scikit-optimize Documentation

Release 0.9.0

The scikit-optimize Contributors.

Oct 12, 2021
CHAPTER ONE

WELCOME TO SCIKIT-OPTIMIZE

1.1 Installation

scikit-optimize requires:

- Python >= 3.6
- NumPy (>= 1.13.3)
- SciPy (>= 0.19.1)
- joblib (>= 0.11)
- scikit-learn >= 0.20
- matplotlib >= 2.0.0

The newest release can be installed via pip:

```bash
$ pip install scikit-optimize
```

or via conda:

```bash
$ conda install -c conda-forge scikit-optimize
```

The newest development version of scikit-optimize can be installed by:

```bash
$ pip install git+https://github.com/scikit-optimize/scikit-optimize.git
```

1.1.1 Development version

The library is still experimental and under heavy development. The development version can be installed through:

```bash
git clone https://github.com/scikit-optimize/scikit-optimize.git
cd scikit-optimize
pip install -r requirements.txt
python setup.py develop
```

Run the tests by executing `pytest` in the top level directory.
1.2 Release History

Release notes for all scikit-optimize releases are linked in this this page.

1.2.1 Version 0.9.0

October 2021

• [Fix] skopt.learning.gaussian_process.gpr.GaussianProcessRegressor for sklearn >= 0.23. #943
• Change skip= parameter in skopt.sampler.sobol.Sobol initial point generator. #955
• [Feature] skopt.callbacks.HollowIterationsStopper callback. #917
• [Feature] skopt.callbacks.ThresholdStopper callback. #1000
• [Fix] Fix skopt.searchcv.BayesSearchCV for scikit-learn >= 0.24. #988
• [API Change] Deprecate skopt.searchcv.BayesSearchCV parameter iid=. #988
• [Fix] NumPy deprecation errors. #1023
• [Fix] issue with skopt.optimizer.optimizer.Optimizer not being garbage-collectable. #1029
• [Fix] version check in skopt.learning.gaussian_process.gpr.GaussianProcessRegressor for scikit-learn >= 1.0. #1063
• Minor documentation improvements.
• Various small bugs and fixes.

1.2.2 Version 0.8.1

September 2020

• [Fix] GaussianProcessRegressor on sklearn 0.23 normalizes the variance to 1, which needs to reverted on predict.

1.2.3 Version 0.8.0

September 2020

skopt.Optimizer

• [Enhancement] n_jobs support was added to Optimizer and fixed for forest_minimize #884 by Holger Nahrstaedt based on #627 by JPN
skopt.plots

- [Enhancement] Allow dimension selection for plot_objective and plot_evaluations and add plot_histogram and plot_objective_2D. Plot code has been refactored. #848 by Holger Nahrstaedt based on #579 by Hvass-Labs

skopt.sampler

- [Major Feature] Initial sampling generation from latin hypercube, sobol, hammersly and halton is possible and can be set in all optimizers #835 by Holger Nahrstaedt
- [Enhancement] Improve sampler and add grid sampler #851 by Holger Nahrstaedt

skopt.searchcv

- [Fix] Fix library for scikit-learn >= 0.23. numpy MaskArray is replaced by numpy.ma.array. y_normalize=False has been added and initial runs has been increased. :pr: 939 by Lucas Plagwitz

skopt.space

- [Fix] Fix Integer transform and inverse_transform for normalize #880 by Holger Nahrstaedt
- [Enhancement] Add is_constant property to dimension and n_constant_dimensions property to Space #883 by Holger Nahrstaedt
- [Enhancement] Skip constant dimensions for plot_objective and plot_evaluations to allow plots using BayesSearchCV #888 by Holger Nahrstaedt

skopt.utils

- [Fix] Fix Optimizer for full categorical spaces #874 by Holger Nahrstaedt

Miscellaneous

- Improve circle ci #852 by Holger Nahrstaedt
- Add project toml and adapt minimal numpy, scipy, pyyaml and joblib version in setup.py #850 by Holger Nahrstaedt

1.2.4 Version 0.7.2

February 2020
skopt.optimizer

- [Feature] update_next() and get_results() added to Optimize and add more examples #837 by Holger Nahrstaedt and Sigurd Carlsen
- [Fix] Fix random forest regressor (Add missing min_impurity_decrease) #829 by Holger Nahrstaedt

skopt.utils

- [Enhancement] Add expected_minimum_random_sampling #830 by Holger Nahrstaedt
- [Fix] Return ordereddict in point_asdict and add some more unit tests. #840 by Holger Nahrstaedt
- [Enhancement] Added check_list_types and check_dimension_names #803 by Hvass-Labs and Holger Nahrstaedt

skopt.plots

- [Enhancement] Add more parameter to plot_objective and more plot examples #830 by Holger Nahrstaedt and Sigurd Carlsen

skopt.searchcv

- [Fix] Fix searchcv rank (issue #831) #832 by Holger Nahrstaedt

skopt.space

- [Fix] Fix integer normalize by using round() #830 by Holger Nahrstaedt

Miscellaneous

- [Fix] Fix doc examples
- [Fix] Fix license detection in github #827 by Holger Nahrstaedt
- [Enhancement] Add doctest to CI

1.2.5 Version 0.7.1

February 2020

skopt.space

- [Fix] Fix categorical space (issue #821) #823 by Holger Nahrstaedt
- [Enhancement] int can be set as dtype to fix issue #790 #807 by Holger Nahrstaedt
- [Feature] New StringEncoder, can be used in Categoricals
- Remove string conversion in Identity
- [Enhancement] dtype can be set in Integer and Real
1.2.6 Version 0.7

January 2020

**skopt.optimizer**

- [Enhancement] Models queue has now a customizable size (model_queue_size). #803 by Kajetan Tukendorf and Holger Nahrstaedt
- [Enhancement] Add log-uniform prior to Integer space #805 by Alex Liebscher

**skopt.plots**

- [Enhancement] Support for plotting categorical dimensions #806 by jkleint

**skopt.searchcv**

- [Fix] Allow BayesSearchCV to work with sklearn 0.21. #777 by Kit Choi

**Miscellaneous**

- [Fix] Reduce the amount of deprecation warnings in unit tests #808 by Holger Nahrstaedt
- [Fix] Reduce the amount of deprecation warnings in unit tests #802 by Alex Liebscher
- joblib instead of sklearn.externals.joblib #776 by Vince Jankovics
- Improve travis CI unit tests (Different sklearn version are checked) #804 by Holger Nahrstaedt
- Removed versioneer support, to keep things simple and to fix pypi deploy #816 by Holger Nahrstaedt

1.2.7 Version 0.6

Highly composite six.
New features

- `plot_regret` function for plotting the cumulative regret; The purpose of such plot is to access how much an optimizer is effective at picking good points.
- `CheckpointSaver` that can be used to save a checkpoint after each iteration with `skopt.dump`
- `Space.from_yaml()` to allow for external file to define Space parameters

Bug fixes

- Fixed numpy broadcasting issues in `gaussian_ei`, `gaussian_pi`
- Fixed build with newest scikit-learn
- Use native python types inside `BayesSearchCV`
- Include `fit_params` in `BayesSearchCV refit`

Maintenance

- Added `versioneer` support, to reduce changes with new version of the `skopt`

1.2.8 Version 0.5.2

Bug fixes

- Separated `n_points` from `n_jobs` in `BayesSearchCV`
- Dimensions now support boolean np.arrays.

Maintenance

- `matplotlib` is now an optional requirement (install with `pip install 'scikit-optimize[plots]'`)

1.2.9 Version 0.5

High five!

New features

- Single element dimension definition, which can be used to fix the value of a dimension during optimization.
- `total_iterations` property of `BayesSearchCV` that counts total iterations needed to explore all subspaces.
- Add iteration event handler for `BayesSearchCV`, useful for early stopping inside `BayesSearchCV search loop`.
- added `utils.use_named_args` decorator to help with unpacking named dimensions when calling an objective function.
Bug fixes

- Removed redundant estimator fitting inside BayesSearchCV.
- Fixed the log10 transform for Real dimensions that would lead to values being out of bounds.

1.2.10 Version 0.4

Go forth!

New features

- Support early stopping of optimization loop.
- Benchmarking scripts to evaluate performance of different surrogate models.
- Support for parallel evaluations of the objective function via several constant liar strategies.
- BayesSearchCV as a drop in replacement for scikit-learn’s GridSearchCV.
- New acquisition functions “ELps” and “PIps” that takes into account function compute time.

Bug fixes

- Fixed inference of dimensions of type Real.

API changes

- Change interface of GradientBoostingQuantileRegressor’s predict method to match return type of other regressors
- Dimensions of type Real are now inclusive of upper bound.

1.2.11 Version 0.3

Third time’s a charm.

New features

- Accuracy improvements of the optimization of the acquisition function by pre-selecting good candidates as starting points when using acq_optimizer='lbfgs'.
- Support a ask-and-tell interface. Check out the Optimizer class if you need fine grained control over the iterations.
- Parallelize L-BFGS minimization runs over the acquisition function.
- Implement weighted hamming distance kernel for problems with only categorical dimensions.
- New acquisition function gp_hedge that probabilistically chooses one of EI, PI or LCB at every iteration depending upon the cumulative gain.
Bug fixes

- Warnings are now raised if a point is chosen as the candidate optimum multiple times.
- Infinite gradients that were raised in the kernel gradient computation are now fixed.
- Integer dimensions are now normalized to [0, 1] internally in `gp_minimize`.

API Changes

- The default `acq_optimizer` function has changed from "auto" to "lbfgs" in `gp_minimize`.

1.2.12 Version 0.2

New features

- Speed improvements when using `gp_minimize` with `acq_optimizer='lbfgs'` and `acq_optimizer='auto'` when all the search-space dimensions are Real.
- Persistence of minimization results using `skopt.dump` and `skopt.load`.
- Support for using arbitrary estimators that implement a `return_std` argument in their `predict` method by means of `base_minimize` from `skopt.optimizer`.
- Support for tuning noise in `gp_minimize` using the `noise` argument.
- `TimerCallback` in `skopt.callbacks` to log the time between iterations of the minimization loop.

1.2.13 Version 0.1

First light!

New features

- Bayesian optimization via `gp_minimize`.
- Tree-based sequential model-based optimization via `forest_minimize` and `gbrt_minimize`, with support for multi-threading.
- Support of LCB, EI and PI as acquisition functions.
- Plotting functions for inspecting convergence, evaluations and the objective function.
- API for specifying and sampling from a parameter space.
Scikit-Optimize, or skopt, is a simple and efficient library to minimize (very) expensive and noisy black-box functions. It implements several methods for sequential model-based optimization. skopt aims to be accessible and easy to use in many contexts.

The library is built on top of NumPy, SciPy and Scikit-Learn.

We do not perform gradient-based optimization. For gradient-based optimization algorithms look at scipy.optimize here.

Approximated objective function after 50 iterations of gp_minimize. Plot made using plots.plot_objective.

### 2.1 Finding a minimum

Find the minimum of the noisy function $f(x)$ over the range $-2 < x < 2$ with skopt:

```python
>>> import numpy as np
>>> from skopt import gp_minimize
>>> np.random.seed(123)
>>> def f(x):
...     return (np.sin(5 * x[0]) * (1 - np.tanh(x[0] ** 2)) *
...             np.random.randn() * 0.1)
... >>> res = gp_minimize(f, \[-2.0, 2.0\], n_calls=20)
>>> print("x*=%.2f f(x*)=%.2f" % (res.x[0], res.fun))
x*=-0.85  f(x*)=-0.06
```

For more control over the optimization loop you can use the skopt.Optimizer class:

```python
>>> from skopt import Optimizer
>>> opt = Optimizer([\[-2.0, 2.0\]])
>>> for i in range(20):
...     suggested = opt.ask()
...     y = f(suggested)
...     res = opt.tell(suggested, y)
>>> print("x*=%.2f f(x*)=%.2f" % (res.x[0], res.fun))
x*=0.27  f(x*)=-0.15
```

For more read our Bayesian optimization with skopt and the other examples.
3.1 Acquisition

Function to minimize over the posterior distribution.

3.1.1 gaussian_lcb

Use the lower confidence bound to estimate the acquisition values.
The trade-off between exploitation and exploration is left to be controlled by the user through the parameter $kappa$.

3.1.2 gaussian_pi

Use the probability of improvement to calculate the acquisition values.
The conditional probability $P(y=f(x) \mid x)$ form a gaussian with a certain mean and standard deviation approximated by the model.
The PI condition is derived by computing $E[u(f(x))]$ where \( u(f(x)) = 1 \), if \( f(x) < y_{\text{opt}} \) and \( u(f(x)) = 0 \), if \( f(x) > y_{\text{opt}} \).
This means that the PI condition does not care about how “better” the predictions are than the previous values, since it gives an equal reward to all of them.
Note that the value returned by this function should be maximized to obtain the $X$ with maximum improvement.

3.1.3 gaussian_ei

Use the expected improvement to calculate the acquisition values.
The conditional probability $P(y=f(x) \mid x)$ form a gaussian with a certain mean and standard deviation approximated by the model.
The EI condition is derived by computing $E[u(f(x))]$ where \( u(f(x)) = 0 \), if \( f(x) > y_{\text{opt}} \) and \( u(f(x)) = y_{\text{opt}} - f(x) \), if \( f(x) < y_{\text{opt}} \).
This solves one of the issues of the PI condition by giving a reward proportional to the amount of improvement got.
Note that the value returned by this function should be maximized to obtain the $X$ with maximum improvement.
3.2 BayesSearchCV, a GridSearchCV compatible estimator

Use BayesSearchCV as a replacement for scikit-learn’s GridSearchCV.
BayesSearchCV implements a “fit” and a “score” method. It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.
The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.
In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.
Parameters are presented as a list of skopt.space.Dimension objects.

3.3 Callbacks

Monitor and influence the optimization procedure via callbacks.
Callbacks are callables which are invoked after each iteration of the optimizer and are passed the results “so far”. Callbacks can monitor progress, or stop the optimization early by returning True.

3.3.1 Monitoring callbacks

• VerboseCallback
• TimerCallback

3.3.2 Early stopping callbacks

• DeltaXStopper
• DeadlineStopper
• DeltaXStopper
• DeltaYStopper
• EarlyStopper

3.3.3 Other callbacks

• CheckpointSaver
3.4 Optimizer, an ask-and-tell interface

Use the `Optimizer` class directly when you want to control the optimization loop. We refer to this as the ask-and-tell interface. This class is used internally to implement the `skopt’s top level minimization functions`.

3.5 `skopt’s top level minimization functions`

These are easy to get started with. They mirror the `scipy.optimize` API and provide a high level interface to various pre-configured optimizers.

3.5.1 dummy_minimize

Random search by uniform sampling within the given bounds.

3.5.2 forest_minimize

Sequential optimisation using decision trees.

A tree based regression model is used to model the expensive to evaluate function `func`. The model is improved by sequentially evaluating the expensive function at the next best point. Thereby finding the minimum of `func` with as few evaluations as possible.

3.5.3 gbrt_minimize

Sequential optimization using gradient boosted trees.

Gradient boosted regression trees are used to model the (very) expensive to evaluate function `func`. The model is improved by sequentially evaluating the expensive function at the next best point. Thereby finding the minimum of `func` with as few evaluations as possible.

3.5.4 gp_minimize

Bayesian optimization using Gaussian Processes.

If every function evaluation is expensive, for instance when the parameters are the hyperparameters of a neural network and the function evaluation is the mean cross-validation score across ten folds, optimizing the hyperparameters by standard optimization routines would take for ever!

The idea is to approximate the function using a Gaussian process. In other words the function values are assumed to follow a multivariate gaussian. The covariance of the function values are given by a GP kernel between the parameters. Then a smart choice to choose the next parameter to evaluate can be made by the acquisition function over the Gaussian prior which is much quicker to evaluate.
3.6 Plotting tools

Plotting functions can be used to visualize the optimization process.

3.6.1 plot_convergence

`plot_convergence` plots one or several convergence traces.

3.6.2 plot_evaluations

`plot_evaluations` visualize the order in which points where sampled.
3.6.3 plot_objective

`plot_objective` creates pairwise dependence plot of the objective function.

3.6.4 plot_regret

`plot_regret` plot one or several cumulative regret traces.

3.7 Space

`Space` define the optimization space which contains one or multiple dimensions of the following type:

3.7.1 Real

Search space dimension that can take on any real value.
3.7.2 Integer

Search space dimension that can take on integer values.

3.7.3 Categorical

Search space dimension that can take on categorical values.

3.8 Utility functions

This is a list of public utility functions. Other functions in this module are meant for internal use.

3.8.1 use_named_args()

This utility function allows it to use objective functions with named arguments:

```python
>>> # Define the search-space dimensions. They must all have names!
>>> from skopt.space import Real
>>> from skopt.utils import use_named_args
>>> dim1 = Real(name='foo', low=0.0, high=1.0)
>>> dim2 = Real(name='bar', low=0.0, high=1.0)
>>> dim3 = Real(name='baz', low=0.0, high=1.0)
>>> # Gather the search-space dimensions in a list.
>>> dimensions = [dim1, dim2, dim3]
>>> # Define the objective function with named arguments
>>> # and use this function-decorator to specify the
>>> # search-space dimensions.
>>> @use_named_args(dimensions=dimensions)
... def my_objective_function(foo, bar, baz):
...     return foo ** 2 + bar ** 4 + baz ** 8
```

3.8.2 dump()

Store an skopt optimization result into a file.

3.8.3 load()

Reconstruct a skopt optimization result from a file persisted with `dump()`.
4.1 Miscellaneous examples

Miscellaneous and introductory examples for scikit-optimize.

4.1.1 Parallel optimization

Iaroslav Shcherbutyi, May 2017. Reviewed by Manoj Kumar and Tim Head. Reformatted by Holger Nahrstaedt 2020

Introduction

For many practical black box optimization problems expensive objective can be evaluated in parallel at multiple points. This allows to get more objective evaluations per unit of time, which reduces the time necessary to reach good objective values when appropriate optimization algorithms are used, see for example results in\(^1\) and the references therein.

One such example task is a selection of number and activation function of a neural network which results in highest accuracy for some machine learning problem. For such task, multiple neural networks with different combinations of number of neurons and activation function type can be evaluated at the same time in parallel on different cpu cores / computational nodes.

The “ask and tell” API of scikit-optimize exposes functionality that allows to obtain multiple points for evaluation in parallel. Intended usage of this interface is as follows:

1. Initialize instance of the Optimizer class from skopt
2. Obtain n points for evaluation in parallel by calling the ask method of an optimizer instance with the n_points argument set to \( n > 0 \)
3. Evaluate points
4. Provide points and corresponding objectives using the tell method of an optimizer instance
5. Continue from step 2 until eg maximum number of evaluations reached

```
print(__doc__)
import numpy as np
```

\(^1\) https://hal.archives-ouvertes.fr/hal-00732512/document
Example

A minimalistic example that uses joblib to parallelize evaluation of the objective function is given below.

```python
from skopt import Optimizer
from skopt.space import Real
from joblib import Parallel, delayed

# example objective taken from skopt
from skopt.benchmarks import branin

optimizer = Optimizer(
    dimensions=[Real(-5.0, 10.0), Real(0.0, 15.0)],
    random_state=1,
    base_estimator='gp',
)

for i in range(10):
    x = optimizer.ask(n_points=4)  # x is a list of n_points points
    y = Parallel(n_jobs=4)(delayed(branin)(v) for v in x)  # evaluate points in parallel
    optimizer.tell(x, y)

# takes ~ 20 sec to get here
print(min(optimizer.yi))  # print the best objective found
```

Out:

```
0.3982974723981023
```

Note that if n_points is set to some integer > 0 for the ask method, the result will be a list of points, even for n_points = 1. If the argument is set to None (default value) then a single point (but not a list of points) will be returned.

The default “minimum constant liar” parallelization strategy is used in the example, which allows to obtain multiple points for evaluation with a single call to the ask method with any surrogate or acquisition function. Parallelization strategy can be set using the “strategy” argument of ask. For supported parallelization strategies see the documentation of scikit-optimize.

**Total running time of the script:** ( 0 minutes 28.444 seconds)

**Estimated memory usage:** 31 MB

### 4.1.2 Store and load skopt optimization results

Mikhail Pak, October 2016. Reformatted by Holger Nahrstaedt 2020

**Problem statement**

We often want to store optimization results in a file. This can be useful, for example,

- if you want to share your results with colleagues;
- if you want to archive and/or document your work;
- or if you want to postprocess your results in a different Python instance or on another computer.

The process of converting an object into a byte stream that can be stored in a file is called _serialization_. Conversely, _deserialization_ means loading an object from a byte stream.
**Warning:** Deserialization is not secure against malicious or erroneous code. Never load serialized data from untrusted or unauthenticated sources!

```python
print(__doc__)
import numpy as np
import os
import sys

Simple example

We will use the same optimization problem as in the *Bayesian optimization with skopt* notebook:

```python
from skopt import gp_minimize
noise_level = 0.1

def obj_fun(x, noise_level=noise_level):
    return np.sin(5 * x[0]) * (1 - np.tanh(x[0] ** 2)) + np.random.randn() * noise_level

res = gp_minimize(obj_fun,
                  # the function to minimize
                  [(-2.0, 2.0)],
                  # the bounds on each dimension of x
                  x0=[0.],
                  # the starting point
                  acq_func="LCB",
                  # the acquisition function (optional)
                  n_calls=15,
                  # the number of evaluations of f including at x0
                  n_random_starts=3,
                  # the number of random initial points
                  random_state=777)

Out:

/home/circleci/project/skopt/optimizer/optimizer.py:449: UserWarning: The objective has been evaluated at this point before.
warnings.warn("The objective has been evaluated ")

As long as your Python session is active, you can access all the optimization results via the `res` object.

So how can you store this data in a file? `skopt` conveniently provides functions `skopt.dump` and `skopt.load` that handle this for you. These functions are essentially thin wrappers around the `joblib` module’s `joblib.dump` and `joblib.load`.

We will now show how to use `skopt.dump` and `skopt.load` for storing and loading results.

**Using skopt.dump() and skopt.load()**

For storing optimization results into a file, call the `skopt.dump` function:

```python
from skopt import dump, load
dump(res, 'result.pkl')
```

And load from file using `skopt.load`:
res_loaded = load('result.pkl')

res_loaded.fun

Out:
-1.157882608320259

You can fine-tune the serialization and deserialization process by calling `skopt.dump` and `skopt.load` with additional keyword arguments. See the joblib documentation `joblib.dump` and `joblib.load` for the additional parameters.

For instance, you can specify the compression algorithm and compression level (highest in this case):

dump(res, 'result.gz', compress=9)

from os.path import getsize
print('Without compression: {} bytes'.format(getsize('result.pkl')))
print('Compressed with gz: {} bytes'.format(getsize('result.gz')))

Out:
Without compression: 74076 bytes
Compressed with gz: 26987 bytes

### Unserializable objective functions

Notice that if your objective function is non-trivial (e.g. it calls MATLAB engine from Python), it might be not serializable and `skopt.dump` will raise an exception when you try to store the optimization results. In this case you should disable storing the objective function by calling `skopt.dump` with the keyword argument `store_objective=False`:

dump(res, 'result_without_objective.pkl', store_objective=False)

Notice that the entry 'func' is absent in the loaded object but is still present in the local variable:

res_loaded_without_objective = load('result_without_objective.pkl')

print('Loaded object:', res_loaded_without_objective.specs['args'].keys())
print('Local variable:', res.specs['args'].keys())

Out:

Loaded object: dict_keys(['dimensions', 'base_estimator', 'n_calls', 'n_random_starts',
    'n_initial_points', 'initial_point_generator', 'acq_func', 'acq_optimizer', 'x0', 'y0',
    'random_state', 'verbose', 'callback', 'n_points', 'n_restarts_optimizer', 'xi',
    'kappa', 'n_jobs', 'model_queue_size'])
Local variable: dict_keys(['func', 'dimensions', 'base_estimator', 'n_calls', 'n_random_starts',
    'n_initial_points', 'initial_point_generator', 'acq_func', 'acq_optimizer',
    'x0', 'y0', 'random_state', 'verbose', 'callback', 'n_points', 'n_restarts_optimizer',
    'xi', 'kappa', 'n_jobs', 'model_queue_size'])
Possible problems

- **Python versions incompatibility:** In general, objects serialized in Python 2 cannot be deserialized in Python 3 and vice versa.

- **Security issues:** Once again, do not load any files from untrusted sources.

- **Extremely large results objects:** If your optimization results object is extremely large, calling `skopt.dump` with `store_objective=False` might cause performance issues. This is due to creation of a deep copy without the objective function. If the objective function is not critical to you, you can simply delete it before calling `skopt.dump`. In this case, no deep copy is created:

```python
del res.specs['args']['func']
dump(res, 'result_without_objective_2.pkl')
```

**Total running time of the script:** (0 minutes 3.282 seconds)

**Estimated memory usage:** 14 MB

### 4.1.3 Interruptible optimization runs with checkpoints

Christian Schell, Mai 2018 Reformatted by Holger Nahrstaedt 2020

**Problem statement**

Optimization runs can take a very long time and even run for multiple days. If for some reason the process has to be interrupted results are irreversibly lost, and the routine has to start over from the beginning.

With the help of the `callbacks.CheckpointSaver` callback the optimizer’s current state can be saved after each iteration, allowing to restart from that point at any time.

This is useful, for example,

- if you don’t know how long the process will take and cannot hog computational resources forever
- if there might be system failures due to shaky infrastructure (or colleagues…)
- if you want to adjust some parameters and continue with the already obtained results

```python
print(__doc__)
import sys
import numpy as np
np.random.seed(777)
import os
```
**Simple example**

We will use pretty much the same optimization problem as in the *Bayesian optimization with skopt* notebook. Additionally we will instantiate the `callbacks.CheckpointSaver` and pass it to the minimizer:

```
from skopt import gp_minimize
from skopt import callbacks
from skopt.callbacks import CheckpointSaver

noise_level = 0.1

def obj_fun(x, noise_level=noise_level):
    return np.sin(5 * x[0]) * (1 - np.tanh(x[0]**2)) + np.random.randn() * noise_level

checkpoint_saver = CheckpointSaver('./checkpoint.pkl', compress=9)  # keyword arguments will be passed to `skopt.dump`

gp_minimize(obj_fun,  # the function to minimize
            [(-20.0, 20.0)],  # the bounds on each dimension of x
            x0=[-20.],  # the starting point
            acq_func="LCB",  # the acquisition function (optional)
            n_calls=10,  # number of evaluations of f including at x0
            n_random_starts=3,  # the number of random initial points
            callback=[checkpoint_saver],  # a list of callbacks including the checkpoint saver
            random_state=777)
```

Out:

```
fun: -0.17524445239614728
func_vals: array([-0.04682088, -0.08228249, -0.00653801, -0.07133619, 0.09063509, 0.07662367, 0.08260541, -0.13236828, -0.17524445, 0.10024491])
models: [GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=1), n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=65685735)]
```

(continues on next page)
n_restarts_optimizer=2, noise='gaussian',
normalize_y=True, random_state=655685735),
GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
WhiteKernel(noise_level=1),
n_restarts_optimizer=2, noise='gaussian',
normalize_y=True, random_state=655685735)

random_state: RandomState(MT19937) at 0x7F54AF59CB40
space: Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize')])
specs: {
'args': {
'func': <function obj_fun at 0x7f54ae7cbe50>,
'dimensions':
Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize')]),
'base_estimator': GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5),
n_restarts_optimizer=2, noise='gaussian',
normalize_y=True, random_state=655685735),
n_calls': 10, 'n_random_starts': 3, 'n_initial_points': 10, 'initial_point_generator': 'random', 'acq_func': 'LCB', 'acq_optimizer': 'auto', 'x0': [-20.0], 'y0': None, 'random_state':
RandomState(MT19937) at 0x7f54af59cb40, 'verbose': False, 'callback': [<skopt.callbacks.CheckpointSaver object at 0x7f54aeea2460>], 'n_points': 10000, 'n_restarts_optimizer': 5, 'xi': 0.01, 'kappa': 1.96, 'n_jobs': 1, 'model_queue_size': None},
'function': 'base_minimize'}
x: [-18.660711622818603]
x_iters: [[-20.0], [5.857990176187936], [-11.97095004855501], [5.450171667295798],
[10.524218484749973], [-17.111120867646513], [7.251301450238415], [-19.16709880491886],
[-18.660711622818603], [-18.284297243496296]]

Now let's assume this did not finish at once but took some long time: you started this on Friday night, went out for the weekend and now, Monday morning, you're eager to see the results. However, instead of the notebook server you only see a blank page and your colleague Garry tells you that he had had an update scheduled for Sunday noon – who doesn't like updates?

gp_minimize did not finish, and there is no res variable with the actual results!

**Restoring the last checkpoint**

Luckily we employed the callbacks.CheckpointSaver and can now restore the latest result with skopt.load (see Store and load skopt optimization results for more information on that)

```python
from skopt import load

res = load('./checkpoint.pkl')

res.fun
```

Out:

```
-0.17524445239614728
```
The previous results can then be used to continue the optimization process:

```python
x0 = res.x_iters
y0 = res.func_vals

gp_minimize(obj_fun,  # the function to minimize
            [(-20.0, 20.0)],  # the bounds on each dimension of x
            x0=x0,  # already examined values for x
            y0=y0,  # observed values for x0
            acq_func="LCB",  # the acquisition function (optional)
            n_calls=10,  # number of evaluations of f including at x0
            n_random_starts=3,  # the number of random initialization points
            callback=[checkpoint_saver],
            random_state=777)
```

Out:

```
fun: -0.17524445239614728
func_vals: array([-0.04682088, -0.08228249, -0.00653801, -0.07133619, 0.09063509,
                    0.07662367, 0.08260541, -0.13236828, -0.17524445, 0.10024491,
                    0.05448095, 0.18951609, -0.07693575, -0.14030959, -0.06324675,
                    -0.05588737, -0.12323214, -0.04395035, 0.09147873, 0.02650409])
models: [GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735),
         GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) +
                                     WhiteKernel(noise_level=1),
                                     n_restarts_optimizer=2, noise='gaussian',
                                     normalize_y=True, random_state=655685735)]
```

(continues on next page)
n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=655685735)

random_state: RandomState(MT19937) at 0x7F54AF59CB40
space: Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize')])
specs: {'args': {'func': <function obj_fun at 0x7f54ae7cbe50>, 'dimensions':
  Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize')]), 'base_estimator': GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5),
  n_restarts_optimizer=2, noise='gaussian', normalize_y=True, random_state=655685735), 'n_calls': 10, 'n_
  random_starts': 3, 'n_initial_points': 10, 'initial_point_generator': 'random', 'acq_
  func': 'LCB', 'acq_optimizer': 'auto', 'x0': [[-20.0], [5.857990176187936], [-11.97095004855501], [5.450171667295798], [10.524218484749973], [-17.111120867646513], [7.251301450238415], [-19.16709880491886], [-18.660711622818603], [-18.284297243496926]],
'y0': array([-0.04682088, -0.08228249, -0.00653801, -0.07133619, 0.09063509, 0.07662367, 0.08260541, -0.13236828, -0.17524445, 0.10024491]), 'random_state':
RandomState(MT19937) at 0x7F54AF59CB40, 'verbose': False, 'callback': [<skopt.callbacks.CheckpointSaver object at 0x7f54aeea2460>], 'n_points': 10000, 'n_restarts_
optimizer': 5, 'xi': 0.01, 'kappa': 1.96, 'n_jobs': 1, 'model_queue_size': None},
  'function': 'base_minimize'}
x: [-18.660711622818603]
x_iters: [[-20.0], [5.857990176187936], [-11.97095004855501], [5.450171667295798], [10.524218484749973], [-17.111120867646513], [7.251301450238415], [-19.16709880491886], [-18.660711622818603], [-18.284297243496926], [5.857990176187936], [-11.97095004855501], [5.450171667295798], [19.095152570513417], [-18.99431276746093], [-19.391720111674047], [-18.851948436512373]]

Possible problems

- **changes in search space**: You can use this technique to interrupt the search, tune the search space and continue the optimization. Note that the optimizers will complain if x0 contains parameter values not covered by the dimension definitions, so in many cases shrinking the search space will not work without deleting the offending runs from x0 and y0.
- **see Store and load skopt optimization results**

for more information on how the results get saved and possible caveats

**Total running time of the script:** ( 0 minutes 3.468 seconds)

**Estimated memory usage:** 17 MB

### 4.1.4 Tuning a scikit-learn estimator with skopt

Gilles Louppe, July 2016 Katie Malone, August 2016 Reformatted by Holger Nahrstaedt 2020

If you are looking for a sklearn.model_selection.GridSearchCV replacement checkout Scikit-learn hyperparameter search wrapper instead.

---

## 4.1. Miscellaneous examples

---
Problem statement

Tuning the hyper-parameters of a machine learning model is often carried out using an exhaustive exploration of (a subset of) the space all hyper-parameter configurations (e.g., using `sklearn.model_selection.GridSearchCV`), which often results in a very time consuming operation.

In this notebook, we illustrate how to couple `gp_minimize` with sklearn’s estimators to tune hyper-parameters using sequential model-based optimisation, hopefully resulting in equivalent or better solutions, but within fewer evaluations.

Note: scikit-optimize provides a dedicated interface for estimator tuning via `BayesSearchCV` class which has a similar interface to those of `sklearn.model_selection.GridSearchCV`. This class uses functions of skopt to perform hyperparameter search efficiently. For example usage of this class, see `Scikit-learn hyperparameter search wrapper` example notebook.

```python
print(__doc__)
import numpy as np
```

Objective

To tune the hyper-parameters of our model we need to define a model, decide which parameters to optimize, and define the objective function we want to minimize.

```python
from sklearn.datasets import load_boston
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import cross_val_score

boston = load_boston()
X, y = boston.data, boston.target
n_features = X.shape[1]

# gradient boosted trees tend to do well on problems like this
reg = GradientBoostingRegressor(n_estimators=50, random_state=0)
```

Out:

```
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-˓
→linux-x86_64.egg/sklearn/utils/deprecation.py:87: FutureWarning: Function load_boston␣
˓
→is deprecated; `load_boston` is deprecated in 1.0 and will be removed in 1.2.

The Boston housing prices dataset has an ethical problem. You can refer to the documentation of this function for further details.

The scikit-learn maintainers therefore strongly discourage the use of this dataset unless the purpose of the code is to study and educate about ethical issues in data science and machine learning.

In this case special case, you can fetch the dataset from the original source:

```python
import pandas as pd
import numpy as np

data_url = "http://lib.stat.cmu.edu/datasets/boston"
```

(continues on next page)
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])target = raw_df.values[1::2, 2]

Alternative datasets include the California housing dataset (i.e. func: `~sklearn.datasets.fetch_california_housing`) and the Ames housing dataset. You can load the datasets as follows:

```python
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
```

for the California housing dataset and:

```python
from sklearn.datasets import fetch_openml
housing = fetch_openml(name="house_prices", as_frame=True)
```

for the Ames housing dataset.

warnings.warn(msg, category=FutureWarning)

Next, we need to define the bounds of the dimensions of the search space we want to explore and pick the objective. In this case the cross-validation mean absolute error of a gradient boosting regressor over the Boston dataset, as a function of its hyper-parameters.

```python
from skopt.space import Real, Integer
from skopt.utils import use_named_args

# The list of hyper-parameters we want to optimize. For each one we define the
# bounds, the corresponding scikit-learn parameter name, as well as how to
# sample values from that dimension ("log-uniform" for the learning rate)
space = [
    Integer(1, 5, name='max_depth'),
    Real(1e-5, 1e0, "log-uniform", name='learning_rate'),
    Integer(1, n_features, name='max_features'),
    Integer(2, 100, name='min_samples_split'),
    Integer(1, 100, name='min_samples_leaf')
]

# this decorator allows your objective function to receive a the parameters as
# keyword arguments. This is particularly convenient when you want to set
# scikit-learn estimator parameters
@use_named_args(space)
def objective(**params):
    reg.set_params(**params)

    return -np.mean(cross_val_score(reg, X, y, cv=5, n_jobs=-1,
                                     scoring="neg_mean_absolute_error"))
```

4.1. Miscellaneous examples

29
Optimize all the things!

With these two pieces, we are now ready for sequential model-based optimisation. Here we use gaussian process-based optimisation.

```python
from skopt import gp_minimize
res_gp = gp_minimize(objective, space, n_calls=50, random_state=0)
"Best score=%.4f" % res_gp.fun
```

Out:

```
'Best score=2.9062'
```

```python
print("\n**Best parameters:**
- max_depth=%d
- learning_rate=%.6f
- max_features=%d
- min_samples_split=%d
- min_samples_leaf=%d"
% (res_gp.x[0], res_gp.x[1],
    res_gp.x[2], res_gp.x[3],
    res_gp.x[4]))
```

Out:

```
Best parameters:
- max_depth=5
- learning_rate=0.143650
- max_features=9
- min_samples_split=100
- min_samples_leaf=1
```

Convergence plot

```python
from skopt.plots import plot_convergence
plot_convergence(res_gp)
```
Bayesian optimization is used to tune parameters for walking robots or other experiments that are not a simple (expensive) function call.

Tim Head, February 2017. Reformatted by Holger Nahrstaedt 2020

They often follow a pattern a bit like this:

1. ask for a new set of parameters
2. walk to the experiment and program in the new parameters
3. observe the outcome of running the experiment
4. walk back to your laptop and tell the optimizer about the outcome
5. go to step 1
A setup like this is difficult to implement with the *minimize() function interface. This is why scikit-optimize has an ask-and-tell interface that you can use when you want to control the execution of the optimization loop.

This notebook demonstrates how to use the ask and tell interface.

```python
print(__doc__)

import numpy as np
np.random.seed(1234)
import matplotlib.pyplot as plt
from skopt.plots import plot_gaussian_process

The Setup

We will use a simple 1D problem to illustrate the API. This is a little bit artificial as you normally would not use the ask-and-tell interface if you had a function you can call to evaluate the objective.

```python
from skopt.learning import ExtraTreesRegressor
from skopt import Optimizer

noise_level = 0.1
```

Our 1D toy problem, this is the function we are trying to minimize

```python
def objective(x, noise_level=noise_level):
    return np.sin(5 * x[0]) * (1 - np.tanh(x[0] ** 2)) + np.random.randn() * noise_level

def objective_wo_noise(x, noise_level=0):
    return objective(x, noise_level=0)
```

Here a quick plot to visualize what the function looks like:

```python
# Plot f(x) + contours
plt.set_cmap("viridis")
x = np.linspace(-2, 2, 400).reshape(-1, 1)
fx = np.array([objective(x_i, noise_level=0.0) for x_i in x])
plt.plot(x, fx, "r--", label="True (unknown)")
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate((fx_i - 1.9600 * noise_level for fx_i in fx,
                         [fx_i + 1.9600 * noise_level for fx_i in fx[::-1]])),
         alpha=.2, fc="r", ec="None")
plt.legend()
plt.grid()
plt.show()
```
Now we setup the `Optimizer` class. The arguments follow the meaning and naming of the `_minimize()` functions. An important difference is that you do not pass the objective function to the optimizer.

```python
opt = Optimizer([(-2.0, 2.0)], "GP", acq_func="EI",
                acq_optimizer="sampling",
                initial_point_generator="lhs")

# To obtain a suggestion for the point at which to evaluate the objective
# you call the ask() method of opt:
next_x = opt.ask()
print(next_x)
```

Out:

```
[-0.7315058981975282]
```

In a real world use case you would probably go away and use this parameter in your experiment and come back a while later with the result. In this example we can simply evaluate the objective function and report the value back to the optimizer:

```python
f_val = objective(next_x)
opt.tell(next_x, f_val)
```

Out:
Like `minimize()` the first few points are suggestions from the initial point generator as there is no data yet with which to fit a surrogate model.

```python
def objective(x):
    return x ** 2 + 2 * x + 1

for i in range(9):
    next_x = opt.ask()
    f_val = objective(next_x)
    res = opt.tell(next_x, f_val)
```

We can now plot the random suggestions and the first model that has been fit:

```python
_ = plot_gaussian_process(res, objective=objective_wo_noise,
                          noise_level=noise_level,
                          show_next_point=False,
                          show_acq_func=True)
plt.show()
```
Let us sample a few more points and plot the optimizer again:

```python
for i in range(10):
    next_x = opt.ask()
    f_val = objective(next_x)
    res = opt.tell(next_x, f_val)

_ = plot_gaussian_process(res, objective=objective_wo_noise,
                          noise_level=noise_level,
                          show_next_point=True,
                          show_acq_func=True)
plt.show()
```
By using the `Optimizer` class directly you get control over the optimization loop.

You can also pickle your `Optimizer` instance if you want to end the process running it and resume it later. This is handy if your experiment takes a very long time and you want to shutdown your computer in the meantime:

```python
import pickle

with open('my-optimizer.pkl', 'wb') as f:
    pickle.dump(opt, f)

with open('my-optimizer.pkl', 'rb') as f:
    opt_restored = pickle.load(f)
```

Total running time of the script: (0 minutes 3.124 seconds)

Estimated memory usage: 15 MB
4.1.6 Comparing surrogate models

Tim Head, July 2016. Reformatted by Holger Nahrstaedt 2020

Bayesian optimization or sequential model-based optimization uses a surrogate model to model the expensive to evaluate function \( \text{func} \). There are several choices for what kind of surrogate model to use. This notebook compares the performance of:

- gaussian processes,
- extra trees, and
- random forests

as surrogate models. A purely random optimization strategy is also used as a baseline.

```python
print(__doc__)
import numpy as np
np.random.seed(123)
import matplotlib.pyplot as plt

def branin(x, noise_level=0.):
    return _branin(x) + noise_level * np.random.randn()

from matplotlib.colors import LogNorm

def plot_branin():
    fig, ax = plt.subplots()
    x1_values = np.linspace(-5, 10, 100)
    x2_values = np.linspace(0, 15, 100)
    x_ax, y_ax = np.meshgrid(x1_values, x2_values)
    vals = np.c_[x_ax.ravel(), y_ax.ravel()]
    fx = np.reshape([branin(val) for val in vals], (100, 100))

    cm = ax.pcolormesh(x_ax, y_ax, fx,
                      norm=LogNorm(vmin=fx.min(), vmax=fx.max()),
                      cmap='viridis_r')

    minima = np.array([[-np.pi, 12.275], [+np.pi, 2.275], [9.42478, 2.475]])
    ax.plot(minima[:, 0], minima[:, 1], "r.", markersize=14,
            lw=0, label="Minima")

    cb = fig.colorbar(cm)
    cb.set_label("f(x)")

    return fig
```

(continues on next page)
This shows the value of the two-dimensional branin function and the three minima.
Objective

The objective of this example is to find one of these minima in as few iterations as possible. One iteration is defined as one call to the `benchmarks.branin` function.

We will evaluate each model several times using a different seed for the random number generator. Then compare the average performance of these models. This makes the comparison more robust against models that get “lucky”.

```python
going = partial(branin, noise_level=2.0)
bounds = [(-5.0, 10.0), (0.0, 15.0)]
n_calls = 60

def run(minimizer, n_iter=5):
    return [minimizer(func, bounds, n_calls=n_calls, random_state=n)
            for n in range(n_iter)]

# Random search
dummy_res = run(dummy_minimize)

# Gaussian processes
gp_res = run(gp_minimize)

# Random forest
rf_res = run(partial(forest_minimize, base_estimator="RF"))

# Extra trees
et_res = run(partial(forest_minimize, base_estimator="ET"))
```

Note that this can take a few minutes.

```python
from skopt.plots import plot_convergence
plot = plot_convergence(("dummy_minimize", dummy_res),
                        ("gp_minimize", gp_res),
                        ("forest_minimize('rf')", rf_res),
                        ("forest_minimize('et')", et_res),
                        true_minimum=0.397887, yscale="log")
plot.legend(loc="best", prop={'size': 6}, numpoints=1)
```
This plot shows the value of the minimum found (y axis) as a function of the number of iterations performed so far (x axis). The dashed red line indicates the true value of the minimum of the `benchmarks.branin` function.

For the first ten iterations all methods perform equally well as they all start by creating ten random samples before fitting their respective model for the first time. After iteration ten the next point at which to evaluate `benchmarks.branin` is guided by the model, which is where differences start to appear.

Each minimizer only has access to noisy observations of the objective function, so as time passes (more iterations) it will start observing values that are below the true value simply because they are fluctuations.

**Total running time of the script:** (3 minutes 10.745 seconds)

**Estimated memory usage:** 69 MB
4.1.7 Bayesian optimization with skopt

Gilles Louppe, Manoj Kumar July 2016. Reformatted by Holger Nahrstaedt 2020

Problem statement

We are interested in solving

\[ x^* = \arg \min_x f(x) \]

under the constraints that

- \( f \) is a black box for which no closed form is known (nor its gradients);
- \( f \) is expensive to evaluate;
- and evaluations of \( y = f(x) \) may be noisy.

Disclaimer. If you do not have these constraints, then there is certainly a better optimization algorithm than Bayesian optimization.

This example uses \texttt{plots.plot gaussian process} which is available since version 0.8.

Bayesian optimization loop

For \( t = 1 : T \):

1. Given observations \((x_i, y_i = f(x_i))\) for \( i = 1 : t \), build a probabilistic model for the objective \( f \). Integrate out all possible true functions, using Gaussian process regression.

2. Optimize a cheap acquisition/utility function \( u \) based on the posterior distribution for sampling the next point. \( x_{t+1} = \arg \min_x u(x) \) Expose uncertainty to balance exploration against exploitation.

3. Sample the next observation \( y_{t+1} \) at \( x_{t+1} \).

Acquisition functions

Acquisition functions \( u(x) \) specify which sample \( x \): should be tried next:

- Expected improvement (default): \(-EI(x) = -E[f(x) - f(x_t^*)]\)
- Lower confidence bound: \( LCB(x) = \mu_{GP}(x) + \kappa \sigma_{GP}(x) \)
- Probability of improvement: \(-PI(x) = -P(f(x) \geq f(x_t^*) + \kappa) \)

where \( x_t^* \) is the best point observed so far.

In most cases, acquisition functions provide knobs (e.g., \( \kappa \)) for controlling the exploration-exploitation trade-off. - Search in regions where \( \mu_{GP}(x) \) is high (exploitation) - Probe regions where uncertainty \( \sigma_{GP}(x) \) is high (exploration)

```python
print(__doc__)
import numpy as np
np.random.seed(237)
import matplotlib.pyplot as plt
from skopt.plots import plot_gaussian_process
```
**Toy example**

Let assume the following noisy function $f$:

```python
noise_level = 0.1

def f(x, noise_level=noise_level):
    return np.sin(5 * x[0]) * (1 - np.tanh(x[0] ** 2))
    + np.random.randn() * noise_level
```

**Note.** In skopt, functions $f$ are assumed to take as input a 1D vector $x$: represented as an array-like and to return a scalar $f(x)$:

```python
# Plot f(x) + contours
x = np.linspace(-2, 2, 400).reshape(-1, 1)
fx = [f(x_i, noise_level=0.0) for x_i in x]
plt.plot(x, fx, "r--", label="True (unknown)")
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate((
                    [fx_i - 1.9600 * noise_level for fx_i in fx],
                    [fx_i + 1.9600 * noise_level for fx_i in fx[::-1]])),
            alpha=.2, fc="r", ec="None")
plt.legend()
plt.grid()
plt.show()
```
Bayesian optimization based on gaussian process regression is implemented in `gp_minimize` and can be carried out as follows:

```python
from skopt import gp_minimize
res = gp_minimize(f,  # the function to minimize
                  [(-2.0, 2.0)],  # the bounds on each dimension of x
                  acq_func="EI",  # the acquisition function
                  n_calls=15,  # the number of evaluations of f
                  n_random_starts=5,  # the number of random initialization points
                  noise=0.1**2,  # the noise level (optional)
                  random_state=1234)  # the random seed
```

Accordingly, the approximated minimum is found to be:

```
x^*=%4.4f, f(x^*)=%4.4f  % (res.x[0], res.fun)
```

Out:

```
x^*=-0.3552, f(x^*)=-1.0079
```

For further inspection of the results, attributes of the `res` named tuple provide the following information:

- `x` [float]: location of the minimum.
- `fun` [float]: function value at the minimum.
- `models`: surrogate models used for each iteration.
- `x_iters` [array]: location of function evaluation for each iteration.
- `func_vals` [array]: function value for each iteration.
- `space` [Space]: the optimization space.
- `specs` [dict]: parameters passed to the function.

```
print(res)
```

Out:

```
fun: -1.0079192431413255
func_vals: array([ 0.03716044, 0.00673852, 0.63515442, -0.16042062, 0.10695907, -0.24436726, -0.5863053 , 0.05238728, -1.00791924, -0.98466748, -0.86259915, 0.18102445, -0.10782771, 0.00815673, -0.79756402])
models: [GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=0.01), n_restarts_optimizer=2, noise=0.010000000000000002, normalize_y=True, random_state=822569775), GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=0.01), n_restarts_optimizer=2, noise=0.010000000000000002, normalize_y=True, random_state=822569775), GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=0.01), n_restarts_optimizer=2, noise=0.010000000000000002, normalize_y=True, random_state=822569775), GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=0.01), n_restarts_optimizer=2, noise=0.010000000000000002, normalize_y=True, random_state=822569775), GaussianProcessRegressor(kernel=1**2 * Matern(length_scale=1, nu=2.5) + WhiteKernel(noise_level=0.01), n_restarts_optimizer=2, noise=0.010000000000000002, normalize_y=True, random_state=822569775)]
```

(continues on next page)
Together these attributes can be used to visually inspect the results of the minimization, such as the convergence trace or the acquisition function at the last iteration:

```python
from skopt.plots import plot_convergence
```

(continued on next page)
Let us now visually examine

1. The approximation of the fit gp model to the original function.
2. The acquisition values that determine the next point to be queried.

```python
plt.rcParams["figure.figsize"] = (8, 14)

def f_wo_noise(x):
    return f(x, noise_level=0)
```

Plot the 5 iterations following the 5 random points

```python
for n_iter in range(5):
    # Plot true function.
    plt.subplot(5, 2, 2*n_iter+1)
```
if n_iter == 0:
    show_legend = True
else:
    show_legend = False

ax = plot_gaussian_process(res, n_calls=n_iter,
                           objective=f_wo_noise,
                           noise_level=noise_level,
                           show_legend=show_legend, show_title=False,
                           show_next_point=False, show_acq_func=False)

ax.set_ylabel('')
ax.set_xlabel('')

# Plot EI(x)
plt.subplot(5, 2, 2*n_iter+2)
ax = plot_gaussian_process(res, n_calls=n_iter,
                           show_legend=show_legend, show_title=False,
                           show_mu=False, show_acq_func=True,
                           show_observations=False,
                           show_next_point=True)

ax.set_ylabel('')
ax.set_xlabel('')

plt.show()
4.1. Miscellaneous examples
The first column shows the following:

1. The true function.
2. The approximation to the original function by the gaussian process model
3. How sure the GP is about the function.

The second column shows the acquisition function values after every surrogate model is fit. It is possible that we do not choose the global minimum but a local minimum depending on the minimizer used to minimize the acquisition function.

At the points closer to the points previously evaluated at, the variance dips to zero.

Finally, as we increase the number of points, the GP model approaches the actual function. The final few points are clustered around the minimum because the GP does not gain anything more by further exploration:

```python
plt.rcParams["figure.figsize"] = (6, 4)

# Plot f(x) + contours
_ = plot_gaussian_process(res, objective=f_wo_noise, noise_level=noise_level)
plt.show()
```

$x^* = -0.3552, f(x^*) = -1.0079$

**Total running time of the script:** (0 minutes 3.873 seconds)

**Estimated memory usage:** 9 MB
4.1.8 Scikit-learn hyperparameter search wrapper

Iaroslav Shcherbatyi, Tim Head and Gilles Louppe. June 2017. Reformatted by Holger Nahrstaedt 2020

Introduction

This example assumes basic familiarity with scikit-learn.

Search for parameters of machine learning models that result in best cross-validation performance is necessary in almost all practical cases to get a model with best generalization estimate. A standard approach in scikit-learn is using sklearn.model_selection.GridSearchCV class, which takes a set of values for every parameter to try, and simply enumerates all combinations of parameter values. The complexity of such search grows exponentially with the addition of new parameters. A more scalable approach is using sklearn.model_selection.RandomizedSearchCV, which however does not take advantage of the structure of a search space.

Scikit-optimize provides a drop-in replacement for sklearn.model_selection.GridSearchCV, which utilizes Bayesian Optimization where a predictive model referred to as “surrogate” is used to model the search space and utilized to arrive at good parameter values combination as soon as possible.

Note: for a manual hyperparameter optimization example, see “Hyperparameter Optimization” notebook.

```
print(__doc__)
import numpy as np
np.random.seed(123)
import matplotlib.pyplot as plt

Minimal example

A minimal example of optimizing hyperparameters of SVC (Support Vector machine Classifier) is given below.

```
from skopt import BayesSearchCV
from sklearn.datasets import load_digits
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

X, y = load_digits(n_class=10, return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75, test_size=.25, random_state=0)

# log-uniform: understand as search over p = exp(x) by varying x
opt = BayesSearchCV(SVC(),
    {
        'C': (1e-6, 1e+6, 'log-uniform'),
        'gamma': (1e-6, 1e+1, 'log-uniform'),
        'degree': (1, 8),  # integer valued parameter
        'kernel': ['linear', 'poly', 'rbf'],  # categorical parameter
    },
    n_iter=32,
    cv=3)

opt.fit(X_train, y_train)
```

(continues on next page)
print("val. score: %s" % opt.best_score_)
print("test score: %s" % opt.score(X_test, y_test))

Out:

val. score: 0.985894580549369
test score: 0.9822222222222222

Advanced example

In practice, one wants to enumerate over multiple predictive model classes, with different search spaces and number of evaluations per class. An example of such search over parameters of Linear SVM, Kernel SVM, and decision trees is given below.

```python
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram
from sklearn.datasets import load_digits
from sklearn.svm import LinearSVC, SVC
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split

X, y = load_digits(n_class=10, return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# pipeline class is used as estimator to enable
# search over different model types
pipe = Pipeline([('model', SVC())])

# single categorical value of 'model' parameter is
# sets the model class
# We will get ConvergenceWarnings because the problem is not well-conditioned.
# But that's fine, this is just an example.
linsvc_search = {
    'model': [LinearSVC(max_iter=1000)],
    'model__C': (1e-6, 1e+6, 'log-uniform'),
}

# explicit dimension classes can be specified like this
svc_search = {
    'model': Categorical([SVC()]),
    'model__C': Real(1e-6, 1e+6, prior='log-uniform'),
    'model__gamma': Real(1e-6, 1e+1, prior='log-uniform'),
    'model__degree': Integer(1,8),
    'model__kernel': Categorical(['linear', 'poly', 'rbf']),
}
```

(continues on next page)
opt = BayesSearchCV(
    pipe,
    # (parameter space, # of evaluations)
    [(svc_search, 40), (linsvc_search, 16)],
    cv=3
)

opt.fit(X_train, y_train)

print("val. score: %s" % opt.best_score_)
print("test score: %s" % opt.score(X_test, y_test))
print("best params: %s" % str(opt.best_params_))

Out:

/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-...
lin...1199: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn(4.1. Miscellaneous examples 51
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
˓
linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to␣
˓
converge, increase the number of iterations.
warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
\-linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to
˓converge, increase the number of iterations.
  warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
\-linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to
˓converge, increase the number of iterations.
  warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
\-linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to
˓converge, increase the number of iterations.
  warnings.warn(
/home/circleci/miniconda/envs/testenv/lib/python3.9/site-packages/scikit_learn-1.0-py3.9-
\-linux-x86_64.egg/sklearn/svm/_base.py:1199: ConvergenceWarning: Liblinear failed to
˓converge, increase the number of iterations.
  val. score: 0.985894580549369
 test score: 0.9822222222222222
 best params: OrderedDict([('model', SVC(C=0.41571471424085416, gamma=1.0560013164213486,˓kernel='poly')), ('model__C', 0.41571471424085416), ('model__degree', 3), ('model__gamma', 1.0560013164213486), ('model__kernel', 'poly'))]

Partial Dependence plot of the objective function for SVC

_ = plot_objective(opt.optimizer_results_[0],
                     dimensions=['C', 'degree', 'gamma', 'kernel'],
                     n_minimum_search=int(1e8))

plt.show()
Plot of the histogram for LinearSVC

```python
_= plot_histogram(opt.optimizer_results_[1], 1)
plt.show()
```
Progress monitoring and control using callback argument of fit method

It is possible to monitor the progress of *BayesSearchCV* with an event handler that is called on every step of subspace exploration. For single job mode, this is called on every evaluation of model configuration, and for parallel mode, this is called when n_jobs model configurations are evaluated in parallel.

Additionally, exploration can be stopped if the callback returns `True`. This can be used to stop the exploration early, for instance when the accuracy that you get is sufficiently high.

An example usage is shown below.

```python
from skopt import BayesSearchCV

from sklearn.datasets import load_iris
from sklearn.svm import SVC

X, y = load_iris(return_X_y=True)

searchcv = BayesSearchCV(
    SVC(gamma='scale'),
    search_spaces={
        'C': (0.01, 100.0, 'log-uniform')},
    n_iter=10,
    cv=3
)
```

(continues on next page)
# callback handler

def on_step(optim_result):
    score = -optim_result['fun']
    print("best score: %s" % score)
    if score >= 0.98:
        print('Interrupting!')
    return True

searchcv.fit(X, y, callback=on_step)

Out:

best score: 0.98
Interrupting!

BayesSearchCV(cv=3, estimator=SVC(), n_iter=10,
search_spaces={'C': (0.01, 100.0, 'log-uniform')})

Counting total iterations that will be used to explore all subspaces

Subspaces in previous examples can further increase in complexity if you add new model subspaces or dimensions for feature extraction pipelines. For monitoring of progress, you would like to know the total number of iterations it will take to explore all subspaces. This can be calculated with total_iterations property, as in the code below.

from skopt import BayesSearchCV

from sklearn.datasets import load_iris
from sklearn.svm import SVC

X, y = load_iris(return_X_y=True)

searchcv = BayesSearchCV(
    SVC(),
    search_spaces=[
        {'C': (0.1, 1.0)}, 19,  # 19 iterations for this subspace
        {'gamma':(0.1, 1.0)}
    ],
    n_iter=23
)

print(searchcv.total_iterations)

Out:

42

Total running time of the script: ( 1 minutes 37.287 seconds)
Estimated memory usage: 10 MB
4.1.9 Exploration vs exploitation

Sigurd Carlen, September 2019. Reformatted by Holger Nahrstaedt 2020

We can control how much the acquisition function favors exploration and exploitation by tweaking the two parameters kappa and xi. Higher values means more exploration and less exploitation and vice versa with low values.

kappa is only used if acq_func is set to “LCB”. xi is used when acq_func is “EI” or “PI”. By default the acquisition function is set to “gp_hedge” which chooses the best of these three. Therefore I recommend not using gp_hedge when tweaking exploration/exploitation, but instead choosing “LCB”, “EI” or “PI”.

The way to pass kappa and xi to the optimizer is to use the named argument “acq_func_kwargs”. This is a dict of extra arguments for the acquisition function.

If you want opt.ask() to give a new acquisition value immediately after tweaking kappa or xi call opt.update_next(). This ensures that the next value is updated with the new acquisition parameters.

This example uses plots.plot_gaussian_process which is available since version 0.8.

```python
print(__doc__)

import numpy as np
np.random.seed(1234)
import matplotlib.pyplot as plt
from skopt.learning import ExtraTreesRegressor
from skopt import Optimizer
from skopt.plots import plot_gaussian_process

Toy example

First we define our objective like in the ask-and-tell example notebook and define a plotting function. We do however only use on initial random point. All points after the first one is therefore chosen by the acquisition function.

```python
noise_level = 0.1

# Our 1D toy problem, this is the function we are trying to # minimize
def objective(x, noise_level=noise_level):
    return np.sin(5 * x[0]) * (1 - np.tanh(x[0] ** 2)) +
    np.random.randn() * noise_level

def objective_wo_noise(x):
    return objective(x, noise_level=0)

opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
    acq_optimizer="sampling")

Plotting parameters

```python
plot_args = {
    "objective": objective_wo_noise,
    "noise_level": noise_level, "show_legend": True,
    "show_title": True, "show_next_point": False,
    "show_acq_func": True}
```
We run an optimization loop with standard settings:

```python
for i in range(30):
    next_x = opt.ask()
    f_val = objective(next_x)
    opt.tell(next_x, f_val)
# The same output could be created with opt.run(objective, n_iter=30)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```

We see that some minima is found and “exploited”

Now let’s try to set kappa and xi using to other values and pass it to the optimizer:

```python
acq_func_kwargs = {"xi": 10000, "kappa": 10000}
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
                acq_optimizer="sampling",
                acq_func_kwarg=acq_func_kwarg)
opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
We see that the points are more random now.

This works both for kappa when using `acq_func="LCB"`:

```python
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
               acq_func="LCB", acq_optimizer="sampling",
               acq_func_kwargs=acq_func_kwargs)
opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
And for $x_i$ when using acq_func="EI": or acq_func="PI":

```python
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
               acq_func="EI", acq_optimizer="sampling",
               acq_func_kwargs=acq_func_kwargs)
opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
We can also favor exploitation:

```python
acq_func_kwars = {"xi": 0.000001, "kappa": 0.001}

opt = Optimizer([-2.0, 2.0], "GP", n_initial_points=3,
          acq_func="LCB", acq_optimizer="sampling",
          acq_func_kwars=acq_func_kwars)

opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
`scikit-optimize Documentation, Release 0.9.0`

```
opt = Optimizer([[-2.0, 2.0]], "GP", n_initial_points=3,
                acq_func="EI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)

opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
scikit-optimize Documentation, Release 0.9.0

\[
x^* = 0.8760, f(x^*) = -0.4154
\]

```python
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
                acq_func="PI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)

opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
Note that negative values does not work with the “PI”-acquisition function but works with “EI”:

```python
acq_func_kwars = {"xi": -1000000000000}
```

```python
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
                 acq_func="PI", acq_optimizer="sampling",
                 acq_func_kwargs=acq_func_kwargs)
```

```python
opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
scikit-optimize Documentation, Release 0.9.0

\[ x^* = 0.8093, f(x^*) = -0.4621 \]

```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
                acq_func="EI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)

opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
Changing kappa and xi on the go

If we want to change kappa or xi at any point during our optimization process we just replace opt.acq_func_kwargs. Remember to call opt.update_next() after the change, in order for next point to be recalculated.

```python
acq_func_kwargs = {'kappa': 0}

opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=3,
                 acq_func="LCB", acq_optimizer="sampling",
                 acq_func_kwargs=acq_func_kwargs)

opt.acq_func_kwargs
```

Out:

```python
{'kappa': 0}
```

```python
opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
scikit-optimize Documentation, Release 0.9.0

4.1. Miscellaneous examples

\[ x^* = -0.2982, f(x^*) = -1.0610 \]

acq_func_kwarg = {"kappa": 100000}

```python
opt.acq_func_kwarg = acq_func_kwarg
opt.update_next()
```

```python
opt.run(objective, n_iter=20)
_ = plot_gaussian_process(opt.get_result(), **plot_args)
```
4.1.10 Use different base estimators for optimization

Sigurd Carlen, September 2019. Reformatted by Holger Nahrstaedt 2020

To use different base estimator or create a regressor with different parameters, we can create a regressor object and set it as kernel.

This example uses `plots.plot_gaussian_process` which is available since version 0.8.

```python
print(__doc__)

import numpy as np
np.random.seed(1234)
import matplotlib.pyplot as plt
from skopt.plots import plot_gaussian_process
from skopt import Optimizer
```
Toy example

Let assume the following noisy function $f$:

```python
noise_level = 0.1

# Our 1D toy problem, this is the function we are trying to
# minimize

def objective(x, noise_level=noise_level):
    return np.sin(5 * x[0]) * (1 - np.tanh(x[0] ** 2))
    + np.random.randn() * noise_level

def objective_wo_noise(x):
    return objective(x, noise_level=0)

opt_gp = Optimizer([(-2.0, 2.0)], base_estimator="GP", n_initial_points=5,
                    acq_optimizer="sampling", random_state=42)
```

```python
def plot_optimizer(res, n_iter, max_iters=5):
    if n_iter == 0:
        show_legend = True
    else:
        show_legend = False
    ax = plt.subplot(max_iters, 2, 2 * n_iter + 1)
    # Plot GP(x) + contours
    ax = plot_gaussian_process(res, ax=ax,
                               objective=objective_wo_noise,
                               noise_level=noise_level,
                               show_legend=show_legend, show_title=True,
                               show_next_point=False, show_acq_func=False)
    ax.set_ylabel(""")
    ax.set_xlabel(""")
    if n_iter < max_iters - 1:
        ax.get_xaxis().set_ticklabels([])
    # Plot EI(x)
    ax = plt.subplot(max_iters, 2, 2 * n_iter + 2)
    ax = plot_gaussian_process(res, ax=ax,
                               noise_level=noise_level,
                               show_legend=show_legend, show_title=False,
                               show_next_point=True, show_acq_func=True,
                               show_observations=False,
                               show_mu=False)
    ax.set_ylabel(""")
    ax.set_xlabel(""")
    if n_iter < max_iters - 1:
        ax.get_xaxis().set_ticklabels([])
```

4.1. Miscellaneous examples 69
GP kernel

```python
fig = plt.figure()
fig.suptitle("Standard GP kernel")
for i in range(10):
    next_x = opt_gp.ask()
    f_val = objective(next_x)
    res = opt_gp.tell(next_x, f_val)
    if i >= 5:
        plot_optimizer(res, n_iter=i-5, max_iters=5)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.plot()
```

Out:

[]

---

Chapter 4. Examples
Test different kernels

```python
from skopt.learning import GaussianProcessRegressor
from skopt.learning.gaussian_process.kernels import ConstantKernel, Matern
# Gaussian process with Matérn kernel as surrogate model

from sklearn.gaussian_process.kernels import (RBF, Matern, RationalQuadratic,
                                            ExpSineSquared, DotProduct,
                                            ConstantKernel)

kernels = [1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-1, 10.0)),
           1.0 * RationalQuadratic(length_scale=1.0, alpha=0.1),
           1.0 * ExpSineSquared(length_scale=1.0, periodicity=3.0,
                                length_scale_bounds=(0.1, 10.0),
                                periodicity_bounds=(1.0, 10.0)),
           ConstantKernel(0.1, (0.01, 10.0))
           * (DotProduct(sigma_0=1.0, sigma_0_bounds=(0.1, 10.0)) ** 2),
           1.0 * Matern(length_scale=1.0, length_scale_bounds=(1e-1, 10.0),
                         nu=2.5)]

for kernel in kernels:
    gpr = GaussianProcessRegressor(kernel=kernel, alpha=noise_level ** 2,
                                    normalize_y=True, noise="gaussian",
                                    n_restarts_optimizer=2)
    opt = Optimizer([[(-2.0, 2.0)]], base_estimator=gpr, n_initial_points=5,
                     acq_optimizer="sampling", random_state=42)
    fig = plt.figure()
    fig.suptitle(repr(kernel))
    for i in range(10):
        next_x = opt.ask()
        f_val = objective(next_x)
        res = opt.tell(next_x, f_val)
        if i >= 5:
            plot_optimizer(res, n_iter=i - 5, max_iters=5)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
```

4.1. Miscellaneous examples
$1^{**2} \times \text{RBF}(\text{length_scale}=1)$

$x^* = -0.5018, f(x^*) = -0.4236$

$x^* = -0.5018, f(x^*) = -0.4236$

$x^* = -0.5018, f(x^*) = -0.4236$

$x^* = -0.5018, f(x^*) = -0.4236$

$x^* = -0.5018, f(x^*) = -0.4236$
\[ 1^{*2} \times \text{RationalQuadratic}(\alpha=0.1, \text{length\_scale}=1) \]

\[ x^* = -0.5018, f(x^*) = -0.4792 \]

\[ x^* = -0.5018, f(x^*) = -0.4792 \]

\[ x^* = -0.5018, f(x^*) = -0.4792 \]

\[ x^* = -0.3767, f(x^*) = -0.8734 \]
\[ 1^{*2} * \text{ExpSineSquared}(\text{length\_scale}=1, \text{periodicity}=3) \]

\[
\begin{align*}
\text{x}^* &= -0.5018, f(x^*) = -0.4078 \\
\text{x}^* &= -0.5018, f(x^*) = -0.4078 \\
\text{x}^* &= -0.5018, f(x^*) = -0.4078 \\
\text{x}^* &= -0.2591, f(x^*) = -1.0230 \\
\text{x}^* &= -0.2591, f(x^*) = -1.0230 \\
\end{align*}
\]
0.316**2 * DotProduct(sigma_0=1)**2

\[ x^* = -0.5018, \ f(x^*) = -0.5936 \]

\[ x^* = 0.5018, \ f(x^*) = 0.5936 \]

\[ x^* = -0.5018, \ f(x^*) = -0.5936 \]

\[ x^* = 0.5018, \ f(x^*) = 0.5936 \]

\[ x^* = -0.5018, \ f(x^*) = -0.5936 \]

\[ x^* = 0.5018, \ f(x^*) = 0.5936 \]
4.2 Initial sampling functions

Examples concerning the \texttt{skopt.sampler} module.

4.2.1 Comparing initial sampling methods

Holger Nahrstaedt 2020 Sigurd Carlsen October 2019

When doing bayesian optimization we often want to reserve some of the early part of the optimization to pure exploration. By default the optimizer suggests purely random samples for the first \texttt{n\_initial\_points} (10 by default). The downside to this is that there is no guarantee that these samples are spread out evenly across all the dimensions.

Sampling methods as Latin hypercube, Sobol', Halton and Hammersly take advantage of the fact that we know beforehand how many random points we want to sample. Then these points can be “spread out” in such a way that each dimension is explored.

See also the example on an integer space \texttt{sphx\_glr\_auto\_examples\_initial\_sampling\_method\_integer.py}

```python
print(__doc__)
import numpy as np```

(continues on next page)
np.random.seed(123)
import matplotlib.pyplot as plt
from skopt.space import Space
from skopt.sampler import Sobol
from skopt.sampler import Lhs
from skopt.sampler import Halton
from skopt.sampler import Hammersly
from skopt.sampler import Grid
from scipy.spatial.distance import pdist

def plot_searchspace(x, title):
    fig, ax = plt.subplots()
    plt.plot(np.array(x)[:, 0], np.array(x)[:, 1], 'bo', label='samples')
    plt.plot(np.array(x)[:, 0], np.array(x)[:, 1], 'bo', markersize=80, alpha=0.5)
    # ax.legend(loc="best", numpoints=1)
    ax.set_xlabel("X1")
    ax.set_xlim([-5, 10])
    ax.set_ylabel("X2")
    ax.set_ylim([0, 15])
    plt.title(title)

n_samples = 10

space = Space([(-5., 10.), (0., 15.)])
# space.set_transformer("normalize")

Random sampling

x = space.rvs(n_samples)
plot_searchspace(x, "Random samples")
pdist_data = []
x_label = []
pdist_data.append(pdist(x).flatten())
x_label.append("random")
Sobol'

```python
sobol = Sobol()
x = sobol.generate(space.dimensions, n_samples)
plot_searchspace(x, "Sobol")
pdist_data.append(pdist(x).flatten())
x_label.append("sobol")
```
Out:
/home/circleci/project/skopt/sampler/sobol.py:246: UserWarning: The balance properties of Sobol' points require n to be a power of 2. 0 points have been previously generated, then: n=0+10=10.
warnings.warn("The balance properties of Sobol' points require ")

Classic Latin hypercube sampling

```python
lhs = Lhs(lhs_type="classic", criterion=None)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'classic LHS')
pdist_data.append(pdist(x).flatten())
x_label.append("lhs")
```
Centered Latin hypercube sampling

```python
lhs = Lhs(lhs_type="centered", criterion=None)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'centered LHS')
pdist_data.append(pdist(x).flatten())
x_label.append("center")
```
Maximin optimized hypercube sampling

```python
lhs = Lhs(criterion="maximin", iterations=10000)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'maximin LHS')
pdist_data.append(pdist(x).flatten())
x_label.append("maximin")
```
Correlation optimized hypercube sampling

```python
lhs = Lhs(criterion="correlation", iterations=10000)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'correlation LHS')
pdist_data.append(pdist(x).flatten())
x_label.append("corr")
```
**Ratio optimized hypercube sampling**

```python
lhs = Lhs(criterion="ratio", iterations=10000)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'ratio LHS')
pdist_data.append(pdist(x).flatten())
x_label.append("ratio")
```
Halton sampling

```python
halton = Halton()
x = halton.generate(space.dimensions, n_samples)
plot_searchspace(x, 'Halton')
pdist_data.append(pdist(x).flatten())
x_label.append("halton")
```
Hammersly sampling

```python
hammersly = Hammersly()
x = hammersly.generate(space.dimensions, n_samples)
plot_searchspace(x, 'Hammersly')
pdist_data.append(pdist(x).flatten())
x_label.append("hammersly")
```
Grid sampling

```python
grid = Grid(border="include", use_full_layout=False)
x = grid.generate(space.dimensions, n_samples)
plot_searchspace(x, 'Grid')
pdist_data.append(pdist(x).flatten())
x_label.append("grid")
```
**Pdist boxplot of all methods**

This boxplot shows the distance between all generated points using Euclidian distance. The higher the value, the better the sampling method. It can be seen that random has the worst performance.

```python
fig, ax = plt.subplots()
ax.boxplot(pdist_data)
plt.grid(True)
plt.ylabel("pdist")
_ = ax.set_ylim(0, 12)
_ = ax.set_xticklabels(x_label, rotation=45, fontsize=8)
```
4.2.2 Comparing initial sampling methods on integer space

Holger Nahrstaedt 2020 Sigurd Carlsen October 2019

When doing baysian optimization we often want to reserve some of the early part of the optimization to pure exploration. By default the optimizer suggests purely random samples for the first n_initial_points (10 by default). The downside to this is that there is no guarantee that these samples are spread out evenly across all the dimensions.

Sampling methods as Latin hypercube, Sobol’, Halton and Hammersly take advantage of the fact that we know beforehand how many random points we want to sample. Then these points can be “spread out” in such a way that each dimension is explored.

See also the example on a real space sphx_glr_auto_examples_initial_sampling_method.py

```python
print(__doc__)
import numpy as np
np.random.seed(1234)
import matplotlib.pyplot as plt
from skopt.space import Space
from skopt.sampler import Sobol
from skopt.sampler import Lhs
```
from skopt.sampler import Halton
from skopt.sampler import Hammersly
from skopt.sampler import Grid
from scipy.spatial.distance import pdist

def plot_searchspace(x, title):
    fig, ax = plt.subplots()
    plt.plot(np.array(x)[:, 0], np.array(x)[:, 1], 'bo', label='samples')
    plt.plot(np.array(x)[:, 0], np.array(x)[:, 1], 'bs', markersize=40, alpha=0.5)
    ax.set_xlabel("X1")
    ax.set_xlim([0, 5])
    ax.set_ylabel("X2")
    ax.set_ylim([0, 5])
    plt.title(title)
    ax.grid(True)

n_samples = 10
space = Space([(0, 5), (0, 5)])

Random sampling

x = space.rvs(n_samples)
plot_searchspace(x, "Random samples")
pdist_data = []
x_label = []
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("random")
```python
sobol = Sobol()
x = sobol.generate(space.dimensions, n_samples)
plot_searchspace(x, "Sobol")
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("sobol")
```
Classic latin hypercube sampling

```python
lhs = Lhs(lhs_type="classic", criterion=\texttt{None})
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'classic LHS')
print("empty fields: \%d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("lhs")
```
Centered latin hypercube sampling

```python
lhs = Lhs(lhs_type="centered", criterion=None)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'centered LHS')
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("center")
```
Maximin optimized hypercube sampling

```python
lhs = Lhs(criterion="maximin", iterations=10000)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, "maximin LHS")
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("maximin")
```
Correlation optimized hypercube sampling

```python
lhs = Lhs(criterion="correlation", iterations=10000)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'correlation LHS')
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("corr")
```
Ratio optimized hypercube sampling

```python
lhs = Lhs(criterion="ratio", iterations=10000)
x = lhs.generate(space.dimensions, n_samples)
plot_searchspace(x, 'ratio LHS')
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("ratio")
```
Halton sampling

```python
halton = Halton()
x = halton.generate(space.dimensions, n_samples)
plot_searchspace(x, 'Halton')
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("halton")
```
Hammersly sampling

```python
hammersly = Hammersly()
x = hammersly.generate(space.dimensions, n_samples)
plot_searchspace(x, 'Hammersly')
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("hammersly")
```
Out:

```
empty fields: 26
```

**Grid sampling**

```python
grid = Grid(border="include", use_full_layout=False)
x = grid.generate(space.dimensions, n_samples)
plot_searchspace(x, 'Grid')
print("empty fields: %d" % (36 - np.size(np.unique(x, axis=0), 0)))
pdist_data.append(pdist(x).flatten())
x_label.append("grid")
```
empty fields: 26

**Pdist boxplot of all methods**

This boxplot shows the distance between all generated points using Euclidian distance. The higher the value, the better the sampling method. It can be seen that random has the worst performance.

```python
fig, ax = plt.subplots()
ax.boxplot(pdist_data)
plt.grid(True)
plt.ylabel("pdist")
_ = ax.set_ylim(0, 6)
_ = ax.set_xticklabels(x_label, rotation=45, fontsize=8)
```
4.2.3 Comparing initial point generation methods

Holger Nahrstaedt 2020

Bayesian optimization or sequential model-based optimization uses a surrogate model to model the expensive to evaluate function func. There are several choices for what kind of surrogate model to use. This notebook compares the performance of:

- Halton sequence,
- Hammersly sequence,
- Sobol’ sequence and
- Latin hypercube sampling

as initial points. The purely random point generation is used as a baseline.
Toy model

We will use the `benchmarks.hart6` function as toy model for the expensive function. In a real world application this function would be unknown and expensive to evaluate.

```python
from skopt.benchmarks import hart6 as hart6_
# redefined `hart6` to allow adding arbitrary "noise" dimensions
def hart6(x, noise_level=0.):
    return hart6_([x[:6]]) + noise_level * np.random.randn()

from skopt.benchmarks import branin as _branin
def branin(x, noise_level=0.):
    return _branin(x) + noise_level * np.random.randn()
```

```python
from matplotlib.pyplot import cm
import time
from skopt import gp_minimize, forest_minimize, dummy_minimize
def plot_convergence(result_list, true_minimum=None, yscale=None, title="Convergence plot ":)
    ax = plt.gca()
    ax.set_title(title)
    ax.set_xlabel("Number of calls $n$")
    ax.set_ylabel(r"$\min f(x)$ after $n$ calls")
    ax.grid()
    if yscale is not None:
        ax.set_yscale(yscale)
    colors = cm.hsv(np.linspace(0.25, 1.0, len(result_list)))
    for results, color in zip(result_list, colors):
        name, results = results
        n_calls = len(results[0].x_iters)
        iterations = range(1, n_calls + 1)
        mins = [[np.min(r.func_vals[:i]) for i in iterations] for r in results]
        ax.plot(iterations, np.mean(mins, axis=0), c=color, label=name)
        if true_minimum:
            ax.axhline(true_minimum, linestyle="--", color="r", lw=1,
            label="True minimum")
    ax.legend(loc="best")
    return ax
```

```python
def run(minimizer, initial_point_generator,
    n_initial_points=10, n_repeats=1):
    return [minimizer(func, bounds, n_initial_points=n_initial_points,
        initial_point_generator=initial_point_generator,
        n_calls=n_calls, random_state=n)
```

(continues on next page)
for n in range(n_repeats)]

def run_measure(initial_point_generator, n_initial_points=10):
    start = time.time()
    # n_repeats must set to a much higher value to obtain meaningful results.
    n_repeats = 1
    res = run(gp_minimize, initial_point_generator,
               n_initial_points=n_initial_points, n_repeats=n_repeats)
    duration = time.time() - start
    # print("%s %s: %.2f s" % (initial_point_generator,
    # str(init_point_gen_kwargs),
    # duration))
    return res

Objective

The objective of this example is to find one of these minima in as few iterations as possible. One iteration is defined as one call to the benchmarks.hart6 function.

We will evaluate each model several times using a different seed for the random number generator. Then compare the average performance of these models. This makes the comparison more robust against models that get “lucky”.

from functools import partial
example = "hart6"

if example == "hart6":
    func = partial(hart6, noise_level=0.1)
    bounds = [(0., 1.), ] * 6
    true_minimum = -3.32237
    n_calls = 40
    n_initial_points = 10
    yscale = None
    title = "Convergence plot - hart6"
else:
    func = partial(branin, noise_level=2.0)
    bounds = [(-5.0, 10.0), (0.0, 15.0)]
    true_minimum = 0.397887
    n_calls = 30
    n_initial_points = 10
    yscale="log"
    title = "Convergence plot - branin"

from skopt.utils import cook_initial_point_generator

# Random search
dummy_res = run_measure("random", n_initial_points)
lhs = cook_initial_point_generator("lhs", criteria="classic", criterion="maximin")
lhs_res = run_measure(lhs, n_initial_points)
lhs2 = cook_initial_point_generator("lhs", criterion="maximin")
lhs2_res = run_measure(lhs2, n_initial_points)
sobol = cook_initial_point_generator("sobol", randomize=False,
    min_skip=1, max_skip=100)
sobol_res = run_measure(sobol, n_initial_points)
halton_res = run_measure("halton", n_initial_points)
hammersly_res = run_measure("hammersly", n_initial_points)
grid_res = run_measure("grid", n_initial_points)

Out:

/home/circleci/project/skopt/sampler/sobol.py:246: UserWarning: The balance properties of Sobol' points require n to be a power of 2. 0 points have been previously generated, then: n=0+10=10.
    warnings.warn("The balance properties of Sobol' points require ")

Note that this can take a few minutes.

plot = plot_convergence([("random", dummy_res),
    ("lhs", lhs_res),
    ("lhs_maximin", lhs2_res),
    ("sobol'", sobol_res),
    ("halton", halton_res),
    ("hammersly", hammersly_res),
    ("grid", grid_res)],
    true_minimum=true_minimum, yscale=yscale, title=title)

plt.show()}
This plot shows the value of the minimum found (y axis) as a function of the number of iterations performed so far (x axis). The dashed red line indicates the true value of the minimum of the `benchmarks.hart6` function.

Test with different n_random_starts values

```python
lhs2 = cook_initial_point_generator("lhs", criterion="maximin")
lhs2_15_res = run_measure(lhs2, 12)
lhs2_20_res = run_measure(lhs2, 14)
lhs2_25_res = run_measure(lhs2, 16)
```

n_random_starts = 10 produces the best results

```python
plot = plot_convergence(["random - 10", dummy_res],
                        ["lhs_maximin - 10", lhs2_res],
                        ["lhs_maximin - 12", lhs2_15_res],
                        ["lhs_maximin - 14", lhs2_20_res],
                        ["lhs_maximin - 16", lhs2_25_res],
                        true_minimum=true_minimum,
                        yscale=yscale,
                        title=title)

plt.show()
```
4.3 Plotting functions

Examples concerning the `skopt.plots` module.

4.3.1 Partial Dependence Plots

Sigurd Carlsen Feb 2019 Holger Nahrstaedt 2020

Plot objective now supports optional use of partial dependence as well as different methods of defining parameter values for dependency plots.
Objective function

Plot objective now supports optional use of partial dependence as well as different methods of defining parameter values for dependency plots.

```python
# Here we define a function that we evaluate.
def funny_func(x):
    s = 0
    for i in range(len(x)):
        s += (x[i] * i) ** 2
    return s
```

Optimisation using decision trees

We run forest_minimize on the function.

```python
bounds = [(-1, 1.),] * 3
n_calls = 150

result = forest_minimize(funny_func, bounds, n_calls=n_calls,
                             base_estimator="ET",
                             random_state=4)
```

Partial dependence plot

Here we see an example of using partial dependence. Even when setting n_points all the way down to 10 from the default of 40, this method is still very slow. This is because partial dependence calculates 250 extra predictions for each point on the plots.

```python
_ = plot_objective(result, n_points=10)
```
It is possible to change the location of the red dot, which normally shows the position of the found minimum. We can set it 'expected_minimum', which is the minimum value of the surrogate function, obtained by a minimum search method.

```python
_ = plot_objective(result, n_points=10, minimum='expected_minimum')
```
Plot without partial dependence

Here we plot without partial dependence. We see that it is a lot faster. Also the values for the other parameters are set to the default “result” which is the parameter set of the best observed value so far. In the case of funny_func this is close to 0 for all parameters.

```python
_ = plot_objective(result, sample_source='result', n_points=10)
```
Modify the shown minimum

Here we try with setting the minimum parameters to something other than “result”. First we try with “expected_minimum” which is the set of parameters that gives the minimum value of the surrogate function, using scipy's minimum search method.

```python
_ = plot_objective(result, n_points=10, sample_source='expected_minimum', minimum='expected_minimum')
```
"expected_minimum_random" is a naive way of finding the minimum of the surrogate by only using random sampling:

```python
_ = plot_objective(result, n_points=10, sample_source='expected_minimum_random', minimum='expected_minimum_random')
```
We can also specify how many initial samples are used for the two different “expected_minimum” methods. We set it to a low value in the next examples to showcase how it affects the minimum for the two methods.

```python
_ = plot_objective(result, n_points=10, sample_source='expected_minimum_random',
                   minimum='expected_minimum_random',
                   n_minimum_search=10)
```
```python
_ = plot_objective(result, n_points=10, sample_source="expected_minimum",
                    minimum='expected_minimum', n_minimum_search=2)
```
Set a minimum location

Lastly we can also define these parameters ourself by parsing a list as the minimum argument:

```python
_ = plot_objective(result, n_points=10, sample_source=[1, -0.5, 0.5],
                    minimum=[1, -0.5, 0.5])
```
4.3.2 Partial Dependence Plots with categorical values

Sigurd Carlsen Feb 2019 Holger Nahrstaedt 2020

Plot objective now supports optional use of partial dependence as well as different methods of defining parameter values for dependency plots.

```python
print(__doc__)
import sys
from skopt.plots import plot_objective
from skopt import forest_minimize
import numpy as np
np.random.seed(123)
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from skopt.space import Integer, Categorical
from skopt import plots, gp_minimize
from skopt.plots import plot_objective

**objective function**

Here we define a function that we evaluate.

```python
def objective(params):
    clf = DecisionTreeClassifier(**{dim.name: val for dim, val in zip(SPACE, params) if dim.name != 'dummy'}))
    return -np.mean(cross_val_score(clf, *load_breast_cancer(return_X_y=True)))
```

**Bayesian optimization**

```python
SPACE = [
    Integer(1, 20, name='max_depth'),
    Integer(2, 100, name='min_samples_split'),
    Integer(5, 30, name='min_samples_leaf'),
    Integer(1, 30, name='max_features'),
    Categorical(list('abc'), name='dummy'),
    Categorical(['gini', 'entropy'], name='criterion'),
    Categorical(list('def'), name='dummy'),
]

result = gp_minimize(objective, SPACE, n_calls=20)
```

**Partial dependence plot**

Here we see an example of using partial dependence. Even when setting n_points all the way down to 10 from the default of 40, this method is still very slow. This is because partial dependence calculates 250 extra predictions for each point on the plots.

```python
_ = plot_objective(result, n_points=10)
```
Plot without partial dependence

Here we plot without partial dependence. We see that it is a lot faster. Also the values for the other parameters are set to the default “result” which is the parameter set of the best observed value so far. In the case of funny_func this is close to 0 for all parameters.

```python
_ = plot_objective(result, sample_source='result', n_points=10)
```
Modify the shown minimum

Here we try with setting the other parameters to something other than “result”. When dealing with categorical dimensions we can’t use ‘expected_minimum’. Therefore we try with “expected_minimum_random” which is a naive way of finding the minimum of the surrogate by only using random sampling. `n_minimum_search` sets the number of random samples, which is used to find the minimum.

```python
_= plot_objective(result, n_points=10, sample_source='expected_minimum_random',
                   minimum='expected_minimum_random', n_minimum_search=10000)
```

4.3. Plotting functions
Set a minimum location

Lastly we can also define these parameters ourselfs by parsing a list as the pars argument:

```
_ = plot_objective(result, n_points=10, sample_source=[15, 4, 7, 15, 'b', 'entropy', 'e'],
                     minimum=[15, 4, 7, 15, 'b', 'entropy', 'e'])
```
Total running time of the script: 0 minutes 14.357 seconds
Estimated memory usage: 52 MB
4.3.3 Visualizing optimization results

Tim Head, August 2016. Reformatted by Holger Nahrstaedt 2020

Bayesian optimization or sequential model-based optimization uses a surrogate model to model the expensive to evaluate objective function :func:. It is this model that is used to determine at which points to evaluate the expensive objective next.

To help understand why the optimization process is proceeding the way it is, it is useful to plot the location and order of the points at which the objective is evaluated. If everything is working as expected, early samples will be spread over the whole parameter space and later samples should cluster around the minimum.

The :func:`plots.plot_evaluations` function helps with visualizing the location and order in which samples are evaluated for objectives with an arbitrary number of dimensions.

The :func:`plots.plot_objective` function plots the partial dependence of the objective, as represented by the surrogate model, for each dimension and as pairs of the input dimensions.

All of the minimizers implemented in :mod:`skopt` return an :class:`OptimizeResult` instance that can be inspected. Both :func:`plots.plot_evaluations` and :func:`plots.plot_objective` are helpers that do just that.

```python
print(__doc__)
import numpy as np
np.random.seed(123)

import matplotlib.pyplot as plt

Toy models

We will use two different toy models to demonstrate how :func:`plots.plot_evaluations` works.

The first model is the :func:`benchmarks.branin` function which has two dimensions and three minima.

The second model is the :func:`hart6` function which has six dimension which makes it hard to visualize. This will show off the utility of :func:`plots.plot_evaluations`.

```python
from skopt.benchmarks import branin
from skopt.benchmarks import hart6

# redefined `hart6` to allow adding arbitrary "noise" dimensions
def hart6(x):
    return hart6_(x[:6])
```

Starting with branin

To start let’s take advantage of the fact that :func:`benchmarks.branin` is a simple function which can be visualised in two dimensions.

```python
from matplotlib.colors import LogNorm

def plot_branin():
    fig, ax = plt.subplots()
```
```python
x1_values = np.linspace(-5, 10, 100)
x2_values = np.linspace(0, 15, 100)
x_ax, y_ax = np.meshgrid(x1_values, x2_values)
vals = np.c_[x_ax.ravel(), y_ax.ravel()]
fx = np.reshape([branin(val) for val in vals], (100, 100))

cm = ax.pcolormesh(x_ax, y_ax, fx,
    norm=LogNorm(vmin=fx.min(),
                 vmax=fx.max()),
    cmap='viridis_r')

minima = np.array([[-np.pi, 12.275], [+np.pi, 2.275], [9.42478, 2.475]])
ax.plot(minima[:, 0], minima[:, 1], 'r.', markersize=14,
        lw=0, label='Minima')

cb = fig.colorbar(cm)
cb.set_label(r'$f(x)$')
ax.legend(loc='best', numpoints=1)

ax.set_xlabel(r'$X_0$')
ax.set_xlim([-5, 10])
ax.set_ylabel(r'$X_1$')
ax.set_ylim([0, 15])

plot_branin()
```
Evaluating the objective function

Next we use an extra trees based minimizer to find one of the minima of the `benchmarks.branin` function. Then we visualize at which points the objective is being evaluated using `plots.plot_evaluations`.

```python
from functools import partial
from skopt.plots import plot_evaluations
from skopt import gp_minimize, forest_minimize, dummy_minimize

bounds = [(-5.0, 10.0), (0.0, 15.0)]
n_calls = 160

forest_res = forest_minimize(branin, bounds, n_calls=n_calls,
```

(continues on next page)
base_estimator="ET", random_state=4)

_ = plot_evaluations(forest_res, bins=10)

`plot_evaluations` creates a grid of size n_dims by n_dims. The diagonal shows histograms for each of the dimensions. In the lower triangle (just one plot in this case) a two dimensional scatter plot of all points is shown. The order in which points were evaluated is encoded in the color of each point. Darker/purple colors correspond to earlier samples and lighter/yellow colors correspond to later samples. A red point shows the location of the minimum found by the optimization process.

You should be able to see that points start clustering around the location of the true minimum. The histograms show that the objective is evaluated more often at locations near to one of the three minima.

Using `plot_objective` we can visualise the one dimensional partial dependence of the surrogate model for each dimension. The contour plot in the bottom left corner shows the two dimensional partial dependence. In this case this is the same as simply plotting the objective as it only has two dimensions.

### Partial dependence plots

Partial dependence plots were proposed by [Friedman (2001)] as a method for interpreting the importance of input features used in gradient boosting machines. Given a function of $k$: variables $y = f(x_1, x_2, ..., x_k)$: the partial dependence of $f$ on the $i$-th variable $x_i$ is calculated as: $\phi(x_i) = \frac{1}{N} \sum_{j=0}^{N} f(x_{i,j}, x_{2,j}, ..., x_{i,j}, ..., x_{k,j})$: with the sum running over a set of $N$ points drawn at random from the search space.

The idea is to visualize how the value of $x_j$: influences the function $f$: after averaging out the influence of all other variables.
The two dimensional partial dependence plot can look like the true objective but it does not have to. As points at which the objective function is being evaluated are concentrated around the suspected minimum the surrogate model sometimes is not a good representation of the objective far away from the minima.

**Random sampling**

Compare this to a minimizer which picks points at random. There is no structure visible in the order in which it evaluates the objective. Because there is no model involved in the process of picking sample points at random, we can not plot the partial dependence of the model.
Working in six dimensions

Visualising what happens in two dimensions is easy, where `plots.plot_evaluations` and `plots.plot_objective` start to be useful is when the number of dimensions grows. They take care of many of the more mundane things needed to make good plots of all combinations of the dimensions.

The next example uses class `benchmarks.hart6` which has six dimensions and shows both `plots.plot_evaluations` and `plots.plot_objective`.

```python
bounds = [(0., 1.),] * 6

forest_res = forest_minimize(hart6, bounds, n_calls=n_calls,
                            base_estimator="ET", random_state=4)

_ = plot_evaluations(forest_res)
_ = plot_objective(forest_res, n_samples=40)
```
Going from 6 to 6+2 dimensions

To make things more interesting let's add two dimension to the problem. As benchmarks.hart6 only depends on six dimensions we know that for this problem the new dimensions will be “flat” or uninformative. This is clearly visible in both the placement of samples and the partial dependence plots.

```python
bounds = [(0., 1.),] * 8
n_calls = 200

forest_res = forest_minimize(hart6, bounds, n_calls=n_calls,
                             base_estimator="ET", random_state=4)

_ = plot_evaluations(forest_res)
_ = plot_objective(forest_res, n_samples=40)
```

(continues on next page)
4.3.4 Partial Dependence Plots 2D

Hvass-Labs Dec 2017 Holger Nahrstaedt 2020

Simple example to show the new 2D plots.

```python
print(__doc__)
import numpy as np
from math import exp

from skopt import gp_minimize
from skopt.space import Real, Categorical, Integer
```

(continues on next page)
from skopt.plots import plot_histogram, plot_objective_2D, plot_objective
from skopt.utils import point_asdict
np.random.seed(123)
import matplotlib.pyplot as plt

dim_learning_rate = Real(name='learning_rate', low=1e-6, high=1e-2, prior='log-uniform')
dim_num_dense_layers = Integer(name='num_dense_layers', low=1, high=5)
dim_num_dense_nodes = Integer(name='num_dense_nodes', low=5, high=512)
dim_activation = Categorical(name='activation', categories=['relu', 'sigmoid'])

dimensions = [dim_learning_rate,
              dim_num_dense_layers,
              dim_num_dense_nodes,
              dim_activation]

def model_fitness(x):
    learning_rate, num_dense_layers, num_dense_nodes, activation = x
    fitness = ((exp(learning_rate) - 1.0) * 1000) ** 2 + \
               (num_dense_layers) ** 2 + \
               (num_dense_nodes/100) ** 2
    fitness *= 1.0 + 0.1 * np.random.rand()

    if activation == 'sigmoid':
        fitness += 10

    return fitness

print(model_fitness(x=default_parameters))

Out:
1.518471835296799

search_result = gp_minimize(func=model_fitness,
                             dimensions=dimensions,
                             n_calls=30,
                             x0=default_parameters,
                             random_state=123)

print(search_result.x)
print(search_result.fun)

Out:
[2.568332629660544e-06, 1, 5, 'relu']
1.0117401773345693
```python
for fitness, x in sorted(zip(search_result.func_vals, search_result.x_iters)):
    print(fitness, x)
```

Out:

```
1.0117401773345693 [2.5683326296760544e-06, 1, 5, 'relu']
1.02011366272145 [4.9296274178364756e-06, 1, 5, 'relu']
1.02082501648667194 [5.447818995527194e-06, 1, 5, 'relu']
1.0216677303833677 [0.00011447839199199741, 1, 5, 'relu']
1.0319553707990767 [3.1995220781684726e-06, 1, 5, 'relu']
1.035006761229889 [9.690337835200007e-05, 1, 5, 'relu']
1.0387852167773188 [3.9177490814751e-06, 1, 5, 'relu']
1.055812553599018 [4.82671260300738e-06, 1, 5, 'relu']
1.0657737857823664 [4.1635874423092e-06, 1, 5, 'relu']
1.0669975953932924 [1e-06, 1, 5, 'relu']
1.0669131656352118 [0.00010264491499511595, 1, 5, 'relu']
1.0751293914333275 [1.5998250006763378e-06, 1, 5, 'relu']
1.0876905175635352 [6.11082549901025e-06, 1, 5, 'relu']
1.1301695294147942 [0.001280379715890307, 1, 19, 'relu']
1.1690663251629732 [0.00105106287289584, 1, 33, 'relu']
1.4602213686635033 [0.001, 1, 64, 'relu']
4.174922771740646 [0.0001122606517861382, 2, 5, 'relu']
14.33754059577632 [4.961649309025573e-06, 2, 44, 'sigmoid']
15.81112245930194 [5.768045960755954e-05, 1, 366, 'relu']
20.75714623673461 [4.664872650162405e-05, 4, 195, 'relu']
20.83105972547271 [3.629134387669892e-06, 3, 323, 'relu']
25.04598550233685 [1.5528231282866148e-05, 3, 380, 'relu']
25.725698564025883 [0.00103490899532338, 4, 264, 'relu']
26.808790139516606 [1e-06, 5, 5, 'relu']
28.09332239815317 [1e-06, 1, 512, 'relu']
31.6780842295837 [9.21458040695478e-05, 4, 213, 'sigmoid']
32.6099725349034 [0.00010264491499511595, 1, 5, 'relu']
108.2413089476986 [0.01, 1, 5, 'relu']
117.225589713700925 [0.008953258961145084, 4, 399, 'relu']
```

date = search_result.space

print(search_result.x_iters)

search_space = {name: space[name][1] for name in space.dimension_names}

print(point_asdict(search_space, default_parameters))

Out:

```
[[0.0001, 1, 64, 'relu'], [0.00010264491499511595, 1, 5, 'relu'], [1e-06, 1, 5, 'relu'], [6.11082549901025e-06, 1, 5, 'relu'], [1.0117401773345693, 2.56833262960544e-06, 1, 5, 'relu']] (continues on next page)
9.296274178364756e-06, 1, 5, 'relu', [4.8261721603000738e-06, 1, 5, 'relu'], [0.00011447839199199741, 1, 5, 'relu']
[4.9296274178364756e-06, 1, 5, 'relu'], [5.447818995527194e-06, 1, 5, 'relu'], [0.001280379715890307, 1, 19, 'relu'], [3.1995220781684726e-06, 1, 5, 'relu'], [9.690337835200007e-05, 1, 5, 'relu'], [3.1995220781684726e-06, 1, 5, 'relu'], [2.56833262960544e-06, 1, 5, 'relu']
```

4.2. Plotting functions
OrderedDict([('activation', 0.0001), ('learning_rate', 1), ('num_dense_layers', 64), ('num_dense_nodes', 'relu')])

print("Plotting now ...")

_ = plot_histogram(result=search_result, dimension_identifier='learning_rate', bins=20)
plt.show()
```python
_ = plot_objective_2D(result=search_result,
    dimension_identifier1='num_dense_layers',
    dimension_identifier2='num_dense_nodes')
plt.show()
```
```python
_ = plot_objective(result=search_result,
    plot_dims=['num_dense_layers',
                'num_dense_nodes'])
plt.show()
```
Total running time of the script: 0 minutes 15.564 seconds
Estimated memory usage: 9 MB
Scikit-Optimize, or skopt, is a simple and efficient library to minimize (very) expensive and noisy black-box functions. It implements several methods for sequential model-based optimization. skopt is reusable in many contexts and accessible.

## 5.1 skopt: module

### 5.1.1 Base classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BayesSearchCV</code></td>
<td>Bayesian optimization over hyper parameters.</td>
</tr>
<tr>
<td><code>Optimizer</code></td>
<td>Run bayesian optimisation loop.</td>
</tr>
<tr>
<td><code>Space</code></td>
<td>Initialize a search space from given specifications.</td>
</tr>
</tbody>
</table>

**class skopt.BayesSearchCV**

```python
skopt.BayesSearchCV(estimator, search_spaces[, ...])
```

Bayesian optimization over hyper parameters.

BayesSearchCV implements a “fit” and a “score” method. It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by `n_iter`.

Parameters are presented as a list of `skopt.space.Dimension` objects.

**Parameters**

- **estimator** [estimator object.] A object of that type is instantiated for each search point. This object is assumed to implement the scikit-learn estimator api. Either estimator needs to provide a `score` function, or `scoring` must be passed.

- **search_spaces** [dict, list of dict or list of tuple containing (dict, int).] One of these cases: 1. dictionary, where keys are parameter names (strings) and values are `skopt.space.Dimension` instances (Real, Integer or Categorical) or any other valid value that defines skopt dimension
2. list of dictionaries: a list of dictionaries, where every dictionary fits the description given in case 1 above. If a list of dictionary objects is given, then the search is performed sequentially for every parameter space with maximum number of evaluations set to self.n_iter. 3. list of (dict, int > 0): an extension of case 2 above, where first element of every tuple is a dictionary representing some search subspace, similarly as in case 2, and second element is a number of iterations that will be spent optimizing over this subspace.

**n_iter** [int, default=50] Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution. Consider increasing n_points if you want to try more parameter settings in parallel.

**optimizer_kwargs** [dict, optional] Dict of arguments passed to Optimizer. For example, 
{
    'base_estimator': 'RF'
} would use a Random Forest surrogate instead of the default Gaussian Process.

**scoring** [string, callable or None, default=None] A string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y). If None, the score method of the estimator is used.

**fit_params** [dict, optional] Parameters to pass to the fit method.

**n_jobs** [int, default=1] Number of jobs to run in parallel. At maximum there are n_points times cv jobs available during each iteration.

**n_points** [int, default=1] Number of parameter settings to sample in parallel. If this does not align with n_iter, the last iteration will sample less points. See also ask()

**pre_dispatch** [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in ‘2*n_jobs’

**cv** [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a (Stratified)KFold,
- An object to be used as a cross-validation generator.
- An iterable yielding train, test splits.

For integer/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used.

**refit** [boolean, default=True] Refit the best estimator with the entire dataset. If “False”, it is impossible to make predictions using this RandomizedSearchCV instance after fitting.

**verbose** [integer] Controls the verbosity: the higher, the more messages.

**random_state** [int or RandomState] Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions.

**error_score** ['raise' (default) or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to ‘raise’, the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.
**return_train_score** [boolean, default=False] If 'True', the cv_results_ attribute will include training scores.

**Attributes**

**cv_results_** [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas DataFrame.

For instance the below given table

```
<table>
<thead>
<tr>
<th>param_kernel</th>
<th>param_gamma</th>
<th>split0_test_score</th>
<th>...</th>
<th>rank_test_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rbf'</td>
<td>0.1</td>
<td>0.8</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>'rbf'</td>
<td>0.2</td>
<td>0.9</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>'rbf'</td>
<td>0.3</td>
<td>0.7</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

will be represented by a cv_results_ dict of:

```python
{
    'param_kernel' : masked_array(data = ['rbf', 'rbf', 'rbf'],
                                   mask = False),
    'param_gamma' : masked_array(data = [0.1, 0.2, 0.3], mask = False),
    'split0_test_score' : [0.8, 0.9, 0.7],
    'split1_test_score' : [0.82, 0.5, 0.7],
    'mean_test_score' : [0.81, 0.7, 0.7],
    'std_test_score' : [0.02, 0.2, 0.1],
    'rank_test_score' : [3, 1, 1],
    'split0_train_score' : [0.8, 0.9, 0.7],
    'split1_train_score' : [0.82, 0.5, 0.7],
    'mean_train_score' : [0.81, 0.7, 0.7],
    'std_train_score' : [0.03, 0.03, 0.04],
    'mean_fit_time' : [0.73, 0.63, 0.43, 0.49],
    'std_fit_time' : [0.01, 0.02, 0.01, 0.01],
    'mean_score_time' : [0.007, 0.06, 0.04, 0.04],
    'std_score_time' : [0.001, 0.002, 0.003, 0.005],
    'params' : [{'kernel': 'rbf', 'gamma': 0.1}, ...]
}
```

NOTE that the key 'params' is used to store a list of parameter settings dict for all the parameter candidates.

The mean_fit_time, std_fit_time, mean_score_time and std_score_time are all in seconds.

**best_estimator_** [estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if refit=False.

**optimizer_results_** [list of OptimizeResult] Contains a OptimizeResult for each search space. The search space parameter are sorted by its name.

**best_score_** [float] Score of best_estimator on the left out data.

**best_params_** [dict] Parameter setting that gave the best results on the hold out data.

**best_index_** [int] The index (of the cv_results_ arrays) which corresponds to the best candidate parameter setting.

The dict at search.cv_results_['params'][search.best_index_] gives the parameter setting for the best model, that gives the highest mean score (search.best_score_).
scikit-optimize Documentation, Release 0.9.0

scorer_  [function] Scorer function used on the held-out data to choose the best parameters for the model.

n_splits_  [int] The number of cross-validation splits (folds/iterations).

See also:

GridSearchCV  Does exhaustive search over a grid of parameters.

Notes

The parameters selected are those that maximize the score of the held-out data, according to the scoring parameter.

If n_jobs was set to a value higher than one, the data is copied for each parameter setting (and not n_jobs times).
This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set pre_dispatch. Then, the memory is copied only pre_dispatch many times. A reasonable value for pre_dispatch is 2 * n_jobs.

Examples

```python
>>> from skopt import BayesSearchCV
>>> # parameter ranges are specified by one of below
>>> from skopt.space import Real, Categorical, Integer
>>> from sklearn.datasets import load_iris
>>> from sklearn.svm import SVC
>>> from sklearn.model_selection import train_test_split

>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75, random_state=0)

>>> # log-uniform: understand as search over p = exp(x) by varying x
>>> opt = BayesSearchCV(SVC(),
                       {'C': Real(1e-6, 1e+6, prior='log-uniform'),
                        'gamma': Real(1e-6, 1e+1, prior='log-uniform'),
                        'degree': Integer(1, 8),
                        'kernel': Categorical(['linear', 'poly', 'rbf']),
                       },
                       n_iter=32,
                       random_state=0)

>>> # executes bayesian optimization
>>> _ = opt.fit(X_train, y_train)

>>> # model can be saved, used for predictions or scoring
>>> print(opt.score(X_test, y_test))
0.973...
```
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decision_function(X)</code></td>
<td>Call decision_function on the estimator with the best found parameters.</td>
</tr>
<tr>
<td><code>fit(X, y, groups, callback)</code></td>
<td>Run fit on the estimator with randomly drawn parameters.</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>inverse_transform(Xt)</code></td>
<td>Call inverse_transform on the estimator with the best found params.</td>
</tr>
<tr>
<td><code>predict(X)</code></td>
<td>Call predict on the estimator with the best found parameters.</td>
</tr>
<tr>
<td><code>predict_log_proba(X)</code></td>
<td>Call predict_log_proba on the estimator with the best found parameters.</td>
</tr>
<tr>
<td><code>predict_proba(X)</code></td>
<td>Call predict_proba on the estimator with the best found parameters.</td>
</tr>
<tr>
<td><code>score(X, y)</code></td>
<td>Return the score on the given data, if the estimator has been refit.</td>
</tr>
<tr>
<td><code>score_samples(X)</code></td>
<td>Call score_samples on the estimator with the best found parameters.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td><code>transform(X)</code></td>
<td>Call transform on the estimator with the best found parameters.</td>
</tr>
</tbody>
</table>

```python
def __init__(estimator, search_spaces, optimizer_kwargs=None, n_iter=50, scoring=None, fit_params=None, n_jobs=1, n_points=1, iid='deprecated', refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', random_state=None, error_score='raise', return_train_score=False):
```

**property classes_**

Class labels.

Only available when refit=True and the estimator is a classifier.

**decision_function(X)**

Call decision_function on the estimator with the best found parameters.

Only available if refit=True and the underlying estimator supports decision_function.

**Parameters**

- `X` [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

**Returns**

- `y_score` [ndarray of shape (n_samples,) or (n_samples, n_classes) or (n_samples, n_classes * (n_classes-1) / 2)] Result of the decision function for X based on the estimator with the best found parameters.

**fit(X, y=None, *, groups=None, callback=None, **fit_params)**

Run fit on the estimator with randomly drawn parameters.

**Parameters**

- `X` [array-like or sparse matrix, shape = [n_samples, n_features]] The training input samples.

- `y` [array-like, shape = [n_samples] or [n_samples, n_output]] Target relative to X for classification or regression (class labels should be integers or strings).
groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

callback: [callable, list of callables, optional] If callable then callback(res) is called after each parameter combination tested. If list of callables, then each callable in the list is called.

get_params(deep=True)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

inverse_transform(Xt)
Call inverse_transform on the estimator with the best found params.
Only available if the underlying estimator implements inverse_transform and refit=True.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

X [{ndarray, sparse matrix} of shape (n_samples, n_features)] Result of the inverse_transform function for Xt based on the estimator with the best found parameters.

property n_features_in_
Number of features seen during fit.
Only available when refit=True.

predict(X)
Call predict on the estimator with the best found parameters.
Only available if refit=True and the underlying estimator supports predict.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred [ndarray of shape (n_samples,)] The predicted labels or values for X based on the estimator with the best found parameters.

predict_log_proba(X)
Call predict_log_proba on the estimator with the best found parameters.
Only available if refit=True and the underlying estimator supports predict_log_proba.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns
**y_pred** [ndarray of shape (n_samples,) or (n_samples, n_classes)] Predicted class log-probabilities for X based on the estimator with the best found parameters. The order of the classes corresponds to that in the fitted attribute `classes_`.

**predict_proba(X)**
Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports predict_proba.

**Parameters**

- **X** [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

**Returns**

- **y_pred** [ndarray of shape (n_samples,) or (n_samples, n_classes)] Predicted class probabilities for X based on the estimator with the best found parameters. The order of the classes corresponds to that in the fitted attribute `classes_`.

**score(X, y=None)**
Return the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

**Parameters**

- **X** [array-like of shape (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **y** [array-like of shape (n_samples, n_output) or (n_samples,), default=None] Target relative to X for classification or regression; None for unsupervised learning.

**Returns**

- **score** [float] The score defined by `scoring` if provided, and the `best_estimator_.score` method otherwise.

**score_samples(X)**
Call score_samples on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports score_samples.

New in version 0.24.

**Parameters**

- **X** [iterable] Data to predict on. Must fulfill input requirements of the underlying estimator.

**Returns**

- **y_score** [ndarray of shape (n_samples,)] The `best_estimator_.score_samples` method.

**set_params(**params)**
Set the parameters of this estimator.

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

**Parameters**

- **params** [dict] Estimator parameters.

**Returns**
**self** [estimator instance] Estimator instance.

**property total_iterations**
Count total iterations that will be taken to explore all subspaces with fit method.

**Returns**

**max_iter**: int, total number of iterations to explore

**transform(X)**
Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports transform and refit=True.

**Parameters**

**X** [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

**Returns**

**Xt** [(ndarray, sparse matrix) of shape (n_samples, n_features)] X transformed in the new space based on the estimator with the best found parameters.

### Examples using skopt.BayesSearchCV

- Scikit-learn hyperparameter search wrapper

**skopt.Optimizer**

**class skopt.Optimizer**

dimensions, base_estimator='GP', n_random_starts=None, n_initial_points=10,
initial_point_generator='random', n_jobs=1, acq_func='gp_hedge',
acq_optimizer='auto', random_state=None, model_queue_size=None,
acq_func_kwargs=None, acq_optimizer_kwargs=None

Run bayesian optimisation loop.

An Optimizer represents the steps of a bayesian optimisation loop. To use it you need to provide your own loop mechanism. The various optimisers provided by skopt use this class under the hood.

Use this class directly if you want to control the iterations of your bayesian optimisation loop.

**Parameters**

**dimensions** [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as

- a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
- a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
- as a list of categories (for Categorical dimensions), or
- an instance of a Dimension object (Real, Integer or Categorical).

**base_estimator** ["GP", "RF", "ET", "GBRT" or sklearn regressor, default: "GP"] Should inherit from sklearn.base.RegressorMixin. In addition the predict method, should have an optional return_std argument, which returns std(Y | x) along with E[Y | x]. If base_estimator is one of ["GP", "RF", "ET", "GBRT"], a default surrogate model of the corresponding type is used corresponding to what is used in the minimize functions.

**n_random_starts** [int, default: 10] Deprecated since version 0.6: use n_initial_points instead.
n_initial_points [int, default: 10] Number of evaluations of func with initialization points before approximating it with base_estimator. Initial point generator can be changed by setting initial_point_generator.

initial_point_generator [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,
- "grid" for a uniform grid sequence

acq_func [string, default: "gp_hedge"] Function to minimize over the posterior distribution. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
- "gp_hedge" Probabilistically choose one of the above three acquisition functions at every iteration.
  - The gains $g_i$ are initialized to zero.
  - At every iteration,
    * Each acquisition function is optimised independently to propose a candidate point $X_i$.
    * Out of all these candidate points, the next point $X_{best}$ is chosen by $softmax(g_i)$
    * After fitting the surrogate model with $(X_{best}, y_{best})$, the gains are updated such that $g_i = \mu(X_i)$
  - "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.
  - "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps"

acq_optimizer [string, "sampling" or "lbfgs", default: "auto"] Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing acq_func with acq_optimizer.

- If set to "auto", then acq_optimizer is configured on the basis of the base_estimator and the space searched over. If the space is Categorical or if the estimator provided based on tree-models then this is set to be "sampling".
- If set to "sampling", then acq_func is optimized by computing acq_func at n_points randomly sampled points.
- If set to "lbfgs", then acq_func is optimized by
  - Sampling n_restarts_optimizer points randomly.
  - "lbfgs" is run for 20 iterations with these points as initial points to find local minima.
– The optimal of these local minima is used to update the prior.

**random_state** [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**n_jobs** [int, default: 1] The number of jobs to run in parallel in the base estimator, if the base_estimator supports n_jobs as parameter and base_estimator was given as string. If -1, then the number of jobs is set to the number of cores.

**acq_func_kwargs** [dict] Additional arguments to be passed to the acquisition function.

**acq_optimizer_kwargs** [dict] Additional arguments to be passed to the acquisition optimizer.

**model_queue_size** [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

**Attributes**

- **Xi** [list] Points at which objective has been evaluated.
- **yi** [scalar] Values of objective at corresponding points in Xi.
- **models** [list] Regression models used to fit observations and compute acquisition function.
- **space** [Space] An instance of skopt.space.Space. Stores parameter search space used to sample points, bounds, and type of parameters.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ask([n_points, strategy])</td>
<td>Query point or multiple points at which objective should be evaluated.</td>
</tr>
<tr>
<td>copy([random_state])</td>
<td>Create a shallow copy of an instance of the optimizer.</td>
</tr>
<tr>
<td>get_result()</td>
<td>Returns the same result that would be returned by opt.tell() but without calling tell.</td>
</tr>
<tr>
<td>run(func[, n_iter])</td>
<td>Execute ask() + tell() n_iter times</td>
</tr>
<tr>
<td>tell(x, y[, fit])</td>
<td>Record an observation (or several) of the objective function.</td>
</tr>
<tr>
<td>update_next()</td>
<td>Updates the value returned by opt.ask().</td>
</tr>
</tbody>
</table>

**__init__** *(dimensions, base_estimator='gp', n_random_starts=None, n_initial_points=10, initial_point_generator='random', n_jobs=1, acq_func='gp_hedge', acq_optimizer='auto', random_state=None, model_queue_size=None, acq_func_kwargs=None, acq_optimizer_kwargs=None)*

**ask**(n_points=None, strategy="cl_min")
Query point or multiple points at which objective should be evaluated.

**n_points** [int or None, default: None] Number of points returned by the ask method. If the value is None, a single point to evaluate is returned. Otherwise a list of points to evaluate is returned of size n_points. This is useful if you can evaluate your objective in parallel, and thus obtain more objective function evaluations per unit of time.

**strategy** [string, default: “cl_min”] Method to use to sample multiple points (see also n_points description). This parameter is ignored if n_points = None. Supported options are "cl_min", "cl_mean" or "cl_max".

- If set to "cl_min", then constant liar strategy is used with lie objective value being minimum of observed objective values. "cl_mean" and "cl_max" means mean and max of values re-
spectively. For details on this strategy see:
https://hal.archives-ouvertes.fr/hal-00732512/document

With this strategy a copy of optimizer is created, which is then asked for a point, and the point
is told to the copy of optimizer with some fake objective (lie), the next point is asked from copy,
it is also told to the copy with fake objective and so on. The type of lie defines different flavours
of cl_x strategies.

copy(random_state=None)
Create a shallow copy of an instance of the optimizer.

Parameters
random_state [int, RandomState instance, or None (default)] Set the random state of the
copy.

get_result()
Returns the same result that would be returned by opt.tell() but without calling tell

Returns
res [OptimizeResult, scipy object] OptimizeResult instance with the required information.

run(func, n_iter=1)
Execute ask() + tell() n_iter times

tell(x, y, fit=True)
Record an observation (or several) of the objective function.

Provide values of the objective function at points suggested by ask() or other points. By default a new
model will be fit to all observations. The new model is used to suggest the next point at which to evaluate
the objective. This point can be retrieved by calling ask().

To add observations without fitting a new model set fit to False.

To add multiple observations in a batch pass a list-of-lists for x and a list of scalars for y.

Parameters
x [list or list-of-lists] Point at which objective was evaluated.

y [scalar or list] Value of objective at x.

fit [bool, default: True] Fit a model to observed evaluations of the objective. A model will
only be fitted after n_initial_points points have been told to the optimizer irrespective
of the value of fit.

update_next()
Updates the value returned by opt.ask(). Useful if a parameter was updated after ask was called.

Examples using skopt.Optimizer

• Parallel optimization
• Async optimization Loop
• Exploration vs exploitation
• Use different base estimators for optimization

5.1. skopt: module
skopt.Space

class skopt.Space(dimensions)
Initialize a search space from given specifications.

Parameters

- **dimensions** [list, shape=(n_dims,)] List of search space dimensions. Each search dimension can be defined either as
  - a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
  - a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
  - as a list of categories (for Categorical dimensions), or
  - an instance of a Dimension object (Real, Integer or Categorical).

Note: The upper and lower bounds are inclusive for Integer dimensions.

Attributes

- **bounds** The dimension bounds, in the original space.
- **dimension_names** Names of all the dimensions in the search-space.
- **is_categorical** Space contains exclusively categorical dimensions
- **is_partly_categorical** Space contains any categorical dimensions
- **is_real** Returns true if all dimensions are Real
- **n_constant_dimensions** Returns the number of constant dimensions which have zero degree of freedom, e.g.
- **n_dims** The dimensionality of the original space.
- **transformed_bounds** The dimension bounds, in the warped space.

Methods

- **distance**(point_a, point_b) Compute distance between two points in this space.
- **from_yaml**(yml_path[, namespace]) Create Space from yaml configuration file
- **get_transformer**() Returns all transformers as list
- **inverse_transform**(Xt) Inverse transform samples from the warped space back to the
- **rvs**(n_samples, random_state) Draw random samples.
- **set_transformer**(transform) Sets the transformer of all dimension objects to transform
- **set_transformer_by_type**(transform, dim_type) Sets the transformer of dim_type objects to transform
- **transform**(X) Transform samples from the original space into a warped space.

__init__**(dimensions)
property bounds
    The dimension bounds, in the original space.

property dimension_names
    Names of all the dimensions in the search-space.

distance(point_a, point_b)
    Compute distance between two points in this space.

    Parameters
        point_a [array] First point.
        point_b [array] Second point.

classmethod from_yaml(yml_path, namespace=None)
    Create Space from yaml configuration file

    Parameters
        yml_path [str] Full path to yaml configuration file, example YaML below:
            Space:
                • Integer: low: -5 high: 5
                • Categorical: categories: - a - b
                • Real: low: 1.0 high: 5.0 prior: log-uniform

        namespace [str, default=None] Namespace within configuration file to use, will use first
            namespace if not provided

    Returns
        space [Space] Instantiated Space object

get_transformer()
    Returns all transformers as list

inverse_transform(Xt)
    Inverse transform samples from the warped space back to the original space.

    Parameters
        Xt [array of floats, shape=(n_samples, transformed_n_dims)] The samples to inverse transform.

    Returns
        X [list of lists, shape=(n_samples, n_dims)] The original samples.

property is_categorical
    Space contains exclusively categorical dimensions

property is_partly_categorical
    Space contains any categorical dimensions

property is_real
    Returns true if all dimensions are Real

property n_constant_dimensions
    Returns the number of constant dimensions which have zero degree of freedom, e.g. an Integer dimensions
    with (0., 0.) as bounds.
property n_dims
   The dimensionality of the original space.

rvs(n_samples=1, random_state=None)
   Draw random samples.
   The samples are in the original space. They need to be transformed before being passed to a model or
   minimizer by space.transform().

   Parameters
     n_samples [int, default=1] Number of samples to be drawn from the space.
     random_state [int, RandomState instance, or None (default)] Set random state to something
     other than None for reproducible results.

   Returns
     points [list of lists, shape=(n_points, n_dims)] Points sampled from the space.

set_transformer(transform)
   Sets the transformer of all dimension objects to transform.

   Parameters
     transform [str or list of str] Sets all transformer. when transform is a string. Otherwise,
     transform must be a list with strings with the same length as dimensions

set_transformer_by_type(transform, dim_type)
   Sets the transformer of dim_type objects to transform.

   Parameters
     transform [str] Sets all transformer of type dim_type to transform
     dim_type [type]
       Can be skopt.space.Real, skopt.space.Integer or skopt.space.Categorical

transform(X)
   Transform samples from the original space into a warped space.

   Note: this transformation is expected to be used to project samples into a suitable space for numerical
   optimization.

   Parameters
     X [list of lists, shape=(n_samples, n_dims)] The samples to transform.

   Returns
     Xt [array of floats, shape=(n_samples, transformed_n_dims)] The transformed samples.

property transformed_bounds
   The dimension bounds, in the warped space.

property transformed_n_dims
   The dimensionality of the warped space.
Examples using `skopt.Space`

- Comparing initial sampling methods
- Comparing initial sampling methods on integer space

5.1.2 Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dummy_minimize(func, dimensions[, n_calls, ...])</code></td>
<td>Random search by uniform sampling within the given bounds.</td>
</tr>
<tr>
<td><code>dump(res, filename[, store_objective])</code></td>
<td>Store an skopt optimization result into a file.</td>
</tr>
<tr>
<td><code>expected_minimum(res[, n_random_starts, ...])</code></td>
<td>Compute the minimum over the predictions of the last surrogate model.</td>
</tr>
<tr>
<td><code>expected_minimum_random_sampling(res[, ...])</code></td>
<td>Minimum search by doing naive random sampling. Returns the parameters that gave the minimum function value.</td>
</tr>
<tr>
<td><code>forest_minimize(func, dimensions[,...])</code></td>
<td>Sequential optimisation using decision trees.</td>
</tr>
<tr>
<td><code>gbrt_minimize(func, dimensions[...,])</code></td>
<td>Sequential optimization using gradient boosted trees.</td>
</tr>
<tr>
<td><code>gp_minimize(func, dimensions[...,])</code></td>
<td>Bayesian optimization using Gaussian Processes.</td>
</tr>
<tr>
<td><code>load(filename, **kwargs)</code></td>
<td>Reconstruct a skopt optimization result from a file persisted with <code>skopt.dump</code>.</td>
</tr>
</tbody>
</table>

`skopt.dummy_minimize`

`skopt.dummy_minimize(func, dimensions, n_calls=100, initial_point_generator='random', x0=None, y0=None, random_state=None, verbose=False, callback=None, model_queue_size=None, init_point_gen_kwargs=None)`

Random search by uniform sampling within the given bounds.

Parameters

- **func** [callable] Function to minimize. Should take a single list of parameters and return the objective value.

  If you have a search-space where all dimensions have names, then you can use `skopt.utils.use_named_args()` as a decorator on your objective function, in order to call it directly with the named arguments. See `use_named_args` for an example.

- **dimensions** [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as

  - a `(lower_bound, upper_bound)` tuple (for Real or Integer dimensions),
  - a `(lower_bound, upper_bound, prior)` tuple (for Real dimensions),
  - as a list of categories (for Categorical dimensions), or
  - an instance of a Dimension object (Real, Integer or Categorical).

- **n_calls** [int, default: 100] Number of calls to `func` to find the minimum.

- **initial_point_generator** [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either

  - "random" for uniform random numbers,
  - "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "1hs" for a latin hypercube sequence,
- "grid" for a uniform grid sequence

**x0** [list, list of lists or None] Initial input points.
- If it is a list of lists, use it as a list of input points.
- If it is a list, use it as a single initial input point.
- If it is None, no initial input points are used.

**y0** [list, scalar or None] Evaluation of initial input points.
- If it is a list, then it corresponds to evaluations of the function at each element of **x0**: the i-th element of **y0** corresponds to the function evaluated at the i-th element of **x0**.
- If it is a scalar, then it corresponds to the evaluation of the function at **x0**.
- If it is None and **x0** is provided, then the function is evaluated at each element of **x0**.

**random_state** [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**verbose** [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

**callback** [callable, list of callables, optional] If callable then callback(res) is called after each call to func. If list of callables, then each callable in the list is called.

**model_queue_size** [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

**Returns**

**res** [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:
- **x** [list]: location of the minimum.
- **fun** [float]: function value at the minimum.
- **x_iters** [list of lists]: location of function evaluation for each iteration.
- **func_vals** [array]: function value for each iteration.
- **space** [Space]: the optimisation space.
- **specs** [dict]: the call specifications.
- **rng** [RandomState instance]: State of the random state at the end of minimization.


**See also:**

functions *skopt.gp_minimize, skopt.forest_minimize, skopt.gbrt_minimize*
Examples using `skopt.dummy_minimize`

- Comparing surrogate models
- Visualizing optimization results

### `skopt.dump`

```python
skopt.dump(res, filename, store_objective=True, **kwargs)
```

Store an skopt optimization result into a file.

**Parameters**

- `res` [OptimizeResult, scipy object] Optimization result object to be stored.
- `filename` [string or pathlib.Path] The path of the file in which it is to be stored. The compression method corresponding to one of the supported filename extensions (`.z`, `.gz`, `.bz2`, `.xz` or `.lzma`) will be used automatically.
- `store_objective` [boolean, default=True] Whether the objective function should be stored. Set `store_objective` to `False` if your objective function (`.specs['args']['func']`) is unserializable (i.e. if an exception is raised when trying to serialize the optimization result).

Notice that if `store_objective` is set to `False`, a deep copy of the optimization result is created, potentially leading to performance problems if `res` is very large. If the objective function is not critical, one can delete it before calling `skopt.dump()` and thus avoid deep copying of `res`.

- `**kwargs` [other keyword arguments] All other keyword arguments will be passed to `joblib.dump`.

### Examples using `skopt.dump`

- Store and load skopt optimization results

### `skopt.expected_minimum`

```python
skopt.expected_minimum(res, n_random_starts=20, random_state=None)
```

Compute the minimum over the predictions of the last surrogate model. Uses `expected_minimum_random_sampling` with `n_random_starts = 100000`, when the space contains any categorical values.

**Note:** The returned minimum may not necessarily be an accurate prediction of the minimum of the true objective function.

**Parameters**

- `res` [OptimizeResult, scipy object] The optimization result returned by a `skopt` minimizer.
- `n_random_starts` [int, default=20] The number of random starts for the minimization of the surrogate model.
- `random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.
Returns
- x [list] location of the minimum.
- fun [float] the surrogate function value at the minimum.

`skopt.expected_minimum_random_sampling`

`skopt.expected_minimum_random_sampling(res, n_random_starts=100000, random_state=None)`

Minimum search by doing naive random sampling. Returns the parameters that gave the minimum function value. Can be used when the space contains any categorical values.

**Note:** The returned minimum may not necessarily be an accurate prediction of the minimum of the true objective function.

**Parameters**
- res [OptimizeResult, scipy object] The optimization result returned by a skopt minimizer.
- n_random_starts [int, default=100000] The number of random starts for the minimization of the surrogate model.
- random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**Returns**
- x [list] location of the minimum.
- fun [float] the surrogate function value at the minimum.

`skopt.forest_minimize`

`skopt.forest_minimize(func, dimensions, base_estimator='ET', n_calls=100, n_random_starts=None, n_initial_points=10, acq_func='EI', initial_point_generator='random', x0=None, y0=None, random_state=None, verbose=False, callback=None, n_points=10000, xi=0.01, kappa=1.96, n_jobs=1, model_queue_size=None)`

Sequential optimisation using decision trees.

An optimizer model is used to model the expensive to evaluate function `func`. The model is improved by sequentially evaluating the expensive function at the next best point. Thereby finding the minimum of `func` with as few evaluations as possible.

The total number of evaluations, `n_calls`, are performed like the following. If x0 is provided but not y0, then the elements of x0 are first evaluated, followed by `n_initial_points` evaluations. Finally, `n_calls - len(x0) - n_initial_points` evaluations are made guided by the surrogate model. If x0 and y0 are both provided then `n_initial_points` evaluations are first made then `n_calls - n_initial_points` subsequent evaluations are made guided by the surrogate model.

The first `n_initial_points` are generated by the `initial_point_generator`.

**Parameters**
- func [callable] Function to minimize. Should take a single list of parameters and return the objective value.

If you have a search-space where all dimensions have names, then you can use `skopt.utils.use_named_args()` as a decorator on your objective function, in order to call it
directly with the named arguments. See `skopt.utils.use_named_args()` for an example.

**dimensions** [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as

- a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
- a (lower_bound, upper_bound, prior) tuple (for Real dimensions),
- as a list of categories (for Categorical dimensions), or
- an instance of a Dimension object (Real, Integer or Categorical).

**Note:** The upper and lower bounds are inclusive for Integer dimensions.

**base_estimator** [string or Regressor, default: "ET"] The regressor to use as surrogate model. Can be either

- "RF" for random forest regressor
- "ET" for extra trees regressor
- instance of regressor with support for `return_std` in its predict method

The predefined models are initialized with good defaults. If you want to adjust the model parameters pass your own instance of a regressor which returns the mean and standard deviation when making predictions.

**n_calls** [int, default: 100] Number of calls to `func`.

**n_random_starts** [int, default: None] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

Deprecated since version 0.8: use `n_initial_points` instead.

**n_initial_points** [int, default: 10] Number of evaluations of `func` with initialization points before approximating it with `base_estimator`. Initial point generator can be changed by setting `initial_point_generator`.

**initial_point_generator** [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,
- "grid" for a uniform grid sequence

**acq_func** [string, default: "LCB"] Function to minimize over the forest posterior. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
• "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.

• "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps"

x0 [list, list of lists or None] Initial input points.
  • If it is a list of lists, use it as a list of input points.
  • If it is a list, use it as a single initial input point.
  • If it is None, no initial input points are used.

y0 [list, scalar or None] Evaluation of initial input points.
  • If it is a list, then it corresponds to evaluations of the function at each element of x0: the i-th element of y0 corresponds to the function evaluated at the i-th element of x0.
  • If it is a scalar, then it corresponds to the evaluation of the function at x0.
  • If it is None and x0 is provided, then the function is evaluated at each element of x0.

random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

verbose [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

callback [callable, optional] If provided, then callback(res) is called after call to func.

n_points [int, default: 10000] Number of points to sample when minimizing the acquisition function.

xi [float, default: 0.01] Controls how much improvement one wants over the previous best values. Used when the acquisition is either "EI" or "PI".

kappa [float, default: 1.96] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. Used when the acquisition is "LCB".

n_jobs [int, default: 1] The number of jobs to run in parallel for fit and predict. If -1, then the number of jobs is set to the number of cores.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns

res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:
  • x [list]: location of the minimum.
  • fun [float]: function value at the minimum.
  • models: surrogate models used for each iteration.
  • x_iters [list of lists]: location of function evaluation for each iteration.
  • func_vals [array]: function value for each iteration.
  • space [Space]: the optimization space.
  • specs [dict] : the call specifications.

See also:

functions `skopt.gp_minimize`, `skopt.dummy_minimize`, `skopt.gbrt_minimize`.

### Examples using `skopt.forest_minimize`

- Comparing surrogate models
- Partial Dependence Plots
- Visualizing optimization results

**skopt.gbrt_minimize**

`skopt.gbrt_minimize(func, dimensions, base_estimator=None, n_calls=100, n_random_starts=None, n_initial_points=10, initial_point_generator='random', acq_func='EI', acq_optimizer='auto', x0=None, y0=None, random_state=None, verbose=False, callback=None, n_points=10000, xi=0.01, kappa=1.96, n_jobs=1, model_queue_size=None)`

Sequential optimization using gradient boosted trees.

Gradient boosted regression trees are used to model the (very) expensive to evaluate function `func`. The model is improved by sequentially evaluating the expensive function at the next best point. Thereby finding the minimum of `func` with as few evaluations as possible.

The total number of evaluations, `n_calls`, are performed like the following. If `x0` is provided but not `y0`, then the elements of `x0` are first evaluated, followed by `n_initial_points` evaluations. Finally, `n_calls - len(x0) - n_initial_points` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_initial_points` evaluations are first made then `n_calls - n_initial_points` subsequent evaluations are made guided by the surrogate model.

The first `n_initial_points` are generated by the `initial_point_generator`.

**Parameters**

- `func` [callable] Function to minimize. Should take a single list of parameters and return the objective value.
  
  If you have a search-space where all dimensions have names, then you can use `skopt.utils.use_named_args` as a decorator on your objective function, in order to call it directly with the named arguments. See `use_named_args` for an example.

- `dimensions` [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
  
  - a `(lower_bound, upper_bound)` tuple (for Real or Integer dimensions),
  - a `(lower_bound, upper_bound, "prior")` tuple (for Real dimensions),
  - as a list of categories (for Categorical dimensions), or
  - an instance of a Dimension object (Real, Integer or Categorical).

- `base_estimator` [GradientBoostingQuantileRegressor] The regressor to use as surrogate model

- `n_calls` [int, default: 100] Number of calls to `func`. 

---

5.1. skopt: module
**n_random_starts** [int, default: None] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

Deprecated since version 0.8: use `n_initial_points` instead.

**n_initial_points** [int, default: 10] Number of evaluations of `func` with initialization points before approximating it with `base_estimator`. Initial point generator can be changed by setting `initial_point_generator`.

**initial_point_generator** [str, `InitialPointGenerator` instance, default: "random"] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,
- "grid" for a uniform grid sequence

**acq_func** [string, default: "LCB"] Function to minimize over the forest posterior. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
- "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken.
- "PIps" for negated probability of improvement per second.

**x0** [list, list of lists or None] Initial input points.

- If it is a list of lists, use it as a list of input points.
- If it is a list, use it as a single initial input point.
- If it is None, no initial input points are used.

**y0** [list, scalar or None] Evaluation of initial input points.

- If it is a list, then it corresponds to evaluations of the function at each element of `x0` : the i-th element of `y0` corresponds to the function evaluated at the i-th element of `x0`.
- If it is a scalar, then it corresponds to the evaluation of the function at `x0`.
- If it is None and `x0` is provided, then the function is evaluated at each element of `x0`.

**random_state** [int, `RandomState` instance, or None (default)] Set random state to something other than None for reproducible results.

**verbose** [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

**callback** [callable, optional] If provided, then `callback(res)` is called after call to `func`.

**n_points** [int, default: 10000] Number of points to sample when minimizing the acquisition function.

**xi** [float, default: 0.01] Controls how much improvement one wants over the previous best values. Used when the acquisition is either "EI" or "PI".
kappa [float, default: 1.96] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. Used when the acquisition is "LCB".

n_jobs [int, default: 1] The number of jobs to run in parallel for fit and predict. If -1, then the number of jobs is set to the number of cores.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns
res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:

- x [list]: location of the minimum.
- fun [float]: function value at the minimum.
- models: surrogate models used for each iteration.
- x_iters [list of lists]: location of function evaluation for each iteration.
- func_vals [array]: function value for each iteration.
- space [Space]: the optimization space.
- specs [dict]: the call specifications.
- rng [RandomState instance]: State of the random state at the end of minimization.


See also:
functions skopt.gp_minimize, skopt.dummy_minimize, skopt.forest_minimize

skopt.gp_minimize

Bayesian optimization using Gaussian Processes.

If every function evaluation is expensive, for instance when the parameters are the hyperparameters of a neural network and the function evaluation is the mean cross-validation score across ten folds, optimizing the hyperparameters by standard optimization routines would take forever!

The idea is to approximate the function using a Gaussian process. In other words the function values are assumed to follow a multivariate gaussian. The covariance of the function values are given by a GP kernel between the parameters. Then a smart choice to choose the next parameter to evaluate can be made by the acquisition function over the Gaussian prior which is much quicker to evaluate.

The total number of evaluations, n_calls, are performed like the following. If x0 is provided but not y0, then the elements of x0 are first evaluated, followed by n_initial_points evaluations. Finally, n_calls - len(x0) - n_initial_points evaluations are made guided by the surrogate model. If x0 and y0 are both provided then n_initial_points evaluations are first made then n_calls - n_initial_points subsequent evaluations are made guided by the surrogate model.

The first n_initial_points are generated by the initial_point_generator.
Parameters

**func** [callable] Function to minimize. Should take a single list of parameters and return the objective value.

If you have a search-space where all dimensions have names, then you can use `skopt.utils.use_named_args()` as a decorator on your objective function, in order to call it directly with the named arguments. See `use_named_args` for an example.

**dimensions** [[list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as

- a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
- a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
- as a list of categories (for Categorical dimensions), or
- an instance of a Dimension object (Real, Integer or Categorical).

**Note:** The upper and lower bounds are inclusive for Integer dimensions.

**base_estimator** [a Gaussian process estimator] The Gaussian process estimator to use for optimization. By default, a Matern kernel is used with the following hyperparameters tuned.

- All the length scales of the Matern kernel.
- The covariance amplitude that each element is multiplied with.
- Noise that is added to the matern kernel. The noise is assumed to be iid gaussian.

**n_calls** [int, default: 100] Number of calls to `func`.

**n_random_starts** [int, default: None] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

Deprecated since version 0.8: use `n_initial_points` instead.

**n_initial_points** [int, default: 10] Number of evaluations of `func` with initialization points before approximating it with `base_estimator`. Initial point generator can be changed by setting `initial_point_generator`.

**initial_point_generator** [str, InitialPointGenerator instance, default: ‘random’] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,

**acq_func** [string, default: "gp_hedge"] Function to minimize over the gaussian prior. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
• "gp_hedge" Probabilistically choose one of the above three acquisition functions at every iteration. The weightage given to these gains can be set by $\eta$ through `acq_func_kwargs`.
  
  - The gains $g_i$ are initialized to zero.
  - At every iteration,
    * Each acquisition function is optimised independently to propose an candidate point $X_i$.
    * Out of all these candidate points, the next point $X_{best}$ is chosen by $softmax(\eta g_i)$
    * After fitting the surrogate model with $(X_{best}, y_{best})$, the gains are updated such that $g_i = \mu(X_i)$
  
• "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.
  
• "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps"

`acq_optimizer` [string, "sampling" or "lbfgs", default: "lbfgs"] Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing `acq_func` with `acq_optimizer`.

The `acq_func` is computed at `n_points` sampled randomly.

  - If set to "auto", then `acq_optimizer` is configured on the basis of the space searched over. If the space is Categorical then this is set to be "sampling".
  - If set to "sampling", then the point among these `n_points` where the `acq_func` is minimum is the next candidate minimum.
  - If set to "lbfgs", then

    - The `n_restarts_optimizer` no. of points which the acquisition function is least are taken as start points.
    - "lbfgs" is run for 20 iterations with these points as initial points to find local minima.
    - The optimal of these local minima is used to update the prior.

`x0` [list, list of lists or None] Initial input points.

  - If it is a list of lists, use it as a list of input points.
  - If it is a list, use it as a single initial input point.
  - If it is None, no initial input points are used.

`y0` [list, scalar or None] Evaluation of initial input points.

  - If it is a list, then it corresponds to evaluations of the function at each element of `x0` : the i-th element of `y0` corresponds to the function evaluated at the i-th element of `x0`.
  - If it is a scalar, then it corresponds to the evaluation of the function at `x0`.
  - If it is None and `x0` is provided, then the function is evaluated at each element of `x0`.

`random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

`verbose` [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.
callback [callable, list of callables, optional] If callable then callback(res) is called after each call to func. If list of callables, then each callable in the list is called.

n_points [int, default: 10000] Number of points to sample to determine the next “best” point. Useless if acq_optimizer is set to "lbfgs".

n_restarts_optimizer [int, default: 5] The number of restarts of the optimizer when acq_optimizer is "lbfgs".

kappa [float, default: 1.96] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. Used when the acquisition is "LCB".

xi [float, default: 0.01] Controls how much improvement one wants over the previous best values. Used when the acquisition is either "EI" or "PI".

noise [float, default: “gaussian”]
- Use noise=“gaussian” if the objective returns noisy observations. The noise of each observation is assumed to be iid with mean zero and a fixed variance.
- If the variance is known beforehand, this can be set directly to the variance of the noise.
- Set this to a value close to zero (1e-10) if the function is noise-free. Setting to zero might cause stability issues.

n_jobs [int, default: 1] Number of cores to run in parallel while running the lbfgs optimizations over the acquisition function. Valid only when acq_optimizer is set to "lbfgs". Defaults to 1 core. If n_jobs=-1, then number of jobs is set to number of cores.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns

res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:
- x [list]: location of the minimum.
- fun [float]: function value at the minimum.
- models: surrogate models used for each iteration.
- x_iters [list of lists]: location of function evaluation for each iteration.
- func_vals [array]: function value for each iteration.
- space [Space]: the optimization space.
- specs [dict]: the call specifications.
- rng [RandomState instance]: State of the random state at the end of minimization.


See also:
functions skopt.forest_minimize, skopt.dummy_minimize, skopt.gbrt_minimize
Examples using `skopt.gp_minimize`

- Store and load skopt optimization results
- Interruptible optimization runs with checkpoints
- Tuning a scikit-learn estimator with skopt
- Comparing surrogate models
- Bayesian optimization with skopt
- Comparing initial point generation methods
- Partial Dependence Plots with categorical values
- Partial Dependence Plots 2D

**skopt.load**

`skopt.load(filename, **kwargs)`

Reconstruct a skopt optimization result from a file persisted with `skopt.dump`.

**Note:** Notice that the loaded optimization result can be missing the objective function (`.specs['args']['func']`) if `skopt.dump` was called with `store_objective=False`.

**Parameters**

- `filename` [string or `pathlib.Path`]: The path of the file from which to load the optimization result.

- `**kwargs` [other keyword arguments]: All other keyword arguments will be passed to `joblib.load`.

**Returns**

- `res` [`OptimizeResult`, `scipy object`]: Reconstructed `OptimizeResult` instance.

Examples using `skopt.load`

- Store and load skopt optimization results
- Interruptible optimization runs with checkpoints

### 5.2 `skopt.acquisition`: Acquisition

**User guide:** See the `Acquisition` section for further details.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acquisition.gaussian_acquisition_1D(X, model)</code></td>
<td>A wrapper around the acquisition function that is called by <code>fmin_l_bfgs_b</code>.</td>
</tr>
<tr>
<td><code>acquisition.gaussian_ei(X, model[, y_opt, ...])</code></td>
<td>Use the expected improvement to calculate the acquisition values.</td>
</tr>
</tbody>
</table>

continues on next page
Table 6 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acquisition.gaussian_lcb(X, model[, kappa, ...])</code></td>
<td>Use the lower confidence bound to estimate the acquisition values.</td>
</tr>
<tr>
<td><code>acquisition.gaussian_pi(X, model[, y_opt, ...])</code></td>
<td>Use the probability of improvement to calculate the acquisition values.</td>
</tr>
</tbody>
</table>

5.2.1 skopt.acquisition.gaussian_acquisition_1D

`skopt.acquisition.gaussian_acquisition_1D(X, model, y_opt=None, acq_func='LCB', acq_func_kwargs=None, return_grad=True)`

A wrapper around the acquisition function that is called by fmin_l_bfgs_b.

This is because lbfgs allows only 1-D input.

5.2.2 skopt.acquisition.gaussian_ei

`skopt.acquisition.gaussian_ei(X, model, y_opt=0.0, xi=0.01, return_grad=False)`

Use the expected improvement to calculate the acquisition values.

The conditional probability \( P(y=f(x) \mid x) \) form a gaussian with a certain mean and standard deviation approximated by the model.

The EI condition is derived by computing \( E[u(f(x))] \) where \( u(f(x)) = 0 \) if \( f(x) > y_{opt} \) and \( u(f(x)) = y_{opt} - f(x) \) if \( f(x) < y_{opt} \).

This solves one of the issues of the PI condition by giving a reward proportional to the amount of improvement got.

Note that the value returned by this function should be maximized to obtain the \( X \) with maximum improvement.

**Parameters**

- \( X \) [array-like, shape=(n_samples, n_features)] Values where the acquisition function should be computed.
- \( model \) [sklearn estimator that implements predict with return_std] The fit estimator that approximates the function through the method predict. It should have a return_std parameter that returns the standard deviation.
- \( y_{opt} \) [float, default 0] Previous minimum value which we would like to improve upon.
- \( xi \) [float, default=0.01] Controls how much improvement one wants over the previous best values. Useful only when method is set to “EI”.
- \( return\_grad \) [boolean, optional] Whether or not to return the grad. Implemented only for the case where \( X \) is a single sample.

**Returns**

- \( values \) [array-like, shape=(X.shape[0],)] Acquisition function values computed at \( X \).
5.2.3 scikit-optimize.gaussian_lcb

scikit-optimize.gaussian_lcb(X, model, kappa=1.96, return_grad=False)

Use the lower confidence bound to estimate the acquisition values.

The trade-off between exploitation and exploration is left to be controlled by the user through the parameter kappa.

Parameters

- **X** [array-like, shape (n_samples, n_features)] Values where the acquisition function should be computed.
- **model** [sklearn estimator that implements predict with return_std] The fit estimator that approximates the function through the method predict. It should have a return_std parameter that returns the standard deviation.
- **kappa** [float, default 1.96 or ‘inf’] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. If set to ‘inf’, the acquisition function will only use the variance which is useful in a pure exploration setting. Useless if method is not set to “LCB”.
- **return_grad** [boolean, optional] Whether or not to return the grad. Implemented only for the case where X is a single sample.

Returns

- **values** [array-like, shape (X.shape[0],)] Acquisition function values computed at X.
- **grad** [array-like, shape (n_samples, n_features)] Gradient at X.

5.2.4 scikit-optimize.gaussian_pi

scikit-optimize.gaussian_pi(X, model, y_opt=0.0, xi=0.01, return_grad=False)

Use the probability of improvement to calculate the acquisition values.

The conditional probability \( P(y=f(x) \mid x) \) form a gaussian with a certain mean and standard deviation approximated by the model.

The PI condition is derived by computing \( E[u(f(x))] \) where \( u(f(x)) = 1 \), if \( f(x) < y_{opt} \) and \( u(f(x)) = 0 \), if \( f(x) > y_{opt} \).

This means that the PI condition does not care about how “better” the predictions are than the previous values, since it gives an equal reward to all of them.

Note that the value returned by this function should be maximized to obtain the X with maximum improvement.

Parameters

- **X** [array-like, shape=(n_samples, n_features)] Values where the acquisition function should be computed.
- **model** [sklearn estimator that implements predict with return_std] The fit estimator that approximates the function through the method predict. It should have a return_std parameter that returns the standard deviation.
- **y_opt** [float, default 0] Previous minimum value which we would like to improve upon.
- **xi** [float, default=0.01] Controls how much improvement one wants over the previous best values. Useful only when method is set to “EI”
**return_grad** [boolean, optional] Whether or not to return the grad. Implemented only for the case where \( X \) is a single sample.

**Returns**

- **values** [[array-like, shape=(\(X\) shape[0],)] Acquisition function values computed at \( X \).

### 5.3 skopt.benchmarks: A collection of benchmark problems.

A collection of benchmark problems.  

**User guide:** See the benchmarks section for further details.

#### 5.3.1 Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skopt.benchmarks.bench1(x)</code></td>
<td>A benchmark function for test purposes.</td>
</tr>
<tr>
<td><code>skopt.benchmarks.bench1_with_time(x)</code></td>
<td>Same as <code>bench1</code> but returns the computation time (constant).</td>
</tr>
<tr>
<td><code>skopt.benchmarks.bench2(x)</code></td>
<td>A benchmark function for test purposes.</td>
</tr>
<tr>
<td><code>skopt.benchmarks.bench3(x)</code></td>
<td>A benchmark function for test purposes.</td>
</tr>
<tr>
<td><code>skopt.benchmarks.bench4(x)</code></td>
<td>A benchmark function for test purposes.</td>
</tr>
<tr>
<td><code>skopt.benchmarks.bench5(x)</code></td>
<td>A benchmark function for test purposes.</td>
</tr>
<tr>
<td><code>skopt.benchmarks.branin(x[, a, b, c, r, s, t])</code></td>
<td>Brakn-Hoo function is defined on the square ( x_1 \in [-5, 10], x_2 \in [0, 15] ).</td>
</tr>
<tr>
<td><code>skopt.benchmarks.hart6(x[, alpha, P, A])</code></td>
<td>The six dimensional Hartmann function is defined on the unit hypercube.</td>
</tr>
</tbody>
</table>

**skopt.benchmarks.bench1**

`skopt.benchmarks.bench1(x)`  
A benchmark function for test purposes.

\[
f(x) = x ** 2
\]

It has a single minima with \( f(x^*) = 0 \) at \( x^* = 0 \).
skopt.benchmarks.bench1_with_time

skopt.benchmarks.bench1_with_time(x)
   Same as bench1 but returns the computation time (constant).

skopt.benchmarks.bench2

skopt.benchmarks.bench2(x)
   A benchmark function for test purposes.
   
   \[ f(x) = \begin{cases} 
   x^2 & \text{if } x < 0 \\
   (x-5)^2 - 5 & \text{otherwise.} 
   \end{cases} \]

   It has a global minima with \( f(x^*) = -5 \) at \( x^* = 5 \).

skopt.benchmarks.bench3

skopt.benchmarks.bench3(x)
   A benchmark function for test purposes.
   
   \[ f(x) = \sin(5x^*) \cdot (1 - \tanh(x^* \cdot 2)) \]

   It has a global minima with \( f(x^*) \approx -0.9 \) at \( x^* \approx -0.3 \).

skopt.benchmarks.bench4

skopt.benchmarks.bench4(x)
   A benchmark function for test purposes.
   
   \[ f(x) = \text{float}(x^*)^2 \]

   where \( x \) is a string. It has a single minima with \( f(x^*) = 0 \) at \( x^* = "0" \). This benchmark is used for checking support of categorical variables.

skopt.benchmarks.bench5

skopt.benchmarks.bench5(x)
   A benchmark function for test purposes.
   
   \[ f(x) = \text{float}(x[0])^2 + x[1]^2 \]

   where \( x \) is a string. It has a single minima with \( f(x) = 0 \) at \( x[0] = "0" \) and \( x[1] = "0" \) This benchmark is used for checking support of mixed spaces.

skopt.benchmarks.branin

skopt.benchmarks.branin(x, a=1, b=0.12918450914398066, c=1.5915494309189535, r=6, s=10,
   t=0.039788735772973836)

   Branin-Hoo function is defined on the square \( x_1 \in [-5, 10], x_2 \in [0, 15] \).

   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>

5.3. skopt.benchmarks: A collection of benchmark problems.
Examples using `skopt.benchmarks.branin`

- Parallel optimization
- Comparing surrogate models
- Comparing initial point generation methods
- Visualizing optimization results

`skopt.benchmarks.hart6`

`skopt.benchmarks.hart6(x, alpha=\text{array([1., 1.2, 3., 3.2])})`

\[
P = \text{array([[0.1312, 0.1696, 0.5569, 0.0124, 0.8283, 0.5886],
                  [0.2329, 0.4135, 0.8307, 0.3736, 0.1004, 0.9991],
                  [0.2348, 0.1451, 0.3522, 0.2883, 0.3047, 0.665]],
                  [0.4047, 0.8828, 0.8732, 0.5743, 0.1091, 0.0381])])
\]

\[
A = \text{array([[10., 3., 17., 3.5, 1.7, 8.],
                [0.05, 10., 17., 0.1, 8., 14.],
                [3., 3.5, 1.7, 10., 17., 8.],
                [17., 8., 0.05, 10., 0.1, 14.]]})
\]

The six dimensional Hartmann function is defined on the unit hypercube.

It has six local minima and one global minimum \( f(x^*) = -3.32237 \) at \( x^* = (0.20169, 0.15001, 0.476874, 0.275332, 0.311652, 0.6573) \).

More details: <http://www.sfu.ca/~ssurjano/hart6.html>

Examples using `skopt.benchmarks.hart6`

- Comparing initial point generation methods
- Visualizing optimization results

5.4 `skopt.callbacks`: Callbacks

Monitor and influence the optimization procedure via callbacks.

Callbacks are callables which are invoked after each iteration of the optimizer and are passed the results “so far”. Callbacks can monitor progress, or stop the optimization early by returning `True`.

User guide: See the `Callbacks` section for further details.

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>callbacks.CheckpointSaver</code></td>
<td>Save current state after each iteration with <code>skopt.dump</code>.</td>
</tr>
<tr>
<td><code>callbacks.DeadlineStopper</code></td>
<td>Stop the optimization before running out of a fixed budget of time.</td>
</tr>
<tr>
<td><code>callbacks.DeltaXStopper</code></td>
<td>Stop the optimization when (</td>
</tr>
<tr>
<td><code>callbacks.DeltaYStopper</code></td>
<td>Stop the optimization if the <code>n_best</code> minima are within <code>delta</code></td>
</tr>
<tr>
<td><code>callbacks.EarlyStopper</code></td>
<td>Decide to continue or not given the results so far.</td>
</tr>
<tr>
<td><code>callbacks.TimerCallback</code></td>
<td>Log the elapsed time between each iteration of the minimization loop.</td>
</tr>
<tr>
<td><code>callbacks.VerboseCallback</code></td>
<td>Callback to control the verbosity.</td>
</tr>
</tbody>
</table>
5.4.1 skopt.callbacks.CheckpointSaver

class skopt.callbacks.CheckpointSaver(checkpoint_path, **dump_options)

Save current state after each iteration with skopt.dump.

Parameters

   checkpoint_path  [string] location where checkpoint will be saved to;
   dump_options  [string] options to pass on to skopt.dump, like compress=9

Examples

>>> import skopt
>>> def obj_fun(x):
... return x[0]**2
... return x[0]**2
>>> checkpoint_callback = skopt.callbacks.CheckpointSaver("./result.pkl")
>>> skopt.gp_minimize(obj_fun, [(-2, 2)], n_calls=10,
... callback=[checkpoint_callback])

Methods

   __call__(res)

Parameters

   __init__(checkpoint_path, **dump_options)

Examples using skopt.callbacks.CheckpointSaver

   Interruptible optimization runs with checkpoints

5.4.2 skopt.callbacks.DeadlineStopper

class skopt.callbacks.DeadlineStopper(total_time)

Stop the optimization before running out of a fixed budget of time.

Parameters

   total_time  [float] fixed budget of time (seconds) that the optimization must finish within.

Attributes

   iter_time  [list, shape (n_iter,)] iter_time[i-1] gives the time taken to complete iteration i
Methods

__call__(result)

Parameters

__init__(total_time)

5.4.3 `skopt.callbacks.DeltaXStopper`

class `skopt.callbacks.DeltaXStopper`(delta)

Stop the optimization when |x1 - x2| < delta

If the last two positions at which the objective has been evaluated are less than delta apart stop the optimization procedure.

Methods

__call__(result)

Parameters

__init__(delta)

5.4.4 `skopt.callbacks.DeltaYStopper`

class `skopt.callbacks.DeltaYStopper`(delta, n_best=5)

Stop the optimization if the n_best minima are within delta

Stop the optimizer if the absolute difference between the n_best objective values is less than delta.

Methods

__call__(result)

Parameters

__init__(delta, n_best=5)
5.4.5 `skopt.callbacks.EarlyStopper`

class `skopt.callbacks.EarlyStopper`
Decide to continue or not given the results so far.
The optimization procedure will be stopped if the callback returns True.

Methods

__call__(result)

Parameters

__init__(*args, **kwargs)

5.4.6 `skopt.callbacks.TimerCallback`

class `skopt.callbacks.TimerCallback`
Log the elapsed time between each iteration of the minimization loop.
The time for each iteration is stored in the iter_time attribute which you can inspect after the minimization has completed.

Attributes

iter_time [list, shape (n_iter,)] iter_time[i-1] gives the time taken to complete iteration i

Methods

__call__(res)

Parameters

__init__()

5.4.7 `skopt.callbacks.VerboseCallback`

class `skopt.callbacks.VerboseCallback`(n_total, n_init=0, n_random=0)
Callback to control the verbosity.

Parameters

n_init [int, optional] Number of points provided by the user which are yet to be evaluated. This is equal to len(x0) when y0 is None

n_random [int, optional] Number of points randomly chosen.

n_total [int] Total number of func calls.

Attributes
iter_no  [int] Number of iterations of the optimization routine.

Methods

__call__(res)

Parameters

__init__(n_total, n_init=0, n_random=0)

5.5 skopt.learning: Machine learning extensions for model-based optimization.

Machine learning extensions for model-based optimization.

User guide: See the learning section for further details.

| learning.ExtraTreesRegressor([n_estimators, ...) | ExtraTreesRegressor that supports conditional standard deviation. |
| learning.GaussianProcessRegressor([kernel, ...) | GaussianProcessRegressor that allows noise tunability. |
| learning.GradientBoostingQuantileRegressor([...) | Predict several quantiles with one estimator. |
| learning.RandomForestRegressor([...) | RandomForestRegressor that supports conditional std computation. |

5.5.1 skopt.learning.ExtraTreesRegressor

class skopt.learning.ExtraTreesRegressor(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_decrease=0.0, bootstrap=False, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, min_variance=0.0)

ExtraTreesRegressor that supports conditional standard deviation.

Parameters

- **n_estimators** [integer, optional (default=10)] The number of trees in the forest.
- **criterion** [string, optional (default="mse"))] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.
- **max_features** [int, float, string or None, optional (default="auto")]] The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
• If “auto”, then max_features=n_features.
• If “sqrt”, then max_features=sqrt(n_features).
• If “log2”, then max_features=log2(n_features).
• If None, then max_features=n_features.

Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:
  • If int, then consider min_samples_split as the minimum number.
  • If float, then min_samples_split is a percentage and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node:
  • If int, then consider min_samples_leaf as the minimum number.
  • If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:

\[
\frac{N_t}{N} \left( \text{impurity} - \frac{N_{t,R}}{N_t} \times \text{right_impurity} - \frac{N_{t,L}}{N_t} \times \text{left_impurity} \right)
\]

where \( N \) is the total number of samples, \( N_t \) is the number of samples at the current node, \( N_{t,L} \) is the number of samples in the left child, and \( N_{t,R} \) is the number of samples in the right child. \( N, N_t, N_{t,R} \) and \( N_{t,L} \) all refer to the weighted sum, if sample_weight is passed.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees.

oob_score [bool, optional (default=False)] whether to use out-of-bag samples to estimate the R^2 on unseen data.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel for both fit and predict. If -1, then the number of jobs is set to the number of cores.
random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by \texttt{np.random}.

verbose [int, optional (default=0)] Controls the verbosity of the tree building process.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest.

Attributes

estimators_ [list of DecisionTreeRegressor] The collection of fitted sub-estimators.

feature_importances_ [array of shape = [n_features]] The impurity-based feature importances.

n_features_ [int] DEPRECATED: Attribute n_features_ was deprecated in version 1.0 and will be removed in 1.2.

n_outputs_ [int] The number of outputs when fit is performed.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_prediction_ [array of shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set.

Notes

The default values for the parameters controlling the size of the trees (e.g. max_depth, min_samples_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values. The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, max_features=n_features and bootstrap=False, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, random_state has to be fixed.

References

[1]

Methods

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

decision_path(X) Return the decision path in the forest.

fit(X, y[, sample_weight]) Build a forest of trees from the training set (X, y).

get_params([deep]) Get parameters for this estimator.

predict(X[, return_std]) Predict continuous output for X.

score(X, y[, sample_weight]) Return the coefficient of determination of the prediction.

set_params(**params) Set the parameters of this estimator.
Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like, sparse matrix} of 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 [ndarray of 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, sparse matrix} of 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 matrix of shape (n_samples, n_nodes)] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes. The matrix is of CSR format.

n_nodes_ptr [ndarray of shape (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.

property feature_importances_

The impurity-based feature importances.

The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values). See sklearn.inspection.permutation_importance() as an alternative.

Returns

feature_importances_ [ndarray of shape (n_features,)] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit(X, y, sample_weight=None)

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

Parameters

X [{array-like, 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 csr_matrix.
y [array-like of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like of shape (n_samples,), default=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] Fitted estimator.

get_params(deep=True)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

property n_features_
Attribute n_features_ was deprecated in version 1.0 and will be removed in 1.2. Use n_features_in_ instead.

Number of features when fitting the estimator.

Type DEPRECATED

predict(X, return_std=False)
Predict continuous output for X.

Parameters

X [array-like of shape=(n_samples, n_features)] Input data.

return_std [boolean] Whether or not to return the standard deviation.

Returns

predictions [array-like of shape=(n_samples,)] Predicted values for X. If criterion is set to "mse", then predictions[i] ~= mean(y | X[i]).

std [array-like of shape=(n_samples,)] Standard deviation of y at X. If criterion is set to "mse", then std[i] ~= std(y | X[i]).

score(X, y, sample_weight=None)
Return the coefficient of determination of the prediction.

The coefficient of determination $R^2$ is defined as $(1 - \frac{u}{v})$, where $u$ is the residual sum of squares $((y_{true} - y_{pred})** 2).sum()$ and $v$ is the total sum of squares $((y_{true} - y_{true}.mean()) ** 2).sum()$. The 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 of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.
y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] \( R^2 \) of self.predict(X) wrt. y.

Notes

The \( R^2 \) score used when calling score on a regressor uses multioutput='uniform_average' from version 0.23 to keep consistent with default value of r2_score(). This influences the score method of all the multioutput regressors (except for MultiOutputRegressor).

set_params(**params)

Set the parameters of this estimator.

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

Parameters

**params [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.

5.5.2 skopt.learning.GaussianProcessRegressor

class skopt.learning.GaussianProcessRegressor(kernel=None, alpha=1e-10, optimizer='fmin_l_bfgs_b', n_restarts_optimizer=0, normalize_y=False, copy_X_train=True, random_state=None, noise=None)

GaussianProcessRegressor that allows noise tunability.

The implementation is based on Algorithm 2.1 of Gaussian Processes for Machine Learning (GPML) by Rasmussen and Williams.

In addition to standard scikit-learn estimator API, GaussianProcessRegressor:

- allows prediction without prior fitting (based on the GP prior);
- provides an additional method sample_y(X), which evaluates samples drawn from the GPR (prior or posterior) at given inputs;
- exposes a method log_marginal_likelihood(theta), which can be used externally for other ways of selecting hyperparameters, e.g., via Markov chain Monte Carlo.

Parameters

kernel [kernel object] The kernel specifying the covariance function of the GP. If None is passed, the kernel “1.0 * RBF(1.0)” is used as default. Note that the kernel’s hyperparameters are optimized during fitting.

alpha [float or array-like, optional (default: 1e-10)] Value added to the diagonal of the kernel matrix during fitting. Larger values correspond to increased noise level in the observations and reduce potential numerical issue during fitting. If an array is passed, it must have the same number of entries as the data used for fitting and is used as datapoint-dependent noise level. Note that this is equivalent to adding a WhiteKernel with c=alpha. Allowing to specify
the noise level directly as a parameter is mainly for convenience and for consistency with Ridge.

**optimizer** [string or callable, optional (default: “fmin_l_bfgs_b”)] Can either be one of the internally supported optimizers for optimizing the kernel’s parameters, specified by a string, or an externally defined optimizer passed as a callable. If a callable is passed, it must have the signature:

```python
def optimizer(obj_func, initial_theta, bounds):
    # * `obj_func` is the objective function to be maximized, which
    # takes the hyperparameters theta as parameter and an
    # optional flag eval_gradient, which determines if the
    # gradient is returned additionally to the function value
    # * `initial_theta`: the initial value for theta, which can be
    # used by local optimizers
    # * `bounds`: the bounds on the values of theta
    ....
    # Returned are the best found hyperparameters theta and
    # the corresponding value of the target function.
    return theta_opt, func_min
```

Per default, the ‘fmin_l_bfgs_b’ algorithm from scipy.optimize is used. If None is passed, the kernel’s parameters are kept fixed. Available internal optimizers are:

```
'fmin_l_bfgs_b'
```

**n_restarts_optimizer** [int, optional (default: 0)] The number of restarts of the optimizer for finding the kernel’s parameters which maximize the log-marginal likelihood. The first run of the optimizer is performed from the kernel’s initial parameters, the remaining ones (if any) from thetas sampled log-uniform randomly from the space of allowed theta-values. If greater than 0, all bounds must be finite. Note that n_restarts_optimizer == 0 implies that one run is performed.

**normalize_y** [boolean, optional (default: False)] Whether the target values y are normalized, i.e., the mean of the observed target values become zero. This parameter should be set to True if the target values’ mean is expected to differ considerably from zero. When enabled, the normalization effectively modifies the GP’s prior based on the data, which contradicts the likelihood principle; normalization is thus disabled per default.

**copy_X_train** [bool, optional (default: True)] If True, a persistent copy of the training data is stored in the object. Otherwise, just a reference to the training data is stored, which might cause predictions to change if the data is modified externally.

**random_state** [integer or numpy.RandomState, optional] The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

**noise** [string, “gaussian”, optional] If set to “gaussian”, then it is assumed that y is a noisy estimate of $f(x)$ where the noise is gaussian.

**Attributes**

- **X_train_** [array-like, shape = (n_samples, n_features)] Feature values in training data (also required for prediction)
- **y_train_** [array-like, shape = (n_samples, [n_output_dims])] Target values in training data (also required for prediction)
**kernel** _kernel object_  The kernel used for prediction. The structure of the kernel is the same as the one passed as parameter but with optimized hyperparameters

L_ [array-like, shape = (n_samples, n_samples)] Lower-triangular Cholesky decomposition of the kernel in X_train_

alpha_ [array-like, shape = (n_samples,)] Dual coefficients of training data points in kernel space

log_marginal_likelihood_value_ [float] The log-marginal-likelihood of self.kernel_.theta

noise_ [float] Estimate of the gaussian noise. Useful only when noise is set to “gaussian”.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit(X, y)</td>
<td>Fit Gaussian process regression model.</td>
</tr>
<tr>
<td>get_params([deep])</td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td>log_marginal_likelihood([theta, ...])</td>
<td>Return log-marginal likelihood of theta for training data.</td>
</tr>
<tr>
<td>predict(X, return_std, return_cov, ...))</td>
<td>Predict output for X.</td>
</tr>
<tr>
<td>sample_y(X, n_samples, random_state)</td>
<td>Draw samples from Gaussian process and evaluate at X.</td>
</tr>
<tr>
<td>score(X, y, sample_weight)</td>
<td>Return the coefficient of determination of the prediction.</td>
</tr>
<tr>
<td>set_params(**params)</td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

**__init__**(kernel=None, alpha=1e-10, optimizer='fmin_l_bfgs_b', n_restarts_optimizer=0, normalize_y=False, copy_X_train=True, random_state=None, noise=None)

fit(X, y)
Fit Gaussian process regression model.

Parameters

X [array-like, shape = (n_samples, n_features)] Training data

y [array-like, shape = (n_samples, [n_output_dims])] Target values

Returns

self Returns an instance of self.

get_params(deep=True)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

log_marginal_likelihood(theta=theta, eval_gradient=False, clone_kernel=True)
Return log-marginal likelihood of theta for training data.

Parameters

5.5. skopt.learning: Machine learning extensions for model-based optimization.
theta [array-like of shape (n_kernel_params,) default=None] Kernel hyperparameters for which the log-marginal likelihood is evaluated. If None, the precomputed log_marginal_likelihood of self.kernel_.theta is returned.

eval_gradient [bool, default=False] If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. If True, theta must not be None.

cloned_kernel [bool, default=True] If True, the kernel attribute is copied. If False, the kernel attribute is modified, but may result in a performance improvement.

Returns

log_likelihood [float] Log-marginal likelihood of theta for training data.

log_likelihood_gradient [ndarray of shape (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

predict(X, return_std=False, return_cov=False, return_mean_grad=False, return_std_grad=False)
Predict output for X.

In addition to the mean of the predictive distribution, also its standard deviation (return_std=True) or co-variance (return_cov=True), the gradient of the mean and the standard-deviation with respect to X can be optionally provided.

Parameters

X [array-like, shape = (n_samples, n_features)] Query points where the GP is evaluated.

return_std [bool, default: False] If True, the standard-deviation of the predictive distribution at the query points is returned along with the mean.

return_cov [bool, default: False] If True, the covariance of the joint predictive distribution at the query points is returned along with the mean.

return_mean_grad [bool, default: False] Whether or not to return the gradient of the mean. Only valid when X is a single point.

return_std_grad [bool, default: False] Whether or not to return the gradient of the std. Only valid when X is a single point.

Returns

y_mean [array, shape = (n_samples, [n_output_dims])] Mean of predictive distribution a query points

y_std [array, shape = (n_samples,), optional] Standard deviation of predictive distribution at query points. Only returned when return_std is True.

y_cov [array, shape = (n_samples, n_samples), optional] Covariance of joint predictive distribution a query points. Only returned when return_cov is True.

y_mean_grad [shape = (n_samples, n_features)] The gradient of the predicted mean

y_std_grad [shape = (n_samples, n_features)] The gradient of the predicted std.

sample_y(X, n_samples=1, random_state=0)
Draw samples from Gaussian process and evaluate at X.

Parameters

X [array-like of shape (n_samples_X, n_features) or list of object] Query points where the GP is evaluated.
n_samples [int, default=1] Number of samples drawn from the Gaussian process per query point.

random_state [int, RandomState instance or None, default=0] Determines random number generation to randomly draw samples. Pass an int for reproducible results across multiple function calls. See Glossary.

Returns

y_samples [ndarray of shape (n_samples_X, n_samples), or (n_samples_X, n_targets, n_samples)] Values of n_samples samples drawn from Gaussian process and evaluated at query points.

score(X, y, sample_weight=None)
Return the coefficient of determination of the prediction.

The coefficient of determination $R^2$ is defined as $1 - \frac{u}{v}$, where $u$ is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and $v$ is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The 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 of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] $R^2$ of self.predict(X) wrt. y.

Notes

The $R^2$ score used when calling score on a regressor uses multioutput='uniform_average' from version 0.23 to keep consistent with default value of r2_score(). This influences the score method of all the multioutput regressors (except for MultiOutputRegressor).

set_params(**params)
Set the parameters of this estimator.

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

Parameters

**params [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.
Examples using `skopt.learning.GaussianProcessRegressor`

- Use different base estimators for optimization

### 5.5.3 `skopt.learning.GradientBoostingQuantileRegressor`

**class skopt.learning.GradientBoostingQuantileRegressor**

```python
class skopt.learning.GradientBoostingQuantileRegressor(quantiles=[0.16, 0.5, 0.84], base_estimator=None, n_jobs=1, random_state=None)
```

Predict several quantiles with one estimator.

This is a wrapper around `GradientBoostingRegressor`'s quantile regression that allows you to predict several quantiles in one go.

**Parameters**

- **quantiles** [array-like] Quantiles to predict. By default the 16, 50 and 84% quantiles are predicted.
- **base_estimator** [GradientBoostingRegressor instance or None (default)] Quantile regressor used to make predictions. Only instances of `GradientBoostingRegressor` are supported. Use this to change the hyper-parameters of the estimator.
- **n_jobs** [int, default=1] The number of jobs to run in parallel for `fit`. If -1, then the number of jobs is set to the number of cores.
- **random_state** [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**Methods**

- **fit(X, y)** Fit one regressor for each quantile.
- **get_params([deep])** Get parameters for this estimator.
- **predict(X[, return_std, return_quantiles])** Predict.
- **score(X, y[, sample_weight])** Return the coefficient of determination of the prediction.
- **set_params(**params)** Set the parameters of this estimator.

```
__init__(quantiles=[0.16, 0.5, 0.84], base_estimator=None, n_jobs=1, random_state=None)
```

*fit(X, y)*

Fit one regressor for each quantile.

**Parameters**

- **X** [array-like, shape=(n_samples, n_features)] Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.
- **y** [array-like, shape=(n_samples,)] Target values (real numbers in regression)

*get_params*(deep=True)*

Get parameters for this estimator.

**Parameters**

- **deep** [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.
Returns

params [dict] Parameter names mapped to their values.

predict(X, return_std=False, return_quantiles=False)

Predict.

Predict X at every quantile if return_std is set to False. If return_std is set to True, then return the mean and the predicted standard deviation, which is approximated as the (0.84th quantile - 0.16th quantile) divided by 2.0

Parameters

X [array-like, shape=(n_samples, n_features)] where n_samples is the number of samples and n_features is the number of features.

score(X, y, sample_weight=None)

Return the coefficient of determination of the prediction.

The coefficient of determination $R^2$ is defined as $(1 - \frac{u}{v})$, where $u$ is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and $v$ is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The 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 of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] $R^2$ of self.predict(X) wrt. y.

Notes

The $R^2$ score used when calling score on a regressor uses multioutput='uniform_average' from version 0.23 to keep consistent with default value of r2_score(). This influences the score method of all the multioutput regressors (except for MultiOutputRegressor).

set_params(**params)

Set the parameters of this estimator.

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

Parameters

**params [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.
RandomForestRegressor that supports conditional std computation.

Parameters

**n_estimators** [integer, optional (default=10)] The number of trees in the forest.

**criterion** [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are “mse” for the mean squared error, which is equal to variance reduction as feature selection criterion, and “mae” for the mean absolute error.

**max_features** [int, float, string or None, optional (default="auto")]) The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
- If “auto”, then max_features=n_features.
- If “sqrt”, then max_features=\sqrt{n_features}.
- If “log2”, then max_features=log2(n_features).
- If None, then max_features=n_features.

Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

**max_depth** [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

**min_samples_split** [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a percentage and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

**min_samples_leaf** [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node:

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

**min_weight_fraction_leaf** [float, optional (default=0.0)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
**max_leaf_nodes** [int or None, optional (default=None)] Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

**min_impurity_decrease** [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:

\[
\frac{N_t}{N} \times (\text{impurity} - \frac{N_{t_R}}{N_t} \times \text{right_impurity} - \frac{N_{t_L}}{N_t} \times \text{left_impurity})
\]

where \( N \) is the total number of samples, \( N_t \) is the number of samples at the current node, \( N_{t,L} \) is the number of samples in the left child, and \( N_{t,R} \) is the number of samples in the right child. \( N, N_t, N_{t,R}, \text{and } N_{t,L} \) all refer to the weighted sum, if sample_weight is passed.

**bootstrap** [boolean, optional (default=True)] Whether bootstrap samples are used when building trees.

**oob_score** [bool, optional (default=False)] whether to use out-of-bag samples to estimate the \( R^2 \) on unseen data.

**n_jobs** [integer, optional (default=1)] The number of jobs to run in parallel for both fit and predict. If -1, then the number of jobs is set to the number of cores.

**random_state** [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

**verbose** [int, optional (default=0)] Controls the verbosity of the tree building process.

**warm_start** [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest.

**Attributes**

**estimators_** [list of DecisionTreeRegressor] The collection of fitted sub-estimators.

**feature_importances_** [array of shape = [n_features]] The impurity-based feature importances.

**n_features_** [int] DEPRECATED: Attribute n_features_ was deprecated in version 1.0 and will be removed in 1.2.

**n_outputs_** [int] The number of outputs when fit is performed.

**oob_score_** [float] Score of the training dataset obtained using an out-of-bag estimate.

**oob_prediction_** [array of shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set.
Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values. The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, `max_features=n_features` and `bootstrap=False`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[1]

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply(X)</code></td>
<td>Apply trees in the forest to X, return leaf indices.</td>
</tr>
<tr>
<td><code>decision_path(X)</code></td>
<td>Return the decision path in the forest.</td>
</tr>
<tr>
<td><code>fit(X, y[, sample_weight])</code></td>
<td>Build a forest of trees from the training set (X, y).</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>predict(X[, return_std])</code></td>
<td>Predict continuous output for X.</td>
</tr>
<tr>
<td><code>score(X, y[, sample_weight])</code></td>
<td>Return the coefficient of determination of the prediction.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

__init__(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_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, min_variance=0.0)

apply(X)

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

Parameters

- **X**: [[array-like, sparse matrix] of 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**: [ndarray of 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, sparse matrix] of 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 matrix of shape (n_samples, n_nodes)] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes. The matrix is of CSR format.

**n_nodes_ptr** [ndarray of shape (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.

**property feature_importances_**

The impurity-based feature importances.

The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values). See `sklearn.inspection.permutation_importance()` as an alternative.

Returns

**feature_importances_** [ndarray of shape (n_features,)] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

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

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

Parameters

**X** [(array-like, 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 of shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

**sample_weight** [array-like of shape (n_samples,), default=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] Fitted estimator.

**get_params**(deep=True)

Get parameters for this estimator.

Parameters

**deep** [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

**params** [dict] Parameter names mapped to their values.

**property n_features_**

Attribute n_features_ was deprecated in version 1.0 and will be removed in 1.2. Use n_features_in_ instead.

Number of features when fitting the estimator.

**Type** DEPRECATED
predict($X$, return_std=False)
Predict continuous output for $X$.

**Parameters**

$X$ [array of shape = (n_samples, n_features)] Input data.

return_std [boolean] Whether or not to return the standard deviation.

**Returns**

predictions [array-like of shape = (n_samples,)] Predicted values for $X$. If criterion is set to “mse”, then predictions[i] $\sim=\text{mean}(y \mid X[i])$.

std [array-like of shape=(n_samples,)] Standard deviation of $y$ at $X$. If criterion is set to “mse”, then std[i] $\sim=\text{std}(y \mid X[i])$.

score($X$, $y$, sample_weight=None)
Return the coefficient of determination of the prediction.

The coefficient of determination $R^2$ is defined as $(1 - \frac{u}{v})$, where $u$ is the residual sum of squares ($(y_{true} - y_{pred})^2$.sum()) and $v$ is the total sum of squares ($(y_{true} - y_{true}.mean())^2$.sum()). The 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 of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

$y$ [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for $X$.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

**Returns**

score [float] $R^2$ of self.predict($X$) wrt $y$.

**Notes**

The $R^2$ score used when calling score on a regressor uses multioutput='uniform_average' from version 0.23 to keep consistent with default value of r2_score(). This influences the score method of all the multioutput regressors (except for MultiOutputRegressor).

set_params(**params)
Set the parameters of this estimator.

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

**Parameters**

**params [dict] Estimator parameters.

**Returns**

self [estimator instance] Estimator instance.
5.6 skopt.optimizer: Optimizer

User guide: See the Optimizer, an ask-and-tell interface section for further details.

optimizer.Optimizer(dimensions[, ...]) Run bayesian optimisation loop.

5.6.1 skopt.optimizer.Optimizer

class skopt.optimizer.Optimizer(dimensions, base_estimator='gp', n_random_starts=None, 
n_initial_points=10, initial_point_generator='random', n_jobs=1, 
acq_func='gp_hedge', acq_optimizer='auto', random_state=None, 
model_queue_size=None, acq_func_kwars=None, 
acq_optimizer_kwars=None)

Run bayesian optimisation loop.

An Optimizer represents the steps of a bayesian optimisation loop. To use it you need to provide your own loop mechanism. The various optimisers provided by skopt use this class under the hood.

Use this class directly if you want to control the iterations of your bayesian optimisation loop.

Parameters

- **dimensions** [list, shape (nDims,)] List of search space dimensions. Each search dimension can be defined either as
  - a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
  - a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
  - as a list of categories (for Categorical dimensions), or
  - an instance of a Dimension object (Real, Integer or Categorical).

- **base_estimator** ["GP", "RF", "ET", "GBRT" or sklearn regressor, default: "GP"] Should inherit from sklearn.base.RegressorMixin. In addition the predict method, should have an optional return_std argument, which returns std(Y | x) along with E[Y | x]. If base_estimator is one of ["GP", "RF", "ET", "GBRT"], a default surrogate model of the corresponding type is used corresponding to what is used in the minimize functions.

- **n_random_starts** [int, default: 10] Deprecated since version 0.6: use n_initial_points instead.

- **n_initial_points** [int, default: 10] Number of evaluations of func with initialization points before approximating it with base_estimator. Initial point generator can be changed by setting initial_point_generator.

- **initial_point_generator** [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either
  - "random" for uniform random numbers,
  - "sobol" for a Sobol’ sequence,
  - "halton" for a Halton sequence,
  - "hammersly" for a Hammersly sequence,
  - "lhs" for a latin hypercube sequence,
  - "grid" for a uniform grid sequence
acq_func [string, default: "gp_hedge"] Function to minimize over the posterior distribution. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
- "gp_hedge" Probabilistically choose one of the above three acquisition functions at every iteration.
  - The gains $g_i$ are initialized to zero.
  - At every iteration,
    - Each acquisition function is optimised independently to propose an candidate point $X_i$.
    - Out of all these candidate points, the next point $X_{best}$ is chosen by $softmax(g_i)$.
    - After fitting the surrogate model with $(X_{best}, y_{best})$, the gains are updated such that $g_i = \mu(X_i)$.
  - "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.
  - "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps".

docstring: acq_optimizer [string, "sampling" or "lbfgs", default: "auto"] Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing acq_func with acq_optimizer.

- If set to "auto", then acq_optimizer is configured on the basis of the base_estimator and the space searched over. If the space is Categorical or if the estimator provided based on tree-models then this is set to be "sampling".
- If set to "sampling", then acq_func is optimized by computing acq_func at $n_{points}$ randomly sampled points.
- If set to "lbfgs", then acq_func is optimized by
  - Sampling $n_{restarts}_{optimizer}$ points randomly.
  - "lbfgs" is run for 20 iterations with these points as initial points to find local minima.
  - The optimal of these local minima is used to update the prior.

random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

n_jobs [int, default: 1] The number of jobs to run in parallel in the base_estimator, if the base_estimator supports n_jobs as parameter and base_estimator was given as string. If -1, then the number of jobs is set to the number of cores.

acq_func_kwargs [dict] Additional arguments to be passed to the acquisition function.

acq_optimizer_kwargs [dict] Additional arguments to be passed to the acquisition optimizer.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Attributes
Xi  [list] Points at which objective has been evaluated.

yi  [scalar] Values of objective at corresponding points in Xi.

models  [list] Regression models used to fit observations and compute acquisition function.

space  [Space] An instance of skopt.space.Space. Stores parameter search space used to sample points, bounds, and type of parameters.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ask([n_points, strategy])</td>
<td>Query point or multiple points at which objective should be evaluated.</td>
</tr>
<tr>
<td>copy([random_state])</td>
<td>Create a shallow copy of an instance of the optimizer.</td>
</tr>
<tr>
<td>get_result()</td>
<td>Returns the same result that would be returned by opt.tell() but without calling tell.</td>
</tr>
<tr>
<td>run(func[, n_iter])</td>
<td>Execute ask() + tell() n_iter times.</td>
</tr>
<tr>
<td>tell(x, y[, fit])</td>
<td>Record an observation (or several) of the objective function.</td>
</tr>
<tr>
<td>update_next()</td>
<td>Updates the value returned by opt.ask().</td>
</tr>
</tbody>
</table>

__init__ (dimensions, base_estimator='gp', n_random_starts=None, n_initial_points=10,
initial_point_generator='random', n_jobs=1, acq_func='gp_hedge', acq_optimizer='auto',
random_state=None, model_queue_size=None, acq_func_kwargs=None,
acq_optimizer_kwargs=None)

ask(n_points=None, strategy='cl_min')
Query point or multiple points at which objective should be evaluated.

n_points  [int or None, default: None] Number of points returned by the ask method. If the value is None, a single point to evaluate is returned. Otherwise a list of points to evaluate is returned of size n_points. This is useful if you can evaluate your objective in parallel, and thus obtain more objective function evaluations per unit of time.

strategy  [string, default: "cl_min"] Method to use to sample multiple points (see also n_points description). This parameter is ignored if n_points = None. Supported options are "cl_min", "cl_mean" or "cl_max".

- If set to "cl_min", then constant liar strategy is used with lie objective value being minimum of observed objective values. "cl_mean" and "cl_max" means mean and max of values respectively. For details on this strategy see:
  https://hal.archives-ouvertes.fr/hal-00732512/document

With this strategy a copy of optimizer is created, which is then asked for a point, and the point is told to the copy of optimizer with some fake objective (lie), the next point is asked from copy, it is also told to the copy with fake objective and so on. The type of lie defines different flavours of cl_x strategies.

copy(random_state=None)
Create a shallow copy of an instance of the optimizer.

Parameters

random_state  [int, RandomState instance, or None (default)] Set the random state of the copy.
get_result()

Returns the same result that would be returned by opt.tell() but without calling tell

Returns

res [OptimizeResult, scipy object] OptimizeResult instance with the required information.

run(func, n_iter=1)

Execute ask() + tell() n_iter times

tell(x, y, fit=True)

Record an observation (or several) of the objective function.

Provide values of the objective function at points suggested by ask() or other points. By default a new model will be fit to all observations. The new model is used to suggest the next point at which to evaluate the objective. This point can be retrieved by calling ask().

To add observations without fitting a new model set fit to False.

To add multiple observations in a batch pass a list-of-lists for x and a list of scalars for y.

Parameters

x [list or list-of-lists] Point at which objective was evaluated.

y [scalar or list] Value of objective at x.

fit [bool, default: True] Fit a model to observed evaluations of the objective. A model will only be fitted after n_initial_points points have been told to the optimizer irrespective of the value of fit.

update_next()

Updates the value returned by opt.ask(). Useful if a parameter was updated after ask was called.

optimizer.base_minimize(func, dimensions, ...) Base optimizer class

optimizer.dummy_minimize(func, dimensions[, ...]) Random search by uniform sampling within the given bounds.

optimizer.forest_minimize(func, dimensions) Sequential optimization using decision trees.

optimizer.gbrt_minimize(func, dimensions[, ...]) Sequential optimization using gradient boosted trees.

optimizer_gp_minimize(func, dimensions[, ...]) Bayesian optimization using Gaussian Processes.

5.6.2 skopt.optimizer.base_minimize

skopt.optimizer.base_minimize(func, dimensions, base_estimator, n_calls=100, n_random_starts=None, n_initial_points=10, initial_point_generator='random', acq_func='EI', acq_optimizer='lbfgs', x0=None, y0=None, random_state=None, verbose=False, callback=None, n_points=10000, n_restarts_optimizer=5, xi=0.01, kappa=1.96, n_jobs=1, model_queue_size=None)

Base optimizer class

Parameters

func [callable] Function to minimize. Should take a single list of parameters and return the objective value.

If you have a search-space where all dimensions have names, then you can use skopt.utils.use_named_args() as a decorator on your objective function, in order to call it directly with the named arguments. See use_named_args for an example.

dimensions [list, shape (n_dims,)] List of search space dimensions. Each search dimension can
be defined either as:

- a \((\text{lower\_bound}, \text{upper\_bound})\) tuple (for Real or Integer dimensions),
- a \((\text{lower\_bound}, \text{upper\_bound}, "\text{prior}"\)) tuple (for Real dimensions),
- as a list of categories (for Categorical dimensions), or
- an instance of a Dimension object (Real, Integer or Categorical).

**Note:** The upper and lower bounds are inclusive for Integer dimensions.

**base\_estimator** [sklearn regressor] Should inherit from sklearn.base.RegressorMixin. In addition, should have an optional return\_std argument, which returns \(\text{std}(Y | x)\) along with \(E[Y | x]\).

**n\_calls** [int, default: 100] Maximum number of calls to func. An objective function will always be evaluated this number of times; Various options to supply initialization points do not affect this value.

**n\_random\_starts** [int, default: None] Number of evaluations of func with random points before approximating it with base\_estimator.

Deprecated since version 0.8: use n\_initial\_points instead.

**n\_initial\_points** [int, default: 10] Number of evaluations of func with initialization points before approximating it with base\_estimator. Initial point generator can be changed by setting initial\_point\_generator.

**initial\_point\_generator** [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,
- "grid" for a uniform grid sequence

**acq\_func** [string, default: "EI"] Function to minimize over the posterior distribution. Can be either

- "LCB" for lower confidence bound,
- "EI" for negative expected improvement,
- "PI" for negative probability of improvement.
- "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.
- "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps"

**acq\_optimizer** [string, "sampling" or "lbfgs", default: "lbfgs"] Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing acq\_func with acq\_optimizer.
• If set to "sampling", then acq_func is optimized by computing acq_func at n_points randomly sampled points and the smallest value found is used.

• If set to "lbfgs", then
  – The n_restarts_optimizer no. of points which the acquisition function is least are taken as start points.
  – "lbfgs" is run for 20 iterations with these points as initial points to find local minima.
  – The optimal of these local minima is used to update the prior.

x0 [list, list of lists or None] Initial input points.

• If it is a list of lists, use it as a list of input points. If no corresponding outputs y0 are supplied, then len(x0) of total calls to the objective function will be spent evaluating the points in x0. If the corresponding outputs are provided, then they will be used together with evaluated points during a run of the algorithm to construct a surrogate.

• If it is a list, use it as a single initial input point. The algorithm will spend 1 call to evaluate the initial point, if the outputs are not provided.

• If it is None, no initial input points are used.

y0 [list, scalar or None] Objective values at initial input points.

• If it is a list, then it corresponds to evaluations of the function at each element of x0: the i-th element of y0 corresponds to the function evaluated at the i-th element of x0.

• If it is a scalar, then it corresponds to the evaluation of the function at x0.

• If it is None and x0 is provided, then the function is evaluated at each element of x0.

random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

verbose [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

callback [callable, list of callables, optional] If callable then callback(res) is called after each call to func. If list of callables, then each callable in the list is called.

n_points [int, default: 10000] If acq_optimizer is set to "sampling", then acq_func is optimized by computing acq_func at n_points randomly sampled points.

n_restarts_optimizer [int, default: 5] The number of restarts of the optimizer when acq_optimizer is "lbfgs".

xi [float, default: 0.01] Controls how much improvement one wants over the previous best values. Used when the acquisition is either "EI" or "PI".

kappa [float, default: 1.96] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. Used when the acquisition is "LCB".

n_jobs [int, default: 1] Number of cores to run in parallel while running the lbfgs optimizations over the acquisition function and given to the base_estimator. Valid only when acq_optimizer is set to "lbfgs". or when the base_estimator supports n_jobs as parameter and was given as string. Defaults to 1 core. If n_jobs=-1, then number of jobs is set to number of cores.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns
res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:
• x [list]: location of the minimum.
• fun [float]: function value at the minimum.
• models: surrogate models used for each iteration.
• x_iters [list of lists]: location of function evaluation for each iteration.
• func_vals [array]: function value for each iteration.
• space [Space]: the optimization space.
• specs [dict]: the call specifications.
• rng [RandomState instance]: State of the random state at the end of minimization.


5.6.3 skopt.optimizer.dummy_minimize

skopt.optimizer.dummy_minimize(func, dimensions, n_calls=100, initial_point_generator='random', x0=None, y0=None, random_state=None, verbose=False, callback=None, model_queue_size=None, init_point_gen_kwargs=None)

Random search by uniform sampling within the given bounds.

Parameters

func [callable] Function to minimize. Should take a single list of parameters and return the objective value.

If you have a search-space where all dimensions have names, then you can use skopt.utils.use_named_args() as a decorator on your objective function, in order to call it directly with the named arguments. See use_named_args for an example.

dimensions [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
• a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
• a (lower_bound, upper_bound, prior) tuple (for Real dimensions),
• as a list of categories (for Categorical dimensions), or
• an instance of a Dimension object (Real, Integer or Categorical).

n_calls [int, default: 100] Number of calls to func to find the minimum.

initial_point_generator [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either
• "random" for uniform random numbers,
• "sobol" for a Sobol’ sequence,
• "halton" for a Halton sequence,
• "hammersly" for a Hammersly sequence,
• "lhs" for a latin hypercube sequence,
• "grid" for a uniform grid sequence
x0 [list, list of lists or None] Initial input points.
- If it is a list of lists, use it as a list of input points.
- If it is a list, use it as a single initial input point.
- If it is None, no initial input points are used.

y0 [list, scalar or None] Evaluation of initial input points.
- If it is a list, then it corresponds to evaluations of the function at each element of x0: the i-th element of y0 corresponds to the function evaluated at the i-th element of x0.
- If it is a scalar, then it corresponds to the evaluation of the function at x0.
- If it is None and x0 is provided, then the function is evaluated at each element of x0.

random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

verbose [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

callback [callable, list of callables, optional] If callable then callback(res) is called after each call to func. If list of callables, then each callable in the list is called.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns

res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:
- x [list]: location of the minimum.
- fun [float]: function value at the minimum.
- x_iters [list of lists]: location of function evaluation for each iteration.
- func_vals [array]: function value for each iteration.
- space [Space]: the optimisation space.
- specs [dict]: the call specifications.
- rng [RandomState instance]: State of the random state at the end of minimization.


See also:
functions skopt.gp_minimize, skopt.forest_minimize, skopt.gbdt_minimize
5.6.4 `skopt.optimizer.forest_minimize`

The function `skopt.optimizer.forest_minimize` is used for sequential optimisation using decision trees. A tree-based regression model is used to model the expensive to evaluate function `func`. The model is improved by sequentially evaluating the expensive function at the next best point. Thereby finding the minimum of `func` with as few evaluations as possible.

The total number of evaluations, `n_calls`, are performed like the following. If `x0` is provided but not `y0`, then the elements of `x0` are first evaluated, followed by `n_initial_points` evaluations. Finally, `n_calls - len(x0) - n_initial_points` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_initial_points` evaluations are first made then `n_calls - n_initial_points` subsequent evaluations are made guided by the surrogate model.

The first `n_initial_points` are generated by the `initial_point_generator`.

**Parameters**

- `func` [callable] Function to minimize. Should take a single list of parameters and return the objective value.
  
  If you have a search-space where all dimensions have names, then you can use `skopt. utils.use_named_args()` as a decorator on your objective function, in order to call it directly with the named arguments. See `skopt.utils.use_named_args()` for an example.

- `dimensions` [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
  
  - a `(lower_bound, upper_bound)` tuple (for `Real` or `Integer` dimensions),
  - a `(lower_bound, upper_bound, prior)` tuple (for `Real` dimensions),
  - as a list of categories (for `Categorical` dimensions), or
  - an instance of a `Dimension` object (`Real`, `Integer` or `Categorical`).

  **Note:** The upper and lower bounds are inclusive for `Integer` dimensions.

- `base_estimator` [string or `Regressor`, default: "ET"] The regressor to use as surrogate model. Can be either
  
  - "RF" for random forest regressor
  - "ET" for extra trees regressor
  - instance of regressor with support for `return_std` in its predict method

  The predefined models are initialized with good defaults. If you want to adjust the model parameters pass your own instance of a regressor which returns the mean and standard deviation when making predictions.

- `n_calls` [int, default: 100] Number of calls to `func`.  

---

5.6. `skopt.optimizer`: Optimizer
**n_random_starts** [int, default: None] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

Depreciated since version 0.8: use `n_initial_points` instead.

**n_initial_points** [int, default: 10] Number of evaluations of `func` with initialization points before approximating it with `base_estimator`. Initial point generator can be changed by setting `initial_point_generator`.

**initial_point_generator** [str, `InitialPointGenerator` instance, default: "random"] Sets a initial points generator. Can be either
- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a Latin hypercube sequence,
- "grid" for a uniform grid sequence

**acq_func** [string, default: "LCB"] Function to minimize over the forest posterior. Can be either
- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
- "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.
- "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps"

**x0** [list, list of lists or None] Initial input points.
- If it is a list of lists, use it as a list of input points.
- If it is a list, use it as a single initial input point.
- If it is None, no initial input points are used.

**y0** [list, scalar or None] Evaluation of initial input points.
- If it is a list, then it corresponds to evaluations of the function at each element of `x0`: the i-th element of `y0` corresponds to the function evaluated at the i-th element of `x0`.
- If it is a scalar, then it corresponds to the evaluation of the function at `x0`.
- If it is None and `x0` is provided, then the function is evaluated at each element of `x0`.

**random_state** [int, `RandomState` instance, or None (default)] Set random state to something other than None for reproducible results.

**verbose** [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

**callback** [callable, optional] If provided, then `callback(res)` is called after call to `func`.

**n_points** [int, default: 10000] Number of points to sample when minimizing the acquisition function.
xi [float, default: 0.01] Controls how much improvement one wants over the previous best values. Used when the acquisition is either "EI" or "PI".

kappa [float, default: 1.96] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. Used when the acquisition is "LCB".

n_jobs [int, default: 1] The number of jobs to run in parallel for fit and predict. If -1, then the number of jobs is set to the number of cores.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns

res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:

- x [list]: location of the minimum.
- fun [float]: function value at the minimum.
- models: surrogate models used for each iteration.
- x_iters [list of lists]: location of function evaluation for each iteration.
- func_vals [array]: function value for each iteration.
- space [Space]: the optimization space.
- specs [dict]: the call specifications.

For more details related to the OptimizeResult object, refer to http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

See also:

functions skopt.gp_minimize, skopt.dummy_minimize, skopt.gbrt_minimize

5.6.5 skopt.optimizer.gbrt_minimize

skopt.optimizer.gbrt_minimize(func, dimensions, base_estimator=None, n_calls=100, n_random_starts=None, n_initial_points=10, initial_point_generator='random', acq_func='EI', acq_optimizer='auto', x0=None, y0=None, random_state=None, verbose=False, callback=None, n_points=10000, xi=0.01, kappa=1.96, n_jobs=1, model_queue_size=None)

Sequential optimization using gradient boosted trees.

Gradient boosted regression trees are used to model the (very) expensive to evaluate function func. The model is improved by sequentially evaluating the expensive function at the next best point. Thereby finding the minimum of func with as few evaluations as possible.

The total number of evaluations, n_calls, are performed like the following. If x0 is provided but not y0, then the elements of x0 are first evaluated, followed by n_initial_points evaluations. Finally, n_calls - len(x0) - n_initial_points evaluations are made guided by the surrogate model. If x0 and y0 are both provided then n_initial_points evaluations are first made then n_calls - n_initial_points subsequent evaluations are made guided by the surrogate model.

The first n_initial_points are generated by the initial_point_generator.

Parameters
**func** [callable] Function to minimize. Should take a single list of parameters and return the objective value.

If you have a search-space where all dimensions have names, then you can use `skopt.utils.use_named_args` as a decorator on your objective function, in order to call it directly with the named arguments. See `use_named_args` for an example.

**dimensions** [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as

- a `(lower_bound, upper_bound)` tuple (for `Real` or `Integer` dimensions),
- a `(lower_bound, upper_bound, "prior")` tuple (for `Real` dimensions),
- as a list of categories (for `Categorical` dimensions), or
- an instance of a `Dimension` object (`Real`, `Integer` or `Categorical`).

**base_estimator** [GradientBoostingQuantileRegressor] The regressor to use as surrogate model

**n_calls** [int, default: 100] Number of calls to `func`.

**n_random_starts** [int, default: None] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

Depreciated since version 0.8: use `n_initial_points` instead.

**n_initial_points** [int, default: 10] Number of evaluations of `func` with initialization points before approximating it with `base_estimator`. Initial point generator can be changed by setting `initial_point_generator`.

**initial_point_generator** [str, InitialPointGenerator instance, default: "random"] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol' sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,
- "grid" for a uniform grid sequence

**acq_func** [string, default: "LCB"] Function to minimize over the forest posterior. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
- "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken.
- "PIps" for negated probability of improvement per second.

**x0** [list, list of lists or None] Initial input points.

- If it is a list of lists, use it as a list of input points.
- If it is a list, use it as a single initial input point.
- If it is None, no initial input points are used.
y0 [list, scalar or None] Evaluation of initial input points.
  • If it is a list, then it corresponds to evaluations of the function at each element of x0: the
    i-th element of y0 corresponds to the function evaluated at the i-th element of x0.
  • If it is a scalar, then it corresponds to the evaluation of the function at x0.
  • If it is None and x0 is provided, then the function is evaluated at each element of x0.

random_state [int, RandomState instance, or None (default)] Set random state to something
other than None for reproducible results.

verbose [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True
for long optimization runs.

callback [callable, optional] If provided, then callback(res) is called after call to func.

n_points [int, default: 10000] Number of points to sample when minimizing the acquisition
function.

xi [float, default: 0.01] Controls how much improvement one wants over the previous best values.
Used when the acquisition is either "EI" or "PI".

kappa [float, default: 1.96] Controls how much of the variance in the predicted values should be
taken into account. If set to be very high, then we are favouring exploration over exploitation
and vice versa. Used when the acquisition is "LCB".

n_jobs [int, default: 1] The number of jobs to run in parallel for fit and predict. If -1, then
the number of jobs is set to the number of cores.

model_queue_size [int or None, default: None] Keeps list of models only as long as the argument
given. In the case of None, the list has no capped length.

Returns

res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:
  • x [list]: location of the minimum.
  • fun [float]: function value at the minimum.
  • models: surrogate models used for each iteration.
  • x_iters [list of lists]: location of function evaluation for each iteration.
  • func_vals [array]: function value for each iteration.
  • space [Space]: the optimization space.
  • specs [dict]: the call specifications.
  • rng [RandomState instance]: State of the random state at the end of minimization.

For more details related to the OptimizeResult object, refer http://docs.scipy.org/doc/scipy/
reference/generated/scipy.optimize.OptimizeResult.html

See also:

functions skopt.gp_minimize, skopt.dummy_minimize, skopt.forest_minimize

5.6. skopt.optimizer: Optimizer
5.6.6 skopt.optimizer.gp_minimize

skopt.optimizer.gp_minimize(func, dimensions, base_estimator=None, n_calls=100, n_random_starts=None, n_initial_points=10, initial_point_generator='random', acq_func='gp_hedge', acq_optimizer='auto', x0=None, y0=None, random_state=None, verbose=False, callback=None, n_points=10000, n_restarts_optimizer=5, xi=0.01, kappa=1.96, noise='gaussian', n_jobs=1, model_queue_size=None)

Bayesian optimization using Gaussian Processes.

If every function evaluation is expensive, for instance when the parameters are the hyperparameters of a neural network and the function evaluation is the mean cross-validation score across ten folds, optimizing the hyperparameters by standard optimization routines would take for ever!

The idea is to approximate the function using a Gaussian process. In other words the function values are assumed to follow a multivariate gaussian. The covariance of the function values are given by a GP kernel between the parameters. Then a smart choice to choose the next parameter to evaluate can be made by the acquisition function over the Gaussian prior which is much quicker to evaluate.

The total number of evaluations, n_calls, are performed like the following. If x0 is provided but not y0, then the elements of x0 are first evaluated, followed by n_initial_points evaluations. Finally, n_calls - len(x0) - n_initial_points evaluations are made guided by the surrogate model. If x0 and y0 are both provided then n_initial_points evaluations are first made then n_calls - n_initial_points subsequent evaluations are made guided by the surrogate model.

The first n_initial_points are generated by the initial_point_generator.

Parameters

  * **func** [callable] Function to minimize. Should take a single list of parameters and return the objective value.

    If you have a search-space where all dimensions have names, then you can use `skopt.utils.use_named_args()` as a decorator on your objective function, in order to call it directly with the named arguments. See `use_named_args` for an example.

  * **dimensions** [[list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as

      - a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
      - a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
      - as a list of categories (for Categorical dimensions), or
      - an instance of a `Dimension` object (Real, Integer or Categorical).

  * **base_estimator** [a Gaussian process estimator] The Gaussian process estimator to use for optimization. By default, a Matern kernel is used with the following hyperparameters tuned.

    - All the length scales of the Matern kernel.
    - The covariance amplitude that each element is multiplied with.
    - Noise that is added to the matern kernel. The noise is assumed to be iid gaussian.

  * **n_calls** [int, default: 100] Number of calls to `func`. 

Note: The upper and lower bounds are inclusive for Integer dimensions.
**n_random_starts** [int, default: None] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

Deprecated since version 0.8: use `n_initial_points` instead.

**n_initial_points** [int, default: 10] Number of evaluations of `func` with initialization points before approximating it with `base_estimator`. Initial point generator can be changed by setting `initial_point_generator`.

**initial_point_generator** [str, InitialPointGenerator instance, default: ‘random’] Sets a initial points generator. Can be either

- "random" for uniform random numbers,
- "sobol" for a Sobol’ sequence,
- "halton" for a Halton sequence,
- "hammersly" for a Hammersly sequence,
- "lhs" for a latin hypercube sequence,

**acq_func** [string, default: "gp_hedge"] Function to minimize over the gaussian prior. Can be either

- "LCB" for lower confidence bound.
- "EI" for negative expected improvement.
- "PI" for negative probability of improvement.
- "gp_hedge" Probabilistically choose one of the above three acquisition functions at every iteration. The weightage given to these gains can be set by $\eta$ through `acq_func_kwargs`.
  - The gains $g_i$ are initialized to zero.
  - At every iteration,
    * Each acquisition function is optimised independently to propose an candidate point $X_i$.
    * Out of all these candidate points, the next point $X_{best}$ is chosen by $softmax(\eta g_i)$
    * After fitting the surrogate model with $(X_{best}, y_{best})$, the gains are updated such that $g_i = \mu(X_i)$
- "EIps" for negated expected improvement per second to take into account the function compute time. Then, the objective function is assumed to return two values, the first being the objective value and the second being the time taken in seconds.
- "PIps" for negated probability of improvement per second. The return type of the objective function is assumed to be similar to that of "EIps"

**acq_optimizer** [string, "sampling" or "lbfgs", default: "lbfgs"] Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing `acq_func` with `acq_optimizer`.

The `acq_func` is computed at `n_points` sampled randomly.

- If set to "auto", then `acq_optimizer` is configured on the basis of the space searched over. If the space is Categorical then this is set to be "sampling".
- If set to "sampling", then the point among these `n_points` where the `acq_func` is minimum is the next candidate minimum.
- If set to "lbfgs", then
scikit-optimize Documentation, Release 0.9.0

- The `n_restarts_optimizer` no. of points which the acquisition function is least are taken as start points.
- "lbfgs" is run for 20 iterations with these points as initial points to find local minima.
- The optimal of these local minima is used to update the prior.

`x0` [list, list of lists or None] Initial input points.
- If it is a list of lists, use it as a list of input points.
- If it is a list, use it as a single initial input point.
- If it is None, no initial input points are used.

`y0` [list, scalar or None] Evaluation of initial input points.
- If it is a list, then it corresponds to evaluations of the function at each element of `x0` : the i-th element of `y0` corresponds to the function evaluated at the i-th element of `x0`.
- If it is a scalar, then it corresponds to the evaluation of the function at `x0`.
- If it is None and `x0` is provided, then the function is evaluated at each element of `x0`.

`random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

`verbose` [boolean, default: False] Control the verbosity. It is advised to set the verbosity to True for long optimization runs.

`callback` [callable, list of callables, optional] If callable then callback(res) is called after each call to func. If list of callables, then each callable in the list is called.

`n_points` [int, default: 10000] Number of points to sample to determine the next “best” point. Useless if acq_optimizer is set to "lbfgs".

`n_restarts_optimizer` [int, default: 5] The number of restarts of the optimizer when acq_optimizer is "lbfgs".

`kappa` [float, default: 1.96] Controls how much of the variance in the predicted values should be taken into account. If set to be very high, then we are favouring exploration over exploitation and vice versa. Used when the acquisition is "LCB".

`xi` [float, default: 0.01] Controls how much improvement one wants over the previous best values. Used when the acquisition is either "EI" or "PI".

`noise` [float, default: “gaussian”]
- Use noise="gaussian“ if the objective returns noisy observations. The noise of each observation is assumed to be iid with mean zero and a fixed variance.
- If the variance is known before-hand, this can be set directly to the variance of the noise.
- Set this to a value close to zero (1e-10) if the function is noise-free. Setting to zero might cause stability issues.

`n_jobs` [int, default: 1] Number of cores to run in parallel while running the lbfgs optimizations over the acquisition function. Valid only when acq_optimizer is set to "lbfgs". Defaults to 1 core. If n_jobs=-1, then number of jobs is set to number of cores.

`model_queue_size` [int or None, default: None] Keeps list of models only as long as the argument given. In the case of None, the list has no capped length.

Returns
res [OptimizeResult, scipy object] The optimization result returned as a OptimizeResult object. Important attributes are:

- x [list]: location of the minimum.
- fun [float]: function value at the minimum.
- models: surrogate models used for each iteration.
- x_iters [list of lists]: location of function evaluation for each iteration.
- func_vals [array]: function value for each iteration.
- space [Space]: the optimization space.
- specs [dict]: the call specifications.
- rng [RandomState instance]: State of the random state at the end of minimization.


See also:

functions skopt.forest_minimize, skopt.dummy_minimize, skopt.gbrt_minimize

5.7 skopt.plots: Plotting functions.

Plotting functions.

User guide: See the Plotting tools section for further details.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plots.partial_dependence</td>
<td>Calculate the partial dependence for dimensions ( i ) and ( j ) with respect to the objective value, as approximated by ( \text{model} ).</td>
</tr>
<tr>
<td>plots.partial_dependence_1D</td>
<td>Calculate the partial dependence for a single dimension.</td>
</tr>
<tr>
<td>plots.partial_dependence_2D</td>
<td>Calculate the partial dependence for two dimensions in the search-space.</td>
</tr>
<tr>
<td>plots.plot_convergence(*args, **kwargs)</td>
<td>Plot one or several convergence traces.</td>
</tr>
<tr>
<td>plots.plot_evaluations(result[, bins, ...])</td>
<td>Visualize the order in which points were sampled during optimization.</td>
</tr>
<tr>
<td>plots.plot_gaussian_process(res, **kwargs)</td>
<td>Plots the optimization results and the gaussian process for 1-D objective functions.</td>
</tr>
<tr>
<td>plots.plot_objective(result[, levels, ...])</td>
<td>Plot a 2-d matrix with so-called Partial Dependence plots of the objective function.</td>
</tr>
<tr>
<td>plots.plot_objective_2D(result, [...], ...)</td>
<td>Create and return a Matplotlib figure and axes with a landscape contour-plot of the last fitted model of the search-space, overlaid with all the samples from the optimization results, for the two given dimensions of the search-space.</td>
</tr>
<tr>
<td>plots.plot_histogram(result, [...], bins, ...)</td>
<td>Create and return a Matplotlib figure with a histogram of the samples from the optimization results, for a given dimension of the search-space.</td>
</tr>
<tr>
<td>plots.plot_regret(*args, **kwargs)</td>
<td>Plot one or several cumulative regret traces.</td>
</tr>
</tbody>
</table>
5.7.1 skopt.plots.partial_dependence

skopt.plots.partial_dependence(space, model, i=None, j=None, sample_points=None, n_samples=250, n_points=40, x_eval=None)

Calculate the partial dependence for dimensions \(i\) and \(j\) with respect to the objective value, as approximated by model.

The partial dependence plot shows how the value of the dimensions \(i\) and \(j\) influence the model predictions after “averaging out” the influence of all other dimensions.

When \(x_{\text{eval}}\) is not None, the given values are used instead of random samples. In this case, \(n_{\text{samples}}\) will be ignored.

Parameters

- **space** [Space] The parameter space over which the minimization was performed.
- **model** Surrogate model for the objective function.
- **i** [int] The first dimension for which to calculate the partial dependence.
- **j** [int, default=None] The second dimension for which to calculate the partial dependence. To calculate the 1D partial dependence on \(i\) alone set \(j=\text{None}\).
- **sample_points** [np.array, shape=(n_points, n_dims), default=None] Only used when \(x_{\text{eval}}=\text{None}\), i.e. in case partial dependence should be calculated. Randomly sampled and transformed points to use when averaging the model function at each of the \(n_{\text{points}}\) when using partial dependence.
- **n_samples** [int, default=100] Number of random samples to use for averaging the model function at each of the \(n_{\text{points}}\) when using partial dependence. Only used when \(sample_{\text{points}}=\text{None}\) and \(x_{\text{eval}}=\text{None}\).
- **n_points** [int, default=40] Number of points at which to evaluate the partial dependence along each dimension \(i\) and \(j\).
- **x_eval** [list, default=None] \(x_{\text{eval}}\) is a list of parameter values or None. In case \(x_{\text{eval}}\) is not None, the parsed dependence will be calculated using these values. Otherwise, random selected samples will be used.

Returns

For 1D partial dependence:

- **xi** [np.array] The points at which the partial dependence was evaluated.
- **yi** [np.array] The value of the model at each point \(x_i\).

For 2D partial dependence:

- **xi** [np.array, shape=n_points] The points at which the partial dependence was evaluated.
- **yi** [np.array, shape=n_points] The points at which the partial dependence was evaluated.
- **zi** [np.array, shape=(n_points, n_points)] The value of the model at each point \((x_i, y_i)\).

For Categorical variables, the \(xi\) (and \(yi\) for 2D) returned are the indices of the variable in Dimension.categories.
5.7.2 skopt.plots.partial_dependence_1D

`skopt.plots.partial_dependence_1D(space, model, i, samples, n_points=40)`

Calculate the partial dependence for a single dimension.

This uses the given model to calculate the average objective value for all the samples, where the given dimension is fixed at regular intervals between its bounds.

This shows how the given dimension affects the objective value when the influence of all other dimensions are averaged out.

**Parameters**

- `space` [Space] The parameter space over which the minimization was performed.
- `model` Surrogate model for the objective function.
- `i` [int] The dimension for which to calculate the partial dependence.
- `samples` [np.array, shape=(n_points, n_dims)] Randomly sampled and transformed points to use when averaging the model function at each of the n_points when using partial dependence.
- `n_points` [int, default=40] Number of points at which to evaluate the partial dependence along each dimension i.

**Returns**

- `xi` [np.array] The points at which the partial dependence was evaluated.
- `yi` [np.array] The average value of the modelled objective function at each point xi.

5.7.3 skopt.plots.partial_dependence_2D

`skopt.plots.partial_dependence_2D(space, model, i, j, samples, n_points=40)`

Calculate the partial dependence for two dimensions in the search-space.

This uses the given model to calculate the average objective value for all the samples, where the given dimensions are fixed at regular intervals between their bounds.

This shows how the given dimensions affect the objective value when the influence of all other dimensions are averaged out.

**Parameters**

- `space` [Space] The parameter space over which the minimization was performed.
- `model` Surrogate model for the objective function.
- `i` [int] The first dimension for which to calculate the partial dependence.
- `j` [int] The second dimension for which to calculate the partial dependence.
- `samples` [np.array, shape=(n_points, n_dims)] Randomly sampled and transformed points to use when averaging the model function at each of the n_points when using partial dependence.
- `n_points` [int, default=40] Number of points at which to evaluate the partial dependence along each dimension i and j.

**Returns**

- `xi` [np.array, shape=n_points] The points at which the partial dependence was evaluated.
- `yi` [np.array, shape=n_points] The points at which the partial dependence was evaluated.
zi [np.array, shape=(n_points, n_points)] The average value of the objective function at each point (xi, yi).

5.7.4 skopt.plots.plot_convergence

skopt.plots.plot_convergence(*args, **kwargs)

Plot one or several convergence traces.

Parameters

- args[i] [OptimizeResult, list of OptimizeResult, or tuple] The result(s) for which to plot the convergence trace.
  - if OptimizeResult, then draw the corresponding single trace;
  - if list of OptimizeResult, then draw the corresponding convergence traces in transparency, along with the average convergence trace;
  - if tuple, then args[i][0] should be a string label and args[i][1] an OptimizeResult or a list of OptimizeResult.

- ax [Axes, optional] The matplotlib axes on which to draw the plot, or None to create a new one.

- true_minimum [float, optional] The true minimum value of the function, if known.

-yscale [None or string, optional] The scale for the y-axis.

Returns


Examples using skopt.plots.plot_convergence

- Tuning a scikit-learn estimator with skopt
- Comparing surrogate models
- Bayesian optimization with skopt

5.7.5 skopt.plots.plot_evaluations

skopt.plots.plot_evaluations(result, bins=20, dimensions=None, plotDims=None)

Visualize the order in which points were sampled during optimization.

This creates a 2-d matrix plot where the diagonal plots are histograms that show the distribution of samples for each search-space dimension.

The plots below the diagonal are scatter-plots of the samples for all combinations of search-space dimensions.

The order in which samples were evaluated is encoded in each point’s color.

A red star shows the best found parameters.

Parameters

- result [OptimizeResult] The optimization results from calling e.g. gp_minimize().

- bins [int, bins=20] Number of bins to use for histograms on the diagonal.

- dimensions [list of str, default=None] Labels of the dimension variables. None defaults to space.dimensions[i].name, or if also None to ['X_0', 'X_1', ..].
plot_dims [list of str and int, default=None] List of dimension names or dimension indices from the search-space dimensions to be included in the plot. If None then use all dimensions except constant ones from the search-space.

Returns

ax [Matplotlib.Axes] A 2-d matrix of Axes-objects with the sub-plots.

Examples using skopt.plots.plot_evaluations

- Visualizing optimization results

5.7.6 skopt.plots.plot_gaussian_process

skopt.plots.plot_gaussian_process(res, **kwargs)
Plots the optimization results and the gaussian process for 1-D objective functions.

Parameters

res [OptimizeResult] The result for which to plot the gaussian process.
ax [Axes, optional] The matplotlib axes on which to draw the plot, or None to create a new one.
n_calls [int, default: -1] Can be used to evaluate the model at call n_calls.
objective [func, default: None] Defines the true objective function. Must have one input parameter.
n_points [int, default: 1000] Number of data points used to create the plots
noise_level [float, default: 0] Sets the estimated noise level
show_legend [boolean, default: True] When True, a legend is plotted.
show_title [boolean, default: True] When True, a title containing the found minimum value is shown
show_acq_func [boolean, default: False] When True, the acquisition function is plotted
show_next_point [boolean, default: False] When True, the next evaluated point is plotted
show_observations [boolean, default: True] When True, observations are plotted as dots.
show_mu [boolean, default: True] When True, the predicted model is shown.

Returns

ax [Axes] The matplotlib axes.

Examples using skopt.plots.plot_gaussian_process

- Async optimization Loop
- Bayesian optimization with skopt
- Exploration vs exploitation
- Use different base estimators for optimization
5.7.7 `skopt.plots.plot_objective`

`skopt.plots.plot_objective(result, levels=10, n_points=40, n_samples=250, size=2, zscale='linear',
    dimensions=None, sample_source='random', minimum='result',
    n_minimum_search=None, plot_dims=None, show_points=True,
    cmap='viridis_r')`

Plot a 2-d matrix with so-called Partial Dependence plots of the objective function. This shows the influence of each search-space dimension on the objective function.

This uses the last fitted model for estimating the objective function.

The diagonal shows the effect of a single dimension on the objective function, while the plots below the diagonal show the effect on the objective function when varying two dimensions.

The Partial Dependence is calculated by averaging the objective value for a number of random samples in the search-space, while keeping one or two dimensions fixed at regular intervals. This averages out the effect of varying the other dimensions and shows the influence of one or two dimensions on the objective function.

Also shown are small black dots for the points that were sampled during optimization.

A red star indicates per default the best observed minimum, but this can be changed by changing argument `minimum`.

**Note:** The Partial Dependence plot is only an estimation of the surrogate model which in turn is only an estimation of the true objective function that has been optimized. This means the plots show an “estimate of an estimate” and may therefore be quite imprecise, especially if few samples have been collected during the optimization (e.g. less than 100-200 samples), and in regions of the search-space that have been sparsely sampled (e.g. regions away from the optimum). This means that the plots may change each time you run the optimization and they should not be considered completely reliable. These compromises are necessary because we cannot evaluate the expensive objective function order to plot it, so we have to use the cheaper surrogate model to plot its contour. And in order to show search-spaces with 3 dimensions or more in a 2-dimensional plot, we further need to map those dimensions to only 2-dimensions using the Partial Dependence, which also causes distortions in the plots.

**Parameters**

- `result` [OptimizeResult] The optimization results from calling e.g. `gp_minimize()`.
- `levels` [int, default=10] Number of levels to draw on the contour plot, passed directly to `plt.contourf()`.
- `n_points` [int, default=40] Number of points at which to evaluate the partial dependence along each dimension.
- `n_samples` [int, default=250] Number of samples to use for averaging the model function at each of the `n_points` when `sample_method` is set to ‘random’.
- `size` [float, default=2] Height (in inches) of each facet.
- `zscale` [str, default=’linear’] Scale to use for the z axis of the contour plots. Either ‘linear’ or ‘log’.
- `dimensions` [list of str, default=None] Labels of the dimension variables. None defaults to `space.dimensions[i].name`, or if also None to ['X_0', 'X_1', ...].
- `plot_dims` [list of str and int, default=None] List of dimension names or dimension indices from the search-space dimensions to be included in the plot. If None then use all dimensions except constant ones from the search-space.
sample_source  [str or list of floats, default='random'] Defines to samples generation to use for averaging the model function at each of the n_points.

A partial dependence plot is only generated, when sample_source is set to ‘random’ and n_samples is sufficient.

sample_source can also be a list of floats, which is then used for averaging.

Valid strings:
- ‘random’ - n_samples random samples will used
- ‘result’ - Use only the best observed parameters
- ‘expected_minimum’ - Parameters that gives the best minimum Calculated using scipy’s minimize method. This method currently does not work with categorical values.
- ‘expected_minimum_random’ - Parameters that gives the best minimum when using naive random sampling. Works with categorical values.

minimum  [str or list of floats, default='result'] Defines the values for the red points in the plots.

Valid strings:
- ‘result’ - Use best observed parameters
- ‘expected_minimum’ - Parameters that gives the best minimum Calculated using scipy’s minimize method. This method currently does not work with categorical values.
- ‘expected_minimum_random’ - Parameters that gives the best minimum when using naive random sampling. Works with categorical values

n_minimum_search  [int, default=None] Determines how many points should be evaluated to find the minimum when using ‘expected_minimum’ or ‘expected_minimum_random’. Parameter is used when sample_source and/or minimum is set to ‘expected_minimum’ or ‘expected_minimum_random’.

show_points: bool, default=True  Choose whether to show evaluated points in the contour plots.

cmap: str or Colormap, default=‘viridis_r’ Color map for contour plots. Passed directly to plt.contourf()

Returns

ax [Matplotlib.Axes] A 2-d matrix of Axes-objects with the sub-plots.

Examples using skopt.plots.plot_objective

- Scikit-learn hyperparameter search wrapper
- Partial Dependence Plots
- Partial Dependence Plots with categorical values
- Visualizing optimization results
- Partial Dependence Plots 2D
5.7.8 skopt.plots.plot_objective_2D

skopt.plots.plot_objective_2D(result, dimension_identifier1, dimension_identifier2, n_points=40, n_samples=250, levels=10, zscale='linear', sample_source='random', minimum='result', n_minimum_search=None, ax=None)

Create and return a Matplotlib figure and axes with a landscape contour-plot of the last fitted model of the search-space, overlaid with all the samples from the optimization results, for the two given dimensions of the search-space.

This is similar to plot_objective() but only for 2 dimensions whose doc-string also has a more extensive explanation.

Parameters

- result [OptimizeResult] The optimization results e.g. from calling gp_minimize().
- dimension_identifier1 [str or int] Name or index of a dimension in the search-space.
- dimension_identifier2 [str or int] Name or index of a dimension in the search-space.
- n_samples [int, default=250] Number of random samples used for estimating the contour-plot of the objective function.
- n_points [int, default=40] Number of points along each dimension where the partial dependence is evaluated when generating the contour-plots.
- levels [int, default=10] Number of levels to draw on the contour plot.
- zscale [str, default='linear'] Scale to use for the z axis of the contour plots. Either ‘log’ or linear for all other choices.
- ax [Matplotlib.Axes, default: None] When set, everything is plotted inside this axis.

Returns

ax [Matplotlib.Axes] The Matplotlib Axes-object. For example, you can save the plot by calling fig.savefig('file.png')

Examples using skopt.plots.plot_objective_2D

- Partial Dependence Plots 2D

5.7.9 skopt.plots.plot_histogram

skopt.plots.plot_histogram(result, dimension_identifier, bins=20, rotate_labels=0, ax=None)

Create and return a Matplotlib figure with a histogram of the samples from the optimization results, for a given dimension of the search-space.

Parameters

- result [OptimizeResult] The optimization results e.g. from calling gp_minimize().
- dimension_identifier [str or int] Name or index of a dimension in the search-space.
- bins [int, bins=20] Number of bins in the histogram.
- rotate_labels [int, rotate_labels=0] Degree to rotate category-names on the x-axis. Only used for Categorical dimensions.

Returns

Examples using `skopt.plots.plot_histogram`

- Scikit-learn hyperparameter search wrapper
- Partial Dependence Plots 2D

## 5.7.10 skopt.plots.plot_regret

`skopt.plots.plot_regret(*args, **kwargs)`
Plot one or several cumulative regret traces.

### Parameters

- **args** [OptimizeResult, list of OptimizeResult, or tuple] The result(s) for which to plot the cumulative regret trace.
  - if OptimizeResult, then draw the corresponding single trace;
  - if list of OptimizeResult, then draw the corresponding cumulative regret traces in transparency, along with the average cumulative regret trace;
  - if tuple, then args[i][0] should be a string label and args[i][1] an OptimizeResult or a list of OptimizeResult.
- **ax** [Axes, optional] The matplotlib axes on which to draw the plot, or `None` to create a new one.
- **true_minimum** [float, optional] The true minimum value of the function, if known.
- **yscale** [None or string, optional] The scale for the y-axis.

### Returns

- **ax** [Axes] The matplotlib axes.

## 5.8 skopt.utils: Utils functions.

**User guide:** See the `Utility functions` section for further details.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skopt.utils.cook_estimator(base_estimator[, space])</code></td>
<td>Cook a default estimator.</td>
</tr>
<tr>
<td><code>skopt.utils.cook_initial_point_generator(...)</code></td>
<td>Cook a default initial point generator.</td>
</tr>
<tr>
<td><code>skopt.utils.dimensions_aslist(search_space)</code></td>
<td>Convert a dict representation of a search space into a list of dimensions, ordered by <code>sorted(search_space.keys())</code>.</td>
</tr>
<tr>
<td><code>skopt.utils.expected_minimum(res[, ...])</code></td>
<td>Compute the minimum over the predictions of the last surrogate model.</td>
</tr>
<tr>
<td><code>skopt.utils.expected_minimum_random_sampling(res)</code></td>
<td>Minimum search by doing naive random sampling. Returns the parameters that gave the minimum function value.</td>
</tr>
<tr>
<td><code>skopt.utils.dump(res, filename[, store_objective])</code></td>
<td>Store an skopt optimization result into a file.</td>
</tr>
<tr>
<td><code>skopt.utils.load(filename, **kwargs)</code></td>
<td>Reconstruct a skopt optimization result from a file persisted with skopt.dump.</td>
</tr>
<tr>
<td><code>skopt.utils.point_asdict(search_space, point_as_list)</code></td>
<td>Convert the list representation of a point from a search space to the dictionary representation, where keys are dimension names and values are corresponding to the values of dimensions in the list.</td>
</tr>
</tbody>
</table>

continues on next page
Table 25 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>utils.point_aslist(search_space, point_as_dict)</code></td>
<td>Convert a dictionary representation of a point from a search space to the list representation.</td>
</tr>
<tr>
<td><code>utils.use_named_args(dimensions)</code></td>
<td>Wrapper / decorator for an objective function that uses named arguments to make it compatible with optimizers that use a single list of parameters.</td>
</tr>
</tbody>
</table>

5.8.1 skopt.utils.cook_estimator

skopt.utils.cook_estimator(base_estimator, space=None, **kwargs)

Cook a default estimator.

For the special base_estimator called “DUMMY” the return value is None. This corresponds to sampling points at random, hence there is no need for an estimator.

Parameters

- **base_estimator** [“GP”, “RF”, “ET”, “GBRT”, “DUMMY” or sklearn regressor] Should inherit from sklearn.base.RegressorMixin. In addition the predict method should have an optional return_std argument, which returns std(Y | x) along with E[Y | x]. If base_estimator is one of [“GP”, “RF”, “ET”, “GBRT”, “DUMMY”], a surrogate model corresponding to the relevant x_minimize function is created.

- **space** [Space instance] Has to be provided if the base_estimator is a gaussian process. Ignored otherwise.

- **kwargs** [dict] Extra parameters provided to the base_estimator at init time.

5.8.2 skopt.utils.cook_initial_point_generator

skopt.utils.cook_initial_point_generator(generator, **kwargs)

Cook a default initial point generator.

For the special generator called “random” the return value is None.

Parameters


- **kwargs** [dict] Extra parameters provided to the generator at init time.

Examples using skopt.utils.cook_initial_point_generator

- Comparing initial point generation methods
5.8.3 skopt.utils.dimensions_aslist

**skopt.utils.dimensions_aslist(search_space)**

Convert a dict representation of a search space into a list of dimensions, ordered by sorted(search_space.keys()).

**Parameters**

- **search_space** [dict] Represents search space. The keys are dimension names (strings) and values are instances of classes that inherit from the class `skopt.space.Dimension` (Real, Integer or Categorical)

**Returns**

- **params_space_list**: list list of `skopt.space.Dimension` instances.

**Examples**

```python
>>> from skopt.space.space import Real, Integer
>>> from skopt.utils import dimensions_aslist
>>> search_space = {
    'name1': Real(0, 1),
    'name2': Integer(2, 4),
    'name3': Real(-1, 1)
}

>>> dimensions_aslist(search_space)[0]
Real(low=0, high=1, prior='uniform', transform='identity')

>>> dimensions_aslist(search_space)[1]
Integer(low=2, high=4, prior='uniform', transform='identity')

>>> dimensions_aslist(search_space)[2]
Real(low=-1, high=1, prior='uniform', transform='identity')
```

5.8.4 skopt.utils.expected_minimum

**skopt.utils.expected_minimum(res, n_random_starts=20, random_state=None)**

Compute the minimum over the predictions of the last surrogate model. Uses `expected_minimum_random_sampling` with `n_random_starts = 100000`, when the space contains any categorical values.

**Note:** The returned minimum may not necessarily be an accurate prediction of the minimum of the true objective function.

**Parameters**

- **res** [OptimizeResult, scipy object] The optimization result returned by a skopt minimizer.

- **n_random_starts** [int, default=20] The number of random starts for the minimization of the surrogate model.

- **random_state** [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**Returns**

- **x** [list] location of the minimum.

- **fun** [float] the surrogate function value at the minimum.
5.8.5 **skopt.utils.expected_minimum_random_sampling**

`skopt.utils.expected_minimum_random_sampling(res, n_random_starts=100000, random_state=None)`

Minimum search by doing naive random sampling. Returns the parameters that gave the minimum function value. Can be used when the space contains any categorical values.

**Note:** The returned minimum may not necessarily be an accurate prediction of the minimum of the true objective function.

**Parameters**

- `res` [OptimizeResult, scipy object] The optimization result returned by a skopt minimizer.
- `n_random_starts` [int, default=100000] The number of random starts for the minimization of the surrogate model.
- `random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**Returns**

- `x` [list] location of the minimum.
- `fun` [float] the surrogate function value at the minimum.

5.8.6 **skopt.utils.dump**

`skopt.utils.dump(res, filename, store_objective=True, **kwargs)`

Store an skopt optimization result into a file.

**Parameters**

- `res` [OptimizeResult, scipy object] Optimization result object to be stored.
- `filename` [string or pathlib.Path] The path of the file in which it is to be stored. The compression method corresponding to one of the supported filename extensions (`.z`, `.gz`, `.bz2`, `.xz` or `.lzma`) will be used automatically.
- `store_objective` [boolean, default=True] Whether the objective function should be stored. Set `store_objective` to `False` if your objective function (`specs['args']['func']`) is unserializable (i.e. if an exception is raised when trying to serialize the optimization result).

Notice that if `store_objective` is set to `False`, a deep copy of the optimization result is created, potentially leading to performance problems if `res` is very large. If the objective function is not critical, one can delete it before calling skopt.dump() and thus avoid deep copying of `res`.

- `**kwargs` [other keyword arguments] All other keyword arguments will be passed to joblib.dump.
5.8.7 skopt.utils.load

skopt.utils.load(filename, **kwargs)
Reconstruct a skopt optimization result from a file persisted with skopt.dump.

Note: Notice that the loaded optimization result can be missing the objective function (.specs['args'][func]) if skopt.dump was called with store_objective=False.

Parameters

- filename [string or pathlib.Path] The path of the file from which to load the optimization result.
- **kwargs [other keyword arguments] All other keyword arguments will be passed to joblib.load.

Returns


5.8.8 skopt.utils.point_asdict

skopt.utils.point_asdict(search_space, point_as_list)
Convert the list representation of a point from a search space to the dictionary representation, where keys are dimension names and values are corresponding to the values of dimensions in the list.

See also:

skopt.utils.point_aslist

Parameters

- search_space [dict] Represents search space. The keys are dimension names (strings) and values are instances of classes that inherit from the class skopt.space.Dimension (Real, Integer or Categorical)
- point_as_list [list] list with parameter values. The order of parameters in the list is given by sorted(params_space.keys()).

Returns

- params_dict [OrderedDict] dictionary with parameter names as keys to which corresponding parameter values are assigned.

Examples

```python
>>> from skopt.space.space import Real, Integer
>>> from skopt.utils import point_asdict
>>> search_space = {'name1': Real(0,1),
...    'name2': Integer(2,4), 'name3': Real(-1,1)}
>>> point_as_list = [0.66, 3, -0.15]
>>> point_asdict(search_space, point_as_list)
OrderedDict([('name1', 0.66), ('name2', 3), ('name3', -0.15)])
```
Examples using `skopt.utils.point_asdict`

- Partial Dependence Plots 2D

### 5.8.9 `skopt.utils.point_aslist`

`skopt.utils.point_aslist(search_space, point_as_dict)`

Convert a dictionary representation of a point from a search space to the list representation. The list of values is created from the values of the dictionary, sorted by the names of dimensions used as keys.

See also:

`skopt.utils.point_asdict`

**Parameters**

- `search_space` [dict] Represents search space. The keys are dimension names (strings) and values are instances of classes that inherit from the class `skopt.space.Dimension` (Real, Integer or Categorical)

- `point_as_dict` [dict] dict with parameter names as keys to which corresponding parameter values are assigned.

**Returns**

- `point_as_list` [list] list with point values. The order of parameters in the list is given by `sorted(params_space.keys())`.

**Examples**

```python
>>> from skopt.space.space import Real, Integer
>>> from skopt.utils import point_aslist
>>> search_space = {'name1': Real(0, 1), ...
...     'name2': Integer(2, 4), 'name3': Real(-1, 1)}
>>> point_as_dict = {'name1': 0.66, 'name2': 3, 'name3': -0.15}
>>> point_aslist(search_space, point_as_dict)

[0.66, 3, -0.15]
```

### 5.8.10 `skopt.utils.use_named_args`

`skopt.utils.use_named_args(dimensions)`

Wrapper / decorator for an objective function that uses named arguments to make it compatible with optimizers that use a single list of parameters.

Your objective function can be defined as being callable using named arguments: `func(foo=123, bar=3.0, baz='hello')` for a search-space with dimensions named `['foo', 'bar', 'baz']`. But the optimizer will only pass a single list `x` of unnamed arguments when calling the objective function: `func(x=[123, 3.0, 'hello'])`. This wrapper converts your objective function with named arguments into one that accepts a list as argument, while doing the conversion automatically.

The advantage of this is that you don’t have to unpack the list of arguments `x` yourself, which makes the code easier to read and also reduces the risk of bugs if you change the number of dimensions or their order in the search-space.

**Parameters**
dimensions [list(Dimension)] List of Dimension-objects for the search-space dimensions.

Returns

wrapped_func [callable] Wrapped objective function.

Examples

```python
>>> # Define the search-space dimensions. They must all have names!
>>> from skopt.space import Real
>>> from skopt import forest_minimize
>>> from skopt.utils import use_named_args

>>> dim1 = Real(name='foo', low=0.0, high=1.0)
>>> dim2 = Real(name='bar', low=0.0, high=1.0)
>>> dim3 = Real(name='baz', low=0.0, high=1.0)

>>> # Gather the search-space dimensions in a list.
>>> dimensions = [dim1, dim2, dim3]

>>> # Define the objective function with named arguments
>>> # and use this function-decorator to specify the
>>> # search-space dimensions.
>>> @use_named_args(dimensions=dimensions)
... def my_objective_function(foo, bar, baz):
...     return foo ** 2 + bar ** 4 + baz ** 8

>>> # Not the function is callable from the outside as
>>> # `my_objective_function(x)` where `x` is a list of unnamed arguments,
>>> # which then wraps your objective function that is callable as
>>> # `my_objective_function(foo, bar, baz)`.
>>> # The conversion from a list `x` to named parameters `foo`,
>>> # `bar`, `baz`
>>> # is done automatically.

>>> # Run the optimizer on the wrapped objective function which is called
>>> # as `my_objective_function(x)` as expected by `forest_minimize()`.
>>> result = forest_minimize(func=my_objective_function,
...                          dimensions=dimensions,
...                          n_calls=20, base_estimator="ET",
...                          random_state=4)

>>> # Print the best-found results in same format as the expected result.
>>> print("Best fitness: {} ".format(result.fun))
Best fitness: 0.1948080835239698
>>> print("Best parameters: {} ".format(result.x))
Best parameters: [0.44134853091052617, 0.06570954323368307, 0.17586123323419825]
```
Examples using `skopt.utils.use_named_args`

- Tuning a scikit-learn estimator with skopt

## 5.9 `skopt.sampler`: Samplers

Utilities for generating initial sequences

**User guide:** See the sampler section for further details.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sampler.Lhs()</code></td>
<td>Latin hypercube sampling</td>
</tr>
<tr>
<td><code>sampler.Sobol()</code></td>
<td>Generates a new quasirandom Sobol vector with each call.</td>
</tr>
<tr>
<td><code>sampler.Halton()</code></td>
<td>Creates Halton sequence samples.</td>
</tr>
<tr>
<td><code>sampler.Hammersly()</code></td>
<td>Creates Hammersley sequence samples.</td>
</tr>
</tbody>
</table>

### 5.9.1 `skopt.sampler.Lhs`

**class** `skopt.sampler.Lhs(lhs_type='classic', criterion='maximin', iterations=1000)`

Latin hypercube sampling

**Parameters**

- `lhs_type` [str, default='classic']
  - ‘classic’ - a small random number is added
  - ‘centered’ - points are set uniformly in each interval
- `criterion` [str or None, default='maximin'] When set to None, the LHS is not optimized
  - ‘correlation’ : optimized LHS by minimizing the correlation
  - ‘maximin’ : optimized LHS by maximizing the minimal pdist
  - ‘ratio’ : optimized LHS by minimizing the ratio \( \text{max(pdist)} / \text{min(pdist)} \)
- `iterations` [int] Defines the number of iterations for optimizing LHS

**Methods**

- `generate(dimensions, n_samples[, random_state])` Creates latin hypercube samples.
- `set_params(**params)` Set the parameters of this initial point generator.

```
__init__(lhs_type='classic', criterion='maximin', iterations=1000)
```

```
create(dimensions, n_samples, random_state=None)
```

Creates latin hypercube samples.

**Parameters**

- `dimensions` [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
  - a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
• a \((\text{lower\_bound, upper\_bound, "prior"})\) tuple (for \text{Real} dimensions),
• as a list of categories (for \text{Categorical} dimensions), or
• an instance of a \text{Dimension} object (\text{Real, Integer} or \text{Categorical}).

\text{n\_samples} [\text{int}] The order of the LHS sequence. Defines the number of samples.

\text{random\_state} [\text{int, RandomState instance, or None (default)}] Set random state to something other than None for reproducible results.

\text{Returns}

\text{np.array, shape=(n\_dim, n\_samples)} LHS set

\text{set\_params}(**params)

Set the parameters of this initial point generator.

\text{Parameters}

**params [\text{dict}] Generator parameters.

\text{Returns}

——

\text{self} [\text{object}] Generator instance.

\text{Examples using} \text{skopt.sampler.Lhs}

• \text{Comparing initial sampling methods}
• \text{Comparing initial sampling methods on integer space}

\text{5.9.2 skopt.sampler.Sobol}

\text{class skopt.sampler.Sobol}(skip=0, randomize=True)

Generates a new quasirandom Sobol’ vector with each call.

\text{Parameters}

skip [\text{int}] Skipped seed number.
randomize [\text{bool, default=False}] When set to True, random shift is applied.

\text{Notes}

Sobol’ sequences \cite{1} provide \(n = 2^m\) low discrepancy points in \([0, 1)^{dim}\). Scrambling them makes them suitable for singular integrands, provides a means of error estimation, and can improve their rate of convergence.

There are many versions of Sobol’ sequences depending on their ‘direction numbers’. Here, the maximum number of dimension is 40.

The routine adapts the ideas of Antonov and Saleev \cite{2}.

\text{Warning:} Sobol’ sequences are a quadrature rule and they lose their balance properties if one uses a sample size that is not a power of 2, or skips the first point, or thins the sequence \cite{5}.

If \(n = 2^m\) points are not enough then one should take \(2^M\) points for \(M > m\). When scrambling, the number \(R\) of independent replicates does not have to be a power of 2.
Sobol’ sequences are generated to some number $B$ of bits. Then after $2^B$ points have been generated, the sequence will repeat. Currently $B = 30$.

**References**

[1], [2], [3], [4], [5]

**Methods**

- `generate(dimensions, n_samples[, random_state])` Creates samples from Sobol’ set.

- `__init__(skip=0, randomize=True)`

- `generate(dimensions, n_samples, random_state=None)` Creates samples from Sobol’ set.

  **Parameters**

  - `dimensions` [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
    - a `(lower_bound, upper_bound)` tuple (for Real or Integer dimensions),
    - a `(lower_bound, upper_bound, "prior")` tuple (for Real dimensions),
    - as a list of categories (for Categorical dimensions), or
    - an instance of a Dimension object (Real, Integer or Categorical).

  - `n_samples` [int] The order of the Sobol’ sequence. Defines the number of samples.

  - `random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

  **Returns**

  - `sample` [array_like (n_samples, dim)] Sobol’ set.

- `set_params(**params)`

  Set the parameters of this initial point generator.

  **Parameters**

  - `**params` [dict] Generator parameters.

  **Returns**

Examples using `skopt.sampler.Sobol`

- Comparing initial sampling methods
- Comparing initial sampling methods on integer space

### 5.9.3 `skopt.sampler.Halton`

class `skopt.sampler.Halton`(min_skip=0, max_skip=0, primes=None)

Creates Halton sequence samples.

In statistics, Halton sequences are sequences used to generate points in space for numerical methods such as Monte Carlo simulations. Although these sequences are deterministic, they are of low discrepancy, that is, appear to be random for many purposes. They were first introduced in 1960 and are an example of a quasi-random number sequence. They generalise the one-dimensional van der Corput sequences.

For `dim == 1` the sequence falls back to Van Der Corput sequence.

**Parameters**

- `min_skip` [int] Minimum skipped seed number. When `min_skip != max_skip` a random number is picked.
- `max_skip` [int] Maximum skipped seed number. When `min_skip != max_skip` a random number is picked.
- `primes` [tuple, default=None] The (non-)prime base to calculate values along each axis. If empty or None, growing prime values starting from 2 will be used.

**Methods**

- `generate(dimensions, n_samples[, random_state])` Creates samples from Halton set.
- `set_params(**params)` Set the parameters of this initial point generator.

__init__ (min_skip=0, max_skip=0, primes=None)

`generate(dimensions, n_samples, random_state=None)`

Creates samples from Halton set.

**Parameters**

- `dimensions` [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
  - a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
  - a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
  - as a list of categories (for Categorical dimensions), or
  - an instance of a Dimension object (Real, Integer or Categorical).
- `n_samples` [int] The order of the Halton sequence. Defines the number of samples.
- `random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**Returns**
`np.array, shape=(n_dim, n_samples)` Halton set.

```python
set_params(**params)
```
Set the parameters of this initial point generator.

**Parameters**

**params** [dict] Generator parameters.

**Returns**

——


Examples using `skopt.sampler.Halton`

- Comparing initial sampling methods
- Comparing initial sampling methods on integer space

5.9.4 `skopt.sampler.Hammersly`

```python
class skopt.sampler.Hammersly(min_skip=0, max_skip=0, primes=None)
```
Creates Hammersley sequence samples.

The Hammersley set is equivalent to the Halton sequence, except for one dimension is replaced with a regular grid. It is not recommended to generate a Hammersley sequence with more than 10 dimension.

For `dim == 1` the sequence falls back to Van Der Corput sequence.

**Parameters**

- `min_skip` [int, default=-1] Minimum skipped seed number. When `min_skip != max_skip` and both are > -1, a random number is picked.
- `max_skip` [int, default=-1] Maximum skipped seed number. When `min_skip != max_skip` and both are > -1, a random number is picked.
- `primes` [tuple, default=None] The (non-)prime base to calculate values along each axis. If empty, growing prime values starting from 2 will be used.

**References**


**Methods**

- `generate(dimensions, n_samples[, random_state])` Creates samples from Hammersley set.
- `set_params(**params)` Set the parameters of this initial point generator.
- `__init__(min_skip=0, max_skip=0, primes=None)`

- `generate(dimensions, n_samples, random_state=None)` Creates samples from Hammersley set.
Parameters

dimensions [list, shape (n_dims,)] List of search space dimensions. Each search dimension can be defined either as
  • a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
  • a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
  • as a list of categories (for Categorical dimensions), or
  • an instance of a Dimension object (Real, Integer or Categorical).

n_samples [int] The order of the Hammersley sequence. Defines the number of samples.

random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

Returns

np.array, shape=(n_dim, n_samples) Hammersley set.

set_params(**params)
  Set the parameters of this initial point generator.

Parameters

**params [dict] Generator parameters.

Returns

——


Examples using `skopt.sampler.Hammersly`

• Comparing initial sampling methods
• Comparing initial sampling methods on integer space

5.10 `skopt.space.space`: Space

User guide: See the Space section for further details.

| space.space.Categorical(categories[, prior, ...]) | Search space dimension that can take on categorical values. |
| space.space.Dimension() | Base class for search space dimensions. |
| space.space.Integer(low, high[, prior, ...]) | Search space dimension that can take on integer values. |
| space.space.Real(low, high[, prior, base, ...]) | Search space dimension that can take on any real value. |
| space.space.Space(dimensions) | Initialize a search space from given specifications. |
5.10.1 `skopt.space.space.Categorical`

**class `skopt.space.space.Categorical(categories, prior=None, transform=None, name=None)`**

Search space dimension that can take on categorical values.

**Parameters**

- **categories** `[list, shape=(n_categories,)]` Sequence of possible categories.
- **prior** `[list, shape=(categories,), default=None]` Prior probabilities for each category. By default all categories are equally likely.
- **transform** `[“onehot”, “string”, “identity”, “label”, default=“onehot”]`
  - “identity”, the transformed space is the same as the original space.
  - “string”, the transformed space is a string encoded representation of the original space.
  - “label”, the transformed space is a label encoded representation (integer) of the original space.
  - “onehot”, the transformed space is a one-hot encoded representation of the original space.
- **name** `[str or None]` Name associated with dimension, e.g., “colors”.

**Attributes**

- **bounds**
- **is_constant**
- **name**
- **prior**
- **size**
- **transformed_bounds**
- **transformed_size**

**Methods**

- **distance(a, b)**
  Compute distance between category a and b.
  As categories have no order the distance between two points is one if a != b and zero otherwise.

- **inverse_transform(Xt)**
  Inverse transform samples from the warped space back into the original space.

- **rvs([n_samples, random_state])**
  Draw random samples.

- **set_transformer([transform])**
  Define `_rvs` and transformer spaces.

- **transform(X)**
  Transform samples from the original space to a warped space.

- **__init__(categories, prior=None, transform=None, name=None)**

- **distance(a, b)**
  Compute distance between category a and b.

  Parameters
  - **a** [category] First category.
b [category] Second category.

```python
inverse_transform(Xt)
```
Inverse transform samples from the warped space back into the original space.

```python
rvs(n_samples=None, random_state=None)
```
Draw random samples.

**Parameters**

- `n_samples` [int or None] The number of samples to be drawn.
- `random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

```python
set_transformer(transform='onehot')
```
Define _rvs and transformer spaces.

**Parameters**


```python
transform(X)
```
Transform samples form the original space to a warped space.

## 5.10.2 skopt.space.space.Dimension

**class** skopt.space.space.Dimension

Base class for search space dimensions.

**Attributes**

- `bounds`
- `is_constant`
- `name`
- `prior`
- `size`
- `transformed_bounds`
- `transformed_size`

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inverse_transform(Xt)</code></td>
<td>Inverse transform samples from the warped space back into the original space.</td>
</tr>
<tr>
<td><code>rvs(n_samples, random_state)</code></td>
<td>Draw random samples.</td>
</tr>
<tr>
<td><code>transform(X)</code></td>
<td>Transform samples from the original space to a warped space.</td>
</tr>
</tbody>
</table>

```python
set_transformer |
```

```python
__init__(*args, **kwargs)
```

5.10. skopt.space.space: Space
**inverse_transform**($X_t$)
Inverse transform samples from the warped space back into the original space.

**rvs**($n_samples=1$, $random_state=None$)
Draw random samples.

**Parameters**

- **n_samples** [int or None] The number of samples to be drawn.
- **random_state** [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

**transform**($X$)
Transform samples from the original space to a warped space.

## 5.10.3 **skopt.space.space.Integer**

**class** *skopt.space.space.Integer*($low$, $high$, $prior='uniform'$, $base=10$, $transform=None$, $name=None$, $dtype=<class 'numpy.int64'>*)
Search space dimension that can take on integer values.

**Parameters**

- **low** [int] Lower bound (inclusive).
- **high** [int] Upper bound (inclusive).
- **prior** ["uniform" or "log-uniform", default="uniform"] Distribution to use when sampling random integers for this dimension.
  - If "uniform", integers are sampled uniformly between the lower and upper bounds.
  - If "log-uniform", integers are sampled uniformly between log(lower, base) and log(upper, base) where log has base base.
- **base** [int] The logarithmic base to use for a log-uniform prior.
  - Default 10, otherwise commonly 2.
- **transform** ["identity", "normalize", optional] The following transformations are supported.
  - "identity", (default) the transformed space is the same as the original space.
  - "normalize", the transformed space is scaled to be between 0 and 1.
- **name** [str or None] Name associated with dimension, e.g., “number of trees”.
- **dtype** [str or dtype, default=np.int64] integer type which will be used in inverse_transform, can be int, np.int16, np.uint32, np.int32, np.int64 (default). When set to int, inverse_transform returns a list instead of a numpy array

**Attributes**

- **bounds**
- **is_constant**
- **name**
- **prior**
- **size**
- **transformed_bounds**
transformed_size

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>distance(a, b)</code></td>
<td>Compute distance between point a and b.</td>
</tr>
<tr>
<td><code>inverse_transform(Xt)</code></td>
<td>Inverse transform samples from the warped space back into the original space.</td>
</tr>
<tr>
<td><code>rvs(n_samples, random_state)</code></td>
<td>Draw random samples.</td>
</tr>
<tr>
<td><code>set_transformer(transform)</code></td>
<td>Define _rvs and transformer spaces.</td>
</tr>
<tr>
<td><code>transform(X)</code></td>
<td>Transform samples form the original space to a warped space.</td>
</tr>
</tbody>
</table>

```python
__init__(low, high, prior='uniform', base=10, transform=None, name=None, dtype=<class 'numpy.int64'>)
```

**distance(a, b)**

Compute distance between point a and b.

**Parameters**

- `a` [int] First point.
- `b` [int] Second point.

```python
inverse_transform(Xt)
```

Inverse transform samples from the warped space back into the original space.

```python
rvs(n_samples=1, random_state=None)
```

Draw random samples.

**Parameters**

- `n_samples` [int or None] The number of samples to be drawn.
- `random_state` [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

```python
set_transformer(transform='identity')
```

Define _rvs and transformer spaces.

**Parameters**

- `transform` [str] Can be ‘normalize’ or ‘identity’

```python
transform(X)
```

Transform samples form the original space to a warped space.

### 5.10.4 skopt.space.space.Real

```python
class skopt.space.space.Real(low, high, prior='uniform', base=10, transform=None, name=None, dtype=<class 'float'>)
```

Search space dimension that can take on any real value.

**Parameters**

- `low` [float] Lower bound (inclusive).
- `high` [float] Upper bound (inclusive).
prior ["uniform" or “log-uniform”, default=”uniform"] Distribution to use when sampling random points for this dimension.

- If "uniform", points are sampled uniformly between the lower and upper bounds.
- If "log-uniform", points are sampled uniformly between \( \log(\text{lower}, \text{base}) \) and \( \log(\text{upper}, \text{base}) \) where \( \log \) has base \( \text{base} \).

base [int] The logarithmic base to use for a log-uniform prior. - Default 10, otherwise commonly 2.

transform [“identity”, “normalize”, optional] The following transformations are supported.

- “identity”, (default) the transformed space is the same as the original space.
- “normalize”, the transformed space is scaled to be between 0 and 1.

ame [str or None] Name associated with the dimension, e.g., “learning rate”.
dtype [str or dtype, default=float] float type which will be used in inverse_transform, can be float.

Attributes

- bounds
- is_constant
- name
- prior
- size
- transformed_bounds
- transformed_size

Methods

---

**distance(a, b)**

Compute distance between point a and b.

**inverse_transform(Xt)**

Inverse transform samples from the warped space back into the original space.

**rvs([n_samples, random_state])**

Draw random samples.

**set_transformer([transform])**

Define rvs and transformer spaces.

**transform(X)**

Transform samples form the original space to a warped space.

---

```python
__init__(low, high, prior='uniform', base=10, transform=None, name=None, dtype=<class 'float'>)
```

**distance(a, b)**

Compute distance between point a and b.

**Parameters**

- a [float] First point.
- b [float] Second point.

**inverse_transform(Xt)**

Inverse transform samples from the warped space back into the original space.
rvs(n_samples=1, random_state=None)
Draw random samples.

Parameters

n_samples [int or None] The number of samples to be drawn.
random_state [int, RandomState instance, or None (default)] Set random state to something
other than None for reproducible results.

set_transformer(transform='identity')
Define rvs and transformer spaces.

Parameters

transform [str] Can be ‘normalize’ or ‘identity’

transform(X)
Transform samples form the original space to a warped space.

5.10.5 skopt.space.space.Space

class skopt.space.space.Space(dimensions)
Initialize a search space from given specifications.

Parameters

dimensions [list, shape=(n_dims,)] List of search space dimensions. Each search dimension can
be defined either as
- a (lower_bound, upper_bound) tuple (for Real or Integer dimensions),
- a (lower_bound, upper_bound, "prior") tuple (for Real dimensions),
- as a list of categories (for Categorical dimensions), or
- an instance of a Dimension object (Real, Integer or Categorical).

Note: The upper and lower bounds are inclusive for Integer dimensions.

Attributes

bounds The dimension bounds, in the original space.
dimension_names Names of all the dimensions in the search-space.
is_categorical Space contains exclusively categorical dimensions
is_partly_categorical Space contains any categorical dimensions
is_real Returns true if all dimensions are Real
n_constant_dimensions Returns the number of constant dimensions which have zero degree
of freedom, e.g.
n_dims The dimensionality of the original space.
transformed_bounds The dimension bounds, in the warped space.
transformed_n_dims The dimensionality of the warped space.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>distance(point_a, point_b)</code></td>
<td>Compute distance between two points in this space.</td>
</tr>
<tr>
<td><code>from_yaml(yml_path[, namespace])</code></td>
<td>Create Space from yaml configuration file</td>
</tr>
<tr>
<td><code>get_transformer()</code></td>
<td>Returns all transformers as list</td>
</tr>
<tr>
<td><code>inverse_transform(Xt)</code></td>
<td>Inverse transform samples from the warped space back to the original space</td>
</tr>
<tr>
<td><code>rvs([n_samples, random_state])</code></td>
<td>Draw random samples.</td>
</tr>
<tr>
<td><code>set_transformer(transform)</code></td>
<td>Sets the transformer of all dimension objects to transform</td>
</tr>
<tr>
<td><code>set_transformer_by_type(transform, dim_type)</code></td>
<td>Sets the transformer of dim_type objects to transform</td>
</tr>
<tr>
<td><code>transform(X)</code></td>
<td>Transform samples from the original space into a warped space.</td>
</tr>
</tbody>
</table>

__init__(dimensions)

**property bounds**
The dimension bounds, in the original space.

**property dimension_names**
Names of all the dimensions in the search-space.

**distance(point_a, point_b)**
Compute distance between two points in this space.

Parameters

- `point_a` [array] First point.
- `point_b` [array] Second point.

**classmethod from_yaml(yml_path, namespace=None)**
Create Space from yaml configuration file

Parameters

- `yml_path` [str] Full path to yaml configuration file, example YaML below: Space:
  - Integer: low: -5 high: 5
  - Categorical: categories: - a - b
  - Real: low: 1.0 high: 5.0 prior: log-uniform

- `namespace` [str, default=None] Namespace within configuration file to use, will use first namespace if not provided

Returns

- `space` [Space] Instantiated Space object

**get_transformer()**
Returns all transformers as list

**inverse_transform(Xt)**
Inverse transform samples from the warped space back to the original space.

Parameters
Xt [array of floats, shape=(n_samples, transformed_n_dims)] The samples to inverse transform.

Returns

X [list of lists, shape=(n_samples, n_dims)] The original samples.

property is_categorical
Space contains exclusively categorical dimensions

property is_partly_categorical
Space contains any categorical dimensions

property is_real
Returns true if all dimensions are Real

property n_constant_dimensions
Returns the number of constant dimensions which have zero degree of freedom, e.g. an Integer dimensions with (0., 0.) as bounds.

property n_dims
The dimensionality of the original space.

rvs(n_samples=1, random_state=None)
Draw random samples.

The samples are in the original space. They need to be transformed before being passed to a model or minimizer by space.transform().

Parameters

n_samples [int, default=1] Number of samples to be drawn from the space.

random_state [int, RandomState instance, or None (default)] Set random state to something other than None for reproducible results.

Returns

points [list of lists, shape=(n_points, n_dims)] Points sampled from the space.

set_transformer(transform)
Sets the transformer of all dimension objects to transform

Parameters

transform [str or list of str] Sets all transformer, when transform is a string. Otherwise, transform must be a list with strings with the same length as dimensions

set_transformer_by_type(transform, dim_type)
Sets the transformer of dim_type objects to transform

Parameters

transform [str] Sets all transformer of type dim_type to transform

dim_type [type]

Can be skopt.space.Real, skopt.space.Integer or skopt.space.Categorical

transform(X)
Transform samples from the original space into a warped space.

Note: this transformation is expected to be used to project samples into a suitable space for numerical optimization.
scikit-optimize Documentation, Release 0.9.0

Parameters

- **X** [list of lists, shape=(n_samples, n_dims)] The samples to transform.

Returns

- **Xt** [array of floats, shape=(n_samples, transformed_n_dims)] The transformed samples.

**property transformed_bounds**

The dimension bounds, in the warped space.

**property transformed_n_dims**

The dimensionality of the warped space.

```python
space.space.check_dimension(dimension[, ...])
```

Turn a provided dimension description into a dimension object.

---

### 5.10.6 `skopt.space.space.check_dimension`

`skopt.space.space.check_dimension(dimension, transform=None)`

Turn a provided dimension description into a dimension object.

Checks that the provided dimension falls into one of the supported types. For a list of supported types, look at the documentation of `dimension` below.

If `dimension` is already a `Dimension` instance, return it.

**Parameters**

- **dimension** [Dimension] Search space Dimension. Each search dimension can be defined either as
  - a `(lower_bound, upper_bound)` tuple (for `Real` or `Integer` dimensions),
  - a `(lower_bound, upper_bound, "prior")` tuple (for `Real` dimensions),
  - as a list of categories (for `Categorical` dimensions), or
  - an instance of a `Dimension` object (`Real`, `Integer` or `Categorical`).

- **transform** ["identity", “normalize”, “string”, “label”, “onehot” optional]
  - For `Categorical` dimensions, the following transformations are supported.
    - “onehot” (default) one-hot transformation of the original space.
    - “label” integer transformation of the original space
    - “string” string transformation of the original space.
    - “identity” same as the original space.
  - For `Real` and `Integer` dimensions, the following transformations are supported.
    - “identity”, (default) the transformed space is the same as the original space.
    - “normalize”, the transformed space is scaled to be between 0 and 1.

**Returns**

- **dimension** [Dimension] Dimension instance.
5.11 `skopt.space.transformers`: transformers

**User guide:** See the transformers section for further details.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skopt.space.transformers.CategoricalEncoder</code></td>
<td>OneHotEncoder that can handle categorical variables.</td>
</tr>
<tr>
<td><code>skopt.space.transformers.Identity()</code></td>
<td>Identity transform.</td>
</tr>
<tr>
<td><code>skopt.space.transformers.LogN(base)</code></td>
<td>Base N logarithm transform.</td>
</tr>
<tr>
<td><code>skopt.space.transformers.Normalize(low, high[, is_int])</code></td>
<td>Scales each dimension into the interval [0, 1].</td>
</tr>
<tr>
<td><code>skopt.space.transformers.Pipeline(transformers)</code></td>
<td>A lightweight pipeline to chain transformers.</td>
</tr>
<tr>
<td><code>skopt.space.transformers.Transformer()</code></td>
<td>Base class for all 1-D transformers.</td>
</tr>
<tr>
<td><code>skopt.space.transformers.LabelEncoder([X])</code></td>
<td>LabelEncoder that can handle categorical variables.</td>
</tr>
<tr>
<td><code>skopt.space.transformers.StringEncoder([dtype])</code></td>
<td>StringEncoder transform.</td>
</tr>
</tbody>
</table>

### 5.11.1 `skopt.space.transformers.CategoricalEncoder`

**class** `skopt.space.transformers.CategoricalEncoder`

OneHotEncoder that can handle categorical variables.

**Methods**

- **`fit(X)`**
  Fit a list or array of categories.

- **`inverse_transform(Xt)`**
  Inverse transform one-hot encoded categories back to their original representation.

- **`transform(X)`**
  Transform an array of categories to a one-hot encoded representation.

**`__init__()`**

Convert labeled categories into one-hot encoded features.

**`fit(X)`**

Fit a list or array of categories.

**Parameters**

- `X` [array-like, shape=(n_categories,)] List of categories.

**`inverse_transform(Xt)`**

Inverse transform one-hot encoded categories back to their original representation.

**Parameters**

- `Xt` [array-like, shape=(n_samples, n_categories)] One-hot encoded categories.

**Returns**

- `X` [array-like, shape=(n_samples,)] The original categories.

**`transform(X)`**

Transform an array of categories to a one-hot encoded representation.

**Parameters**

- `X` [array-like, shape=(n_categories,)] List of categories.
X  [array-like, shape=(n_samples,)] List of categories.

Returns

Xt  [array-like, shape=(n_samples, n_categories)] The one-hot encoded categories.

5.11.2 skopt.space.transformers.Identity

class skopt.space.transformers.Identity
Identity transform.

Methods

__init__(*args, **kwargs)

5.11.3 skopt.space.transformers.LogN

class skopt.space.transformers.LogN(base)
Base N logarithm transform.

Methods

__init__(base)

5.11.4 skopt.space.transformers.Normalize

class skopt.space.transformers.Normalize(low, high, is_int=False)
Scales each dimension into the interval [0, 1].

Parameters

low  [float] Lower bound.

high  [float] Higher bound.

is_int  [bool, default=False] Round and cast the return value of inverse_transform to integer.
        Set to True when applying this transform to integers.
Methods

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit</td>
</tr>
<tr>
<td>inverse_transform</td>
</tr>
<tr>
<td>transform</td>
</tr>
</tbody>
</table>

`__init__`(low, high, is_int=False)

### 5.11.5 `skopt.space.transformers.Pipeline`

class `skopt.space.transformers.Pipeline(transformers)`

A lightweight pipeline to chain transformers.

**Parameters**

- `transformers` [list] A list of Transformer instances.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit</td>
</tr>
<tr>
<td>inverse_transform</td>
</tr>
<tr>
<td>transform</td>
</tr>
</tbody>
</table>

`__init__`(transformers)

### 5.11.6 `skopt.space.transformers.Transformer`

class `skopt.space.transformers.Transformer`

Base class for all 1-D transformers.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit</td>
</tr>
<tr>
<td>inverse_transform</td>
</tr>
<tr>
<td>transform</td>
</tr>
</tbody>
</table>

`__init__`(*args, **kwargs)`
5.11.7 skopt.space.transformers.LabelEncoder

class skopt.space.transformers.LabelEncoder(X=None)
LabelEncoder that can handle categorical variables.

Methods

**fit**(X)  
Fit a list or array of categories.

**inverse_transform**(Xt)  
Inverse transform integer categories back to their original representation.

**transform**(X)  
Transform an array of categories to a one-hot encoded representation.

__init__ (X=None)

**fit**(X)
Fit a list or array of categories.

Parameters

X [array-like, shape=(n_categories,)] List of categories.

**inverse_transform**(Xt)
Inverse transform integer categories back to their original representation.

Parameters

Xt [array-like, shape=(n_samples, n_categories)] Integer categories.

Returns

X [array-like, shape=(n_samples,)] The original categories.

**transform**(X)
Transform an array of categories to a one-hot encoded representation.

Parameters

X [array-like, shape=(n_samples,)] List of categories.

Returns

Xt [array-like, shape=(n_samples, n_categories)] The integer categories.

5.11.8 skopt.space.transformers.StringEncoder

class skopt.space.transformers.StringEncoder(dtype=class 'str')
StringEncoder transform. The transform will cast everything to a string and the inverse transform will cast to the type defined in dtype.
Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit(X)</code></td>
<td>Fit a list or array of categories.</td>
</tr>
<tr>
<td><code>inverse_transform(Xt)</code></td>
<td>Inverse transform string encoded categories back to their original</td>
</tr>
<tr>
<td><code>transform(X)</code></td>
<td>Transform an array of categories to a string encoded representation.</td>
</tr>
</tbody>
</table>

__init__(dtype=<class 'str'>)

`fit(X)`
Fit a list or array of categories. All elements must be from the same type.

Parameters

- `X` [array-like, shape=(n_categories,)] List of categories.

`inverse_transform(Xt)`
Inverse transform string encoded categories back to their original representation.

Parameters

- `Xt` [array-like, shape=(n_samples,)] String encoded categories.

Returns

- `X` [array-like, shape=(n_samples,)] The original categories.

`transform(X)`
Transform an array of categories to a string encoded representation.

Parameters

- `X` [array-like, shape=(n_samples,)] List of categories.

Returns

- `Xt` [array-like, shape=(n_samples,)] The string encoded categories.
The library is still experimental and under heavy development. Checkout the next milestone for the plans for the next release or look at some easy issues to get started contributing.

The development version can be installed through:

```
git clone https://github.com/scikit-optimize/scikit-optimize.git
cd scikit-optimize
pip install -e.
```

Run all tests by executing pytest in the top level directory.

To only run the subset of tests with short run time, you can use `pytest -m 'fast_test'` (pytest -m 'slow_test' is also possible). To exclude all slow running tests try `pytest -m 'not slow_test'`.

This is implemented using pytest attributes. If a tests runs longer than 1 second, it is marked as slow, else as fast.

All contributors are welcome!

### 6.1 Making a Release

The release procedure is almost completely automated. By tagging a new release travis will build all required packages and push them to PyPI. To make a release create a new issue and work through the following checklist:

- update the version tag in `__init__.py`
- update the version tag mentioned in the README
- check if the dependencies in `setup.py` are valid or need unpinning
- check that the `CHANGELOG.md` is up to date
- did the last build of master succeed?
- create a new release
- ping conda-forge

Before making a release we usually create a release candidate. If the next release is v0.X then the release candidate should be tagged v0.Xrc1 in `__init__.py`. Mark a release candidate as a “pre-release” on GitHub when you tag it.


[4] Bennett Fox, Algorithm 647: Implementation and Relative Efficiency of Quasirandom Sequence Generators,

INDEX

Symbols

__init__() (skopt.BayesSearchCV method), 141
__init__() (skopt.Optimizer method), 146
__init__() (skopt.Space method), 148
__init__() (skopt.callbacks.CheckpointSaver method), 169
__init__() (skopt.callbacks.DeadlineStopper method), 170
__init__() (skopt.callbacks.DeltaXStopper method), 170
__init__() (skopt.callbacks.DeltaYStopper method), 170
__init__() (skopt.callbacks.EarlyStopper method), 171
__init__() (skopt.callbacks.TimerCallback method), 171
__init__() (skopt.callbacks.VerboseCallback method), 172
__init__() (skopt.learning.ExtraTreesRegressor method), 174
__init__() (skopt.learning.GaussianProcessRegressor method), 179
__init__() (skopt.learning.GradientBoostingQuantileRegressor method), 182
__init__() (skopt.learning.RandomForestRegressor method), 186
__init__() (skopt.optimizer.Optimizer method), 191
__init__() (skopt.sampler.Halton method), 223
__init__() (skopt.sampler.Hammersly method), 224
__init__() (skopt.sampler.Lhs method), 220
__init__() (skopt.sampler.Sobol method), 222
__init__() (skopt.space.space.Categorical method), 226
__init__() (skopt.space.space.Dimension method), 227
__init__() (skopt.space.space.Integer method), 229
__init__() (skopt.space.space.Real method), 230
__init__() (skopt.space.space.Space method), 232
__init__() (skopt.space.transformers.CategoricalEncoder method), 235
__init__() (skopt.space.transformers.Identity method), 236
__init__() (skopt.space.transformers.LabelEncoder method), 238
__init__() (skopt.space.transformers.LogN method), 236
__init__() (skopt.space.transformers.Normalize method), 237
__init__() (skopt.space.transformers.Pipeline method), 237
__init__() (skopt.space.transformers.StringEncoder method), 239
__init__() (skopt.space.transformers.Transformer method), 237

A
apply() (skopt.learning.ExtraTreesRegressor method), 175
apply() (skopt.learning.RandomForestRegressor method), 186
ask() (skopt.Optimizer method), 146
ask() (skopt.optimizer.Optimizer method), 191

B
base_minimize() (in module skopt.optimizer), 192
BayesSearchCV (class in skopt), 137
bench1() (in module skopt.benchmarks), 166
bench1_with_time() (in module skopt.benchmarks), 167
bench2() (in module skopt.benchmarks), 167
bench3() (in module skopt.benchmarks), 167
bench4() (in module skopt.benchmarks), 167
bench5() (in module skopt.benchmarks), 167
bounds (skopt.Space property), 148
bounds (skopt.space.space.Space property), 232
branin() (in module skopt.benchmarks), 167

C
Categorical (class in skopt.space.space), 226
CategoricalEncoder (class in skopt.space.transformers), 235
check_dimension() (in module skopt.space.space), 234
CheckpointSaver (class in skopt.callbacks), 169
classes_ (skopt.BayesSearchCV property), 141
cook_estimator() (in module skopt.utils), 214
cook_initial_point_generator() (in module skopt.utils), 214
copy() (skopt.Optimizer method), 147
copy() (skopt.optimizer.Optimizer method), 191

D
DeadlineStopper (class in skopt.callbacks), 169
decision_function() (skopt.BayesSearchCV method), 141
decision_path() (skopt.learning.ExtraTreesRegressor method), 175
decision_path() (skopt.learning.RandomForestRegressor method), 186
DeltaXStopper (class in skopt.callbacks), 170
DeltaYStopper (class in skopt.callbacks), 170
Dimension (class in skopt.space.space), 227
dimension_names (skopt.Space property), 149
dimension_names (skopt.space.Space property), 232
dimensions_aslist() (in module skopt.space), 215
distance() (skopt.Space method), 149
distance() (skopt.space.Space.Categorical method), 226
distance() (skopt.space.space.Integer method), 229
distance() (skopt.space.space.Real method), 230
distance() (skopt.space.space.Space method), 232
dummy_minimize() (in module skopt), 151
dummy_minimize() (in module skopt.optimizer), 195
dump() (in module skopt), 153
dump() (in module skopt.utils), 216

E
EarlyStopper (class in skopt.callbacks), 171
expected_minimum() (in module skopt), 153
expected_minimum() (in module skopt.space.space), 215
expected_minimum_random_sampling() (in module skopt), 154
expected_minimum_random_sampling() (in module skopt.space.space), 216
ExtraTreesRegressor (class in skopt.learning), 172

F
feature_importances_ (skopt.learning.ExtraTreesRegressor property), 175
feature_importances_ (skopt.learning.RandomForestRegressor property), 175
fit() (skopt.BayesSearchCV method), 141
fit() (skopt.learning.ExtraTreesRegressor method), 175
fit() (skopt.learning.GradientBoostingQuantileRegressor method), 179
fit() (skopt.learning.GaussianProcessRegressor method), 182
fit() (skopt.space.transformers.CategoricalEncoder method), 235
fit() (skopt.space.transformers.LabelEncoder method), 238
fit() (skopt.space.transformers.StringEncoder method), 239
forest_minimize() (in module skopt), 154
forest_minimize() (in module skopt.optimizer), 197
from_yaml() (skopt.Space class method), 149
from_yaml() (skopt.space.space.Space class method), 232

G
gaussian_acquisition_1D() (in module skopt.acquisition), 164
gaussian_ei() (in module skopt.acquisition), 164
gaussian_lcb() (in module skopt.acquisition), 165
gaussian_pi() (in module skopt.acquisition), 165
GaussianProcessRegressor (class in skopt.learning), 177
gbtr_minimize() (in module skopt), 157
gbtr_minimize() (in module skopt.optimizer), 199
generate() (skopt.sampler.Hammersly method), 223
generate() (skopt.sampler.Hammersly method), 224
generate() (skopt.sampler.Lhs method), 220
generate() (skopt.sampler.Sobol method), 222
get_params() (skopt.BayesSearchCV method), 142
get_params() (skopt.learning.ExtraTreesRegressor method), 176
get_params() (skopt.learning.GaussianProcessRegressor method), 179
get_params() (skopt.learning.GradientBoostingQuantileRegressor method), 182
get_params() (skopt.learning.RandomForestRegressor method), 187
get_result() (skopt.Optimizer method), 147
get_result() (skopt.optimizer.Optimizer method), 191
get_transformer() (skopt.Space method), 149
get_transformer() (skopt.space.space.Space method), 232
gp_minimize() (in module skopt), 159
gp_minimize() (in module skopt.optimizer), 202
GradientBoostingQuantileRegressor (class in skopt.learning), 182

H
Halton (class in skopt.sampler), 223
Hammersly (class in skopt.sampler), 224
hart6() (in module skopt.benchmarks), 168

I
Identity (class in skopt.space.transformers), 236
Integer (class in skopt.space.space), 228
inverse_transform() (skopt.BayesSearchCV method), 142
inverse_transform() (skopt.Space method), 149
inverse_transform() (skopt.space.space.Categorical method), 227
inverse_transform() (skopt.space.space.Dimension method), 227
inverse_transform() (skopt.space.space.Integer method), 229
inverse_transform() (skopt.space.space.Real method), 230
inverse_transform() (skopt.space.space.Space method), 232
inverse_transform()
    (skopt.space.transformers.CategoricalEncoder method), 235
inverse_transform()
    (skopt.space.transformers.LabelEncoder method), 238
inverse_transform()
    (skopt.space.transformers.StringEncoder method), 239
is_categorical (skopt.Space property), 149
is_categorical (skopt.space.space.Space property), 233
is_partly_categorical (skopt.Space property), 149
is_partly_categorical (skopt.space.space.Space property), 233
is_real (skopt.Space property), 149
is_real (skopt.space.space.Space property), 233

LabelEncoder (class in skopt.space.transformers), 238
Lhs (class in skopt.sampler), 220
load() (in module skopt), 163
load() (in module skopt.utils), 217
log_marginal_likelihood()
    (skopt.learning.GaussianProcessRegressor method), 179
LogN (class in skopt.space.transformers), 236

module
    skopt.acquisition, 163
    skopt.benchmarks, 166
    skopt.callbacks, 168
    skopt.learning, 172
    skopt.optimizer, 189
    skopt.plots, 205
    skopt.sampler, 220
    skopt.space, 225
    skopt.space.transformers, 235
    skopt.utils, 213

N
n_constant_dimensions (skopt.Space property), 149
n_constant_dimensions (skopt.space.space.Space property), 233
n_dims (skopt.Space property), 149
n_dims (skopt.space.space.Space property), 233
n_features_ (skopt.learning.ExtraTreesRegressor property), 176
n_features_ (skopt.learning.RandomForestRegressor property), 187
n_features_in_ (skopt.BayesSearchCV property), 142
Normalize (class in skopt.space.transformers), 236

O
Optimizer (class in skopt), 144
Optimizer (class in skopt.optimizer), 189

P
partial_dependence() (in module skopt.plots), 206
partial_dependence_1D() (in module skopt.plots), 207
partial_dependence_2D() (in module skopt.plots), 207
Pipeline (class in skopt.space.transformers), 237
plot_convergence() (in module skopt.plots), 208
plot_evaluations() (in module skopt.plots), 208
plot_gaussian_process() (in module skopt.plots), 209
plot_histogram() (in module skopt.plots), 212
plot_objective() (in module skopt.plots), 210
plot_objective_2D() (in module skopt.plots), 212
plot_regret() (in module skopt.plots), 213
point_asdict() (in module skopt.utils), 217
point_aslist() (in module skopt.utils), 218
predict() (skopt.BayesSearchCV method), 142
predict()
    (skopt.learning.ExtraTreesRegressor method), 176
predict()
    (skopt.learning.GaussianProcessRegressor method), 180
predict()
    (skopt.learning.GradientBoostingQuantileRegressor method), 183
predict()
    (skopt.learning.RandomForestRegressor method), 187
predict_log_proba() (skopt.BayesSearchCV method), 142
predict_proba() (skopt.BayesSearchCV method), 143

R
RandomForestRegressor (class in skopt.learning), 184
Real (class in skopt.space.space), 229
run() (skopt.Optimizer method), 147
run() (skopt.optimizer.Optimizer method), 192
rvs() (skopt.Space method), 150

Index 247