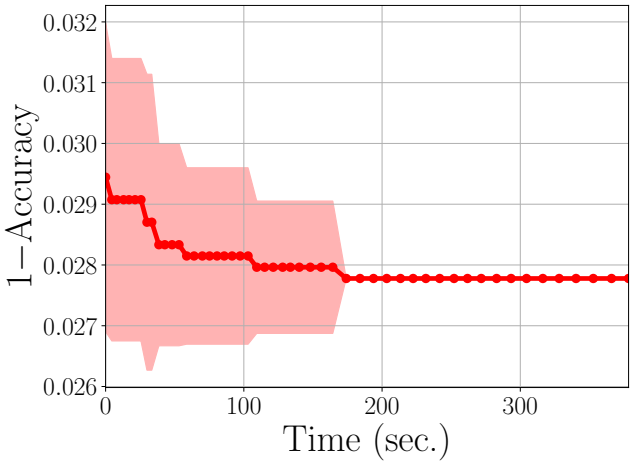
Optimze the objective function, after declaring the *bayeso.bo.BO* object.

```
model_bo = bo.BO(bounds, debug=False)

list_Y = []
list_time = []

for ind_bo in range(0, num_bo):
    print('BO Round:', ind_bo + 1)
    X_final, Y_final, time_final, _, _ = wrappers_bo.run_single_round(
        model_bo, fun_target, num_init, num_iter,
        str_initial_method_bo='uniform', str_sampling_method_ao='uniform',
        num_samples_ao=100, seed=42 * ind_bo)
    list_Y.append(Y_final)
    list_time.append(time_final)

arr_Y = np.array(list_Y)
arr_time = np.array(list_time)

arr_Y = np.expand_dims(np.squeeze(arr_Y), axis=0)
arr_time = np.expand_dims(arr_time, axis=0)
```

Plot the results in terms of the number of iterations and time.

```
utils_plotting.plot_minimum_vs_iter(arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Iteration}',
    str_y_axis=r'$1 - $\textrm{Accuracy}')
utils_plotting.plot_minimum_vs_time(arr_time, arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Time (sec.)}',
    str_y_axis=r'$1 - $\textrm{Accuracy}')
```





Full code:

```
import numpy as np
import xgboost as xgb
import sklearn.datasets
import sklearn.metrics
import sklearn.model_selection

from bayeso import bo
from bayeso.wrappers import wrappers_bo
from bayeso.utils import utils_plotting

digits = sklearn.datasets.load_digits()
data_digits = digits.images
data_digits = np.reshape(data_digits,
    (data_digits.shape[0], data_digits.shape[1] * data_digits.shape[2]))
labels_digits = digits.target

data_train, data_test, labels_train, labels_test = sklearn.model_selection.train_test_
↪split(
    data_digits, labels_digits, test_size=0.3, stratify=labels_digits)

def fun_target(bx):
    model_xgb = xgb.XGBClassifier(
        max_depth=int(bx[0]),
```

(continues on next page)

(continued from previous page)

```
        n_estimators=int(bx[1])
    )
    model_xgb.fit(data_train, labels_train)
    preds_test = model_xgb.predict(data_test)
    return 1.0 - sklearn.metrics.accuracy_score(labels_test, preds_test)

str_fun = 'xgboost'

# (max_depth, n_estimators)
bounds = np.array([[1, 10], [100, 500]])
num_bo = 10
num_iter = 50
num_init = 5

model_bo = bo.BO(bounds, debug=False)

list_Y = []
list_time = []

for ind_bo in range(0, num_bo):
    print('BO Round:', ind_bo + 1)
    X_final, Y_final, time_final, _, _ = wrappers_bo.run_single_round(
        model_bo, fun_target, num_init, num_iter,
        str_initial_method_bo='uniform', str_sampling_method_ao='uniform',
        num_samples_ao=100, seed=42 * ind_bo)
    list_Y.append(Y_final)
    list_time.append(time_final)

arr_Y = np.array(list_Y)
arr_time = np.array(list_time)

arr_Y = np.expand_dims(np.squeeze(arr_Y), axis=0)
arr_time = np.expand_dims(arr_time, axis=0)

utils_plotting.plot_minimum_vs_iter(arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Iteration}',
    str_y_axis=r'$1 - $\textrm{Accuracy}')
utils_plotting.plot_minimum_vs_time(arr_time, arr_Y, [str_fun], num_init, True,
    use_tex=True,
    str_x_axis=r'\textrm{Time (sec.)}',
    str_y_axis=r'$1 - $\textrm{Accuracy}')
```

CHAPTER 8

bayeso

BayesO is a simple, but essential Bayesian optimization package, implemented in Python.

8.1 bayeso.acquisition

It defines acquisition functions.

bayeso.acquisition.**aei**(*pred_mean: numpy.ndarray*, *pred_std: numpy.ndarray*, *Y_train: numpy.ndarray*, *noise: float*, *jitter: float = 1e-05*) → numpy.ndarray

It is an augmented expected improvement criterion.

Parameters

- **pred_mean** (*numpy.ndarray*) – posterior predictive mean function over *X_test*. Shape: (l,).
- **pred_std** (*numpy.ndarray*) – posterior predictive standard deviation function over *X_test*. Shape: (l,).
- **Y_train** (*numpy.ndarray*) – outputs of *X_train*. Shape: (n, 1).
- **noise** (*float*) – noise for augmenting exploration.
- **jitter** (*float, optional*) – jitter for *pred_std*.

Returns acquisition function values. Shape: (l,).

Return type numpy.ndarray

Raises AssertionError

bayeso.acquisition.**ei**(*pred_mean: numpy.ndarray*, *pred_std: numpy.ndarray*, *Y_train: numpy.ndarray*, *jitter: float = 1e-05*) → numpy.ndarray

It is an expected improvement criterion.

Parameters

- **pred_mean** (*numpy.ndarray*) – posterior predictive mean function over *X_test*. Shape: (l,).
- **pred_std** (*numpy.ndarray*) – posterior predictive standard deviation function over *X_test*. Shape: (l,).
- **Y_train** (*numpy.ndarray*) – outputs of *X_train*. Shape: (n, 1).
- **jitter** (*float, optional*) – jitter for *pred_std*.

Returns acquisition function values. Shape: (l,).

Return type numpy.ndarray

Raises AssertionError

bayeso.acquisition.**pi**(*pred_mean: numpy.ndarray*, *pred_std: numpy.ndarray*, *Y_train: numpy.ndarray*, *jitter: float = 1e-05*) → numpy.ndarray

It is a probability improvement criterion.

Parameters

- **pred_mean** (*numpy.ndarray*) – posterior predictive mean function over *X_test*. Shape: (l,).
- **pred_std** (*numpy.ndarray*) – posterior predictive standard deviation function over *X_test*. Shape: (l,).
- **Y_train** (*numpy.ndarray*) – outputs of *X_train*. Shape: (n, 1).
- **jitter** (*float, optional*) – jitter for *pred_std*.

Returns acquisition function values. Shape: (l,).

Return type numpy.ndarray

Raises AssertionError

bayeso.acquisition.**pure_exploit**(*pred_mean: numpy.ndarray*) → numpy.ndarray

It is a pure exploitation criterion.

Parameters **pred_mean** (*numpy.ndarray*) – posterior predictive mean function over *X_test*. Shape: (l,).

Returns acquisition function values. Shape: (l,).

Return type numpy.ndarray

Raises AssertionError

bayeso.acquisition.**pure_explore**(*pred_std: numpy.ndarray*) → numpy.ndarray

It is a pure exploration criterion.

Parameters **pred_std** (*numpy.ndarray*) – posterior predictive standard deviation function over *X_test*. Shape: (l,).

Returns acquisition function values. Shape: (l,).

Return type numpy.ndarray

Raises AssertionError

bayeso.acquisition.**ucb**(*pred_mean: numpy.ndarray*, *pred_std: numpy.ndarray*, *Y_train: Optional[numpy.ndarray] = None*, *kappa: float = 2.0*, *increase_kappa: bool = True*) → numpy.ndarray

It is a Gaussian process upper confidence bound criterion.

Parameters

- **pred_mean** (*numpy.ndarray*) – posterior predictive mean function over *X_test*. Shape: (l,).
- **pred_std** (*numpy.ndarray*) – posterior predictive standard deviation function over *X_test*. Shape: (l,).
- **Y_train** (*numpy.ndarray, optional*) – outputs of *X_train*. Shape: (n, 1).
- **kappa** (*float, optional*) – trade-off hyperparameter between exploration and exploitation.
- **increase_kappa** (*bool., optional*) – flag for increasing a kappa value as *Y_train* grows. If *Y_train* is None, it is ignored, which means *kappa* is fixed.

Returns acquisition function values. Shape: (l,).

Return type numpy.ndarray

Raises AssertionError

8.2 bayeso.bo

It defines a class for Bayesian optimization.

class bayeso.bo.**BO**(*range_X: numpy.ndarray*, *str_cov: str = 'matern52'*, *str_acq: str = 'ei'*, *normalize_Y: bool = True*, *use_ard: bool = True*, *prior_mu: Optional[callable] = None*, *str_optimizer_method_gp: str = 'BFGS'*, *str_optimizer_method_bo: str = 'L-BFGS-B'*, *str_modelselection_method: str = 'ml'*, *debug: bool = False*)

Bases: object

It is a Bayesian optimization class.

Parameters

- **range_X** (*numpy.ndarray*) – a search space. Shape: (d, 2).
- **str_cov** (*str., optional*) – the name of covariance function.
- **str_acq** (*str., optional*) – the name of acquisition function.
- **normalize_Y** (*bool., optional*) – flag for normalizing outputs.
- **use_ard** (*bool., optional*) – flag for automatic relevance determination.
- **prior_mu** (*NoneType, or function, optional*) – None, or prior mean function.
- **str_optimizer_method_gp** (*str., optional*) – the name of optimization method for Gaussian process regression.
- **str_optimizer_method_bo** (*str., optional*) – the name of optimization method for Bayesian optimization.
- **str_modelselection_method** (*str., optional*) – the name of model selection method for Gaussian process regression.
- **debug** (*bool., optional*) – flag for printing log messages.

_get_bounds() → list

It returns list of range tuples, obtained from *self.range_X*.

Returns list of range tuples.

Return type list

_get_samples_grid(*num_grids: int = 50*) → numpy.ndarray
It returns grids of *self.range_X*.

Parameters **num_grids** (*int., optional*) – the number of grids.

Returns grids of *self.range_X*. Shape: (*num_grids*$^{\mathrm{d}}$, d).

Return type numpy.ndarray

Raises AssertionError

_get_samples_latin(*num_samples: int*) → numpy.ndarray
It returns *num_samples* examples sampled from Latin hypercube.

Parameters **num_samples** (*int.*) – the number of samples.

Returns examples sampled from Latin hypercube. Shape: (*num_samples*, d).

Return type numpy.ndarray

Raises AssertionError

_get_samples_sobol(*num_samples: int*, *seed: Optional[int] = None*) → numpy.ndarray
It returns *num_samples* examples sampled from Sobol sequence.

Parameters

- **num_samples** (*int.*) – the number of samples.
- **seed** (*NoneType or int., optional*) – None, or random seed.

Returns examples sampled from Sobol sequence. Shape: (*num_samples*, d).

Return type numpy.ndarray

Raises AssertionError

_get_samples_uniform(*num_samples: int*, *seed: Optional[int] = None*) → numpy.ndarray
It returns *num_samples* examples uniformly sampled.

Parameters

- **num_samples** (*int.*) – the number of samples.
- **seed** (*NoneType or int., optional*) – None, or random seed.

Returns random examples. Shape: (*num_samples*, d).

Return type numpy.ndarray

Raises AssertionError

_optimize(*fun_negative_acquisition: callable*, *str_sampling_method: str*, *num_samples: int*) → Tuple[numpy.ndarray, numpy.ndarray]
It optimizes *fun_negative_function* with *self.str_optimizer_method_bo*. *num_samples* examples are determined by *str_sampling_method*, to start acquisition function optimization.

Parameters

- **fun_objective** (*function*) – negative acquisition function.
- **str_sampling_method** (*str.*) – the name of sampling method.
- **num_samples** (*int.*) – the number of samples.

Returns tuple of next point to evaluate and all candidates determined by acquisition function optimization. Shape: ((d,), (*num_samples*, d)).

Return type (numpy.ndarray, numpy.ndarray)

_optimize_objective(*fun_acquisition: callable*, *X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *X_test: numpy.ndarray*, *cov_X_X: numpy.ndarray*, *inv_cov_X_X: numpy.ndarray*, *hyps: dict*) → numpy.ndarray

It returns acquisition function values over *X_test*.

Parameters

- **fun_acquisition** (*function*) – acquisition function.
- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – inputs. Shape: (l, d) or (l, m, d).
- **cov_X_X** (*numpy.ndarray*) – kernel matrix over *X_train*. Shape: (n, n).
- **inv_cov_X_X** (*numpy.ndarray*) – kernel matrix inverse over *X_train*. Shape: (n, n).
- **hyps** (*dict.*) – dictionary of hyperparameters for Gaussian process.

Returns acquisition function values over *X_test*. Shape: (l,).

Return type numpy.ndarray

get_initials(*str_initial_method: str*, *num_initials: int*, *seed: Optional[int] = None*) → numpy.ndarray

It returns *num_initials* examples, sampled by a sampling method *str_initial_method*.

Parameters

- **str_initial_method** (*str.*) – the name of sampling method.
- **num_initials** (*int., optional*) – the number of samples.
- **seed** (*NoneType or int., optional*) – None, or random seed.

Returns sampled examples. Shape: (*num_samples*, d).

Return type numpy.ndarray

Raises AssertionError

get_samples(*str_sampling_method: str*, *fun_objective: Optional[callable] = None*, *num_samples: int = 100*, *seed: Optional[int] = None*) → numpy.ndarray

It returns *num_samples* examples, sampled by a sampling method *str_sampling_method*.

Parameters

- **str_sampling_method** (*str.*) – the name of sampling method.
- **fun_objective** (*NoneType or function, optional*) – None, or objective function.
- **num_samples** (*int., optional*) – the number of samples.
- **seed** (*NoneType or int., optional*) – None, or random seed.

Returns sampled examples. Shape: (*num_samples*, d).

Return type numpy.ndarray

Raises AssertionError

optimize(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *str_sampling_method: str = 'uniform'*, *num_samples: int = 100*, *str_mlm_method: str = 'regular'*) → Tuple[numpy.ndarray, dict]

It computes acquired example, candidates of acquired examples, acquisition function values for the candidates, covariance matrix, inverse matrix of the covariance matrix, hyperparameters optimized, and execution times.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **str_sampling_method_method** – the name of sampling method for acquisition function optimization.
- **num_samples** (*int., optional*) – the number of samples.
- **str_mlm_method** (*str., optional*) – the name of marginal likelihood maximization method for Gaussian process regression.

Returns acquired example and dictionary of information. Shape: ((d,), dict.).

Return type (numpy.ndarray, dict.)

Raises AssertionError

8.3 bayeso.constants

This file is for declaring various default constants. If you would like to see the details, check out the repository.

8.4 bayeso.covariance

It defines covariance functions and their associated functions.

bayeso.covariance.**choose_fun_cov**(*str_cov: str*, *choose_grad: bool*) → callable

It is for choosing a covariance function or a function for computing gradients of covariance function.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **choose_grad** (*bool.*) – flag for returning a function for the gradients

Returns covariance function, or function for computing gradients of covariance function.

Return type function

Raises AssertionError

bayeso.covariance.**cov_main**(*str_cov: str*, *X: numpy.ndarray*, *Xp: numpy.ndarray*, *hyps: dict*, *same_X_Xp: bool*, *jitter: float = 1e-05*) → numpy.ndarray

It computes kernel matrix over *X* and *Xp*, where *hyps* is given.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **X** (*numpy.ndarray*) – one inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **same_X_Xp** (*bool.*) – flag for checking *X* and *Xp* are same.
- **jitter** (*float, optional*) – jitter for diagonal entries.

Returns kernel matrix over *X* and *Xp*. Shape: (n, m).

Return type numpy.ndarray

Raises AssertionError, ValueError

bayeso.covariance.**cov_matern32**(*X: numpy.ndarray, Xp: numpy.ndarray, lengthscales: Union[numpy.ndarray, float], signal: float*) → numpy.ndarray

It computes Matern 3/2 kernel over *X* and *Xp*, where *lengthscales* and *signal* are given.

Parameters

- **X** (*numpy.ndarray*) – inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **lengthscales** (*numpy.ndarray, or float*) – length scales. Shape: (d,) or ().
- **signal** (*float*) – coefficient for signal.

Returns kernel values over *X* and *Xp*. Shape: (n, m).

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**cov_matern52**(*X: numpy.ndarray, Xp: numpy.ndarray, lengthscales: Union[numpy.ndarray, float], signal: float*) → numpy.ndarray

It computes Matern 5/2 kernel over *X* and *Xp*, where *lengthscales* and *signal* are given.

Parameters

- **X** (*numpy.ndarray*) – inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **lengthscales** (*numpy.ndarray, or float*) – length scales. Shape: (d,) or ().
- **signal** (*float*) – coefficient for signal.

Returns kernel values over *X* and *Xp*. Shape: (n, m).

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**cov_se**(*X: numpy.ndarray, Xp: numpy.ndarray, lengthscales: Union[numpy.ndarray, float], signal: float*) → numpy.ndarray

It computes squared exponential kernel over *X* and *Xp*, where *lengthscales* and *signal* are given.

Parameters

- **X** (*numpy.ndarray*) – inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **lengthscales** (*numpy.ndarray, or float*) – length scales. Shape: (d,) or ().
- **signal** (*float*) – coefficient for signal.

Returns kernel values over *X* and *Xp*. Shape: (n, m).

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**cov_set**(*str_cov: str, X: numpy.ndarray, Xp: numpy.ndarray, lengthscales: Union[numpy.ndarray, float], signal: float*) → numpy.ndarray

It computes set kernel matrix over *X* and *Xp*, where *lengthscales* and *signal* are given.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **X** (*numpy.ndarray*) – one inputs. Shape: (n, m, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (l, m, d).
- **lengthscales** (*numpy.ndarray, or float*) – length scales. Shape: (d,) or ().
- **signal** (*float*) – coefficient for signal.

Returns set kernel matrix over *X* and *Xp*. Shape: (n, l).

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**grad_cov_main**(*str_cov: str*, *X: numpy.ndarray*, *Xp: numpy.ndarray*, *hyps: dict*, *fix_noise: bool*, *same_X_Xp: bool = True*, *jitter: float = 1e-05*) → numpy.ndarray

It computes gradients of kernel matrix over hyperparameters, where *hyps* is given.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **X** (*numpy.ndarray*) – one inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **fix_noise** (*bool.*) – flag for fixing a noise.
- **same_X_Xp** (*bool., optional*) – flag for checking *X* and *Xp* are same.
- **jitter** (*float, optional*) – jitter for diagonal entries.

Returns gradient matrix over hyperparameters. Shape: (n, m, l) where l is the number of hyperparameters.

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**grad_cov_matern32**(*cov_X_Xp: numpy.ndarray*, *X: numpy.ndarray*, *Xp: numpy.ndarray*, *hyps: dict*, *num_hyps: int*, *fix_noise: bool*) → numpy.ndarray

It computes gradients of Matern 3/2 kernel over *X* and *Xp*, where *hyps* is given.

Parameters

- **cov_X_Xp** (*numpy.ndarray*) – covariance matrix. Shape: (n, m).
- **X** (*numpy.ndarray*) – one inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **num_hyps** (*int.*) – the number of hyperparameters == l.
- **fix_noise** (*bool.*) – flag for fixing a noise.

Returns gradient matrix over hyperparameters. Shape: (n, m, l).

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**grad_cov_matern52**(*cov_X_Xp: numpy.ndarray*, *X: numpy.ndarray*, *Xp: numpy.ndarray*, *hyps: dict*, *num_hyps: int*, *fix_noise: bool*) → numpy.ndarray

It computes gradients of Matern 5/2 kernel over *X* and *Xp*, where *hyps* is given.

Parameters

- **cov_X_Xp** (*numpy.ndarray*) – covariance matrix. Shape: (n, m).
- **X** (*numpy.ndarray*) – one inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **num_hyps** (*int.*) – the number of hyperparameters == l.
- **fix_noise** (*bool.*) – flag for fixing a noise.

Returns gradient matrix over hyperparameters. Shape: (n, m, l).

Return type numpy.ndarray

Raises AssertionError

bayeso.covariance.**grad_cov_se**(*cov_X_Xp: numpy.ndarray*, *X: numpy.ndarray*, *Xp: numpy.ndarray*, *hyps: dict*, *num_hyps: int*, *fix_noise: bool*) → numpy.ndarray

It computes gradients of squared exponential kernel over *X* and *Xp*, where *hyps* is given.

Parameters

- **cov_X_Xp** (*numpy.ndarray*) – covariance matrix. Shape: (n, m).
- **X** (*numpy.ndarray*) – one inputs. Shape: (n, d).
- **Xp** (*numpy.ndarray*) – another inputs. Shape: (m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **num_hyps** (*int.*) – the number of hyperparameters == l.
- **fix_noise** (*bool.*) – flag for fixing a noise.

Returns gradient matrix over hyperparameters. Shape: (n, m, l).

Return type numpy.ndarray

Raises AssertionError

CHAPTER 9

bayeso.gp

These files are for implementing Gaussian process regression.

9.1 bayeso.gp.gp

It defines Gaussian process regression.

bayeso.gp.gp.**get_optimized_kernel**(*X_train: numpy.ndarray, Y_train: numpy.ndarray, prior_mu: Optional[callable], str_cov: str, str_framework: str = 'scipy', str_optimizer_method: str = 'BFGS', str_modelselection_method: str = 'ml', fix_noise: bool = True, debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, dict]

This function computes the kernel matrix optimized by optimization method specified, its inverse matrix, and the optimized hyperparameters.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **prior_mu** (*function or NoneType*) – prior mean function or None.
- **str_cov** (*str.*) – the name of covariance function.
- **str_framework** (*str.*) – the name of framework for optimizing kernel hyperparameters.
- **str_optimizer_method** (*str., optional*) – the name of optimization method.
- **str_modelselection_method** (*str., optional*) – the name of model selection method.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of kernel matrix over *X_train*, kernel matrix inverse, and dictionary of hyperparameters.

Return type tuple of (numpy.ndarray, numpy.ndarray, dict.)

Raises AssertionError, ValueError

bayeso.gp.gp.**predict_with_cov**(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *X_test: numpy.ndarray*, *cov_X_X: numpy.ndarray*, *inv_cov_X_X: numpy.ndarray*, *hyps: dict*, *str_cov: str = 'matern52'*, *prior_mu: Optional[callable] = None*, *debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

This function returns posterior mean and posterior standard deviation functions over *X_test*, computed by Gaussian process regression with *X_train*, *Y_train*, *cov_X_X*, *inv_cov_X_X*, and *hyps*.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – inputs. Shape: (l, d) or (l, m, d).
- **cov_X_X** (*numpy.ndarray*) – kernel matrix over *X_train*. Shape: (n, n).
- **inv_cov_X_X** (*numpy.ndarray*) – kernel matrix inverse over *X_train*. Shape: (n, n).
- **hyps** (*dict.*) – dictionary of hyperparameters for Gaussian process.
- **str_cov** (*str., optional*) – the name of covariance function.
- **prior_mu** (*NoneType, or function, optional*) – None, or prior mean function.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of posterior mean function over *X_test*, posterior standard deviation function over *X_test*, and posterior covariance matrix over *X_test*. Shape: ((l, 1), (l, 1), (l, l)).

Return type tuple of (numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.gp.gp.**predict_with_hyps**(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *X_test: numpy.ndarray*, *hyps: dict*, *str_cov: str = 'matern52'*, *prior_mu: Optional[callable] = None*, *debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

This function returns posterior mean and posterior standard deviation functions over *X_test*, computed by Gaussian process regression with *X_train*, *Y_train*, and *hyps*.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – inputs. Shape: (l, d) or (l, m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for Gaussian process.
- **str_cov** (*str., optional*) – the name of covariance function.
- **prior_mu** (*NoneType, or function, optional*) – None, or prior mean function.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of posterior mean function over *X_test*, posterior standard deviation function over *X_test*, and posterior covariance matrix over *X_test*. Shape: ((l, 1), (l, 1), (l, l)).

Return type tuple of (numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.gp.gp.**predict_with_optimized_hyps**(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *X_test: numpy.ndarray*, *str_cov: str = 'matern52'*, *str_optimizer_method: str = 'BFGS'*, *prior_mu: Optional[callable] = None*, *fix_noise: float = True*, *debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

This function returns posterior mean and posterior standard deviation functions over *X_test*, computed by the Gaussian process regression optimized with *X_train* and *Y_train*.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – inputs. Shape: (l, d) or (l, m, d).
- **str_cov** (*str., optional*) – the name of covariance function.
- **str_optimizer_method** (*str., optional*) – the name of optimization method.
- **prior_mu** (*NoneType, or function, optional*) – None, or prior mean function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of posterior mean function over *X_test*, posterior standard deviation function over *X_test*, and posterior covariance matrix over *X_test*. Shape: ((l, 1), (l, 1), (l, l)).

Return type tuple of (numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.gp.gp.**sample_functions**(*mu: numpy.ndarray*, *Sigma: numpy.ndarray*, *num_samples: int = 1*) → numpy.ndarray

It samples *num_samples* functions from multivariate Gaussian distribution (mu, Sigma).

Parameters

- **mu** (*numpy.ndarray*) – mean vector. Shape: (n,).
- **Sigma** (*numpy.ndarray*) – covariance matrix. Shape: (n, n).
- **num_samples** (*int., optional*) – the number of sampled functions

Returns sampled functions. Shape: (num_samples, n).

Return type numpy.ndarray

Raises AssertionError

9.2 bayeso.gp.gp_common

It defines functions for Gaussian process regression.

bayeso.gp.gp_common.**get_kernel_cholesky**(*X_train: numpy.ndarray*, *hyps: dict*, *str_cov: str*, *fix_noise: bool = True*, *use_gradient: bool = False*, *debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

This function computes a kernel inverse with Cholesky decomposition.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for Gaussian process.
- **str_cov** (*str.*) – the name of covariance function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **use_gradient** (*bool., optional*) – flag for computing and returning gradients of negative log marginal likelihood.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of kernel matrix over *X_train*, lower matrix computed by Cholesky decomposition, and gradients of kernel matrix. If *use_gradient* is False, gradients of kernel matrix would be None.

Return type tuple of (numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.gp.gp_common.**get_kernel_inverse**(*X_train: numpy.ndarray*, *hyps: dict*, *str_cov: str*, *fix_noise: bool = True*, *use_gradient: bool = False*, *debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

This function computes a kernel inverse without any matrix decomposition techniques.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **hyps** (*dict.*) – dictionary of hyperparameters for Gaussian process.
- **str_cov** (*str.*) – the name of covariance function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **use_gradient** (*bool., optional*) – flag for computing and returning gradients of negative log marginal likelihood.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of kernel matrix over *X_train*, kernel matrix inverse, and gradients of kernel matrix. If *use_gradient* is False, gradients of kernel matrix would be None.

Return type tuple of (numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

9.3 bayeso.gp.gp_gpytorch

It is Gaussian process regression implementations with GPyTorch.

class bayeso.gp.gp_gpytorch.**ExactGPModel**(*str_cov*, *prior_mu*, *X_train*, *Y_train*, *likelihood*)

Bases: gpytorch.models.exact_gp.ExactGP

forward(*X*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

bayeso.gp.gp_gpytorch.**get_optimized_kernel**(*X_train: numpy.ndarray, Y_train: numpy.ndarray, prior_mu: Optional[callable], str_cov: str, fix_noise: bool = True, num_iters: int = 1000, debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, dict]

This function computes the kernel matrix optimized by optimization method specified, its inverse matrix, and the optimized hyperparameters, using GPyTorch.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **prior_mu** (*function or NoneType*) – prior mean function or None.
- **str_cov** (*str.*) – the name of covariance function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **num_iters** (*int., optional*) – the number of iterations for optimizing negative log likelihood.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of kernel matrix over *X_train*, kernel matrix inverse, and dictionary of hyperparameters.

Return type tuple of (numpy.ndarray, numpy.ndarray, dict.)

Raises AssertionError, ValueError

9.4 bayeso.gp.gp_scipy

It is Gaussian process regression implementations with SciPy.

bayeso.gp.gp_scipy.**get_optimized_kernel**(*X_train: numpy.ndarray, Y_train: numpy.ndarray, prior_mu: Optional[callable], str_cov: str, str_optimizer_method: str = 'BFGS', str_modelselection_method: str = 'ml', fix_noise: bool = True, debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, dict]

This function computes the kernel matrix optimized by optimization method specified, its inverse matrix, and the optimized hyperparameters.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).

- **prior_mu** (*function or NoneType*) – prior mean function or None.
- **str_cov** (*str.*) – the name of covariance function.
- **str_optimizer_method** (*str., optional*) – the name of optimization method.
- **str_modelselection_method** (*str., optional*) – the name of model selection method.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of kernel matrix over *X_train*, kernel matrix inverse, and dictionary of hyperparameters.

Return type tuple of (numpy.ndarray, numpy.ndarray, dict.)

Raises AssertionError, ValueError

bayeso.gp.gp_scipy.**neg_log_ml**(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *hyps: numpy.ndarray*, *str_cov: str*, *prior_mu_train: numpy.ndarray*, *fix_noise: bool = True*, *use_cholesky: bool = True*, *use_gradient: bool = True*, *debug: bool = False*) → Union[float, Tuple[float, float]]

This function computes a negative log marginal likelihood.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **hyps** (*numpy.ndarray*) – hyperparameters for Gaussian process. Shape: (h,).
- **str_cov** (*str.*) – the name of covariance function.
- **prior_mu_train** (*numpy.ndarray*) – the prior values computed by get_prior_mu(). Shape: (n, 1).
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **use_cholesky** (*bool., optional*) – flag for using a cholesky decomposition.
- **use_gradient** (*bool., optional*) – flag for computing and returning gradients of negative log marginal likelihood.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns negative log marginal likelihood, or (negative log marginal likelihood, gradients of the likelihood).

Return type float, or tuple of (float, float)

Raises AssertionError

bayeso.gp.gp_scipy.**neg_log_pseudo_l_loocv**(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *hyps: numpy.ndarray*, *str_cov: str*, *prior_mu_train: numpy.ndarray*, *fix_noise: bool = True*, *debug: bool = False*) → float

It computes a negative log pseudo-likelihood using leave-one-out cross-validation.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).

- **hyps** (*numpy.ndarray*) – hyperparameters for Gaussian process. Shape: (h,).
- **str_cov** (*str.*) – the name of covariance function.
- **prior_mu_train** (*numpy.ndarray*) – the prior values computed by get_prior_mu(). Shape: (n, 1).
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns negative log pseudo-likelihood.

Return type float

Raises AssertionError

9.5 bayeso.gp.gp_tensorflow

It is Gaussian process regression implementations with TensorFlow.

bayeso.gp.gp_tensorflow.**get_optimized_kernel**(*X_train: numpy.ndarray, Y_train: numpy.ndarray, prior_mu: Optional[callable], str_cov: str, fix_noise: bool = True, num_iters: int = 1000, debug: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray, dict]

This function computes the kernel matrix optimized by optimization method specified, its inverse matrix, and the optimized hyperparameters, using TensorFlow and TensorFlow probability.

Parameters

- **X_train** (*numpy.ndarray*) – inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – outputs. Shape: (n, 1).
- **prior_mu** (*function or NoneType*) – prior mean function or None.
- **str_cov** (*str.*) – the name of covariance function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **num_iters** (*int., optional*) – the number of iterations for optimizing negative log likelihood.
- **debug** (*bool., optional*) – flag for printing log messages.

Returns a tuple of kernel matrix over *X_train*, kernel matrix inverse, and dictionary of hyperparameters.

Return type tuple of (numpy.ndarray, numpy.ndarray, dict.)

Raises AssertionError, ValueError

CHAPTER 10

bayeso.utils

These files are for implementing utilities for various features.

10.1 bayeso.utils.utils_bo

It is utilities for Bayesian optimization.

bayeso.utils.utils_bo.**check_hyps_convergence**(*list_hyps: list*, *hyps: dict*, *str_cov: str*, *fix_noise: bool*, *ratio_threshold: float = 0.05*) → bool

It checks convergence of hyperparameters for Gaussian process regression.

Parameters

- **list_hyps** (*list*) – list of historical hyperparameters for Gaussian process regression.
- **hyps** (*dict.*) – dictionary of hyperparameters for acquisition function.
- **str_cov** (*str.*) – the name of covariance function.
- **fix_noise** (*bool.*) – flag for fixing a noise.
- **ratio_threshold** (*float, optional*) – ratio of threshold for checking convergence.

Returns flag for checking convergence. If converged, it is True.

Return type bool.

Raises AssertionError

bayeso.utils.utils_bo.**check_optimizer_method_bo**(*str_optimizer_method_bo: str*, *dim: int*, *debug: bool*) → str

It checks the availability of optimization methods. It helps to run Bayesian optimization, even though additional optimization methods are not installed or there exist the conditions some of optimization methods cannot be run.

Parameters

- **str_optimizer_method_bo** (*str.*) – the name of optimization method for Bayesian optimization.
- **dim** (*int.*) – dimensionality of the problem we solve.
- **debug** (*bool.*) – flag for printing log messages.

Returns available *str_optimizer_method_bo*.

Return type str.

Raises AssertionError

bayeso.utils.utils_bo.**choose_fun_acquisition**(*str_acq: str*, *hyps: dict*) → callable

It chooses and returns an acquisition function.

Parameters

- **str_acq** (*str.*) – the name of acquisition function.
- **hyps** (*dict.*) – dictionary of hyperparameters for acquisition function.

Returns acquisition function.

Return type function

Raises AssertionError

bayeso.utils.utils_bo.**get_best_acquisition_by_evaluation**(*initials: numpy.ndarray*, *fun_objective: callable*) → numpy.ndarray

It returns the best acquisition with respect to values of *fun_objective*. Here, the best acquisition is a minimizer of *fun_objective*.

Parameters

- **initials** (*numpy.ndarray*) – inputs. Shape: (n, d).
- **fun_objective** (*function*) – an objective function.

Returns the best example of *initials*. Shape: (1, d).

Return type numpy.ndarray

Raises AssertionError

bayeso.utils.utils_bo.**get_best_acquisition_by_history**(*X: numpy.ndarray*, *Y: numpy.ndarray*) → Tuple[numpy.ndarray, float]

It returns the best acquisition that has shown minimum result, and its minimum result.

Parameters

- **X** (*numpy.ndarray*) – historical queries. Shape: (n, d).
- **Y** (*numpy.ndarray*) – the observations of *X*. Shape: (n, 1).

Returns a tuple of the best query and its result.

Return type (numpy.ndarray, float)

Raises AssertionError

bayeso.utils.utils_bo.**get_next_best_acquisition**(*points: numpy.ndarray*, *acquisitions: numpy.ndarray*, *points_evaluated: numpy.ndarray*) → numpy.ndarray

It returns the next best acquired example.

Parameters

- **points** (*numpy.ndarray*) – inputs for acquisition function. Shape: (n, d).
- **acquisitions** (*numpy.ndarray*) – acquisition function values over *points*. Shape: (n,).
- **points_evaluated** (*numpy.ndarray*) – examples evaluated so far. Shape: (m, d).

Returns next best acquired point. Shape: (d,).

Return type numpy.ndarray

Raises AssertionError

10.2 bayeso.utils.utils_common

It is utilities for common features.

bayeso.utils.utils_common.**get_grids**(*ranges: numpy.ndarray*, *num_grids: int*) → numpy.ndarray

It returns grids of given *ranges*, where each of dimension has *num_grids* partitions.

Parameters

- **ranges** (*numpy.ndarray*) – ranges. Shape: (d, 2).
- **num_grids** (*int.*) – the number of partitions per dimension.

Returns grids of given *ranges*. Shape: (num_grids^d, d).

Return type numpy.ndarray

Raises AssertionError

bayeso.utils.utils_common.**get_minimum**(*Y_all: numpy.ndarray*, *num_init: int*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

It returns accumulated minima at each iteration, their arithmetic means over rounds, and their standard deviations over rounds, which is widely used in Bayesian optimization community.

Parameters

- **Y_all** (*numpy.ndarray*) – historical function values. Shape: (r, t) where r is the number of Bayesian optimization rounds and t is the number of iterations including initial points for each round. For example, if we run 50 iterations with 5 initial examples and repeat this procedure 3 times, r would be 3 and t would be 55 (= 50 + 5).
- **num_init** (*int.*) – the number of initial points.

Returns tuple of accumulated minima, their arithmetic means over rounds, and their standard deviations over rounds. Shape: ((r, t - *num_init* + 1), (t - *num_init* + 1,), (t - *num_init* + 1,)).

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.utils.utils_common.**get_time**(*time_all: numpy.ndarray*, *num_init: int*, *include_init: bool*) → numpy.ndarray

It returns the means of accumulated execution times over rounds.

Parameters

- **time_all** (*numpy.ndarray*) – execution times for all Bayesian optimization rounds. Shape: (r, t) where r is the number of Bayesian optimization rounds and t is the number of iterations (including initial points if *include_init* is True, or excluding them if *include_init* is False) for each round.
- **num_init** (*int.*) – the number of initial points. If *include_init* is False, it is ignored even if it is provided.
- **include_init** (*bool.*) – flag for describing whether execution times to observe initial examples have been included or not.

Returns arithmetic means of accumulated execution times over rounds. Shape: (t - *num_init*,) if *include_init* is True. (t,), otherwise.

Return type numpy.ndarray

Raises AssertionError

bayeso.utils.utils_common.**validate_types**(*func: callable*) → callable

It is a decorator for validating the number of types, which are declared for typing.

Parameters **func** (*callable*) – an original function.

Returns a callable decorator.

Return type callable

Raises AssertionError

10.3 bayeso.utils.utils_covariance

It is utilities for covariance functions.

bayeso.utils.utils_covariance.**_get_list_first**() → list

It provides list of strings. The strings in that list require two hyperparameters, *signal* and *lengthscales*. We simply call it as *list_first*.

Returns list of strings, which satisfy some requirements we mentioned above.

Return type list

bayeso.utils.utils_covariance.**convert_hyps**(*str_cov: str*, *hyps: dict*, *fix_noise: bool = False*) → numpy.ndarray

It converts hyperparameters dictionary, *hyps* to numpy array.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.

Returns converted array of the hyperparameters given by *hyps*.

Return type numpy.ndarray

Raises AssertionError

bayeso.utils.utils_covariance.**get_hyps**(*str_cov: str*, *dim: int*, *use_ard: bool = True*) → dict

It returns a dictionary of default hyperparameters for covariance function, where *str_cov* and *dim* are given. If *use_ard* is True, the length scales would be *dim*-dimensional vector.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **dim** (*int.*) – dimensionality of the problem we are solving.
- **use_ard** (*bool., optional*) – flag for automatic relevance determination.

Returns dictionary of default hyperparameters for covariance function.

Return type dict.

Raises AssertionError

bayeso.utils.utils_covariance.**get_range_hyps**(*str_cov: str*, *dim: int*, *use_ard: bool = True*, *fix_noise: bool = False*) → list

It returns default optimization ranges of hyperparameters for Gaussian process regression.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **dim** (*int.*) – dimensionality of the problem we are solving.
- **use_ard** (*bool., optional*) – flag for automatic relevance determination.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.

Returns list of default optimization ranges for hyperparameters.

Return type list

Raises AssertionError

bayeso.utils.utils_covariance.**restore_hyps**(*str_cov: str*, *hyps: numpy.ndarray*, *fix_noise: bool = False*, *noise: float = 0.01*) → dict

It restores hyperparameters array, *hyps* to dictionary.

Parameters

- **str_cov** (*str.*) – the name of covariance function.
- **hyps** (*numpy.ndarray*) – array of hyperparameters for covariance function.
- **fix_noise** (*bool., optional*) – flag for fixing a noise.
- **noise** (*float, optional*) – fixed noise value.

Returns restored dictionary of the hyperparameters given by *hyps*.

Return type dict.

Raises AssertionError

bayeso.utils.utils_covariance.**validate_hyps_arr**(*hyps: numpy.ndarray*, *str_cov: str*, *dim: int*) → Tuple[numpy.ndarray, bool]

It validates hyperparameters array, *hyps*.

Parameters

- **hyps** (*numpy.ndarray*) – array of hyperparameters for covariance function.
- **str_cov** (*str.*) – the name of covariance function.
- **dim** (*int.*) – dimensionality of the problem we are solving.

Returns a tuple of valid hyperparameters and validity flag.

Return type (numpy.ndarray, bool.)

Raises AssertionError

bayeso.utils.utils_covariance.**validate_hyps_dict**(*hyps: dict*, *str_cov: str*, *dim: int*) → Tuple[dict, bool]

It validates hyperparameters dictionary, *hyps*.

Parameters

- **hyps** (*dict.*) – dictionary of hyperparameters for covariance function.
- **str_cov** (*str.*) – the name of covariance function.
- **dim** (*int.*) – dimensionality of the problem we are solving.

Returns a tuple of valid hyperparameters and validity flag.

Return type (dict., bool.)

Raises AssertionError

10.4 bayeso.utils.utils_gp

It is utilities for Gaussian process regression.

bayeso.utils.utils_gp.**check_str_cov**(*str_fun: str*, *str_cov: str*, *shape_X1: tuple*, *shape_X2: tuple = None*) → None

It is for validating the shape of X1 (and optionally the shape of X2).

Parameters

- **str_fun** (*str.*) – the name of function.
- **str_cov** (*str.*) – the name of covariance function.
- **shape_X1** (*tuple*) – the shape of X1.
- **shape_X2** (*NoneType or tuple, optional*) – None, or the shape of X2.

Returns None, if it is valid. Raise an error, otherwise.

Return type NoneType

Raises AssertionError, ValueError

bayeso.utils.utils_gp.**get_prior_mu**(*prior_mu: Optional[callable]*, *X: numpy.ndarray*) → numpy.ndarray

It computes the prior mean function values over inputs X.

Parameters

- **prior_mu** (*function or NoneType*) – prior mean function or None.
- **X** (*numpy.ndarray*) – inputs for prior mean function. Shape: (n, d) or (n, m, d).

Returns zero array, or array of prior mean function values. Shape: (n, 1).

Return type numpy.ndarray

Raises AssertionError

10.5 bayeso.utils.utils_logger

It is utilities for loggers.

bayeso.utils.utils_logger.**get_logger**(*str_name: str*) → logging.Logger
It returns a logger to record the messages generated in our package.

Parameters **str_name** (*str.*) – a logger name.

Returns a logger.

Return type logging.Logger

Raises AssertionError

bayeso.utils.utils_logger.**get_str_array**(*arr: numpy.ndarray*) → str
It converts an array into string. It can take one-dimensional, two-dimensional, and three-dimensional arrays.

Parameters **arr** (*numpy.ndarray*) – an array to be converted.

Returns a string.

Return type str.

Raises AssertionError

bayeso.utils.utils_logger.**get_str_array_1d**(*arr: numpy.ndarray*) → str
It converts a one-dimensional array into string.

Parameters **arr** (*numpy.ndarray*) – an array to be converted.

Returns a string.

Return type str.

Raises AssertionError

bayeso.utils.utils_logger.**get_str_array_2d**(*arr: numpy.ndarray*) → str
It converts a two-dimensional array into string.

Parameters **arr** (*numpy.ndarray*) – an array to be converted.

Returns a string.

Return type str.

Raises AssertionError

bayeso.utils.utils_logger.**get_str_array_3d**(*arr: numpy.ndarray*) → str
It converts a three-dimensional array into string.

Parameters **arr** (*numpy.ndarray*) – an array to be converted.

Returns a string.

Return type str.

Raises AssertionError

bayeso.utils.utils_logger.**get_str_hyps**(*hyps: dict*) → str
It converts a dictionary of hyperparameters into string.

Parameters **hyps** (*dict.*) – a hyperparameter dictionary to be converted.

Returns a string.

Return type str.

Raises AssertionError

10.6 bayeso.utils.utils_plotting

It is utilities for plotting figures.

bayeso.utils.utils_plotting.**_save_figure**(*path_save: str*, *str_postfix: str*, *str_prefix: str = ''*) → None

It saves a figure.

Parameters

- **path_save** (*str.*) – path for saving a figure.
- **str_postfix** (*str.*) – the name of postfix.
- **str_prefix** (*str., optional*) – the name of prefix.

Returns None.

Return type NoneType

bayeso.utils.utils_plotting.**_set_ax_config**(*ax: matplotlib.axes._subplots.AxesSubplot*, *str_x_axis: str*, *str_y_axis: str*, *size_labels: int = 32*, *size_ticks: int = 22*, *xlim_min: Optional[float] = None*, *xlim_max: Optional[float] = None*, *draw_box: bool = True*, *draw_zero_axis: bool = False*, *draw_grid: bool = True*) → None

It sets an axis configuration.

Parameters

- **ax** (*matplotlib.axes._subplots.AxesSubplot*) – inputs for acquisition function. Shape: (n, d).
- **str_x_axis** (*str.*) – the name of x axis.
- **str_y_axis** (*str.*) – the name of y axis.
- **size_labels** (*int., optional*) – label size.
- **size_ticks** (*int., optional*) – tick size.
- **xlim_min** (*NoneType or float, optional*) – None, or minimum for x limit.
- **xlim_max** (*NoneType or float, optional*) – None, or maximum for x limit.
- **draw_box** (*bool., optional*) – flag for drawing a box.
- **draw_zero_axis** (*bool., optional*) – flag for drawing a zero axis.
- **draw_grid** (*bool., optional*) – flag for drawing grids.

Returns None.

Return type NoneType

bayeso.utils.utils_plotting.**_set_font_config**(*use_tex: bool*) → None

It sets a font configuration.

Parameters **use_tex** (*bool.*) – flag for using latex.

Returns None.

Return type NoneType

bayeso.utils.utils_plotting.**_show_figure**(*pause_figure: bool, time_pause: Union[int, float]*) → None

It shows a figure.

Parameters

- **pause_figure** (*bool.*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float*) – pausing time.

Returns None.

Return type NoneType

bayeso.utils.utils_plotting.**plot_bo_step**(*X_train: numpy.ndarray, Y_train: numpy.ndarray, X_test: numpy.ndarray, Y_test: numpy.ndarray, mean_test: numpy.ndarray, std_test: numpy.ndarray, path_save: Optional[str] = None, str_postfix: Optional[str] = None, str_x_axis: str = 'x', str_y_axis: str = 'y', num_init: Optional[int] = None, use_tex: bool = False, draw_zero_axis: bool = False, pause_figure: bool = True, time_pause: Union[int, float] = 2.0, range_shade: float = 1.96*) → None

It is for plotting Bayesian optimization results step by step.

Parameters

- **X_train** (*numpy.ndarray*) – training inputs. Shape: (n, 1).
- **Y_train** (*numpy.ndarray*) – training outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – test inputs. Shape: (m, 1).
- **Y_test** (*numpy.ndarray*) – true test outputs. Shape: (m, 1).
- **mean_test** (*numpy.ndarray*) – posterior predictive mean function values over *X_test*. Shape: (m, 1).
- **std_test** (*numpy.ndarray*) – posterior predictive standard deviation function values over *X_test*. Shape: (m, 1).
- **path_save** (*NoneType or str., optional*) – None, or path for saving a figure.
- **str_postfix** (*NoneType or str., optional*) – None, or the name of postfix.
- **str_x_axis** (*str., optional*) – the name of x axis.
- **str_y_axis** (*str., optional*) – the name of y axis.
- **num_init** (*NoneType or int., optional*) – None, or the number of initial examples.
- **use_tex** (*bool., optional*) – flag for using latex.
- **draw_zero_axis** (*bool., optional*) – flag for drawing a zero axis.
- **pause_figure** (*bool., optional*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float, optional*) – pausing time.
- **range_shade** (*float, optional*) – shade range for standard deviation.

Returns None.

Return type NoneType

Raises AssertionError

bayeso.utils.utils_plotting.**plot_bo_step_with_acq**(*X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *X_test: numpy.ndarray*, *Y_test: numpy.ndarray*, *mean_test: numpy.ndarray*, *std_test: numpy.ndarray*, *acq_test: numpy.ndarray*, *path_save: Optional[str] = None*, *str_postfix: Optional[str] = None*, *str_x_axis: str = 'x'*, *str_y_axis: str = 'y'*, *str_acq_axis: str = 'acq.'*, *num_init: Optional[int] = None*, *use_tex: bool = False*, *draw_zero_axis: bool = False*, *pause_figure: bool = True*, *time_pause: Union[int, float] = 2.0*, *range_shade: float = 1.96*) → None

It is for plotting Bayesian optimization results step by step.

Parameters

- **X_train** (*numpy.ndarray*) – training inputs. Shape: (n, 1).
- **Y_train** (*numpy.ndarray*) – training outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – test inputs. Shape: (m, 1).
- **Y_test** (*numpy.ndarray*) – true test outputs. Shape: (m, 1).
- **mean_test** (*numpy.ndarray*) – posterior predictive mean function values over *X_test*. Shape: (m, 1).
- **std_test** (*numpy.ndarray*) – posterior predictive standard deviation function values over *X_test*. Shape: (m, 1).
- **acq_test** (*numpy.ndarray*) – acquisition funcion values over *X_test*. Shape: (m, 1).
- **path_save** (*NoneType or str., optional*) – None, or path for saving a figure.
- **str_postfix** (*NoneType or str., optional*) – None, or the name of postfix.
- **str_x_axis** (*str., optional*) – the name of x axis.
- **str_y_axis** (*str., optional*) – the name of y axis.
- **str_acq_axis** (*str., optional*) – the name of acquisition function axis.
- **num_init** (*NoneType or int., optional*) – None, or the number of initial examples.
- **use_tex** (*bool., optional*) – flag for using latex.
- **draw_zero_axis** (*bool., optional*) – flag for drawing a zero axis.
- **pause_figure** (*bool., optional*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float, optional*) – pausing time.
- **range_shade** (*float, optional*) – shade range for standard deviation.

Returns None.

Return type NoneType

Raises AssertionError

bayeso.utils.utils_plotting.**plot_gp_via_distribution**(*X_train: numpy.ndarray, Y_train: numpy.ndarray, X_test: numpy.ndarray, mean_test: numpy.ndarray, std_test: numpy.ndarray, Y_test: Optional[numpy.ndarray] = None, path_save: Optional[str] = None, str_postfix: Optional[str] = None, str_x_axis: str = 'x', str_y_axis: str = 'y', use_tex: bool = False, draw_zero_axis: bool = False, pause_figure: bool = True, time_pause: Union[int, float] = 2.0, range_shade: float = 1.96, colors: list = ['red', 'green', 'blue', 'orange', 'olive', 'purple', 'darkred', 'limegreen', 'deepskyblue', 'lightsalmon', 'aquamarine', 'navy', 'rosybrown', 'darkkhaki', 'darkslategray']*) → None

It is for plotting Gaussian process regression.

Parameters

- **X_train** (*numpy.ndarray*) – training inputs. Shape: (n, 1).
- **Y_train** (*numpy.ndarray*) – training outputs. Shape: (n, 1).
- **X_test** (*numpy.ndarray*) – test inputs. Shape: (m, 1).
- **mean_test** (*numpy.ndarray*) – posterior predictive mean function values over *X_test*. Shape: (m, 1).
- **std_test** (*numpy.ndarray*) – posterior predictive standard deviation function values over *X_test*. Shape: (m, 1).
- **Y_test** (*NoneType or numpy.ndarray, optional*) – None, or true test outputs. Shape: (m, 1).
- **path_save** (*NoneType or str., optional*) – None, or path for saving a figure.
- **str_postfix** (*NoneType or str., optional*) – None, or the name of postfix.
- **str_x_axis** (*str., optional*) – the name of x axis.
- **str_y_axis** (*str., optional*) – the name of y axis.
- **use_tex** (*bool., optional*) – flag for using latex.
- **draw_zero_axis** (*bool., optional*) – flag for drawing a zero axis.
- **pause_figure** (*bool., optional*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float, optional*) – pausing time.
- **range_shade** (*float, optional*) – shade range for standard deviation.

- **colors** (*list, optional*) – list of colors.

Returns None.

Return type NoneType

Raises AssertionError

bayeso.utils.utils_plotting.**plot_gp_via_sample**(*X: numpy.ndarray, Ys: numpy.ndarray, path_save: Optional[str] = None, str_postfix: Optional[str] = None, str_x_axis: str = 'x', str_y_axis: str = 'y', use_tex: bool = False, draw_zero_axis: bool = False, pause_figure: bool = True, time_pause: Union[int, float] = 2.0, colors: list = ['red', 'green', 'blue', 'orange', 'olive', 'purple', 'darkred', 'limegreen', 'deepskyblue', 'lightsalmon', 'aquamarine', 'navy', 'rosybrown', 'darkkhaki', 'darkslategray']*) → None

It is for plotting sampled functions from multivariate distributions.

Parameters

- **X** (*numpy.ndarray*) – training inputs. Shape: (n, 1).
- **Ys** (*numpy.ndarray*) – training outputs. Shape: (m, n).
- **path_save** (*NoneType or str., optional*) – None, or path for saving a figure.
- **str_postfix** (*NoneType or str., optional*) – None, or the name of postfix.
- **str_x_axis** (*str., optional*) – the name of x axis.
- **str_y_axis** (*str., optional*) – the name of y axis.
- **use_tex** (*bool., optional*) – flag for using latex.
- **draw_zero_axis** (*bool., optional*) – flag for drawing a zero axis.
- **pause_figure** (*bool., optional*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float, optional*) – pausing time.
- **colors** (*list, optional*) – list of colors.

Returns None.

Return type NoneType

Raises AssertionError

bayeso.utils.utils_plotting.**plot_minimum_vs_iter**(*minima: numpy.ndarray, list_str_label: list, num_init: int, draw_std: bool, include_marker: bool = True, include_legend: bool = False, use_tex: bool = False, path_save: Optional[str] = None, str_postfix: Optional[str] = None, str_x_axis: str = 'Iteration', str_y_axis: str = 'Minimum function value', pause_figure: bool = True, time_pause: Union[int, float] = 2.0, range_shade: float = 1.96, markers: list = ['.', 'x', '*', '+', '^', 'v', '<', '>', 'd', ',', '8', 'h', '1', '2', '3'], colors: list = ['red', 'green', 'blue', 'orange', 'olive', 'purple', 'darkred', 'limegreen', 'deepskyblue', 'lightsalmon', 'aquamarine', 'navy', 'rosybrown', 'darkkhaki', 'darkslategray']*) → None

It is for plotting optimization results of Bayesian optimization, in terms of iterations.

Parameters

- **minima** (*numpy.ndarray*) – function values over acquired examples. Shape: (b, r, n) where b is the number of experiments, r is the number of rounds, and n is the number of iterations per round.
- **list_str_label** (*list*) – list of label strings. Shape: (b,).
- **num_init** (*int.*) – the number of initial examples < n.
- **draw_std** (*bool.*) – flag for drawing standard deviations.
- **include_marker** (*bool., optional*) – flag for drawing markers.
- **include_legend** (*bool., optional*) – flag for drawing a legend.
- **use_tex** (*bool., optional*) – flag for using latex.
- **path_save** (*NoneType or str., optional*) – None, or path for saving a figure.
- **str_postfix** (*NoneType or str., optional*) – None, or the name of postfix.
- **str_x_axis** (*str., optional*) – the name of x axis.
- **str_y_axis** (*str., optional*) – the name of y axis.
- **pause_figure** (*bool., optional*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float, optional*) – pausing time.
- **range_shade** (*float, optional*) – shade range for standard deviation.
- **markers** (*list, optional*) – list of markers.
- **colors** (*list, optional*) – list of colors.

Returns None.

Return type NoneType

Raises AssertionError

bayeso.utils.utils_plotting.**plot_minimum_vs_time**(*times: numpy.ndarray, minima: numpy.ndarray, list_str_label: list, num_init: int, draw_std: bool, include_marker: bool = True, include_legend: bool = False, use_tex: bool = False, path_save: Optional[str] = None, str_postfix: Optional[str] = None, str_x_axis: str = 'Time (sec.)', str_y_axis: str = 'Minimum function value', pause_figure: bool = True, time_pause: Union[int, float] = 2.0, range_shade: float = 1.96, markers: list = ['.', 'x', '*', '+', '^', 'v', '<', '>', 'd', ',', '8', 'h', '1', '2', '3'], colors: list = ['red', 'green', 'blue', 'orange', 'olive', 'purple', 'darkred', 'limegreen', 'deepskyblue', 'lightsalmon', 'aquamarine', 'navy', 'rosybrown', 'darkkhaki', 'darkslategray']*) → None

It is for plotting optimization results of Bayesian optimization, in terms of execution time.

Parameters

- **times** (*numpy.ndarray*) – execution times. Shape: (b, r, n), or (b, r, *num_init* + n) where b is the number of experiments, r is the number of rounds, and n is the number of iterations per round.
- **minima** (*numpy.ndarray*) – function values over acquired examples. Shape: (b, r, *num_init* + n) where b is the number of experiments, r is the number of rounds, and n is the number of iterations per round.
- **list_str_label** (*list*) – list of label strings. Shape: (b,).
- **num_init** (*int.*) – the number of initial examples.
- **draw_std** (*bool.*) – flag for drawing standard deviations.
- **include_marker** (*bool., optional*) – flag for drawing markers.
- **include_legend** (*bool., optional*) – flag for drawing a legend.
- **use_tex** (*bool., optional*) – flag for using latex.
- **path_save** (*NoneType or str., optional*) – None, or path for saving a figure.
- **str_postfix** (*NoneType or str., optional*) – None, or the name of postfix.
- **str_x_axis** (*str., optional*) – the name of x axis.
- **str_y_axis** (*str., optional*) – the name of y axis.
- **pause_figure** (*bool., optional*) – flag for pausing before closing a figure.
- **time_pause** (*int. or float, optional*) – pausing time.
- **range_shade** (*float, optional*) – shade range for standard deviation.
- **markers** (*list, optional*) – list of markers.
- **colors** (*list, optional*) – list of colors.

Returns None.

Return type NoneType

Raises AssertionError

CHAPTER 11

bayeso.wrappers

These files are for implementing various wrappers.

11.1 bayeso.wrappers.wrappers_bo

It defines wrappers for Bayesian optimization.

bayeso.wrappers.wrappers_bo.**run_single_round**(*model_bo: bayeso.bo.BO*, *fun_target: callable*, *num_init: int*, *num_iter: int*, *str_initial_method_bo: str = 'uniform'*, *str_sampling_method_ao: str = 'uniform'*, *num_samples_ao: int = 100*, *str_mlm_method: str = 'regular'*, *seed: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

It optimizes *fun_target* for *num_iter* iterations with given *model_bo* and *num_init* initial examples. Initial examples are sampled by *get_initials* method in *model_bo*. It returns the optimization results and execution times.

Parameters

- **model_bo** (bayeso.bo.BO) – Bayesian optimization model.
- **fun_target** (*function*) – a target function.
- **num_init** (*int.*) – the number of initial examples for Bayesian optimization.
- **num_iter** (*int.*) – the number of iterations for Bayesian optimization.
- **str_initial_method_bo** (*str., optional*) – the name of initialization method for sampling initial examples in Bayesian optimization.
- **str_sampling_method_ao** (*str., optional*) – the name of initialization method for acquisition function optimization.

- **num_samples_ao** (*int., optional*) – the number of samples for acquisition function optimization. If L-BFGS-B is used as an acquisition function optimization method, it is employed.
- **str_mlm_method** (*str., optional*) – the name of marginal likelihood maximization method for Gaussian process regression.
- **seed** (*NoneType or int., optional*) – None, or random seed.

Returns tuple of acquired examples, their function values, overall execution times per iteration, execution time consumed in Gaussian process regression, and execution time consumed in acquisition function optimization. Shape: ((*num_init* + *num_iter*, d), (*num_init* + *num_iter*, 1), (*num_init* + *num_iter*,), (*num_iter*,), (*num_iter*,)), or ((*num_init* + *num_iter*, m, d), (*num_init* + *num_iter*, m, 1), (*num_init* + *num_iter*,), (*num_iter*,), (*num_iter*,)), where d is a dimensionality of the problem we are solving and m is a cardinality of sets.

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.wrappers.wrappers_bo.**run_single_round_with_all_initial_information**(*model_bo: bayeso.bo.BO*, *fun_target: callable*, *X_train: numpy.ndarray*, *Y_train: numpy.ndarray*, *num_iter: int*, *str_sampling_method_ac str* = *'uniform'*, *num_samples_ao: int* = *100*, *str_mlm_method: str* = *'regular'*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

It optimizes *fun_target* for *num_iter* iterations with given *model_bo*. It returns the optimization results and execution times.

Parameters

- **model_bo** (bayeso.bo.BO) – Bayesian optimization model.

- **fun_target** (*function*) – a target function.
- **X_train** (*numpy.ndarray*) – initial inputs. Shape: (n, d) or (n, m, d).
- **Y_train** (*numpy.ndarray*) – initial outputs. Shape: (n, 1).
- **num_iter** (*int.*) – the number of iterations for Bayesian optimization.
- **str_sampling_method_ao** (*str., optional*) – the name of initialization method for acquisition function optimization.
- **num_samples_ao** (*int., optional*) – the number of samples for acquisition function optimization. If L-BFGS-B is used as an acquisition function optimization method, it is employed.
- **str_mlm_method** (*str., optional*) – the name of marginal likelihood maximization method for Gaussian process regression.

Returns tuple of acquired examples, their function values, overall execution times per iteration, execution time consumed in Gaussian process regression, and execution time consumed in acquisition function optimization. Shape: ((n + *num_iter*, d), (n + *num_iter*, 1), (*num_iter*,), (*num_iter*,), (*num_iter*,)), or ((n + *num_iter*, m, d), (n + *num_iter*, m, 1), (*num_iter*,), (*num_iter*,), (*num_iter*,)).

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

bayeso.wrappers.wrappers_bo.**run_single_round_with_initial_inputs**(*model_bo: bayeso.bo.BO*, *fun_target: callable*, *X_train: numpy.ndarray*, *num_iter: int*, *str_sampling_method_ao: str = 'uniform'*, *num_samples_ao: int = 100*, *str_mlm_method: str = 'regular'*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

It optimizes *fun_target* for *num_iter* iterations with given *model_bo* and initial inputs *X_train*. It returns the optimization results and execution times.

Parameters

- **model_bo** (bayeso.bo.BO) – Bayesian optimization model.
- **fun_target** (*function*) – a target function.
- **X_train** (*numpy.ndarray*) – initial inputs. Shape: (n, d) or (n, m, d).
- **num_iter** (*int.*) – the number of iterations for Bayesian optimization.

- **str_sampling_method_ao** (*str., optional*) – the name of initialization method for acquisition function optimization.
- **num_samples_ao** (*int., optional*) – the number of samples for acquisition function optimization. If L-BFGS-B is used as an acquisition function optimization method, it is employed.
- **str_mlm_method** (*str., optional*) – the name of marginal likelihood maximization method for Gaussian process regression.

Returns tuple of acquired examples, their function values, overall execution times per iteration, execution time consumed in Gaussian process regression, and execution time consumed in acquisition function optimization. Shape: ((n + *num_iter*, d), (n + *num_iter*, 1), (n + *num_iter*,), (*num_iter*,), (*num_iter*,)), or ((n + *num_iter*, m, d), (n + *num_iter*, m, 1), (n + *num_iter*,), (*num_iter*,), (*num_iter*,)).

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray)

Raises AssertionError

Python Module Index

b

Index

Symbols

A

B

C

E

F

G

N

O

P

R

S

U

V