project-template Documentation

Release 0.0.1

Vighnesh Birodkar

May 30, 2017

Contents

Fluentopt is a flexible hyper-parameter optimization library.

Most hyper-parameter optimization libraries impose three main restrictions :

- they control the optimization loop
- they force the inputs to be represented by vectors
- the priors are very restricted, e.g gaussian, uniform or discrete uniform

the goal of fluentopt is to provide hyper-parameter optimization library where :

- the optimization loop is controlled by the user (but we will provide also helpers).
- the inputs can be represented by a python dictionary to express conditionals rather than just a list (or vector), but in case not needed they can also just be a list or a scalar. The dictionaries can contain strings, varying length lists and special objects like 'None'.
- the priors of the hyper-parameters are not restricted to some pre-defined probability distributions. Users will just provide samplers as a python function, that is, a function that takes a seed and returns a python dictionary.

CHAPTER 1

API Documentation

Base

This module contains a base class of optimizers. This module purpose is to describe the API that optimizers should follow.

class `fluentopt.base.`**`Optimizer`**
Optimizer base class

`suggest`()
Use the surrogate to suggest a next input to evaluate
Returns a dict, a list or a scalar

`update`(*x*, *y*)
Update the surrogate used by the optimizer using a single evaluation.
Parameters **x: dict, or list or scalar**

`update_many`(*xlist*, *ylist*)
Update the surrogate used by the optimizer using a list of evaluations.
Parameters **xlist** : list of dicts, or list of lists or list of scalars

Optimizers

class `fluentopt.random.`**`RandomSearch`**(*sampler*, *random_state=None*)
a random search optimizer. This optimizer is completely random, it does not use any surrogate model. It uses a *sampler*, which have to be a callable (e.g function) that takes a random state (integer) as input and returns a random sample. The *suggest* method just calls *sampler* each time.
Parameters **sampler** : callable
a callable used to sample an input for further evaluation. it takes one argument, a random number generator following the API of numpy.random and returns a dict, a list or a scalar.
random_state : int or None, optional

controls the random seed used by *sampler*.

Attributes

*input_history_*	(list of inputs evaluated) *output_history_* : outputs corresponding to the evaluated inputs

update(*x*, *y*)

Update the surrogate used by the optimizer using a single evaluation.

Parameters x: dict, or list or scalar

class fluentopt.bandit.**Bandit**(*sampler*, *model=<fluentopt.transformers.Wrapper object>*, *nb_suggestions=100*, *score=<function ucb_maximize>*, *random_state=None*)

a bandit based optimizer which uses a surrogate to model the mapping between inputs and outputs. Each time *suggest* is called, a total of *nb_suggestions* inputs are sampled from *sampler*. A score is then calculated for each sampled input, then the next input to evaluate is the one which has the maximum score (reward).

Parameters sampler : callable

a callable used to sample an input for further evaluation. it takes one argument, a random number generator following the API of numpy.random and returns a dict, a list or a scalar.

model : scikit-learn like model instance, optional

default is fluentopt.transformers.Wrapper(GaussianProcessRegressor(normalize_y=True)). Alternatives :

- fluentopt.transformers.Wrapper(fluentopt.utils.RandomForestRegressorWithUncertainty())
- **or use another model which supports returning uncertainty in prediction:** fluentopt.transformers.Wrapper(your_model())
- you can also extend or change the Wrapper, the goal of the wrapper is to feed a vectorized input to the wrapped model.

nb_suggestions : int, optional[default=100]

number of random samples to draw from the *sampler* in each call of *suggest* to select the next input to evaluate.

score : callable, optional[default=ucb_maximize]

score function to use when selecting the next input to evaluate. it takes two arguments, a model and a list of inputs. it returns a list of scores. Available scores are : *ucb_maximize*, *ucb_minimize*.

random_state : int or None, optional

controls the random seed used by *sampler*.

Attributes

input_history_	(list of inputs evaluated) **output_history_**: outputs corresponding to the evaluated inputs

get_scores(*inputs*)

use *score* to get the list of scores of the *inputs*

update(*x*, *y*)

Update the surrogate used by the optimizer using a single evaluation.

Parameters x: dict, or list or scalar

Scores

fluentopt.bandit.**ucb_maximize**(*model*, *inputs*, *kappa=1.96*)

UCB score that can be used as the *score* parameter of the *Bandit* optimizer. Use this score if the objective is maximization with ucb. UCB scores assume that the model can return std, that is, *model.predict* shoud accept a *return_std* parameter. An exception will be thrown if this is not the case.

Parameters **model** : scikit-learn like estimator with return_std

inputs : numpy array

kappa : float

controls the tradeoff between exploration and exploitation (higher value = more exploration)

fluentopt.bandit.**ucb_minimize**(*model*, *inputs*, *kappa=1.96*)

UCB score that can be used as the *score* parameter of the *Bandit* optimizer. Use this score if the objective is minimization. UCB scores assume that the model can return std, that is, *model.predict* shoud accept a *return_std* parameter. An exception will be thrown if this is not the case.

model : scikit-learn like estimator with return_std inputs : numpy array kappa : float

controls the tradeoff between exploration and exploitation (higher value = more exploration)

Transformers

this module contains transfomers that can vectorize data like dicts, lists of varying length.

class fluentopt.transformers.**Wrapper**(*model*, *transform_X=<function vectorize>*, *transform_y=<function Wrapper.<lambda>>*)

wraps a scikit-learn like estimator *model* to transform inputs and outputs using *transform_X* and *transform_y*. This is used to vectorize easily inputs that are passed to the model.

Parameters **model** : scikit-learn like estimator instance to wrap

transform_X : callable

used to transform the inputs before passing them to fit and predict

transform_y : callable

used to transform the outputs before passing them to fit

fluentopt.transformers.**vectorize**(*X*)

vectorizes *X* depending on its type:

- if it is a list of dicts, use *vectorize_list_of_dicts*.
- it it is a list of lists of varying length across examples, use *vectorize_list_of_varying_length_lists*.
- if it is a list of fixed length lists or list of scalars, just convert to numpy array.

Parameters **'X'** : a list of dicts or a list of varying length lists or a list of fixed length lists or list of scalars.

Returns 2D numpy array.

`fluentopt.transformers.`**`vectorize_list_of_dicts`**(*dlist*)
vectorize a list of dicts all columns are considered. rows that have missing columns will be replaced by *np.nan*.

Parameters **dlist** : list of dicts

Returns 2D numpy array

Utils

This module contains utility functions used by other modules. It contains mostly validation functions to check for validity of the parameters that a function or a class gets as an input.

`fluentopt.utils.`**`check_sampler`**(*sampler*)
check whether sampler is a callable

`fluentopt.utils.`**`flatten_dict`**(*D*)
converts a deep dict *D* into a flattened version *d*. it uses a recursive algo:

–start with an empty *d*

–iterate through the keys and values of D:

* **if the current value is a dict then update** *d* with the flattened version of the value by calling *flatten_dict* on the value
* **if the current value is a list or a tuple add all elements** of the list with keys *key_i* where i is the index of the element of the list and values the value of the list at index i.
* else just copy the key and value of *D* into *d*

`fluentopt.utils.`**`dict_vectorizer`**(*dlist*, *colnames*, *missing=nan*)
Converts a list of dicts into a numpy array. the i-th dimension of the vector will correspond to colnames[i]. if a column does not exist in a dict from *dlist*, it takes the value defined by *missing*.

Parameters **dlist** : list of dicts

colnames : list of strings

list of columns to use. the order of the columns in the resulting numpy will correspond to the order in *colnames*.

missing : scalar

the value to use for missing columns.

Returns 2D numpy array

class fluentopt.utils.**RandomForestRegressorWithUncertainty**(*n_estimators=10*, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_split=1e-07*, *bootstrap=True*, *oob_score=False*, *n_jobs=1*, *random_state=None*, *verbose=0*, *warm_start=False*)

an extension of RandomForestRegressor with support of returning uncertainty. it just takes the trees and compute the std of the predicted values for each tree.

apply(*X*)

Apply trees in the forest to X, return leaf indices.

Parameters **X** : array-like or sparse matrix, shape = [n_samples, n_features]

The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns **X_leaves** : array_like, shape = [n_samples, n_estimators]

For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path(*X*)

Return the decision path in the forest

New in version 0.18.

Parameters **X** : array-like or sparse matrix, shape = [n_samples, n_features]

The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns **indicator** : sparse csr array, shape = [n_samples, n_nodes]

Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr : array of size (n_estimators + 1,)

The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns **feature_importances_** : array, shape = [n_features]

fit(*X*, *y*, *sample_weight=None*)

Build a forest of trees from the training set (X, y).

Parameters **X** : array-like or sparse matrix of shape = [n_samples, n_features]

The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y : array-like, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels in classification, real numbers in regression).

sample_weight : array-like, shape = [n_samples] or None

Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns **self** : object

Returns self.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters **X** : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

Returns **X_new** : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** : boolean, optional

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** : mapping of string to any

Parameter names mapped to their values.

score(*X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0 and it can be negative (because

the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters **X** : array-like, shape = (n_samples, n_features)

Test samples.

y : array-like, shape = (n_samples) or (n_samples, n_outputs)

True values for X.

sample_weight : array-like, shape = [n_samples], optional

Sample weights.

Returns **score** : float

R^2 of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns self

transform(**args*, ***kwargs*)

DEPRECATED: Support to use estimators as feature selectors will be removed in version 0.19. Use SelectFromModel instead.

Reduce X to its most important features.

Uses `coef_` or `feature_importances_` to determine the most important features. For models with a `coef_` for each class, the absolute sum over the classes is used.

Parameters **X** : array or scipy sparse matrix of shape [n_samples, n_features]

The input samples.

threshold [string, float or None, optional (default=None)] The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median" (resp. "mean"), then the threshold value is the median (resp. the mean) of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. If None and if available, the object attribute `threshold` is used. Otherwise, "mean" is used by default.

Returns **X_r** : array of shape [n_samples, n_selected_features]

The input samples with only the selected features.

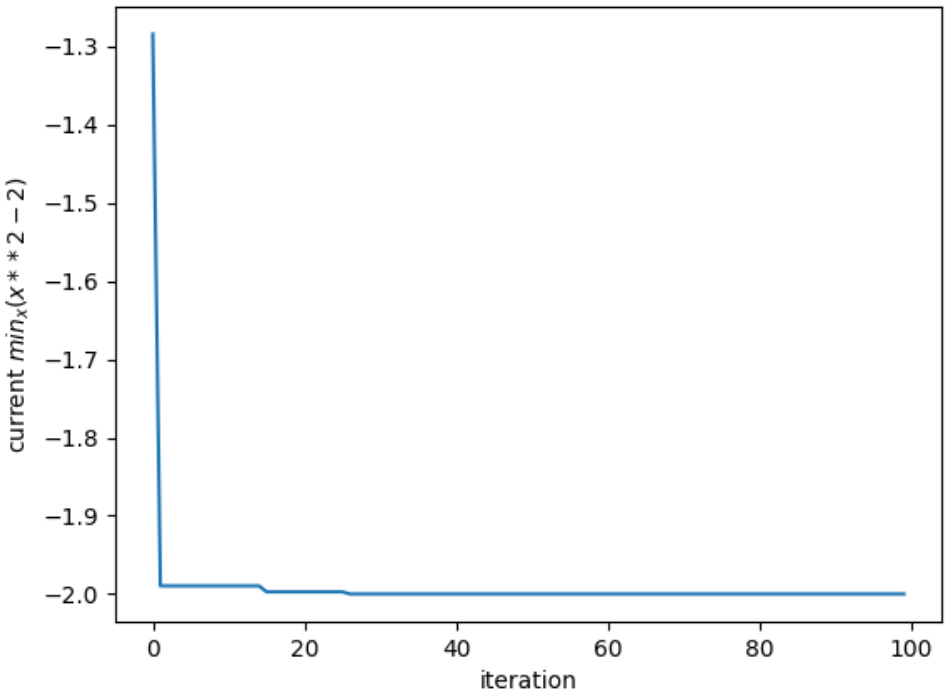
CHAPTER 2

General examples

Introductory examples.

Plotting random search behavior

Show the evolution of the current min with random search



Out:

```
best input : -0.00, best output : -2.00
```

```
from fluentopt import RandomSearch
import numpy as np
import matplotlib.pyplot as plt
from scipy import minimum

np.random.seed(42)


def sampler(rng):
    return rng.uniform(-1, 1)


def feval(x):
    return (x ** 2 - 2)

opt = RandomSearch(sampler=sampler)
n_iter = 100

for _ in range(n_iter):
    x = opt.suggest()
    y = feval(x)
    opt.update(x=x, y=y)

idx = np.argmin(opt.output_history_)
best_input = opt.input_history_[idx]
best_output = opt.output_history_[idx]
print('best input : {:.2f}, best output : {:.2f}'.format(best_input, best_
↪output))
iters = np.arange(len(opt.output_history_))
plt.plot(iters, minimum.accumulate(opt.output_history_))
plt.xlabel('iteration')
plt.ylabel('current $min_x({x**2-2})$')
plt.show()
```

Total running time of the script: (0 minutes 0.069 seconds)

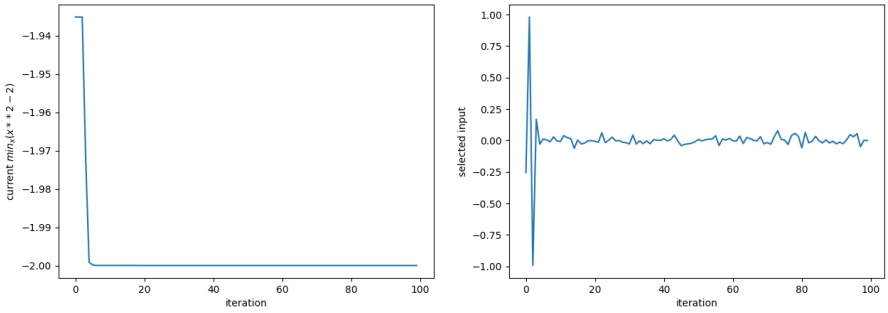
Download Python source code: plot_random_search.py

Download Jupyter notebook: plot_random_search.ipynb

Generated by Sphinx-Gallery

Plotting bandit ucb behavior

Show the evolution of the current min and the selected input with bandit ucb.



Out:

```
best input : 0.00, best output : -2.00
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import minimum

from fluentopt import Bandit
from fluentopt.bandit import ucb_minimize

np.random.seed(42)


def sampler(rng):
    return rng.uniform(-1, 1)


def feval(x):
    return (x ** 2 - 2)

opt = Bandit(sampler=sampler, score=ucb_minimize)
n_iter = 100
for _ in range(n_iter):
    x = opt.suggest()
    y = feval(x)
    opt.update(x=x, y=y)

idx = np.argmin(opt.output_history_)
best_input = opt.input_history_[idx]
best_output = opt.output_history_[idx]
print('best input : {:.2f}, best output : {:.2f}'.format(best_input, best_
↪output))
iters = np.arange(len(opt.output_history_))
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

ax1.plot(iters, minimum.accumulate(opt.output_history_))
ax1.set_xlabel('iteration')
ax1.set_ylabel('current $min_x({x**2-2})$')

ax2.plot(iters, opt.input_history_)
```

```
ax2.set_xlabel('iteration')
ax2.set_ylabel('selected input')

plt.show()
```

Total running time of the script: (0 minutes 2.127 seconds)

Download Python source code: plot_bandit_ucb.py

Download Jupyter notebook: plot_bandit_ucb.ipynb

Generated by Sphinx-Gallery

Hyperband

An example with usage of hyperband.

Out:

```
Total nb. evaluations : 206
Best nb. of iterations : 99
Best params : {'learning_rate': 0.19091103115034602, 'max_depth': 5}
Best mse : 6.416
R2 : 0.907
```

```
import numpy as np

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.datasets import load_boston

from fluentopt.hyperband import hyperband

if __name__ == '__main__':
    np.random.seed(42)
    data = load_boston()
    X = data['data']
    y = data['target']
    ind = np.arange(len(X))
    np.random.shuffle(ind)
    X = X[ind]
    y = y[ind]
    X_train = X[0:400]
    y_train = y[0:400]
    X_test = X[400:]
    y_test = y[400:]

    def sample(rng):
        return {'max_depth': rng.randint(1, 10), 'learning_rate': rng.
↪uniform(0, 1)}

    def run_batch(batch):
        for num_iters, params in batch:
```

```
            max_depth = params['max_depth']
            learning_rate = params['learning_rate']
            num_iters = int(num_iters)
            reg = GradientBoostingRegressor(
                learning_rate=learning_rate,
                max_depth=max_depth,
                n_estimators=num_iters)
            reg.fit(X_train, y_train)
            mse = ((reg.predict(X_test) - y_test)**2).mean()
            yield mse

    input_hist, output_hist = hyperband(sample, run_batch, max_iter=100,␣
↪random_state=42)
    idx = np.argmin(output_hist)
    nb_iter, params = input_hist[idx]
    mse = output_hist[idx]
    print('Total nb. evaluations : {}'.format(len(input_hist)))
    print('Best nb. of iterations : {}'.format(int(nb_iter)))
    print('Best params : {}'.format(params))
    print('Best mse : {:.3f}'.format(mse))
    r2 = 1.0 - mse / y_test.var()
    print('R2 : {:.3f}'.format(r2))
```

Total running time of the script: (0 minutes 2.784 seconds)

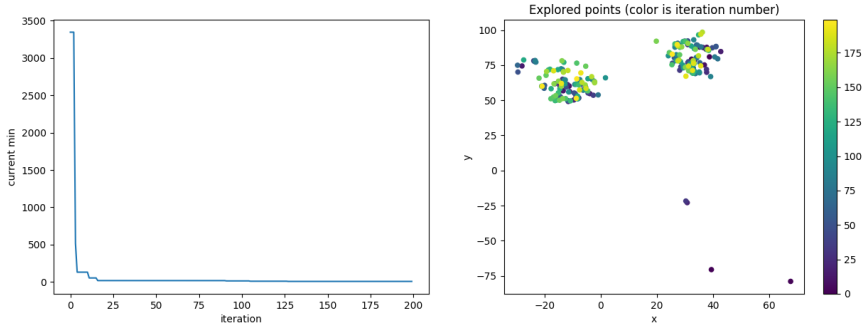
Download Python source code: plot_hyperband.py

Download Jupyter notebook: plot_hyperband.ipynb

Generated by Sphinx-Gallery

Dict format

An example with usage an input with a dict format.



Out:

```
best input : {'b': 58.1415013066466, 'a': -14.604190035269198}, best␣
↪output : 7.49
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import minimum

from fluentopt import Bandit
from fluentopt.bandit import ucb_minimize

np.random.seed(42)


def branin(a=1, b=5.1 / (4 * np.pi**2), c=5. / np.pi,
           r=6, s=10, t=1. / (8 * np.pi)):
    """Branin-Hoo function is defined on the square x1 ∈ [-5, 10], x2 ∈ [0,
↪15].
    It has three minima with f(x*) = 0.397887 at x* = (-pi, 12.275),
    (+pi, 2.275), and (9.42478, 2.475).
    More details: <http://www.sfu.ca/~ssurjano/branin.html>

    This code is adapted from : https://github.com/scikit-optimize/scikit-
↪optimize
    """
    def f(d):
        x, y = d['a'], d['b']
        return (a * (y - b * x ** 2 + c * x - r) ** 2 +
                s * (1 - t) * np.cos(x) + s)
    return f


def sampler(rng):
    return {'a': rng.uniform(-100, 100),
            'b': rng.uniform(-100, 100)}

feval = branin()

opt = Bandit(sampler=sampler, score=ucb_minimize)
n_iter = 200
for _ in range(n_iter):
    x = opt.suggest()
    y = feval(x)
    opt.update(x=x, y=y)

idx = np.argmin(opt.output_history_)
best_input = opt.input_history_[idx]
best_output = opt.output_history_[idx]
print('best input : {}, best output : {:.2f}'.format(best_input, best_
↪output))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
iters = np.arange(len(opt.output_history_))
ax1.plot(iters, minimum.accumulate(opt.output_history_))
ax1.set_xlabel('iteration')
ax1.set_ylabel('current min')
X = [[inp['a'], inp['b']] for inp in opt.input_history_]
X = np.array(X)

sc = ax2.scatter(X[:, 0], X[:, 1], c=iters, cmap='viridis', s=20)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
```

```
ax2.set_title('Explored points (color is iteration number)')
fig.colorbar(sc)
plt.show()
```

Total running time of the script: (0 minutes 3.553 seconds)

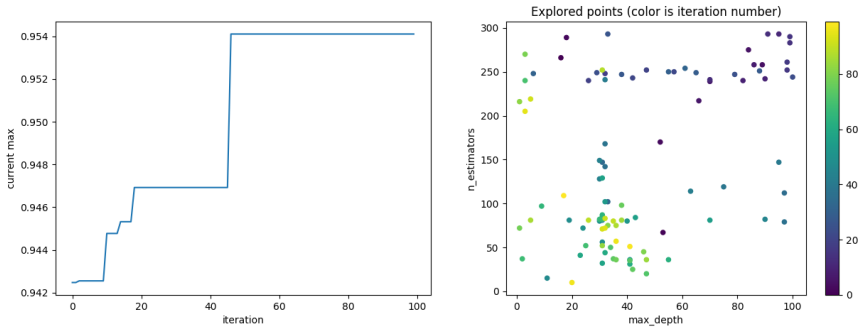
Download Python source code: plot_dict_format.py

Download Jupyter notebook: plot_dict_format.ipynb

Generated by Sphinx-Gallery

Hyper-parameter optimization example with random forests

An example with usage an input with a dict format.



Out:

```
iter 0...
iter 1...
iter 2...
iter 3...
iter 4...
iter 5...
iter 6...
iter 7...
iter 8...
iter 9...
iter 10...
iter 11...
iter 12...
iter 13...
iter 14...
iter 15...
iter 16...
iter 17...
iter 18...
iter 19...
iter 20...
iter 21...
iter 22...
iter 23...
iter 24...
```

```
iter 25...
iter 26...
iter 27...
iter 28...
iter 29...
iter 30...
iter 31...
iter 32...
iter 33...
iter 34...
iter 35...
iter 36...
iter 37...
iter 38...
iter 39...
iter 40...
iter 41...
iter 42...
iter 43...
iter 44...
iter 45...
iter 46...
iter 47...
iter 48...
iter 49...
iter 50...
iter 51...
iter 52...
iter 53...
iter 54...
iter 55...
iter 56...
iter 57...
iter 58...
iter 59...
iter 60...
iter 61...
iter 62...
iter 63...
iter 64...
iter 65...
iter 66...
iter 67...
iter 68...
iter 69...
iter 70...
iter 71...
iter 72...
iter 73...
iter 74...
iter 75...
iter 76...
iter 77...
iter 78...
iter 79...
iter 80...
iter 81...
iter 82...
```

```
iter 83...
iter 84...
iter 85...
iter 86...
iter 87...
iter 88...
iter 89...
iter 90...
iter 91...
iter 92...
iter 93...
iter 94...
iter 95...
iter 96...
iter 97...
iter 98...
iter 99...
best input : {'max_depth': 31, 'n_estimators': 81}, best output : 0.95
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import maximum

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

from fluentopt import Bandit
from fluentopt.bandit import ucb_maximize
from fluentopt.utils import RandomForestRegressorWithUncertainty
from fluentopt.transformers import Wrapper

from sklearn.datasets import load_breast_cancer

np.random.seed(42)

data = load_breast_cancer()
data_X, data_y = data['data'], data['target']


def sampler(rng):
    return {'max_depth': rng.randint(1, 100), 'n_estimators': rng.
↪randint(1, 300)}


def feval(d):
    max_depth = d['max_depth']
    n_estimators = d['n_estimators']
    clf = RandomForestClassifier(n_jobs=-1, max_depth=max_depth, n_
↪estimators=n_estimators)
    scores = cross_val_score(clf, data_X, data_y, cv=5, scoring='accuracy
↪')
    return np.mean(scores) - np.std(scores)
```

```
opt = Bandit(sampler=sampler, score=ucb_maximize,
             model=Wrapper(RandomForestRegressorWithUncertainty()))
n_iter = 100
for i in range(n_iter):
    print('iter {}...'.format(i))
    x = opt.suggest()
    y = feval(x)
    opt.update(x=x, y=y)

idx = np.argmax(opt.output_history_)
best_input = opt.input_history_[idx]
best_output = opt.output_history_[idx]
print('best input : {}, best output : {:.2f}'.format(best_input, best_
↪output))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
iters = np.arange(len(opt.output_history_))
ax1.plot(iters, maximum.accumulate(opt.output_history_))
ax1.set_xlabel('iteration')
ax1.set_ylabel('current max')
X = [[inp['max_depth'], inp['n_estimators']] for inp in opt.input_history_
↪]
X = np.array(X)

sc = ax2.scatter(X[:, 0], X[:, 1], c=iters, cmap='viridis', s=20)
ax2.set_xlabel('max_depth')
ax2.set_ylabel('n_estimators')
ax2.set_title('Explored points (color is iteration number)')
fig.colorbar(sc)
plt.show()
```

Total running time of the script: (4 minutes 30.655 seconds)

Download Python source code: plot_hyperopt.py

Download Jupyter notebook: plot_hyperopt.ipynb

Generated by Sphinx-Gallery

Download all examples in Python source code: auto_examples_python.zip

Download all examples in Jupyter notebooks: auto_examples_jupyter.zip

Generated by Sphinx-Gallery

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index