scikit-optimize Documentation

Release 0.7.4

The scikit-optimize Contributors.

Feb 23, 2020

CONTENTS

CHAPTER

ONE

WELCOME TO SCIKIT-OPTIMIZE

1.1 Installation

scikit-optimize supports Python 3.5 or newer.

The newest release can be installed via pip:

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

or via conda:

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

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

```
$ 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:

```
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.

Versions 0.7.3 and 0.7.4 fixes the missing LICENSE in the package source

1.2.1 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

1.2.2 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

Miscellaneous

- Sphinx documentation #809 by Holger Nahrstaedt
- notebooks are replaced by sphinx-gallery #811 by Holger Nahrstaedt
- Improve sphinx doc #819 by Holger Nahrstaedt
- Old pdoc scripts are removed and replaced by sphinx #822 by Holger Nahrstaedt

1.2.3 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

1.2.4 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`

1.2.5 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]'`)

1.2.6 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.

1.2.7 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 stategies.
- BayesSearchCV as a drop in replacement for scikit-learn's GridSearchCV.
- New acquisition functions "EIps" 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.

1.2.8 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`.

1.2.9 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.

1.2.10 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.

CHAPTER

TWO

GETTING STARTED

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

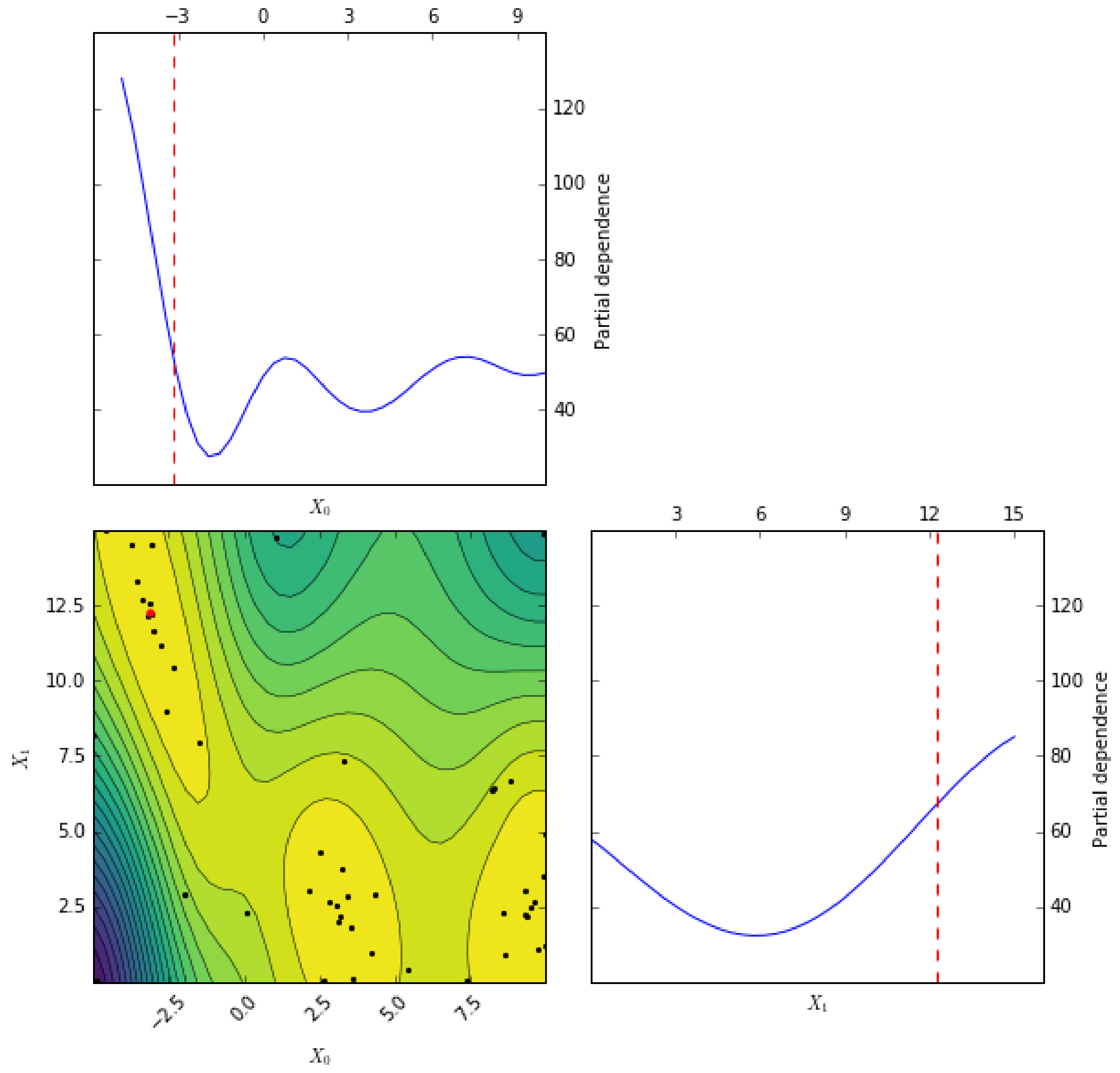
Find the minimum of the noisy function `f(x)` over the range `-2 < x < 2` with `skopt`:

```
import numpy as np
from skopt import gp_minimize

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)])
```

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

```
from skopt import Optimizer

opt = Optimizer([(-2.0, 2.0)])

for i in range(20):
    suggested = opt.ask()
    y = f(suggested)
    opt.tell(suggested, y)
    print('iteration:', i, suggested, y)
```

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


−3
0
3
6
9
120
100
80
60
40
Partial dependence
X_0
12.5
10.0
7.5
5.0
2.5
X_1
−2.5
0.0
2.5
5.0
7.5
X_0
3
6
9
12
15
120
100
80
60
40
Partial dependence
X_1

CHAPTER

THREE

USER GUIDE

3.1 Acquisition

3.2 BayesSearchCV, a GridSearchCV compatible estimator

Use `BayesSearchCV` as a replacement for scikit-learn's GridSearchCV.

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.

- *`dummy_minimize`*
- *`forest_minimize`*
- *`gbrt_minimize`*
- *`gp_minimize`*

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.

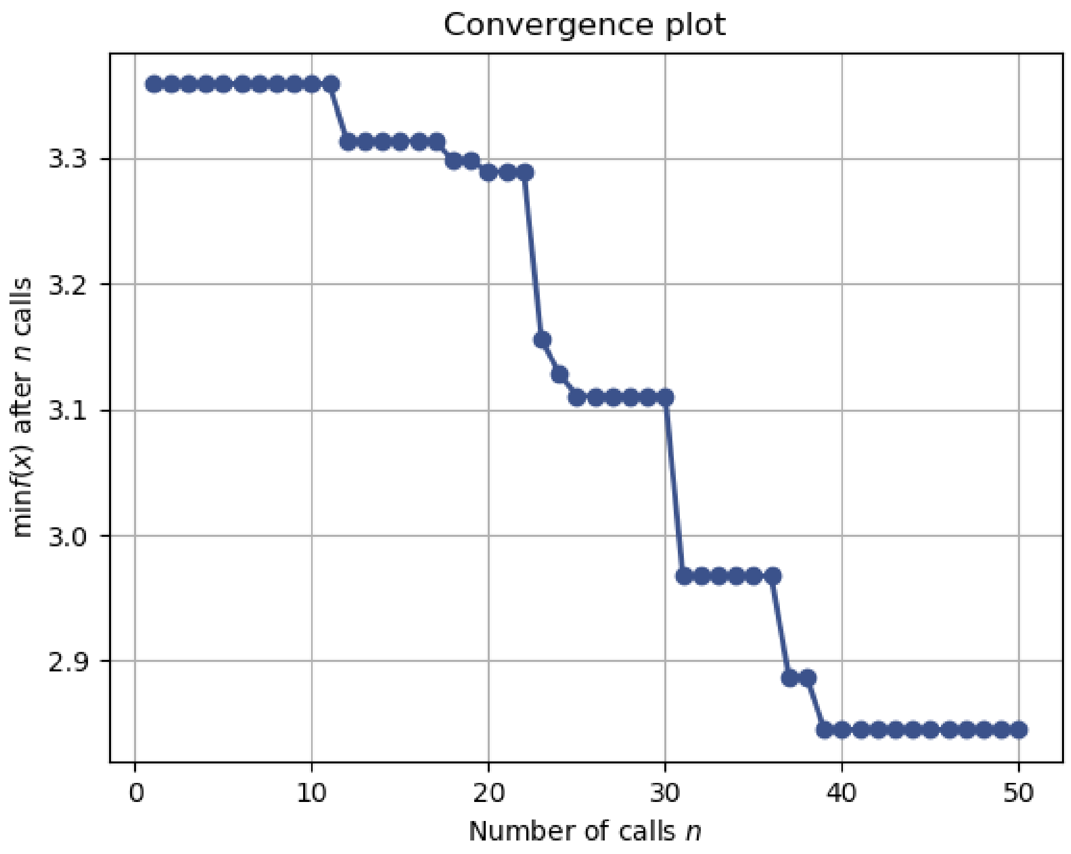
3.6.4 plot_regret

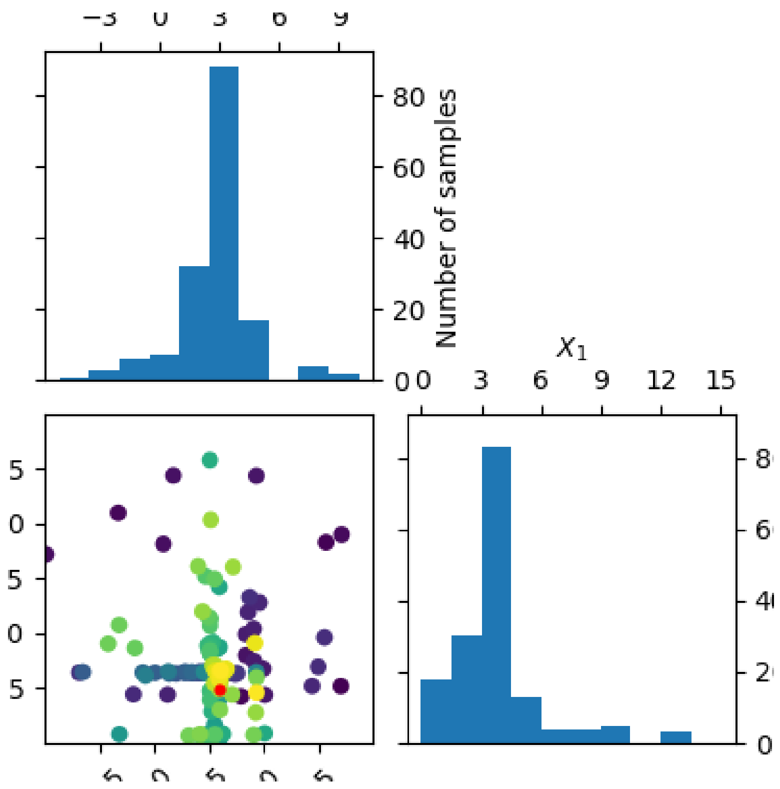
`plot_regret` plot one or several cumulative regret traces.

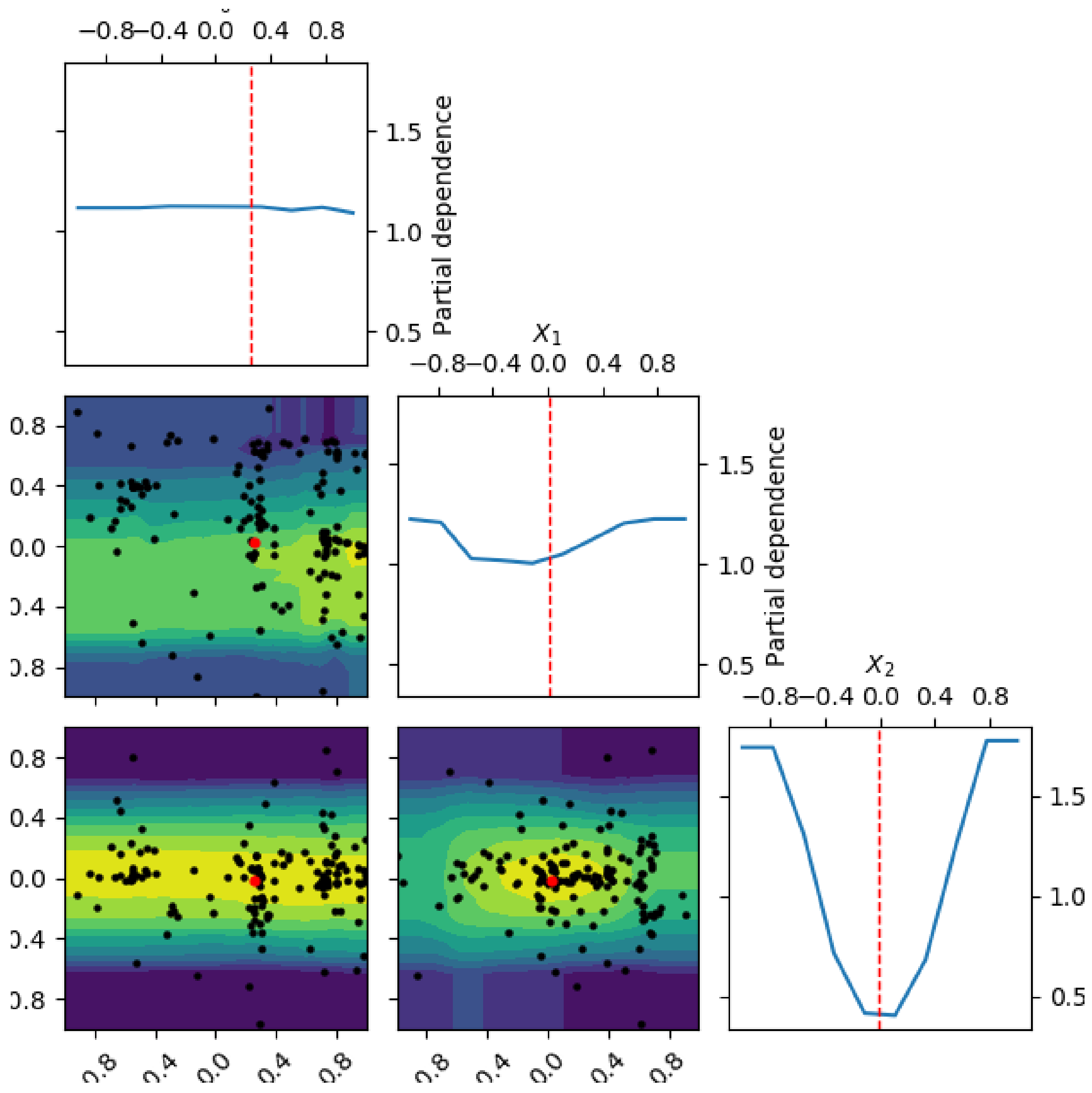
3.7 Space define the optimization space

3.8 Utility functions

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


Convergence plot
min f(x) after n calls
3.3
3.2
3.1
3.0
2.9
0
10
20
30
40
50
Number of calls n


−3
0
3
6
9
80
60
40
20
0
Number of samples
X_1
0
3
6
9
12
15
5
0
5
0
5



Partial dependence
X_1
Partial dependence
X_2

CHAPTER

FOUR

EXAMPLES

4.1 Miscellaneous examples

Miscellaneous and introductory examples for scikit-optimize.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.1 Parallel optimization

Iaroslav Shcherbatyi, 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.

```
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:

```
0.39803969617957335
```

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"[1] 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 27.735 seconds)

Estimated memory usage: 29 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.2 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.

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 less 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.

```
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.

```
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)
```

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.

```
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(10**-5, 10**0, "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"))
```

Optimize all the things!

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

```
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=2.8451'
```

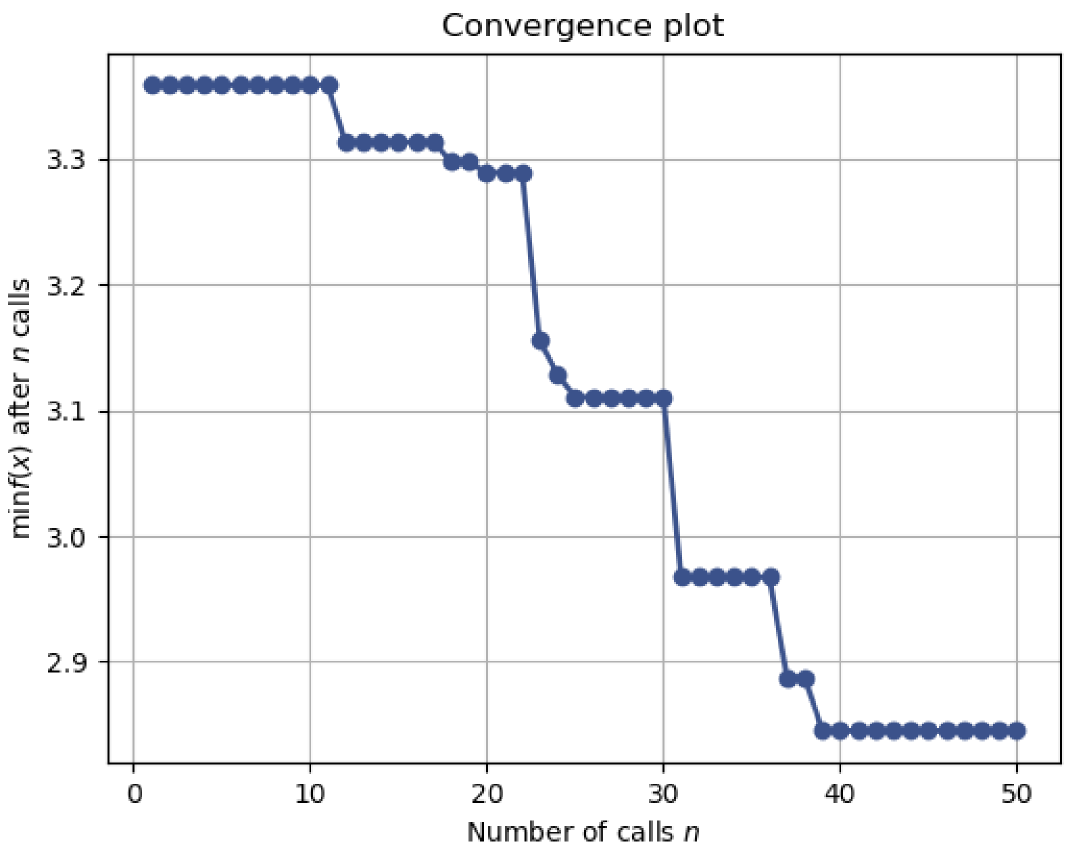
```
print("""Best parameters:
- max_depth=%d
- learning_rate=%.6f
- max_features=%d
- min_samples_split=%d
- min_samples_leaf=%d""" % (res_gp.x[0], res_gp.x[1],
                            res_gp.x[2], res_gp.x[3],
                            res_gp.x[4]))
```

Out:

```
Best parameters:
- max_depth=5
- learning_rate=0.119428
- max_features=9
- min_samples_split=2
- min_samples_leaf=1
```

Convergence plot

```
from skopt.plots import plot_convergence

plot_convergence(res_gp)
```



Out:

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7f8320cacdc0>
```

Total running time of the script: (0 minutes 30.711 seconds)

Estimated memory usage: 33 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.3 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 an 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!

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

# The followings are hacks to allow sphinx-gallery to run the example.
sys.path.insert(0, os.getcwd())
main_dir = os.path.basename(sys.modules['__main__'].__file__)
IS_RUN_WITH_SPHINX_GALLERY = main_dir != os.getcwd()
```

Simple example

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

```
from skopt import gp_minimize
noise_level = 0.1

if IS_RUN_WITH_SPHINX_GALLERY:
    # When this example is run with sphinx gallery, it breaks the pickling
    # capacity for multiprocessing backend so we have to modify the way we
    # define our functions. This has nothing to do with the example.
    from utils import obj_fun
else:
    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=0,  # the number of random initialization points
                  random_state=777)
```

Out:

```
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: 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:

```
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:

```
-0.2423455753391654
```

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: 80120 bytes
Compressed with gz:  19024 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
↪', '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', '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 it is not critical to you, you can simply delete it before calling *skopt.dump*. In this case, no deep copy is created:

```
del res.specs['args']['func']

dump(res, 'result_without_objective_2.pkl')
```

Total running time of the script: (0 minutes 2.774 seconds)

Estimated memory usage: 11 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.4 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 `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.

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

Toy model

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

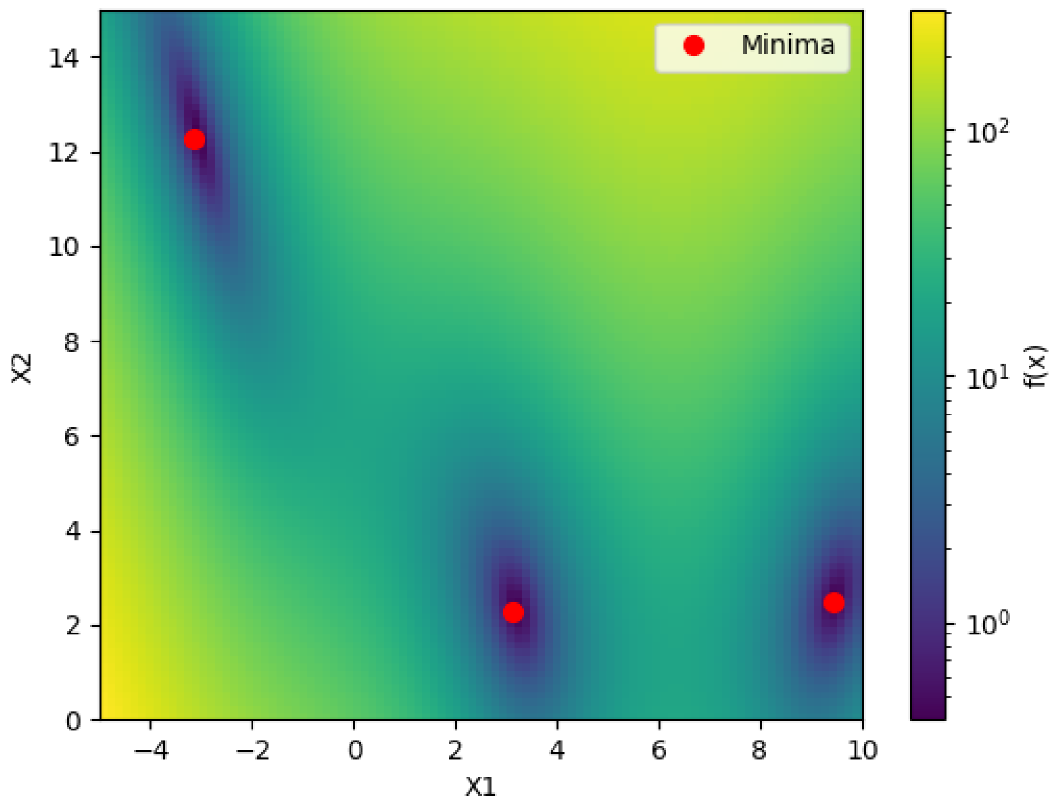
```
from skopt.benchmarks import branin as _branin

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()))

    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)")

    ax.legend(loc="best", numpoints=1)

    ax.set_xlabel("X1")
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlim([-5, 10])
    ax.set_ylabel("X2")
    ax.set_ylim([0, 15])


plot_branin()
```



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.

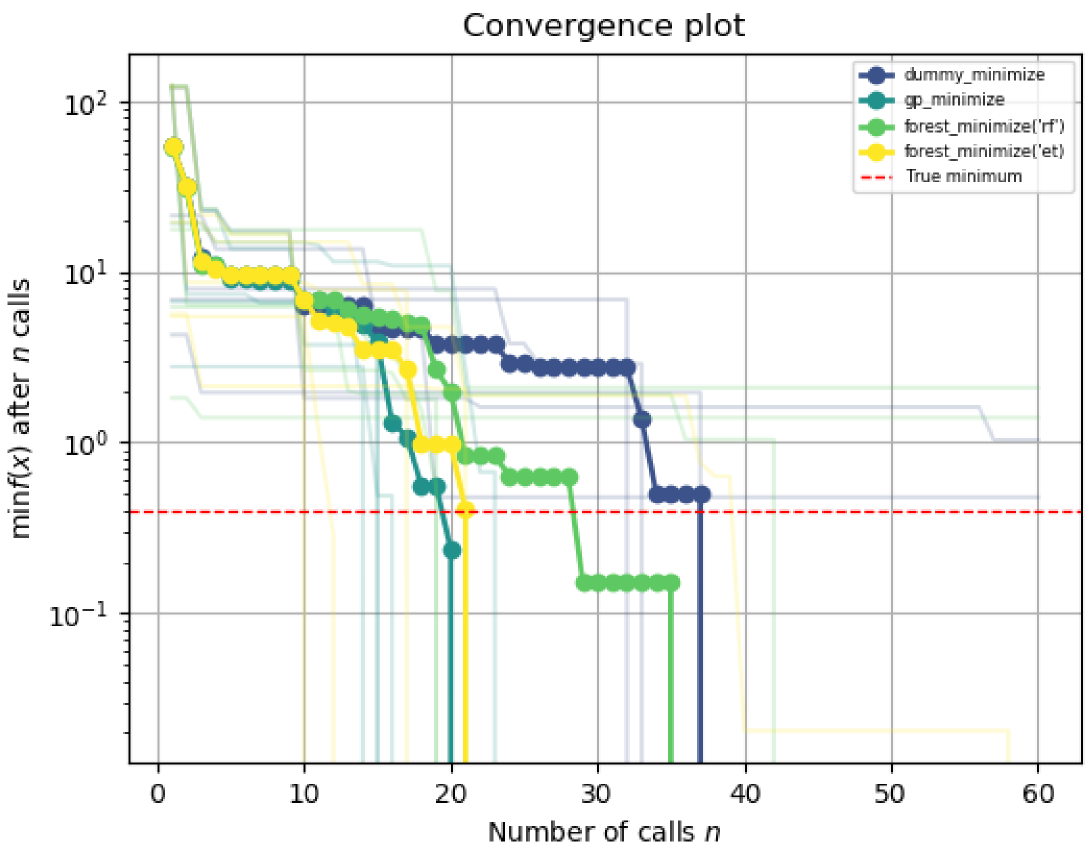
We will evaluate each model several times using a different seed for the random number generator. Then compare the average performance of these models. This makes the comparison more robust against models that get "lucky".

```
from functools import partial
from skopt import gp_minimize, forest_minimize, dummy_minimize

func = 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.

```
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)
```



Out:

```
<matplotlib.legend.Legend object at 0x7f8322d9f2e0>
```

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 22.036 seconds)

Estimated memory usage: 65 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.5 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

```
print(__doc__)
import sys
import numpy as np
np.random.seed(777)
import os

# The followings are hacks to allow sphinx-gallery to run the example.
sys.path.insert(0, os.getcwd())
main_dir = os.path.basename(sys.modules['__main__'].__file__)
IS_RUN_WITH_SPHINX_GALLERY = main_dir != os.getcwd()
```

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

if IS_RUN_WITH_SPHINX_GALLERY:
    # When this example is run with sphinx gallery, it breaks the pickling
    # capacity for multiprocessing backend so we have to modify the way we
    # define our functions. This has nothing to do with the example.
    from utils import obj_fun
else:
    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,                   # the number of evaluations of f␣
↪including at x0
              n_random_starts=0,           # the number of random initialization␣
↪points
```

(continues on next page)

(continued from previous page)

```
              callback=[checkpoint_saver], # a list of callbacks including the␣
↪checkpoint saver
              random_state=777);
```

Out:

```
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "

         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(alpha=1e-10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
```

(continues on next page)

(continued from previous page)

```
                          random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735)]
 random_state: RandomState(MT19937) at 0x7F8322CE7B40
        space: Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize
↪')])
        specs: {'args': {'func': <function obj_fun at 0x7f8320d43d30>, 'dimensions':␣
↪Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize')]), 'base_
↪estimator': GaussianProcessRegressor(alpha=1e-10, copy_X_train=True,
                          kernel=1**2 * Matern(length_scale=1, nu=2.5),
                          n_restarts_optimizer=2, noise='gaussian',
                          normalize_y=True, optimizer='fmin_l_bfgs_b',
                          random_state=655685735), 'n_calls': 10, 'n_random_starts': 0,
↪ 'acq_func': 'LCB', 'acq_optimizer': 'auto', 'x0': [-20.0], 'y0': None, 'random_
↪state': RandomState(MT19937) at 0x7F8322CE7B40, 'verbose': False, 'callback': [
↪<skopt.callbacks.CheckpointSaver object at 0x7f831a509100>], 'n_points': 10000, 'n_
↪restarts_optimizer': 5, 'xi': 0.01, 'kappa': 1.96, 'n_jobs': 1, 'model_queue_size':␣
↪None}, 'function': 'base_minimize'}
            x: [20.0]
      x_iters: [[-20.0], [20.0], [20.0], [-20.0], [-20.0], [20.0], [-20.0], [20.0],␣
↪[20.0], [20.0]]
```

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)

```
from skopt import load

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

res.fun
```

Out:

```
-0.17524445239614728
```

Continue the search

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

```
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,           # the number of evaluations of f including at x0
            n_random_starts=0,    # the number of random initialization points
            callback=[checkpoint_saver],
            random_state=777);
```

Out:

```
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
```

(continues on next page)

(continued from previous page)

```
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "
/home/circleci/project/skopt/optimizer/optimizer.py:409: UserWarning: The objective␣
↪has been evaluated at this point before.
  warnings.warn("The objective has been evaluated "

          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, -0.12332314, -0.04395035,  0.09147873,  0.02650409])
       models: [GaussianProcessRegressor(alpha=1e-10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                         kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                         n_restarts_optimizer=2, noise='gaussian',
                         normalize_y=True, optimizer='fmin_l_bfgs_b',
                         random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
```

(continues on next page)

(continued from previous page)

```
                        kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                        n_restarts_optimizer=2, noise='gaussian',
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                        kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                        n_restarts_optimizer=2, noise='gaussian',
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                        kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                        n_restarts_optimizer=2, noise='gaussian',
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                        kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                        n_restarts_optimizer=2, noise='gaussian',
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=655685735), GaussianProcessRegressor(alpha=1e-
↪10, copy_X_train=True,
                        kernel=1**2 * Matern(length_scale=1, nu=2.5) +␣
↪WhiteKernel(noise_level=1),
                        n_restarts_optimizer=2, noise='gaussian',
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=655685735)]
random_state: RandomState(MT19937) at 0x7F8322CE7B40
       space: Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize
↪')])
       specs: {'args': {'func': <function obj_fun at 0x7f8320d43d30>, 'dimensions':␣
↪Space([Real(low=-20.0, high=20.0, prior='uniform', transform='normalize')]), 'base_
↪estimator': GaussianProcessRegressor(alpha=1e-10, copy_X_train=True,
                        kernel=1**2 * Matern(length_scale=1, nu=2.5),
                        n_restarts_optimizer=2, noise='gaussian',
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=655685735), 'n_calls': 10, 'n_random_starts': 0,
↪ 'acq_func': 'LCB', 'acq_optimizer': 'auto', 'x0': [[-20.0], [20.0], [20.0], [-20.
↪0], [-20.0], [20.0], [-20.0], [20.0], [20.0], [20.0]], '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 0x7F8322CE7B40, 'verbose': False, 'callback': [
↪<skopt.callbacks.CheckpointSaver object at 0x7f831a509100>], 'n_points': 10000, 'n_
↪restarts_optimizer': 5, 'xi': 0.01, 'kappa': 1.96, 'n_jobs': 1, 'model_queue_size':␣
↪None}, 'function': 'base_minimize'}
           x: [20.0]
     x_iters: [[-20.0], [20.0], [20.0], [-20.0], [-20.0], [20.0], [-20.0], [20.0],␣
↪[20.0], [20.0], [20.0], [20.0], [-20.0], [-20.0], [-20.0], [-20.0], [-20.0], [-20.
↪0], [-20.0], [-20.0]]
```

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 3.680 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.6 Async optimization Loop

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 a 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.

```
print(__doc__)

import numpy as np
np.random.seed(1234)

import matplotlib.pyplot as plt
```

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.

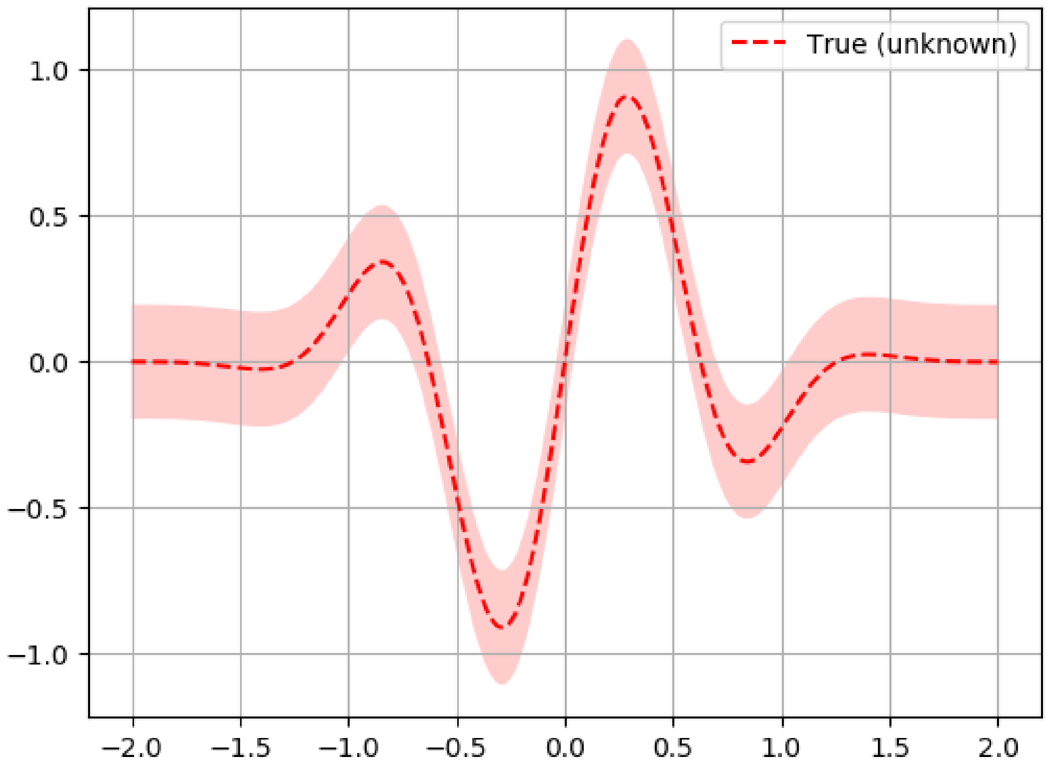
```
from skopt.learning import ExtraTreesRegressor
from skopt import Optimizer

noise_level = 0.1
```

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

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

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

```
# 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.

```
opt = Optimizer([(-2.0, 2.0)], "ET", acq_optimizer="sampling")

# To obtain a suggestion for the point at which to evaluate the objective
```

(continues on next page)

(continued from previous page)

```
# you call the ask() method of opt:

next_x = opt.ask()
print(next_x)
```

Out:

```
[-1.7121321838148869]
```

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:

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

Out:

```
         fun: -0.032758350111535384
   func_vals: array([-0.03275835])
      models: []
random_state: RandomState(MT19937) at 0x7F833EAFAD40
       space: Space([Real(low=-2.0, high=2.0, prior='uniform', transform='identity')])
       specs: None
           x: [-1.7121321838148869]
     x_iters: [[-1.7121321838148869]]
```

Like ***_minimize()** the first few points are random suggestions as there is no data yet with which to fit a surrogate model.

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

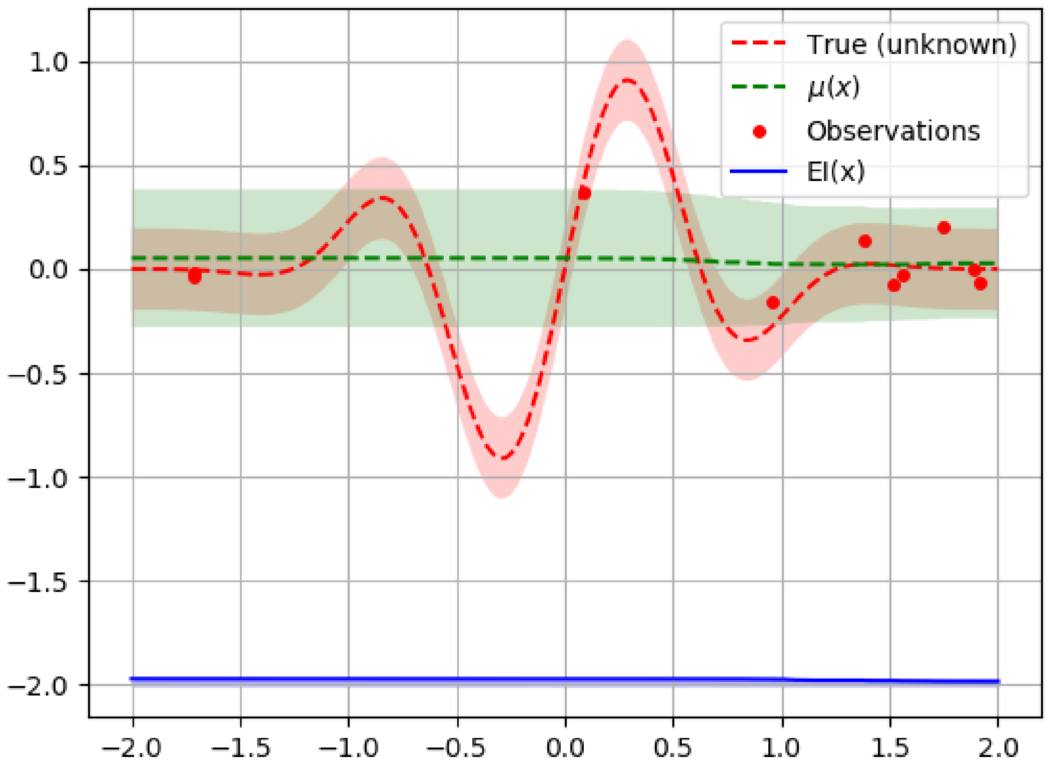
We can now plot the random suggestions and the first model that has been fit:

```
from skopt.acquisition import gaussian_ei


def plot_optimizer(opt, x, fx):
    model = opt.models[-1]
    x_model = opt.space.transform(x.tolist())

    # Plot true function.
    plt.plot(x, fx, "r--", label="True (unknown)")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([fx - 1.9600 * noise_level,
                             fx[::-1] + 1.9600 * noise_level]),
             alpha=.2, fc="r", ec="None")

    # Plot Model(x) + contours
    y_pred, sigma = model.predict(x_model, return_std=True)
    plt.plot(x, y_pred, "g--", label=r"$\mu(x)$")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([y_pred - 1.9600 * sigma,
```

(continues on next page)

(continued from previous page)

```
                                (y_pred + 1.9600 * sigma)[::-1]]),
             alpha=.2, fc="g", ec="None")

    # Plot sampled points
    plt.plot(opt.Xi, opt.yi,
             "r.", markersize=8, label="Observations")

    acq = gaussian_ei(x_model, model, y_opt=np.min(opt.yi))
    # shift down to make a better plot
    acq = 4 * acq - 2
    plt.plot(x, acq, "b", label="EI(x)")
    plt.fill_between(x.ravel(), -2.0, acq.ravel(), alpha=0.3, color='blue')

    # Adjust plot layout
    plt.grid()
    plt.legend(loc='best')


plot_optimizer(opt, x, fx)
```



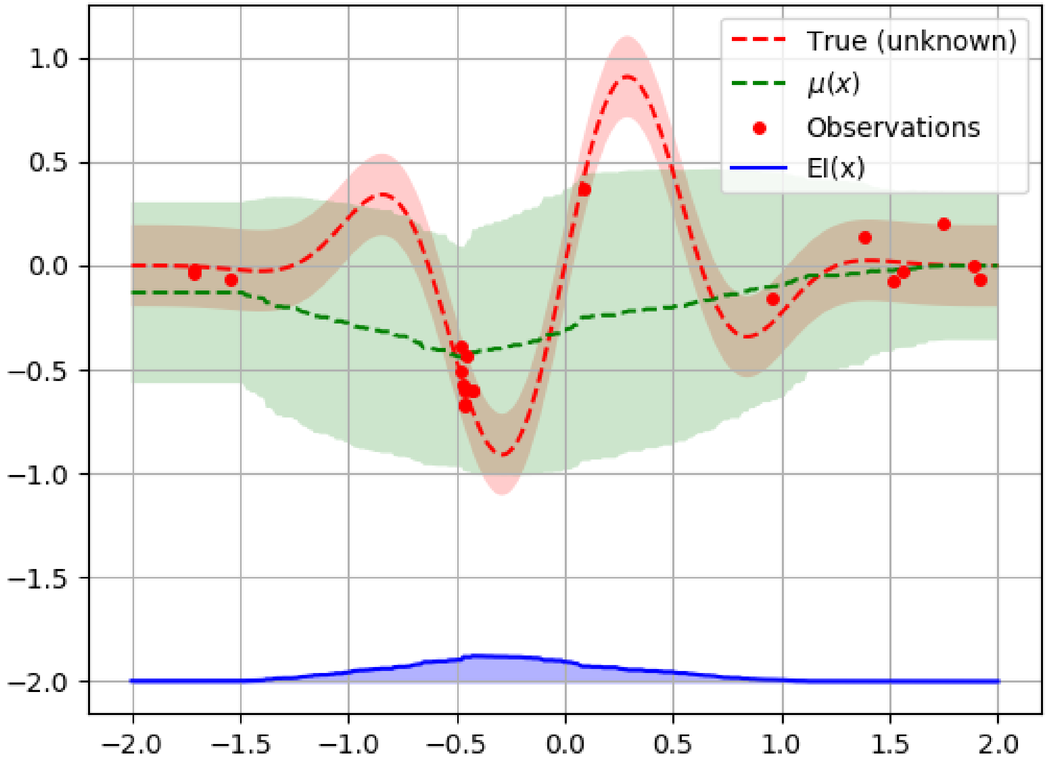
Let us sample a few more points and plot the optimizer again:

```
for i in range(10):
    next_x = opt.ask()
```

(continues on next page)

(continued from previous page)

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

plot_optimizer(opt, x, fx)
```



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:

```
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 5.175 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.7 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
```

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(10, 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)

print("val. score: %s" % opt.best_score_)
print("test score: %s" % opt.score(X_test, y_test))
```

Out:

```
val. score: 0.991833704528582
test score: 0.9933333333333333
```

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.

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

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(10, 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']),
}

opt = BayesSearchCV(
    pipe,
    [(svc_search, 20), (linsvc_search, 16)], # (parameter space, # of evaluations)
    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))
```

Out:

```
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
/home/circleci/miniconda/envs/testenv/lib/python3.8/site-packages/scikit_learn-0.22.1-
↪py3.8-linux-x86_64.egg/sklearn/svm/_base.py:946: ConvergenceWarning: Liblinear␣
↪failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
val. score: 0.9851521900519673
test score: 0.9822222222222222
```

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.

```
from skopt import BayesSearchCV

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

X, y = load_iris(True)

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


# callback handler
def on_step(optim_result):
    score = searchcv.best_score_
    print("best score: %s" % score)
    if score >= 0.98:
        print('Interrupting!')
        return True


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

Out:

```
best score: 0.9466666666666667
best score: 0.9733333333333334
best score: 0.9733333333333334
best score: 0.9733333333333334
best score: 0.9733333333333334
best score: 0.9733333333333334
best score: 0.98
Interrupting!

BayesSearchCV(cv=3, error_score='raise',
              estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                            class_weight=None, coef0=0.0,
                            decision_function_shape='ovr', degree=3,
                            gamma='scale', kernel='rbf', max_iter=-1,
                            probability=False, random_state=None,
                            shrinking=True, tol=0.001, verbose=False),
              fit_params=None, iid=True, n_iter=10, n_jobs=1, n_points=1,
              optimizer_kwargs=None, pre_dispatch='2*n_jobs', random_state=None,
              refit=True, return_train_score=False, scoring=None,
              search_spaces={'C': (0.01, 100.0, 'log-uniform')}, verbose=0)
```

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(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: (0 minutes 50.878 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.8 Exploration vs exploitation

Sigurd Carlen, September 2019. Reformatted by Holger Nahrstaedt 2020

We can control how much the acqusition 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 acqusition 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 aqcuisittion function.

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

```
print(__doc__)

import numpy as np
np.random.seed(1234)
import matplotlib.pyplot as plt
```

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 afterthe first one is therefore choosen by the acquisition function.

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

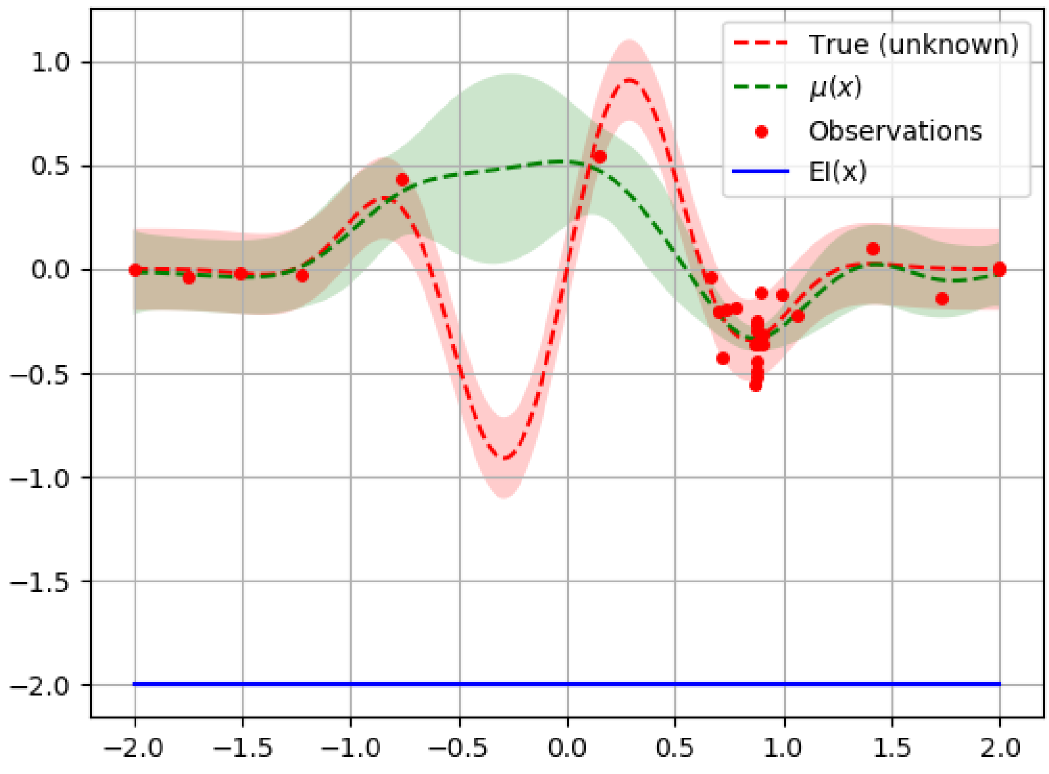
```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points = 1,
                acq_optimizer="sampling")
```

```
x = np.linspace(-2, 2, 400).reshape(-1, 1)
fx = np.array([objective(x_i, noise_level=0.0) for x_i in x])
```

```
from skopt.acquisition import gaussian_ei
def plot_optimizer(opt, x, fx):
    model = opt.models[-1]
    x_model = opt.space.transform(x.tolist())

    # Plot true function.
    plt.plot(x, fx, "r--", label="True (unknown)")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([fx - 1.9600 * noise_level,
                             fx[::-1] + 1.9600 * noise_level]),
             alpha=.2, fc="r", ec="None")

    # Plot Model(x) + contours
    y_pred, sigma = model.predict(x_model, return_std=True)
    plt.plot(x, y_pred, "g--", label=r"$\mu(x)$")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([y_pred - 1.9600 * sigma,
                             (y_pred + 1.9600 * sigma)[::-1]]),
             alpha=.2, fc="g", ec="None")

    # Plot sampled points
    plt.plot(opt.Xi, opt.yi,
             "r.", markersize=8, label="Observations")

    acq = gaussian_ei(x_model, model, y_opt=np.min(opt.yi))
    # shift down to make a better plot
    acq = 4 * acq - 2
    plt.plot(x, acq, "b", label="EI(x)")
    plt.fill_between(x.ravel(), -2.0, acq.ravel(), alpha=0.3, color='blue')

    # Adjust plot layout
    plt.grid()
    plt.legend(loc='best')
```

We run a an optimization loop with standard settings

```
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_optimizer(opt, x, fx)
```



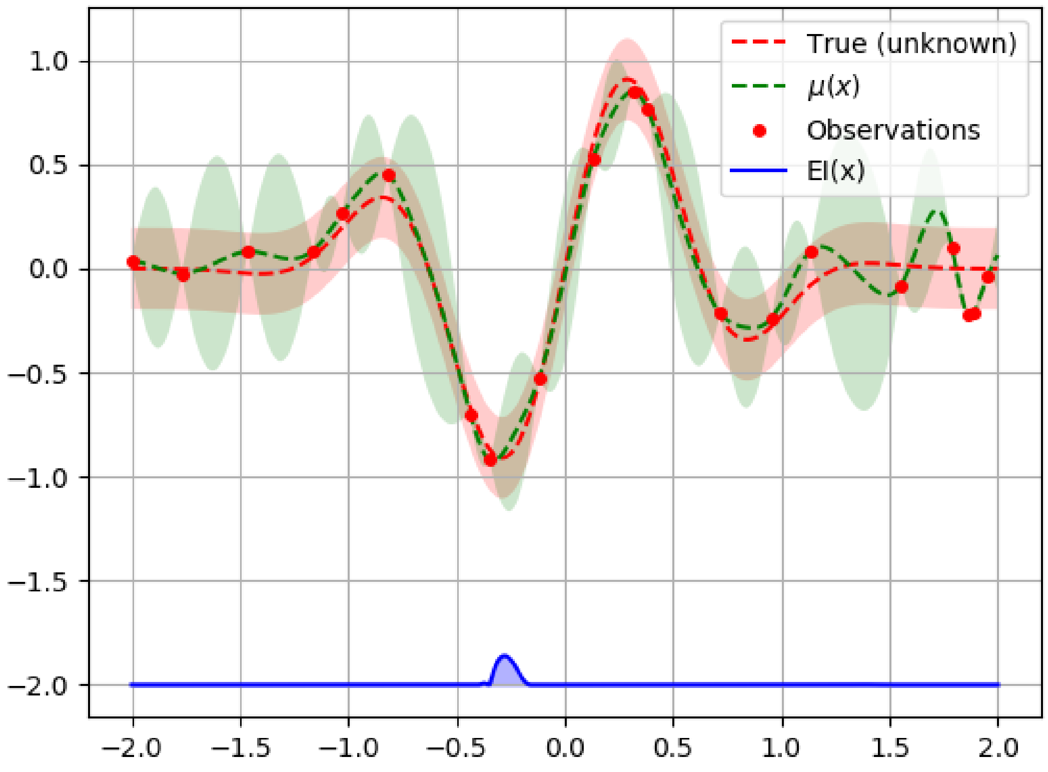
We see that some minima is found and "exploited"

Now lets try to set kappa and xi using'to other values and pass it to the optimizer:

```
acq_func_kwargs = {"xi": 10000, "kappa": 10000}
```

```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```

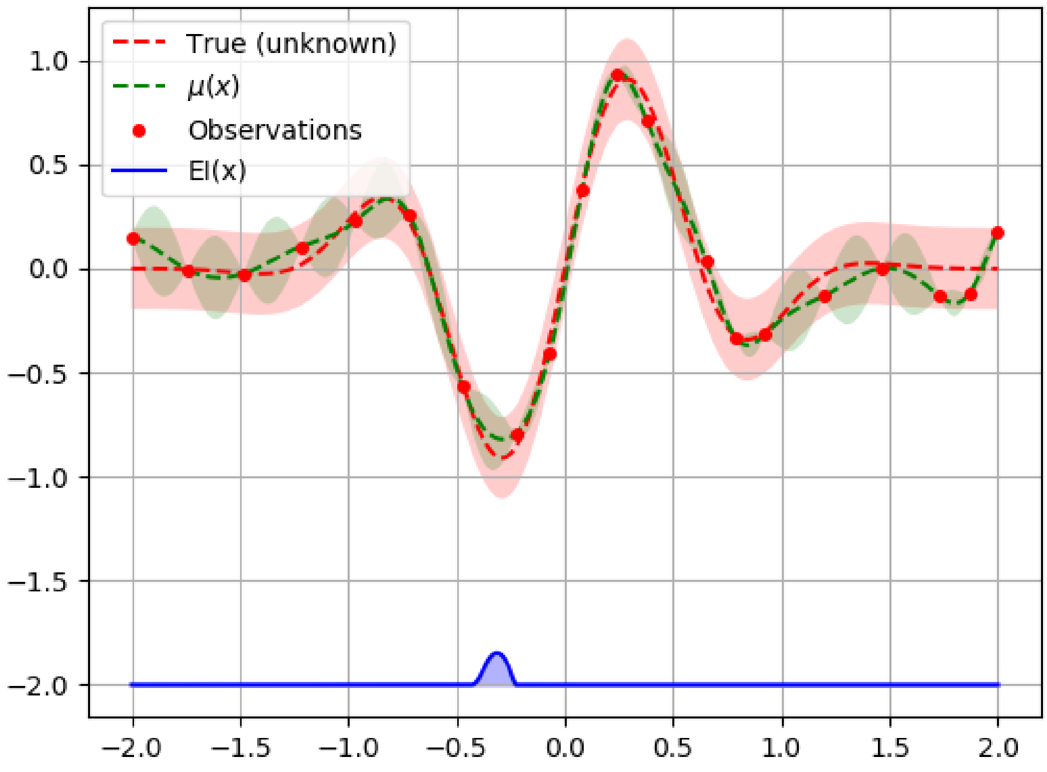


We see that the points are more random now.

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

```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="LCB", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

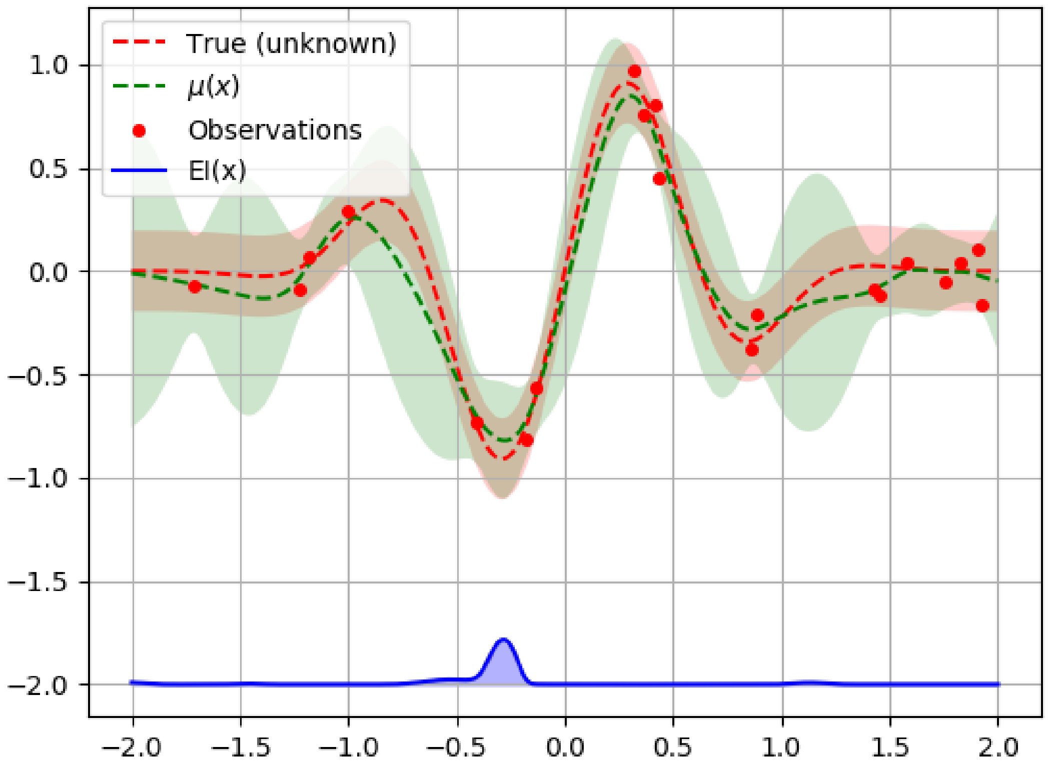
```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



And for xi when using acq_func="EI": or acq_func="PI":

```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="PI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```

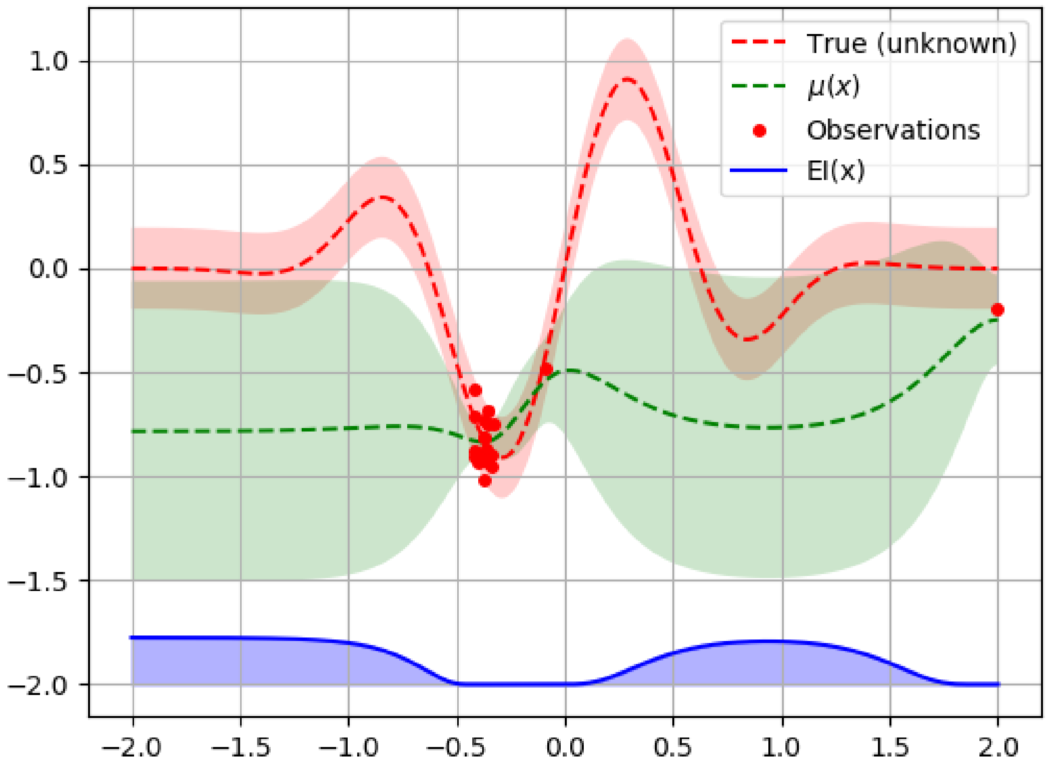


We can also favor exploitaton:

```
acq_func_kwargs = {"xi": 0.000001, "kappa": 0.001}
```

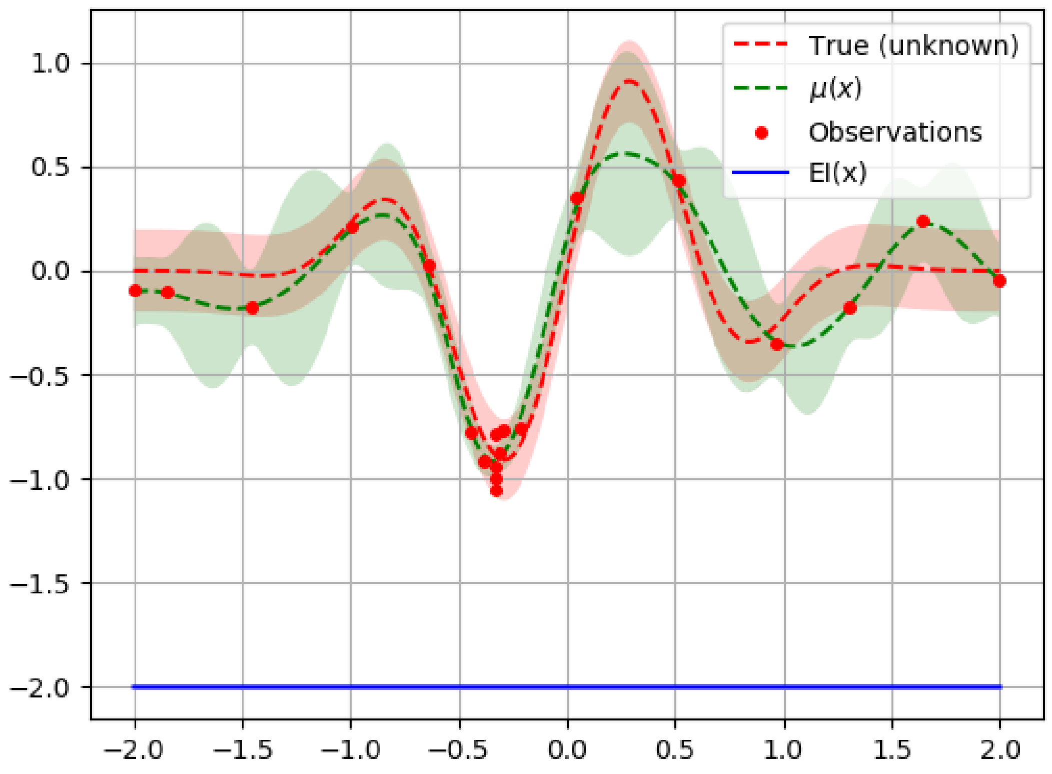
```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="LCB", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



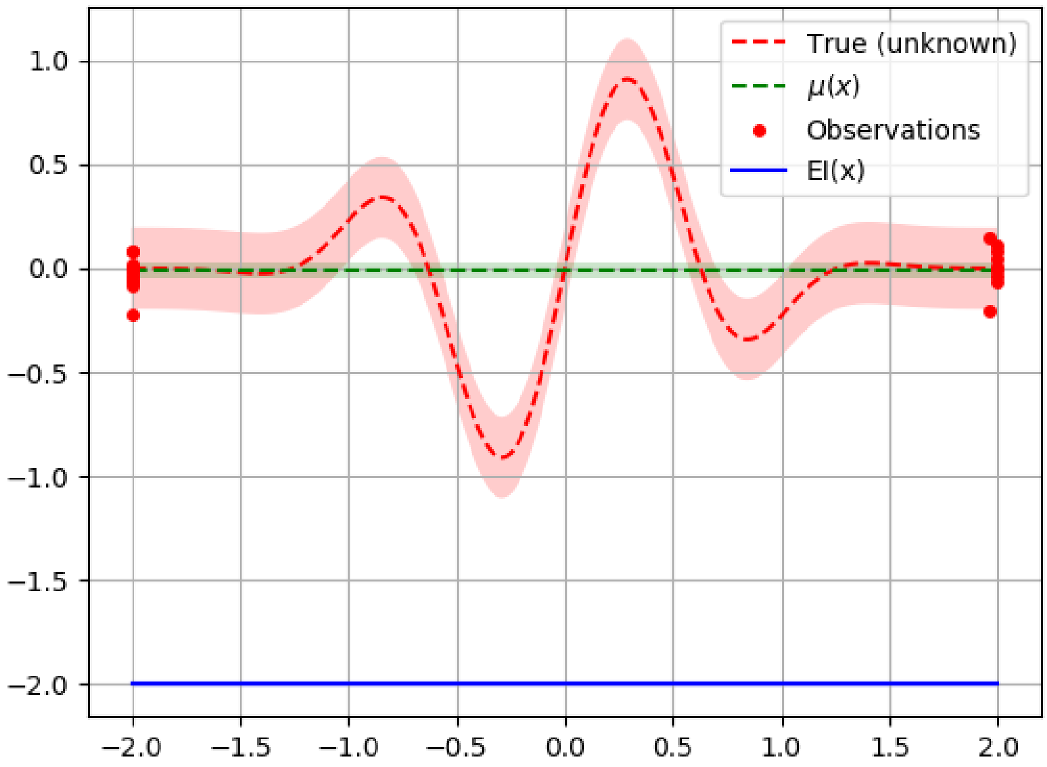
```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="EI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="PI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```

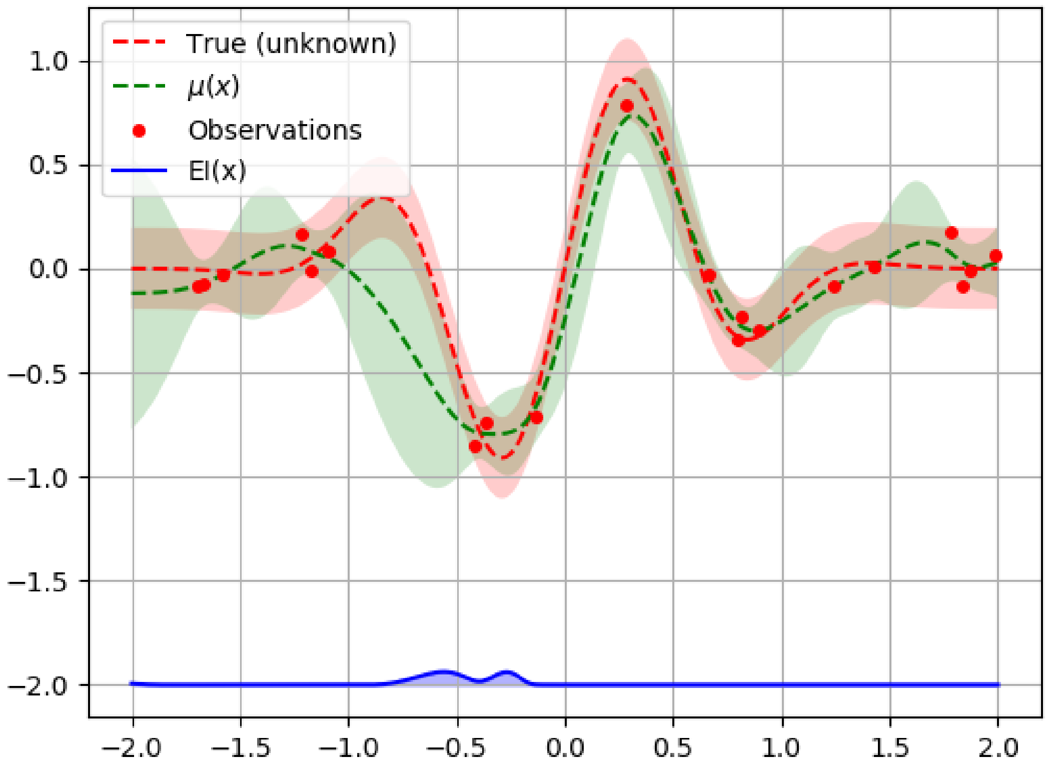


Note that negative values does not work with the "PI"-acquisition function but works with "EI":

```
acq_func_kwargs = {"xi": -1000000000000}
```

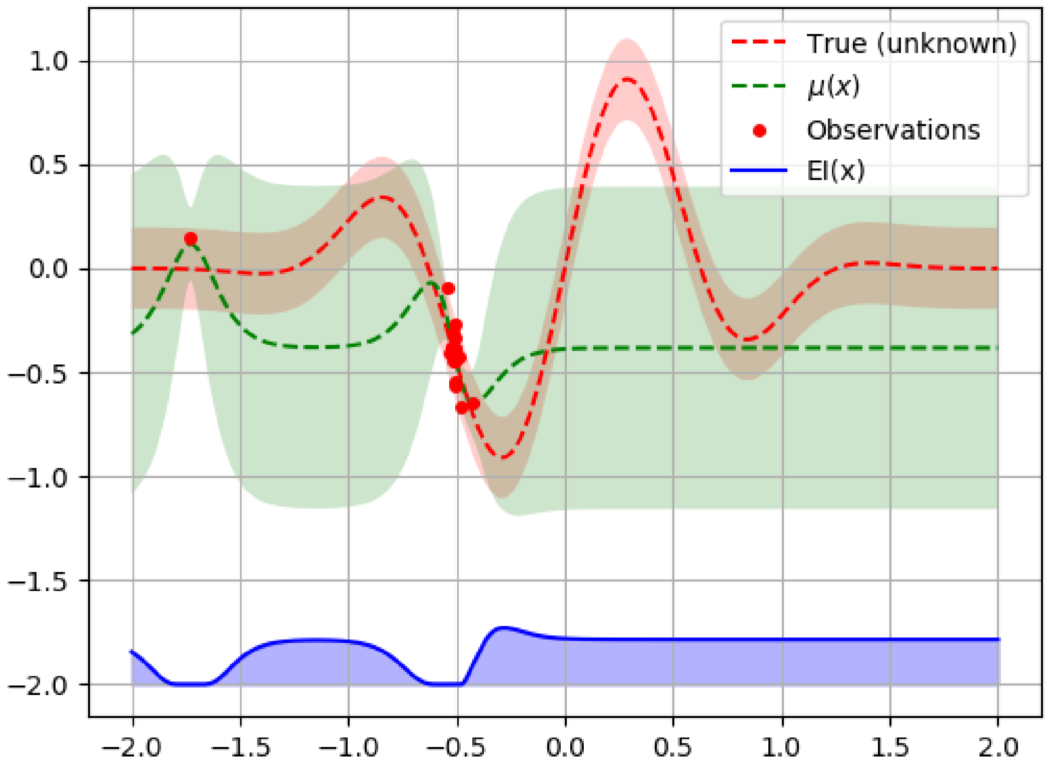
```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="PI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="EI", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



Changing kappa and xi on the go

If we want to change kappa or ki 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.

```
acq_func_kwargs = {"kappa": 0}
```

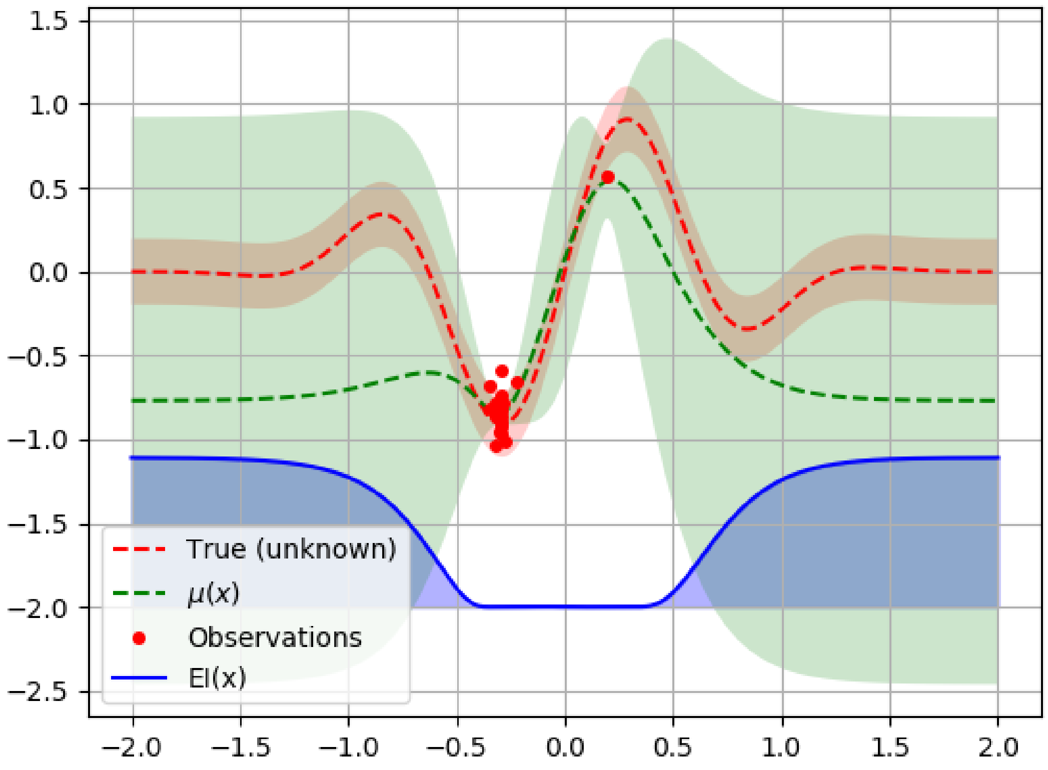
```
opt = Optimizer([(-2.0, 2.0)], "GP", n_initial_points=1,
                acq_func="LCB", acq_optimizer="sampling",
                acq_func_kwargs=acq_func_kwargs)
```

```
opt.acq_func_kwargs
```

Out:

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

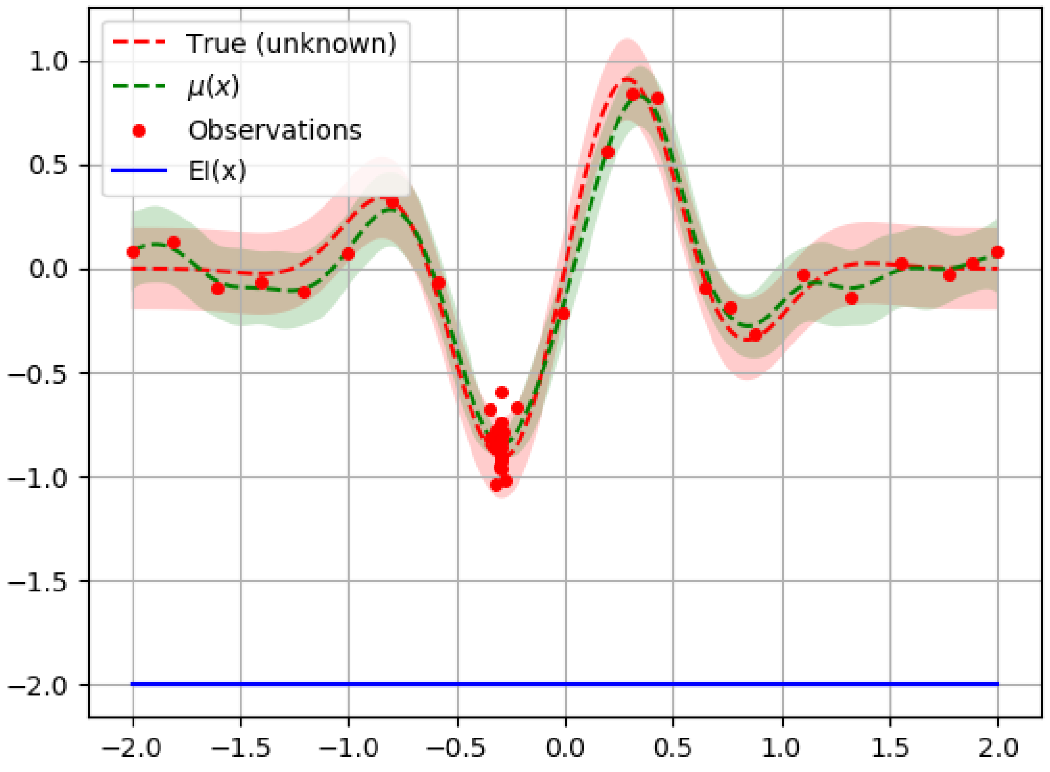
```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



```
acq_func_kwargs = {"kappa": 100000}
```

```
opt.acq_func_kwargs = acq_func_kwargs
opt.update_next()
```

```
opt.run(objective, n_iter=20)
plot_optimizer(opt, x, fx)
```



Total running time of the script: (0 minutes 34.924 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.1.9 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.

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.

$$x_{t+1} = arg \min_x u(x)$$

   Exploit 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):** $-EI(x) = -\mathbb{E}[f(x) - f(x_t^+)]$
- 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., κ) 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)

```
print(__doc__)

import numpy as np
np.random.seed(237)
import matplotlib.pyplot as plt
```

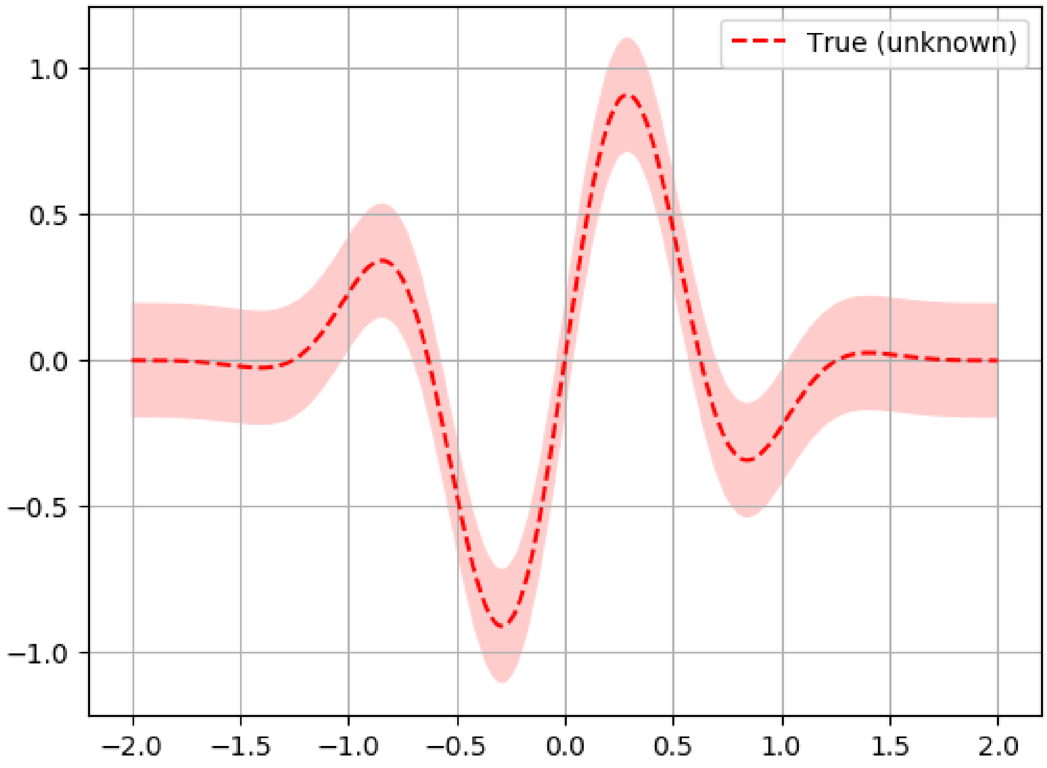
Toy example

Let assume the following noisy function f:

```
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 $f(x)$:.

```
# 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:

```
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^*=%.4f, f(x^*)=%.4f" % (res.x[0], res.fun)
```

Out:

```
'x^*=-0.3508, f(x^*)=-1.0147'
```

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: -1.0146594081392317
    func_vals: array([ 0.03716044,  0.00673852,  0.63515442, -0.16042062,  0.10695907,
       -0.23193728, -0.60259431, -0.04943778, -1.01465941, -0.98480886,
       -0.87449015,  0.18102445, -0.10782771,  0.01197229, -0.80618926])
       models: [GaussianProcessRegressor(alpha=1e-10, copy_X_train=True,
                         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, optimizer='fmin_l_bfgs_b',
                         random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                         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, optimizer='fmin_l_bfgs_b',
                         random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                         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, optimizer='fmin_l_bfgs_b',
                         random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                         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, optimizer='fmin_l_bfgs_b',
                         random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                         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, optimizer='fmin_l_bfgs_b',
                         random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
```

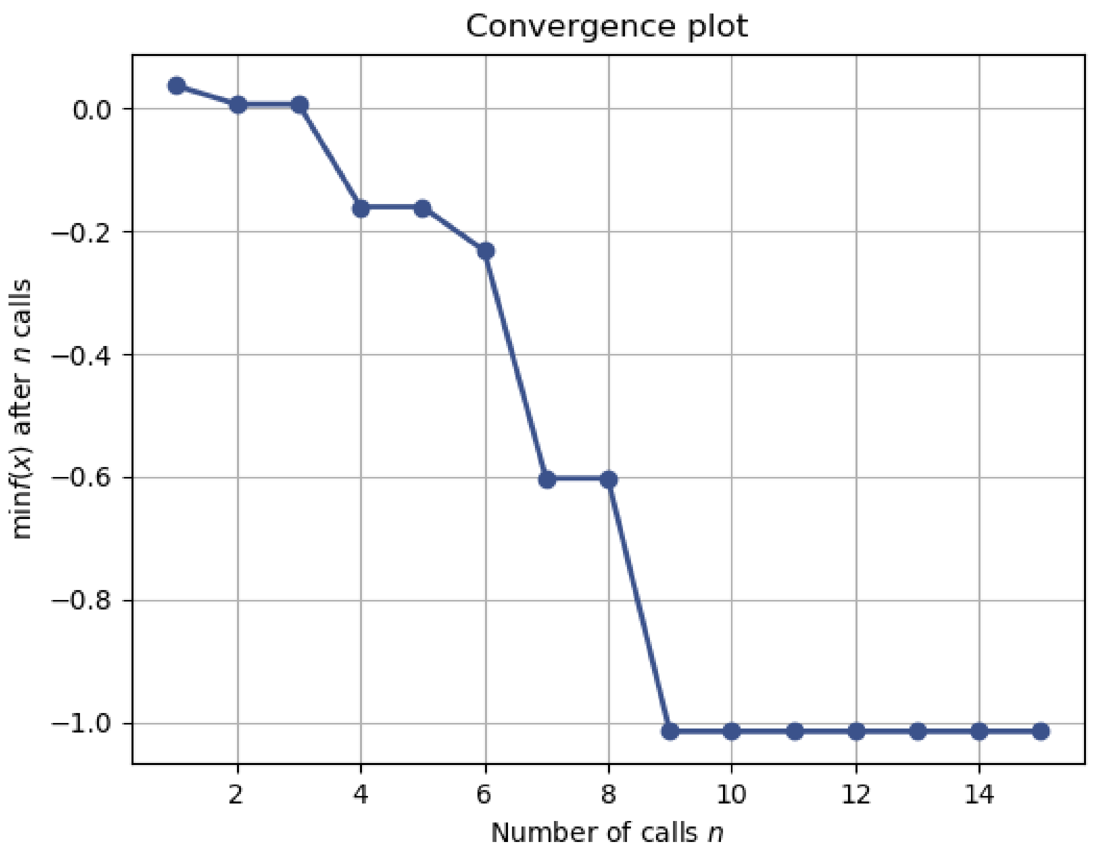
(continues on next page)

(continued from previous page)

```
                        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, optimizer='fmin_l_bfgs_b',
                        random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                        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, optimizer='fmin_l_bfgs_b',
                        random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                        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, optimizer='fmin_l_bfgs_b',
                        random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                        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, optimizer='fmin_l_bfgs_b',
                        random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                        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, optimizer='fmin_l_bfgs_b',
                        random_state=822569775), GaussianProcessRegressor(alpha=1e-10,
↪ copy_X_train=True,
                        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, optimizer='fmin_l_bfgs_b',
                        random_state=822569775)]
random_state: RandomState(MT19937) at 0x7F8322CE7B40
       space: Space([Real(low=-2.0, high=2.0, prior='uniform', transform='normalize
↪')])
       specs: {'args': {'func': <function f at 0x7f832819e430>, 'dimensions':␣
↪Space([Real(low=-2.0, high=2.0, prior='uniform', transform='normalize')]), 'base_
↪estimator': GaussianProcessRegressor(alpha=1e-10, copy_X_train=True,
                        kernel=1**2 * Matern(length_scale=1, nu=2.5),
                        n_restarts_optimizer=2, noise=0.010000000000000002,
                        normalize_y=True, optimizer='fmin_l_bfgs_b',
                        random_state=822569775), 'n_calls': 15, 'n_random_starts': 5,
↪'acq_func': 'EI', 'acq_optimizer': 'auto', 'x0': None, 'y0': None, 'random_state':␣
↪RandomState(MT19937) at 0x7F8322CE7B40, 'verbose': False, 'callback': None, 'n_
↪points': 10000, 'n_restarts_optimizer': 5, 'xi': 0.01, 'kappa': 1.96, 'n_jobs': 1,
↪'model_queue_size': None}, 'function': 'base_minimize'}
           x: [-0.35076964188527904]
     x_iters: [[-0.009345334109402526], [1.2713537644662787], [0.4484475787090836],␣
↪[1.0854396754496047], [1.4426790855107496], [0.9698921802985794], [-0.
↪4464493263345517], [-0.6474638284799423], [-0.35076964188527904], [-0.
↪28714767658880325], [-0.2968537755362253], [-2.0], [2.0], [-1.3149517825054502], [-
↪0.32181607448732485]]
```

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:

```
from skopt.plots import plot_convergence
plot_convergence(res);
```



Out:

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7f831a6b7ac0>
```

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.

```
from skopt.acquisition import gaussian_ei

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

x = np.linspace(-2, 2, 400).reshape(-1, 1)
x_gp = res.space.transform(x.tolist())
fx = np.array([f(x_i, noise_level=0.0) for x_i in x])
```

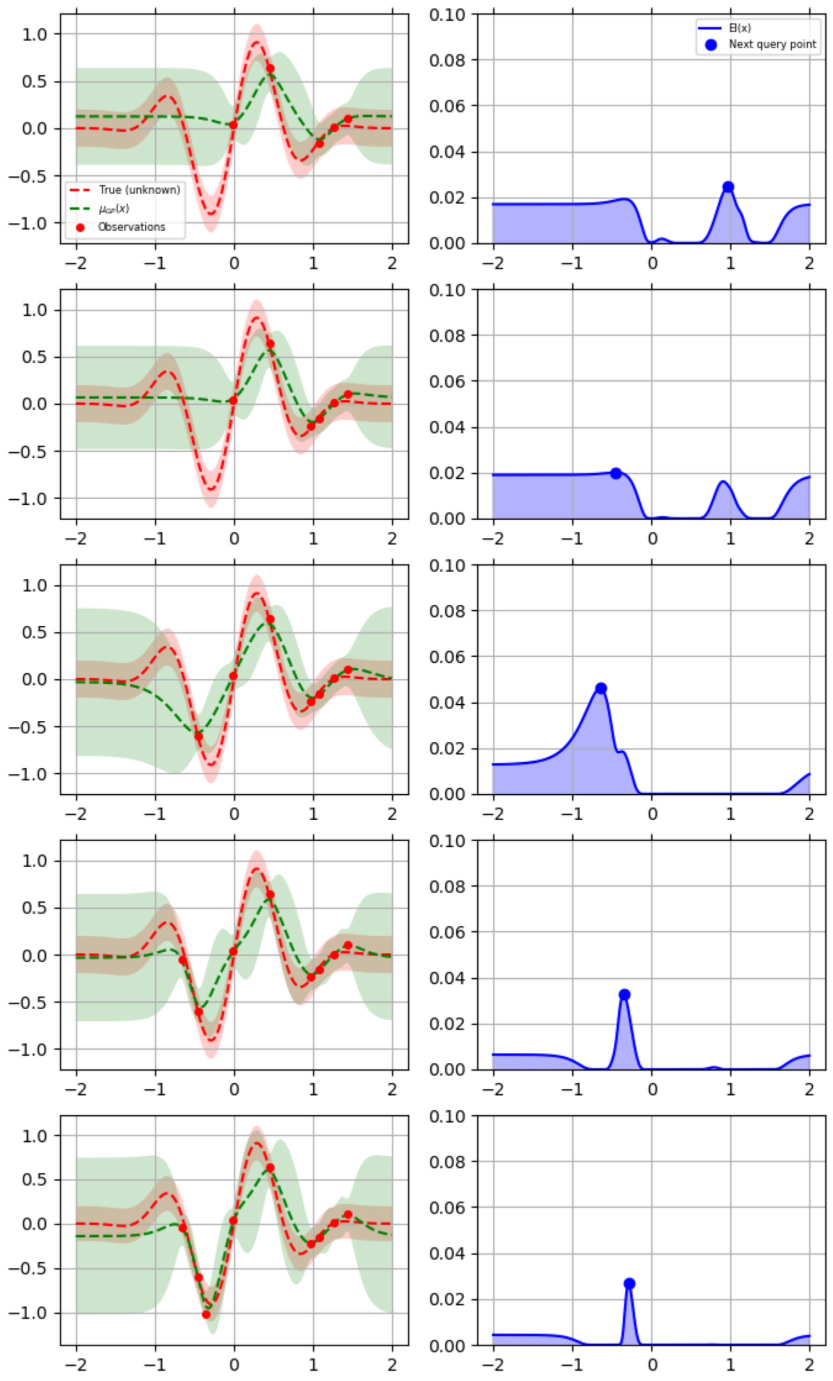
Plot the 5 iterations following the 5 random points

```
for n_iter in range(5):
    gp = res.models[n_iter]
```

(continues on next page)

(continued from previous page)

```
    curr_x_iters = res.x_iters[:5+n_iter]
    curr_func_vals = res.func_vals[:5+n_iter]

    # Plot true function.
    plt.subplot(5, 2, 2*n_iter+1)
    plt.plot(x, fx, "r--", label="True (unknown)")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([fx - 1.9600 * noise_level,
                             fx[::-1] + 1.9600 * noise_level]),
             alpha=.2, fc="r", ec="None")

    # Plot GP(x) + contours
    y_pred, sigma = gp.predict(x_gp, return_std=True)
    plt.plot(x, y_pred, "g--", label=r"$\mu_{GP}(x)$")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([y_pred - 1.9600 * sigma,
                             (y_pred + 1.9600 * sigma)[::-1]]),
             alpha=.2, fc="g", ec="None")

    # Plot sampled points
    plt.plot(curr_x_iters, curr_func_vals,
             "r.", markersize=8, label="Observations")

    # Adjust plot layout
    plt.grid()

    if n_iter == 0:
        plt.legend(loc="best", prop={'size': 6}, numpoints=1)

    if n_iter != 4:
        plt.tick_params(axis='x', which='both', bottom='off',
                        top='off', labelbottom='off')

    # Plot EI(x)
    plt.subplot(5, 2, 2*n_iter+2)
    acq = gaussian_ei(x_gp, gp, y_opt=np.min(curr_func_vals))
    plt.plot(x, acq, "b", label="EI(x)")
    plt.fill_between(x.ravel(), -2.0, acq.ravel(), alpha=0.3, color='blue')

    next_x = res.x_iters[5+n_iter]
    next_acq = gaussian_ei(res.space.transform([next_x]), gp,
                           y_opt=np.min(curr_func_vals))
    plt.plot(next_x, next_acq, "bo", markersize=6, label="Next query point")

    # Adjust plot layout
    plt.ylim(0, 0.1)
    plt.grid()

    if n_iter == 0:
        plt.legend(loc="best", prop={'size': 6}, numpoints=1)

    if n_iter != 4:
        plt.tick_params(axis='x', which='both', bottom='off',
                        top='off', labelbottom='off')

plt.show()
```


True (unknown)
$\mu_{GP}(x)$
Observations
EI(x)
Next query point

The first column shows the following:

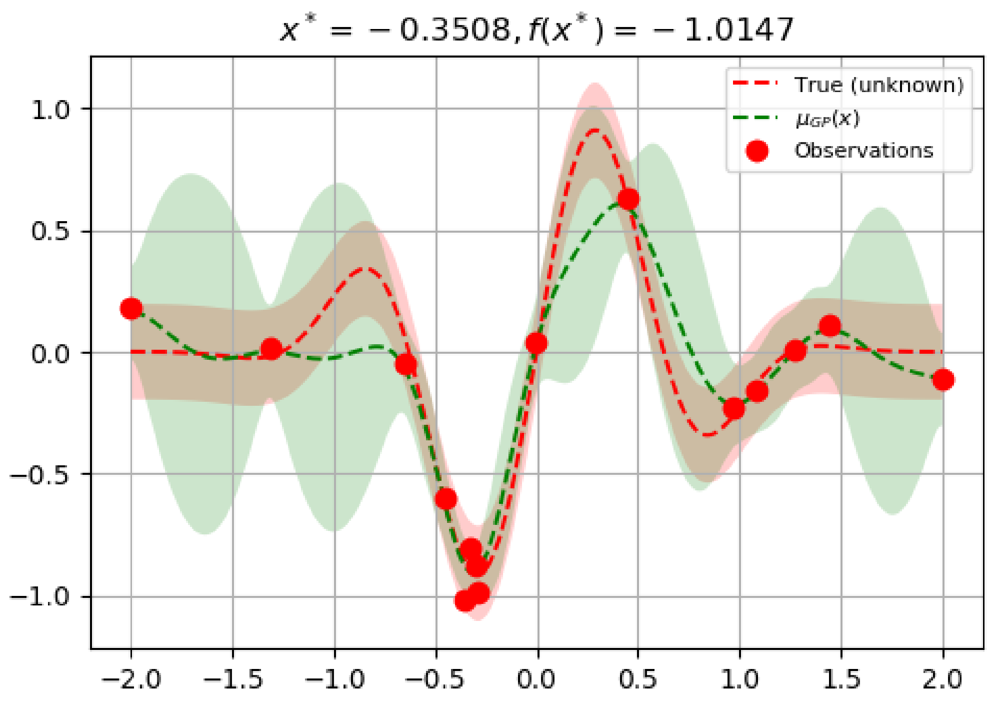
1. The true function.
2. The approximation to the original function by the gaussian process model
3. How sure the GP is about the function.

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:

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

# Plot f(x) + contours
x = np.linspace(-2, 2, 400).reshape(-1, 1)
x_gp = res.space.transform(x.tolist())

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")

# Plot GP(x) + contours
gp = res.models[-1]
y_pred, sigma = gp.predict(x_gp, return_std=True)

plt.plot(x, y_pred, "g--", label=r"$\mu_{GP}(x)$")
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate([y_pred - 1.9600 * sigma,
                         (y_pred + 1.9600 * sigma)[::-1]]),
         alpha=.2, fc="g", ec="None")

# Plot sampled points
plt.plot(res.x_iters,
         res.func_vals,
         "r.", markersize=15, label="Observations")

plt.title(r"$x^* = %.4f, f(x^*) = %.4f$" % (res.x[0], res.fun))
plt.legend(loc="best", prop={'size': 8}, numpoints=1)
plt.grid()

plt.show()
```



Total running time of the script: (0 minutes 3.780 seconds)

Estimated memory usage: 8 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

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.

```
print(__doc__)

import numpy as np
np.random.seed(1234)
import matplotlib.pyplot as plt
```

Toy example

Let assume the following noisy function f:

```
noise_level = 0.1

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

(continues on next page)

(continued from previous page)

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

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

```
x = np.linspace(-2, 2, 400).reshape(-1, 1)
fx = np.array([objective(x_i, noise_level=0.0) for x_i in x])
```

```
from skopt.acquisition import gaussian_ei

def plot_optimizer(res, next_x, x, fx, n_iter, max_iters=5):
    x_gp = res.space.transform(x.tolist())
    gp = res.models[-1]
    curr_x_iters = res.x_iters
    curr_func_vals = res.func_vals

    # Plot true function.
    ax = plt.subplot(max_iters, 2, 2 * n_iter + 1)
    plt.plot(x, fx, "r--", label="True (unknown)")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([fx - 1.9600 * noise_level,
                             fx[::-1] + 1.9600 * noise_level]),
             alpha=.2, fc="r", ec="None")
    if n_iter < max_iters - 1:
        ax.get_xaxis().set_ticklabels([])
    # Plot GP(x) + contours
    y_pred, sigma = gp.predict(x_gp, return_std=True)
    plt.plot(x, y_pred, "g--", label=r"$\mu_{GP}(x)$")
    plt.fill(np.concatenate([x, x[::-1]]),
             np.concatenate([y_pred - 1.9600 * sigma,
                             (y_pred + 1.9600 * sigma)[::-1]]),
             alpha=.2, fc="g", ec="None")

    # Plot sampled points
    plt.plot(curr_x_iters, curr_func_vals,
             "r.", markersize=8, label="Observations")
    plt.title(r"x* = %.4f, f(x*) = %.4f" % (res.x[0], res.fun))
    # Adjust plot layout
    plt.grid()

    if n_iter == 0:
        plt.legend(loc="best", prop={'size': 6}, numpoints=1)

    if n_iter != 4:
        plt.tick_params(axis='x', which='both', bottom='off',
                        top='off', labelbottom='off')

    # Plot EI(x)
    ax = plt.subplot(max_iters, 2, 2 * n_iter + 2)
    acq = gaussian_ei(x_gp, gp, y_opt=np.min(curr_func_vals))
    plt.plot(x, acq, "b", label="EI(x)")
```

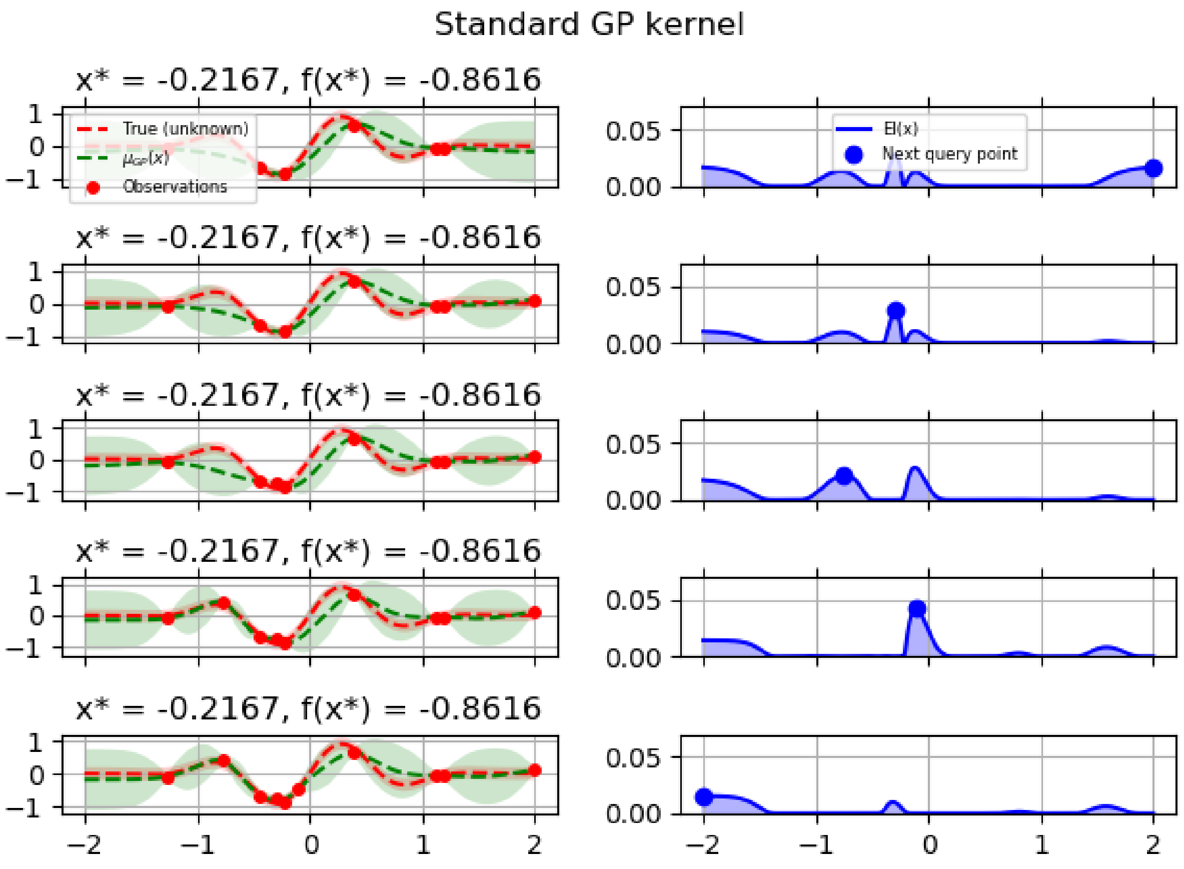
(continues on next page)

(continued from previous page)

```
    plt.fill_between(x.ravel(), -2.0, acq.ravel(), alpha=0.3, color='blue')

    if n_iter < max_iters - 1:
        ax.get_xaxis().set_ticklabels([])

    next_acq = gaussian_ei(res.space.transform([next_x]), gp,
                           y_opt=np.min(curr_func_vals))
    plt.plot(next_x, next_acq, "bo", markersize=6, label="Next query point")

    # Adjust plot layout
    plt.ylim(0, 0.07)
    plt.grid()
    if n_iter == 0:
        plt.legend(loc="best", prop={'size': 6}, numpoints=1)

    if n_iter != 4:
        plt.tick_params(axis='x', which='both', bottom='off',
                        top='off', labelbottom='off')
```

GP kernel

```
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, opt_gp._next_x, x, fx, n_iter=i-5, max_iters=5)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.plot()
```



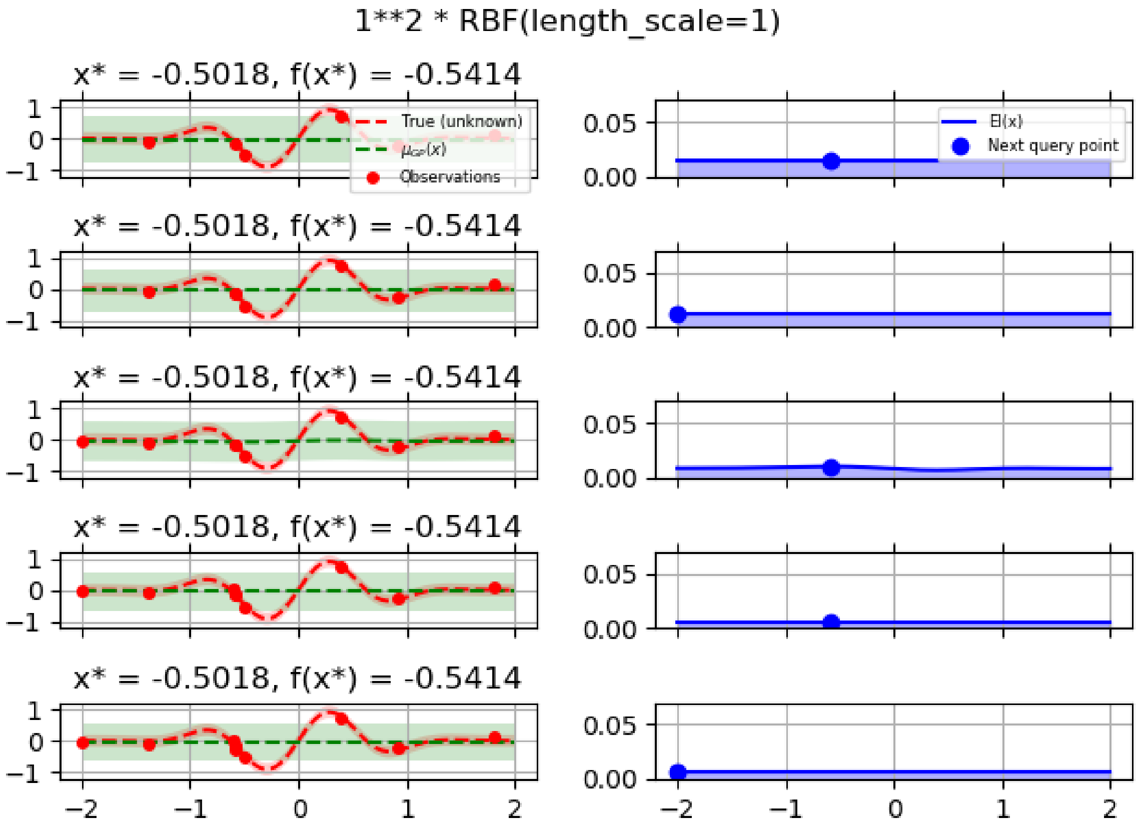
Out:

```
[]
```

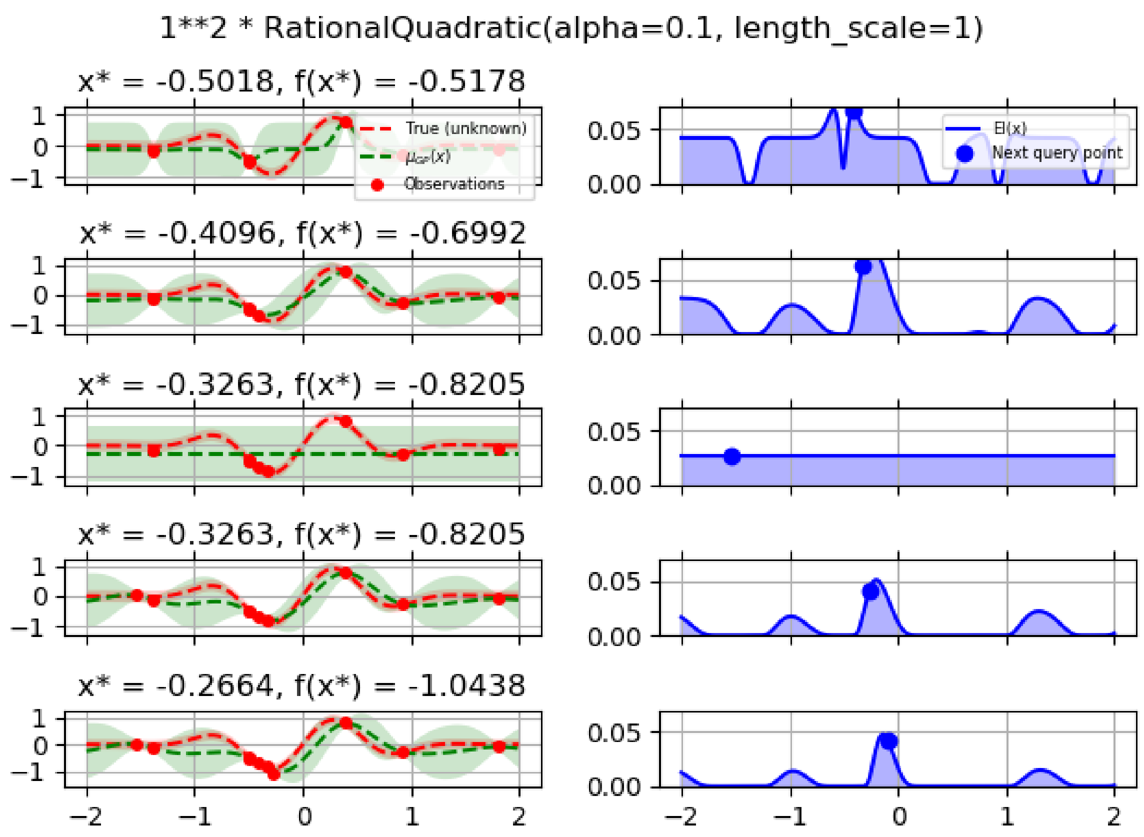
Test different kernels

```
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)]
```

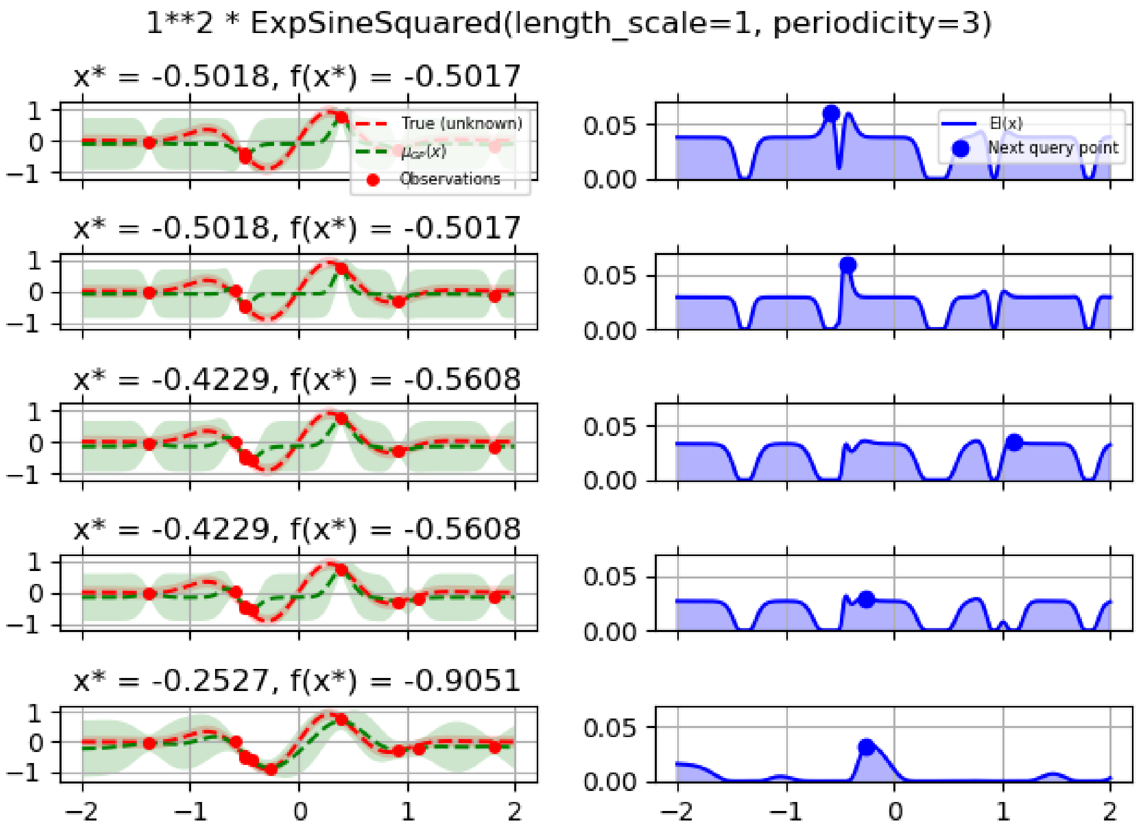
```python
for kernel in kernels:
    gpr = GaussianProcessRegressor(kernel=kernel, alpha=noise_level ** 2,
                                   normalize_y=True, noise="gaussian",
                                   n_restarts_optimizer=2
                                   )
    opt = Optimizer([(-2.0, 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, opt._next_x, x, fx, n_iter=i - 5, max_iters=5)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
```



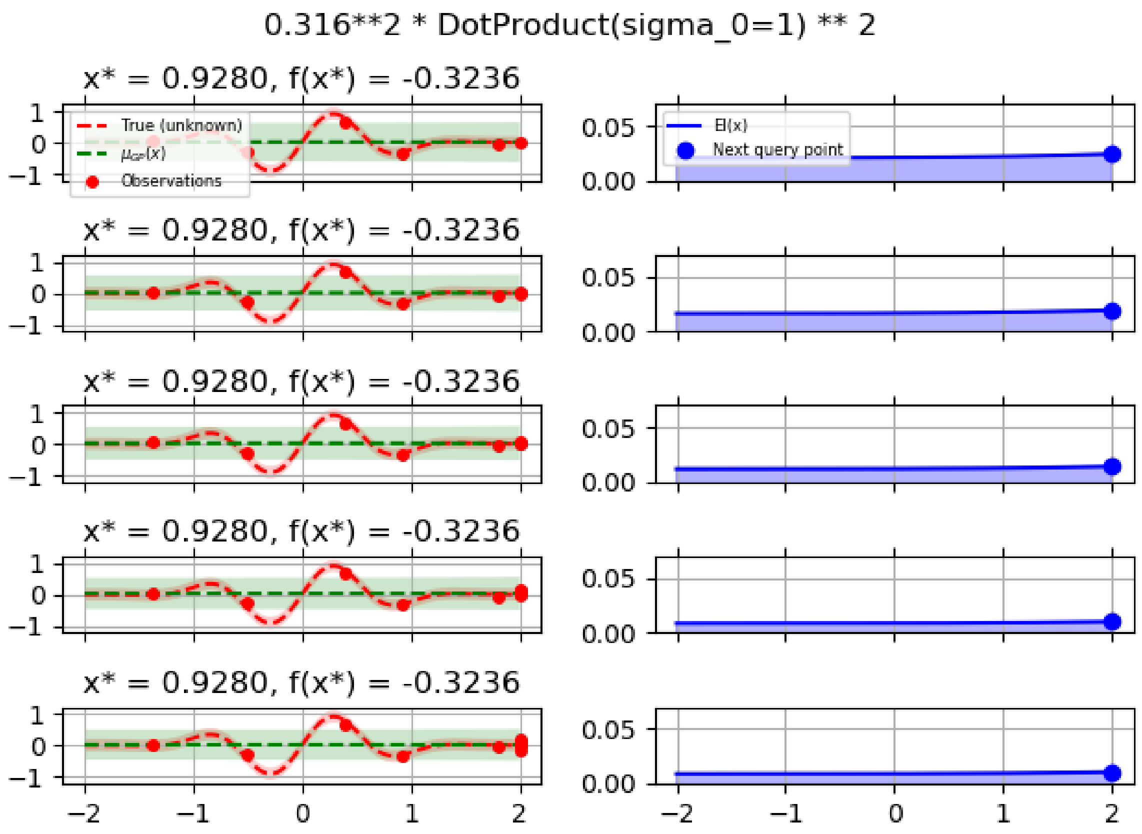
•


1**2 * RationalQuadratic(alpha=0.1, length_scale=1)
x* = -0.5018, f(x*) = -0.5178
x* = -0.4096, f(x*) = -0.6992
x* = -0.3263, f(x*) = -0.8205
x* = -0.3263, f(x*) = -0.8205
x* = -0.2664, f(x*) = -1.0438
True (unknown)
$\mu_{GP}(x)$
Observations
EI(x)
Next query point

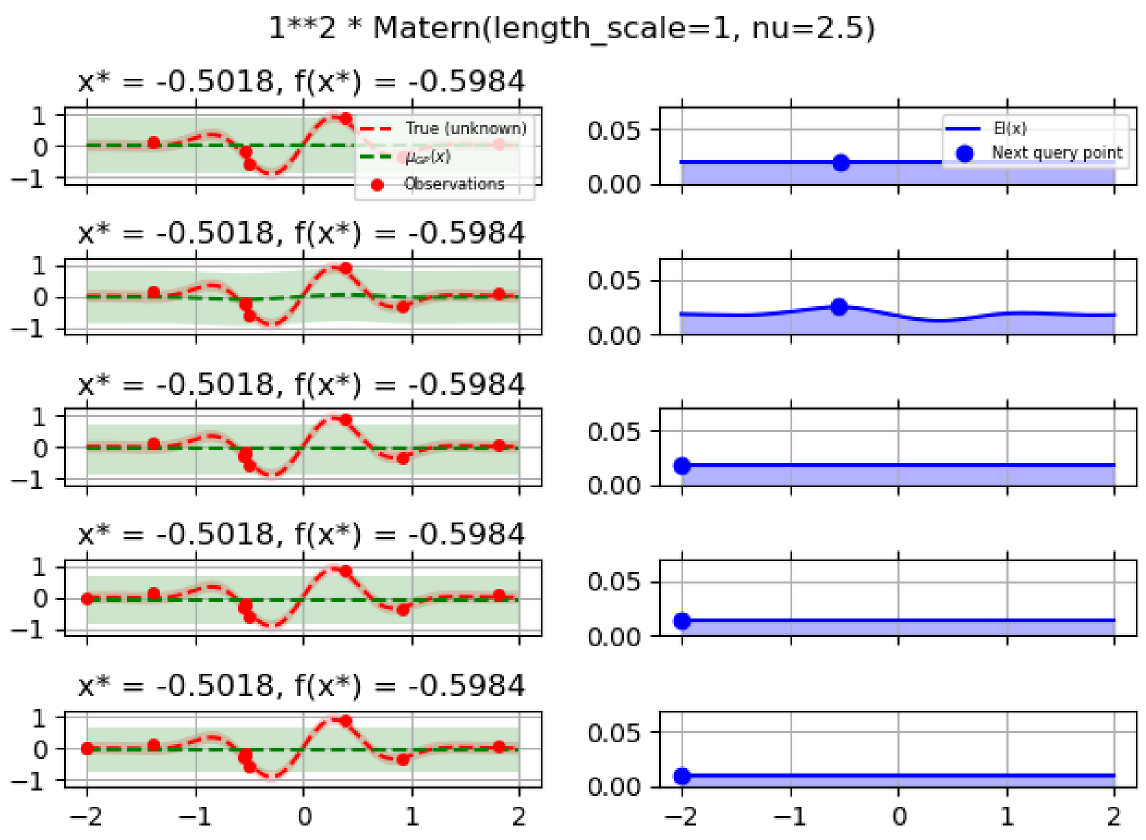
•


1**2 * ExpSineSquared(length_scale=1, periodicity=3)
x* = -0.5018, f(x*) = -0.5017
x* = -0.5018, f(x*) = -0.5017
x* = -0.4229, f(x*) = -0.5608
x* = -0.4229, f(x*) = -0.5608
x* = -0.2527, f(x*) = -0.9051
True (unknown)
$\mu_{GP}(x)$
Observations
EI(x)
Next query point

•


0.316**2 * DotProduct(sigma_0=1) ** 2
x* = 0.9280, f(x*) = -0.3236
True (unknown)
μGP(x)
Observations
EI(x)
Next query point

•



•

Total running time of the script: (0 minutes 9.161 seconds)

Estimated memory usage: 13 MB

4.2 Plotting functions

Examples concerning the *skopt.plots* module.

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.2.1 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.

```
print(__doc__)
import sys
from skopt.plots import plot_objective
from skopt import forest_minimize
import numpy as np
```

(continues on next page)

(continued from previous page)

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

Objective function

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

```
# 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

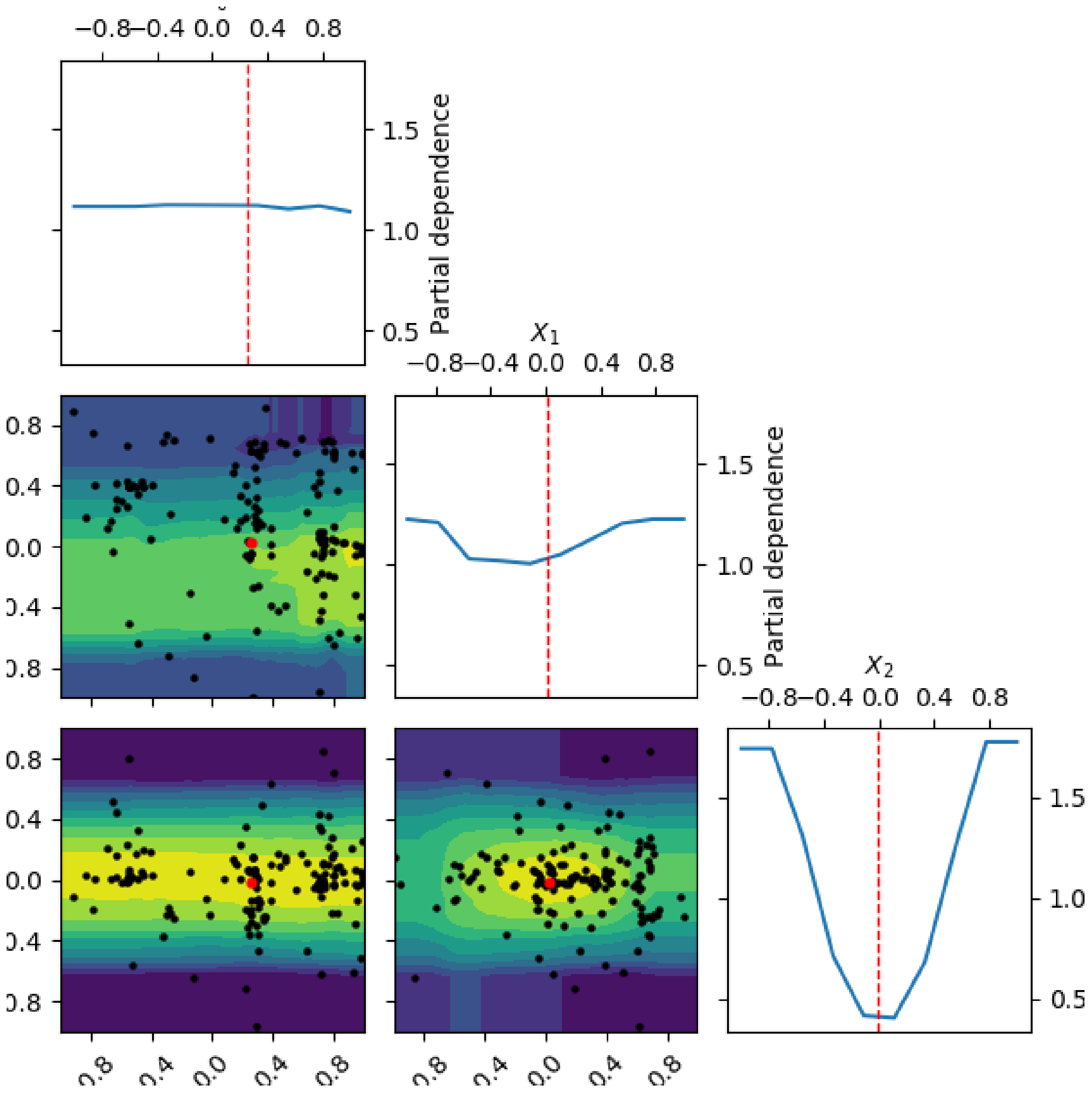
We run forest_minimize on the function

```
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

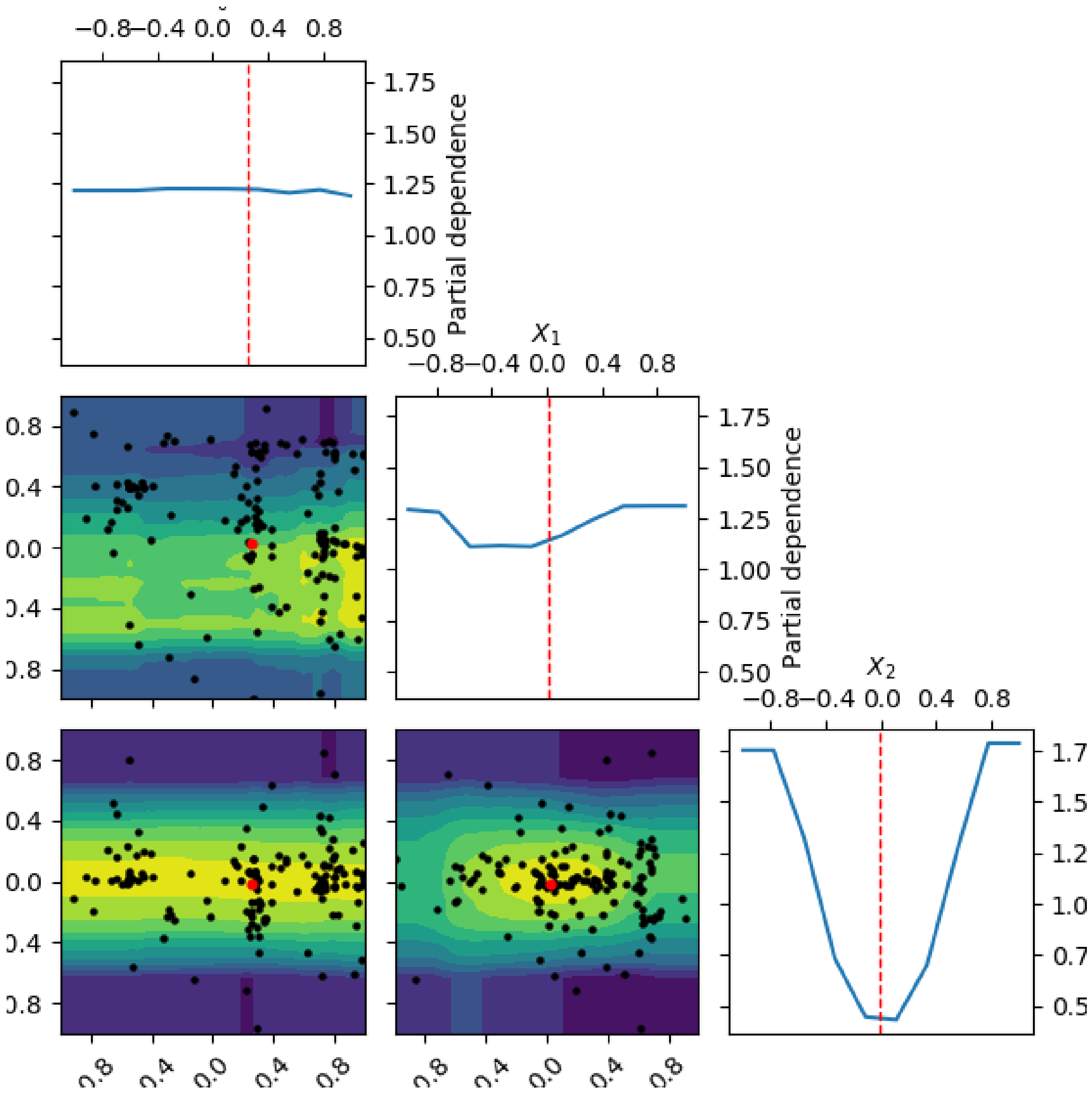
Here we see an example of using partial dependence. Even when setting n_points all the way down to 10 from the default of 40, this method is still very slow. This is because partial dependence calculates 250 extra predictions for each point on the plots.

```
_ = 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.

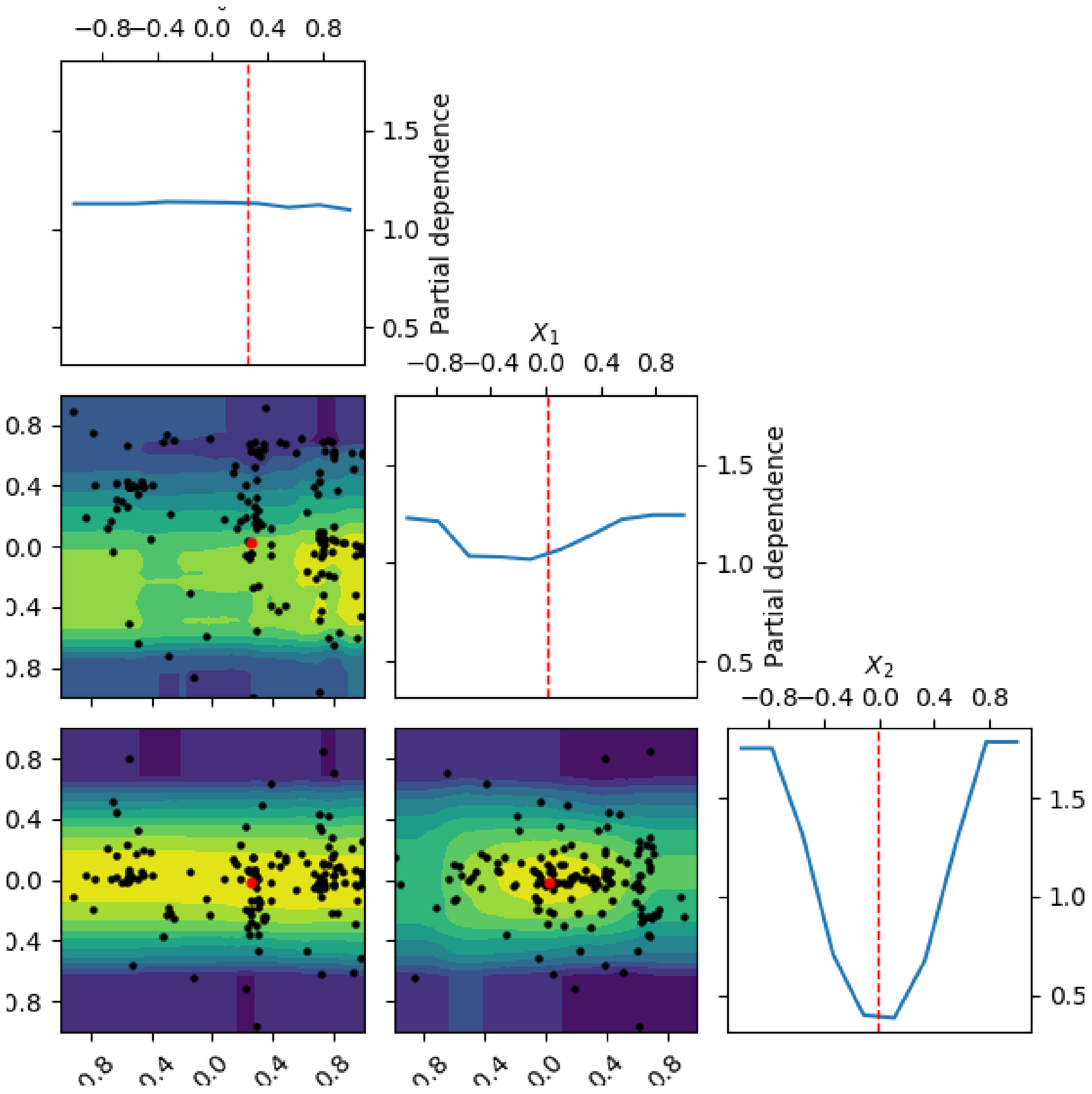
```
_ = plot_objective(result, n_points=10, minimum='expected_minimum')
```



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.

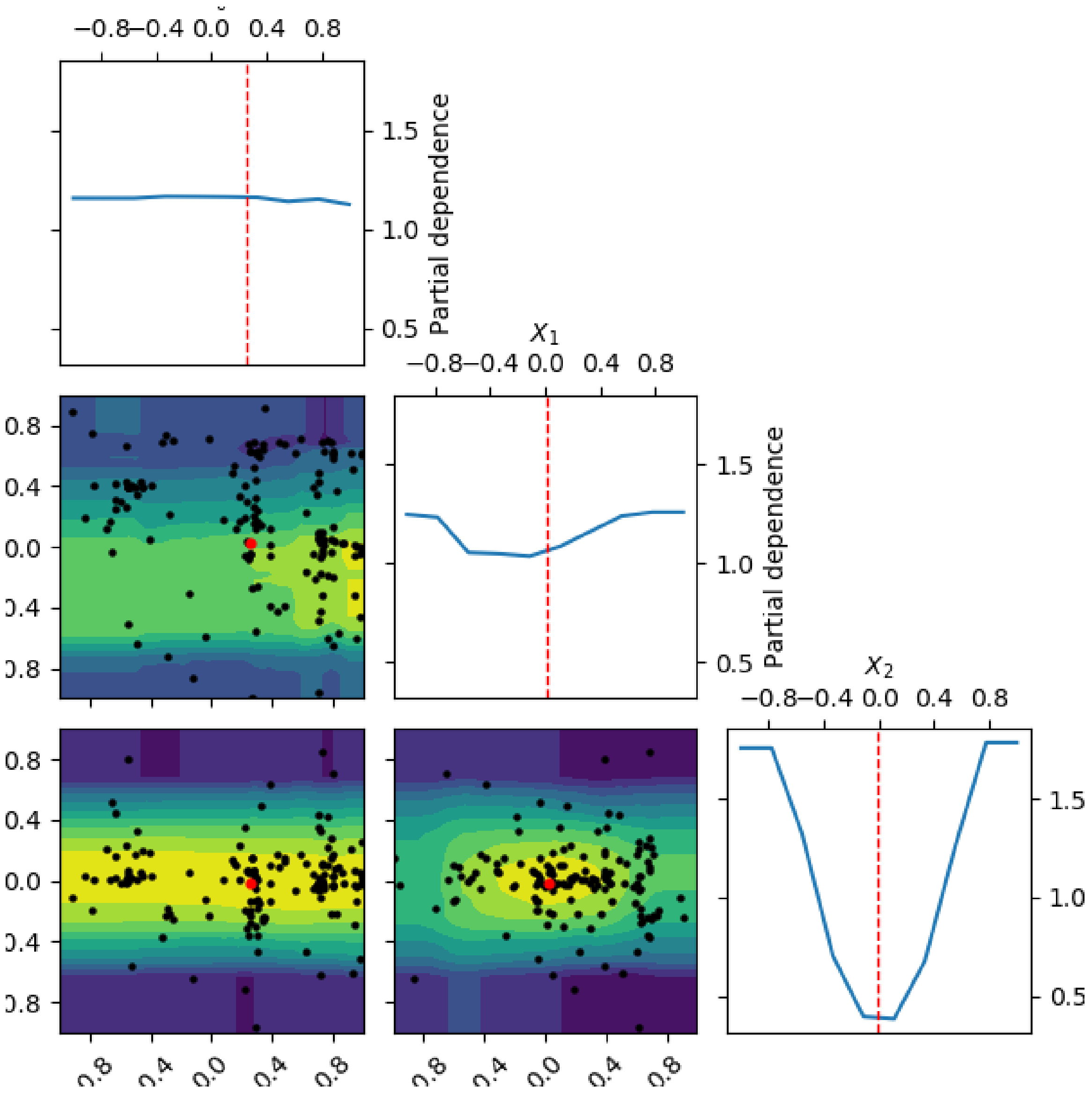
```
_ = plot_objective(result,  sample_source='result', n_points=10)
```



Modify the shown minimum

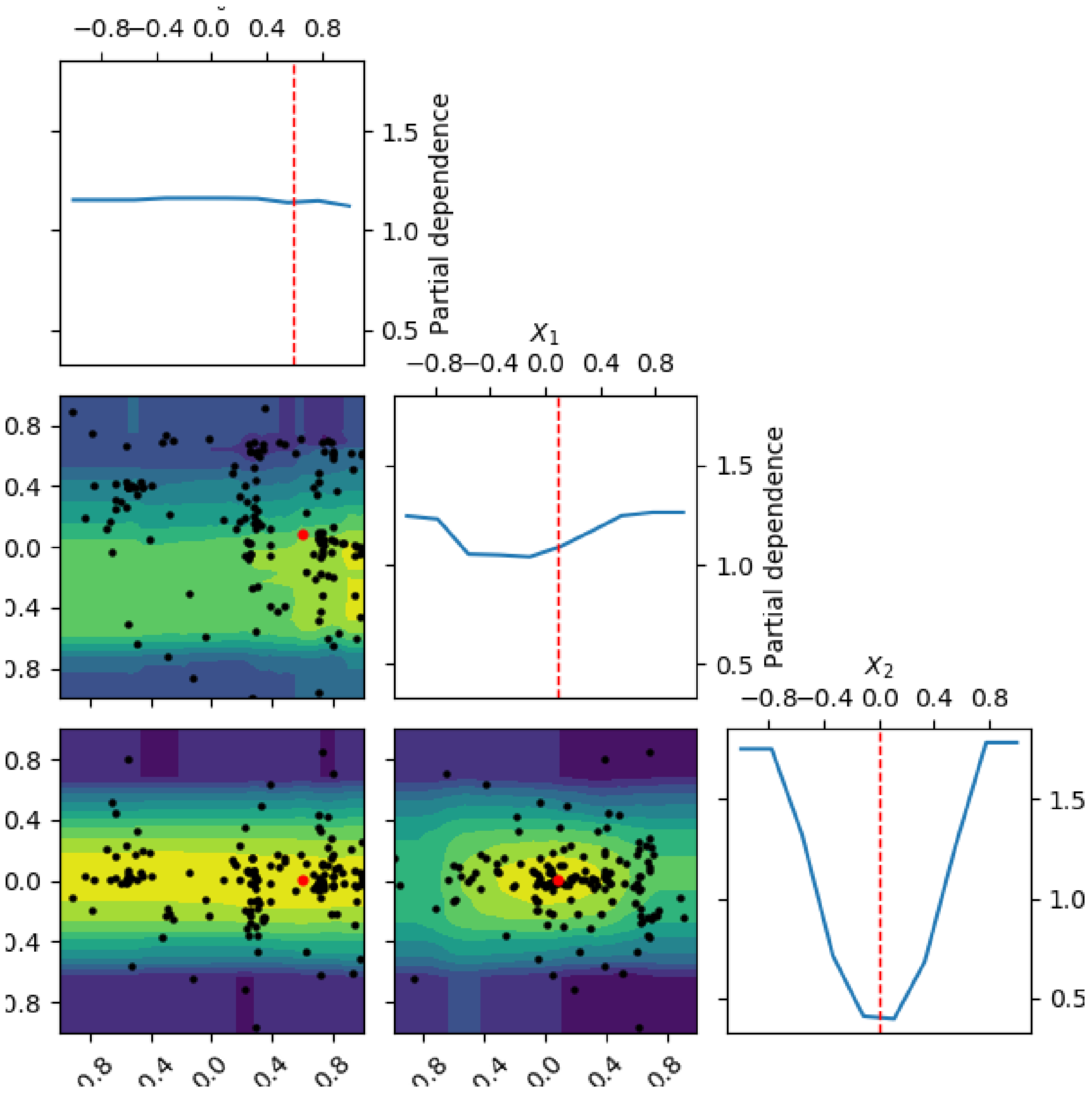
Here we try with setting the `minimum` parameters to something other than “result”. First we try with “expected_minimum” which is the set of parameters that gives the miniumum value of the surrogate function, using scipys minimum search method.

```
_ = plot_objective(result, n_points=10, sample_source='expected_minimum',
                   minimum='expected_minimum')
```



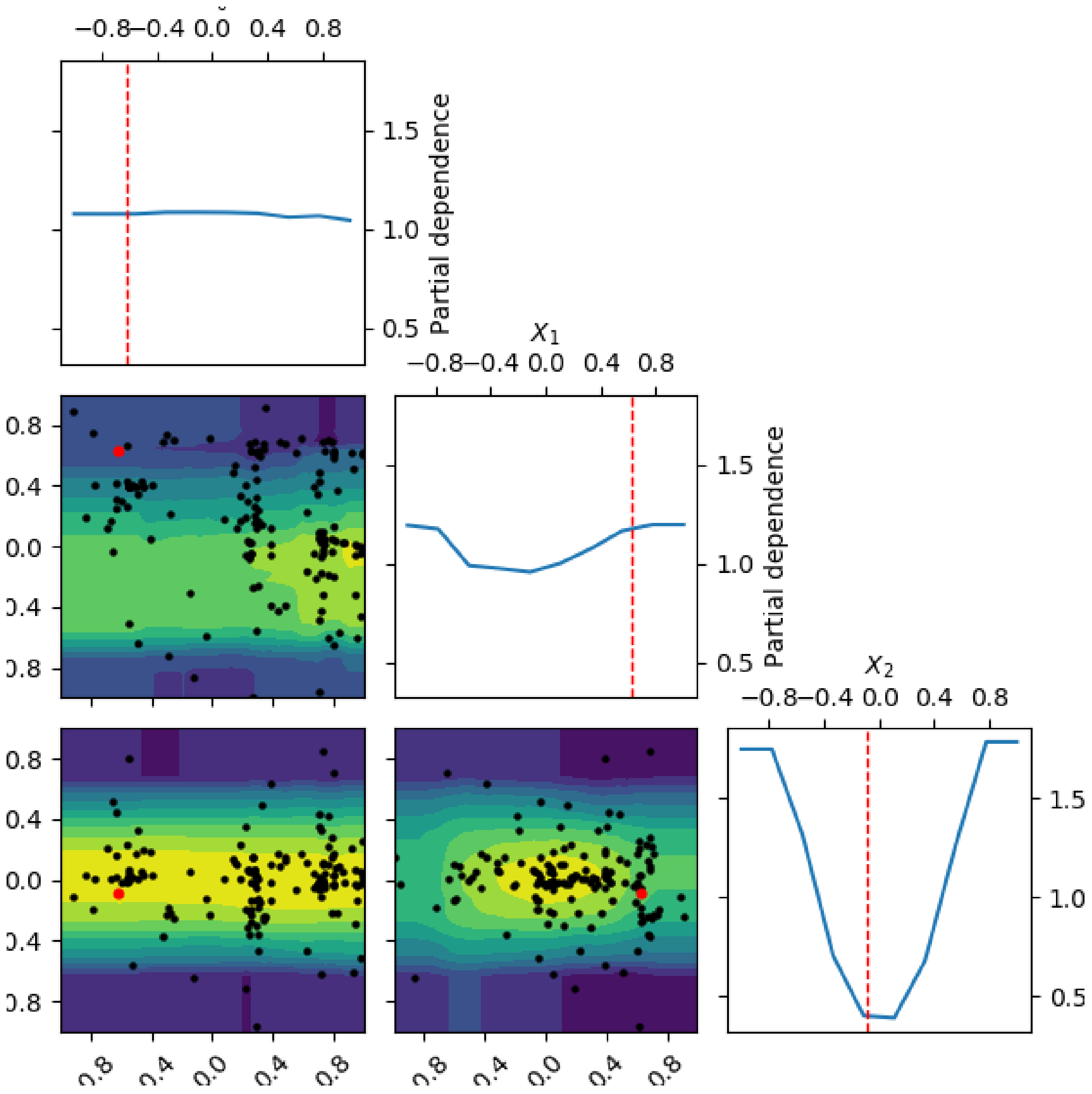
"expected_minimum_random" is a naive way of finding the minimum of the surrogate by only using random sampling:

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

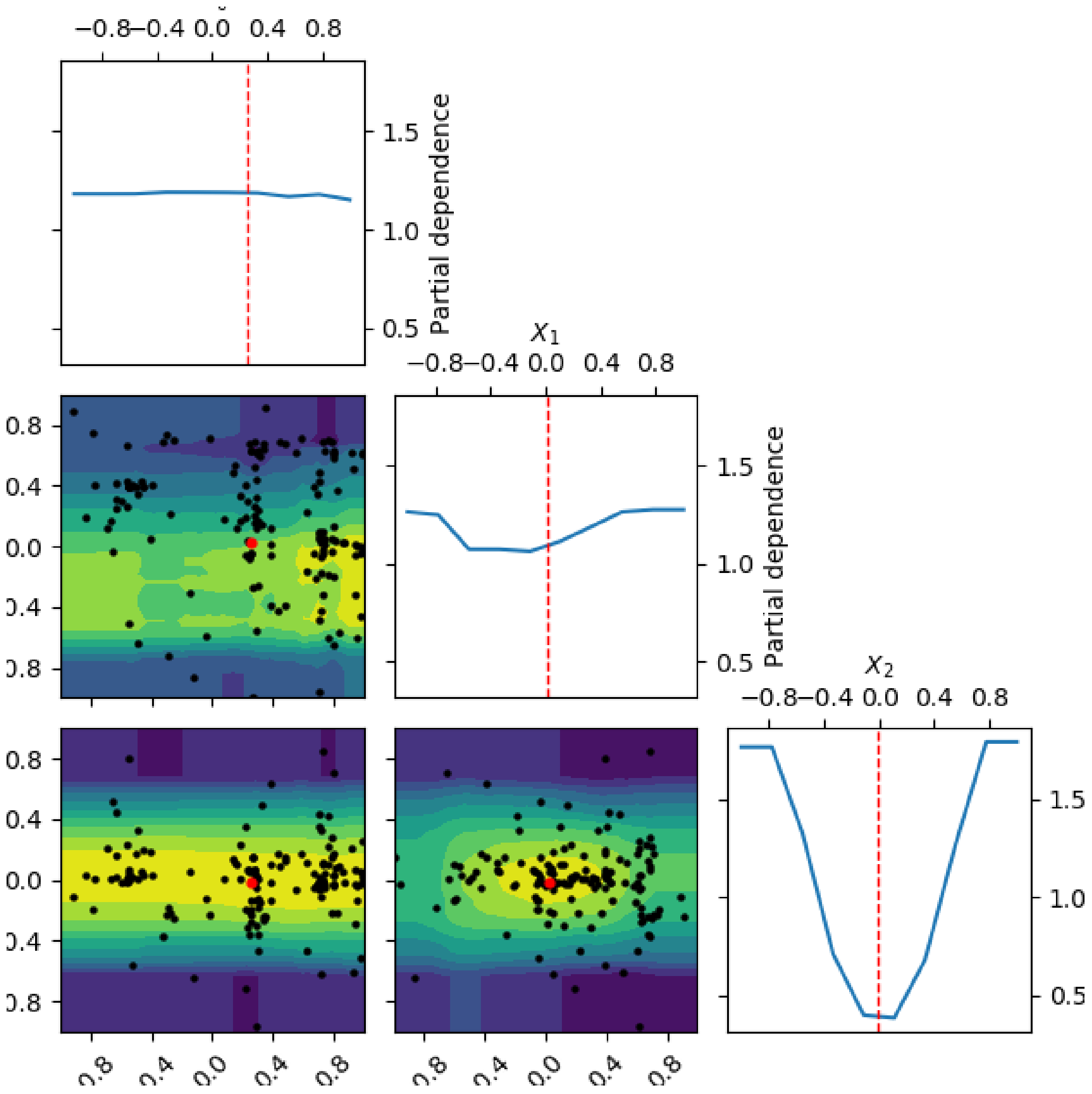


We can also specify how many initial samples are used for the two different "expected_minimum" methods. We set it to a low value in the next examples to showcase how it affects the minimum for the two methods.

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



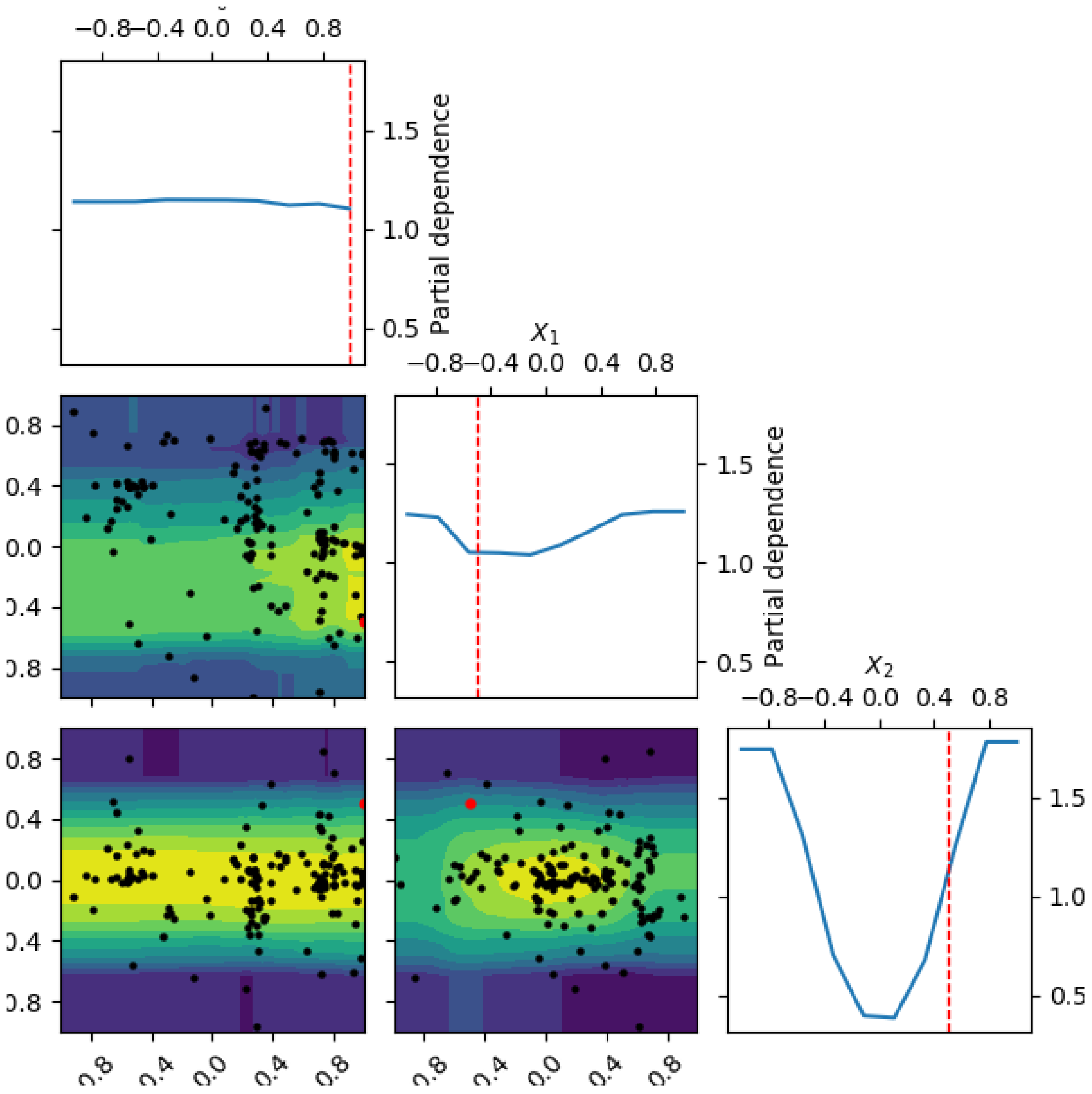
```
_ = plot_objective(result, n_points=10, sample_source="expected_minimum",
                   minimum='expected_minimum', n_minimum_search=2)
```



Set a minimum location

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

```
_ = plot_objective(result, n_points=10, sample_source=[1, -0.5, 0.5],
                   minimum=[1, -0.5, 0.5])
```



Total running time of the script: (4 minutes 7.363 seconds)

Estimated memory usage: 9 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.2.2 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.

```
print(__doc__)
import sys
```

(continues on next page)

(continued from previous page)

```
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.

```
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(True)))
```

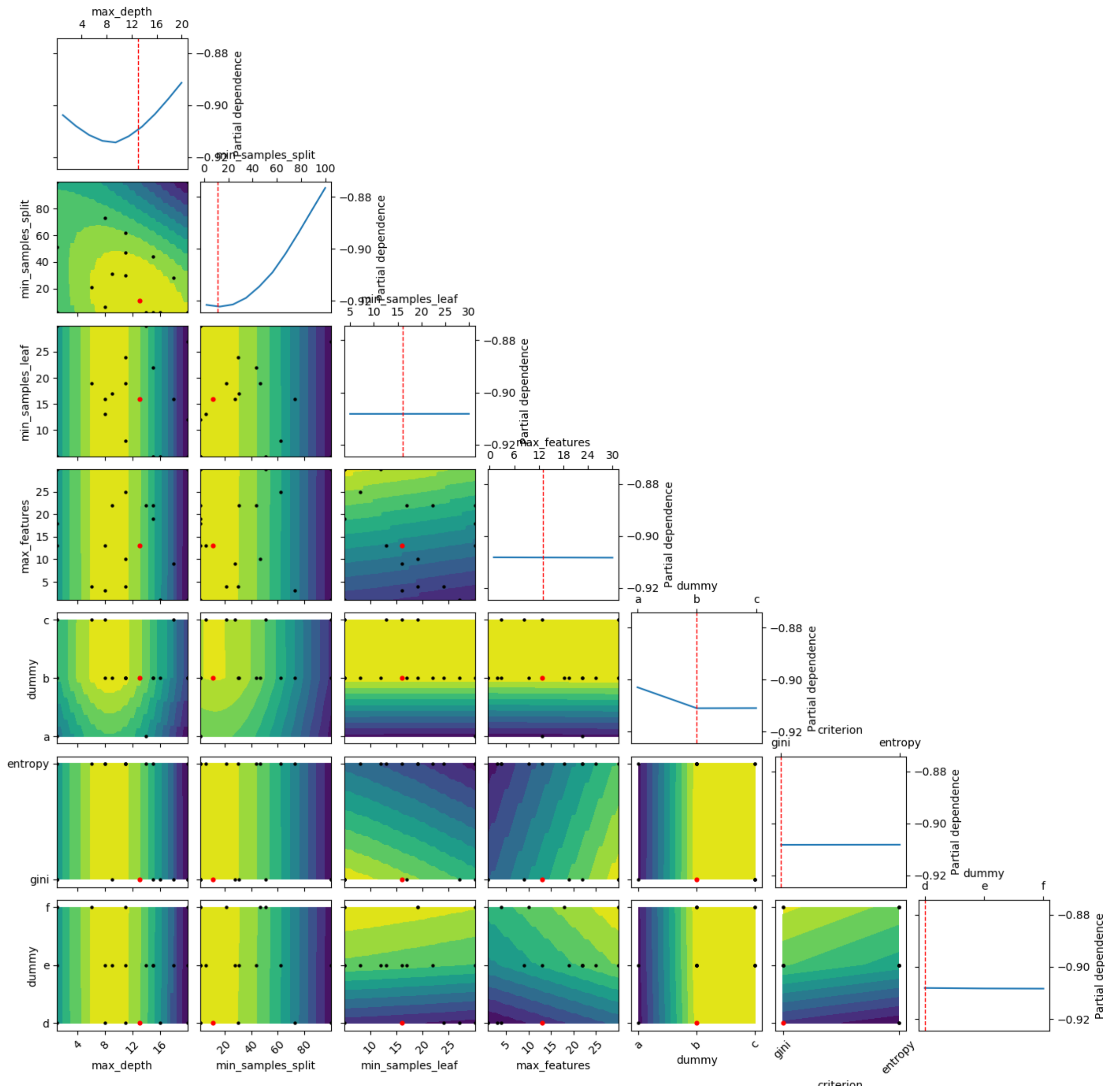
Bayesian optimization

```
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.

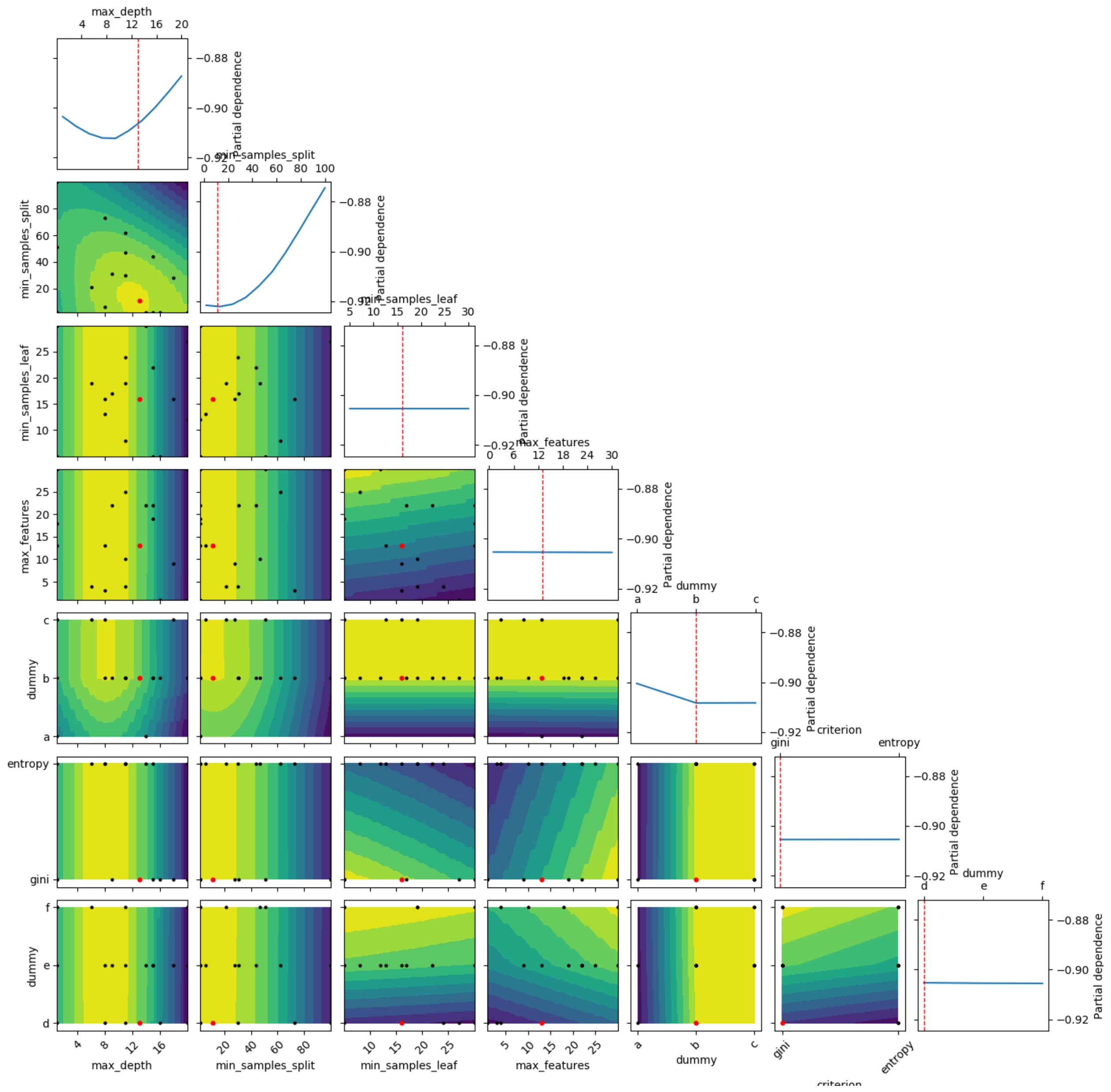
```
_ = plot_objective(result, n_points=10)
```



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.

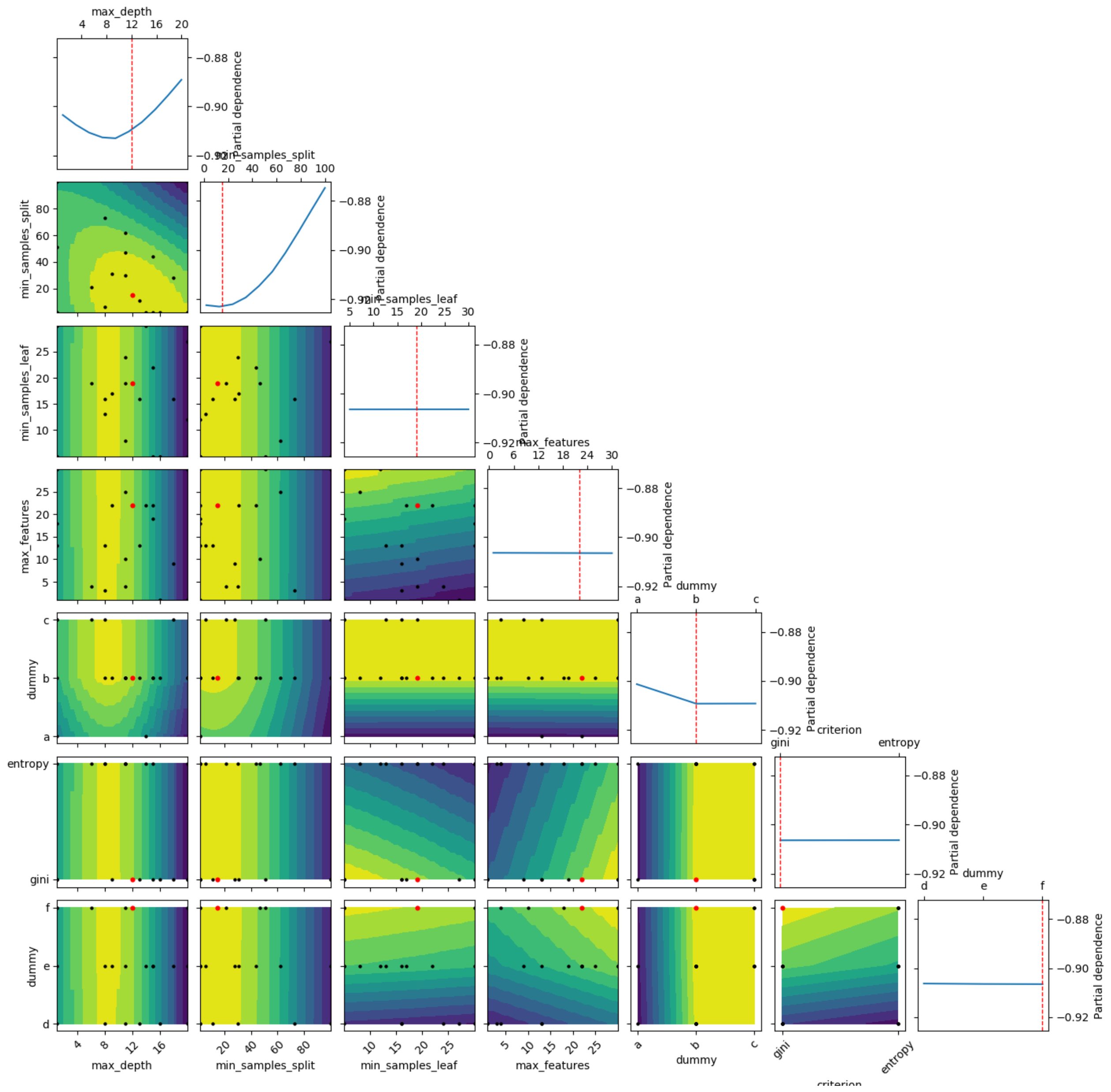
```
_ = plot_objective(result,  sample_source='result', n_points=10)
```



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

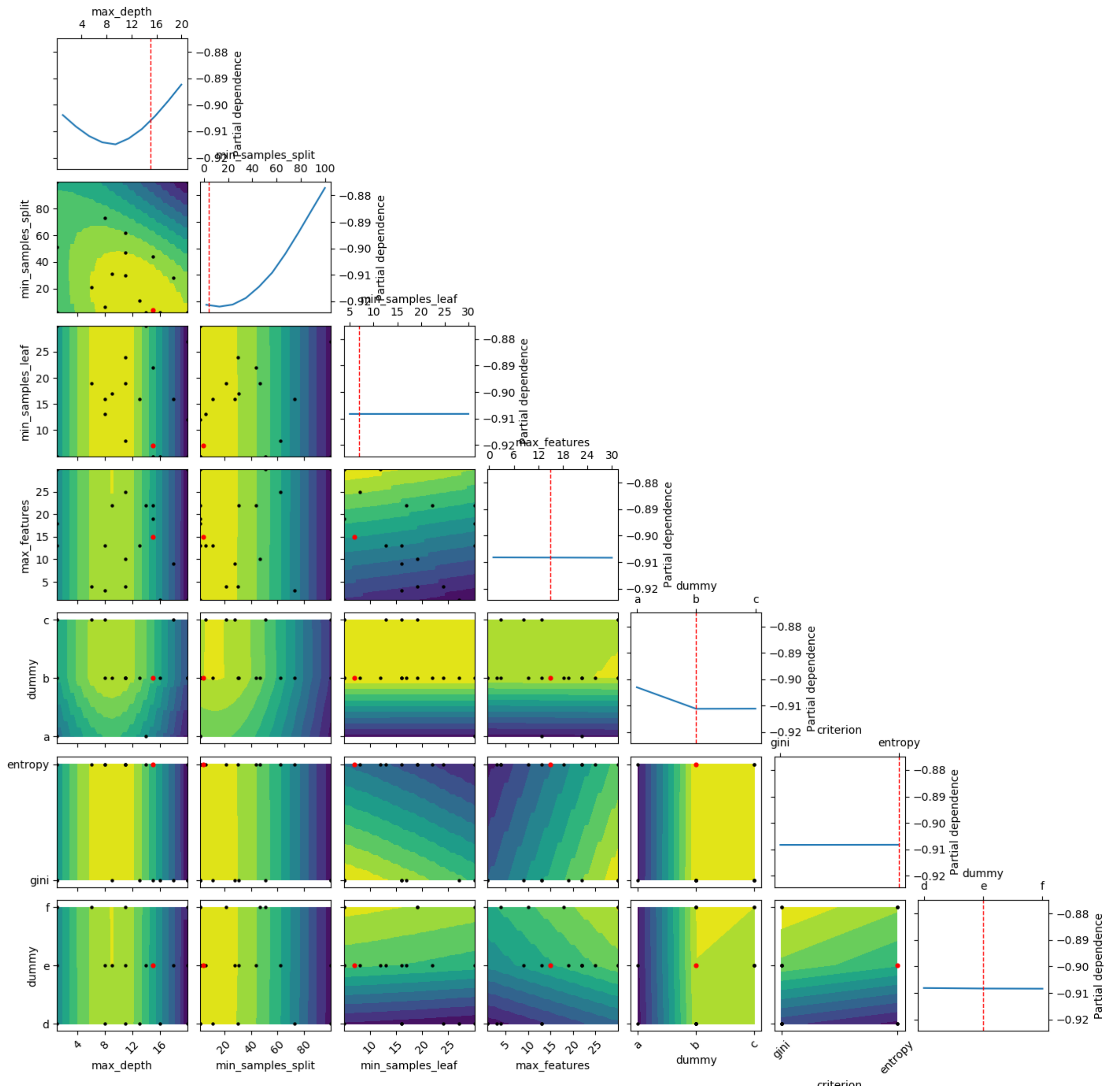
```
_ = plot_objective(result, n_points=10, sample_source='expected_minimum_random',
                   minimum='expected_minimum_random', n_minimum_search=10000)
```



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 25.236 seconds)

Estimated memory usage: 34 MB

Note: Click *here* to download the full example code or to run this example in your browser via Binder

4.2.3 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 *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 *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 `skopt` return an [`OptimizeResult`]() instance that can be inspected. Both *plots.plot_evaluations* and *plots.plot_objective* are helpers that do just that

```
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 *plots.plot_evaluations* works.

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

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

```
from skopt.benchmarks import branin as branin
from skopt.benchmarks import hart6 as hart6_


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

Starting with `branin`

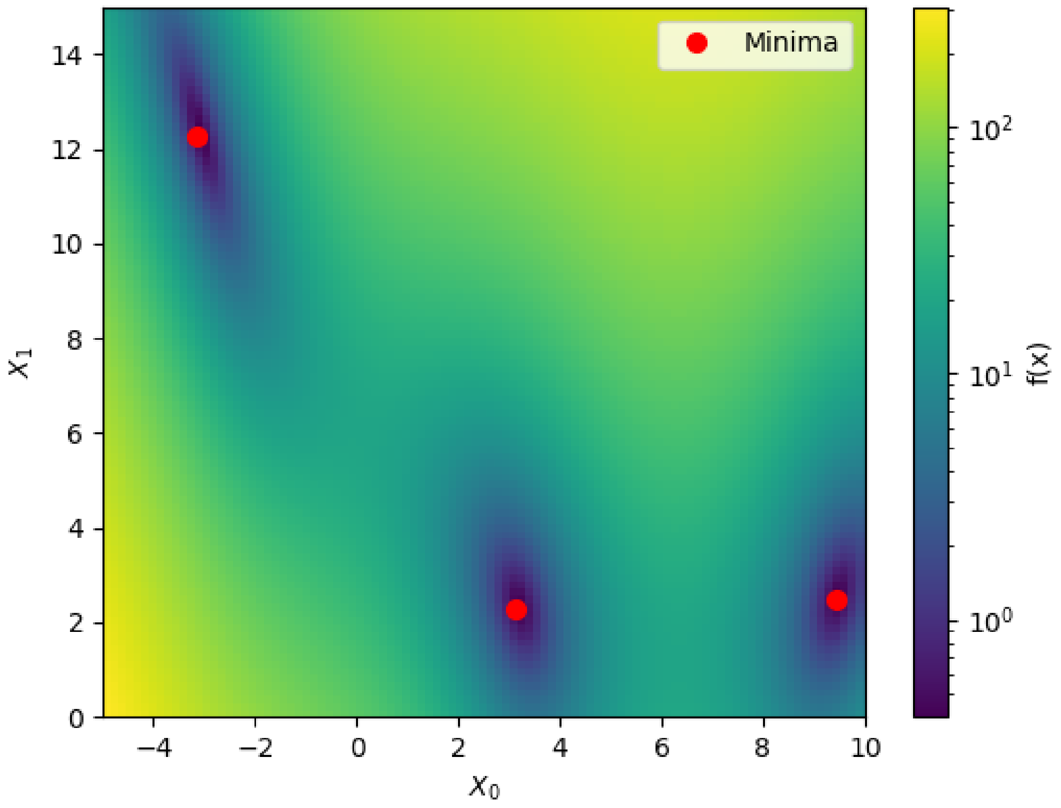
To start let's take advantage of the fact that *benchmarks.branin* is a simple function which can be visualised in two dimensions.

```
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()))
```

(continues on next page)

(continued from previous page)

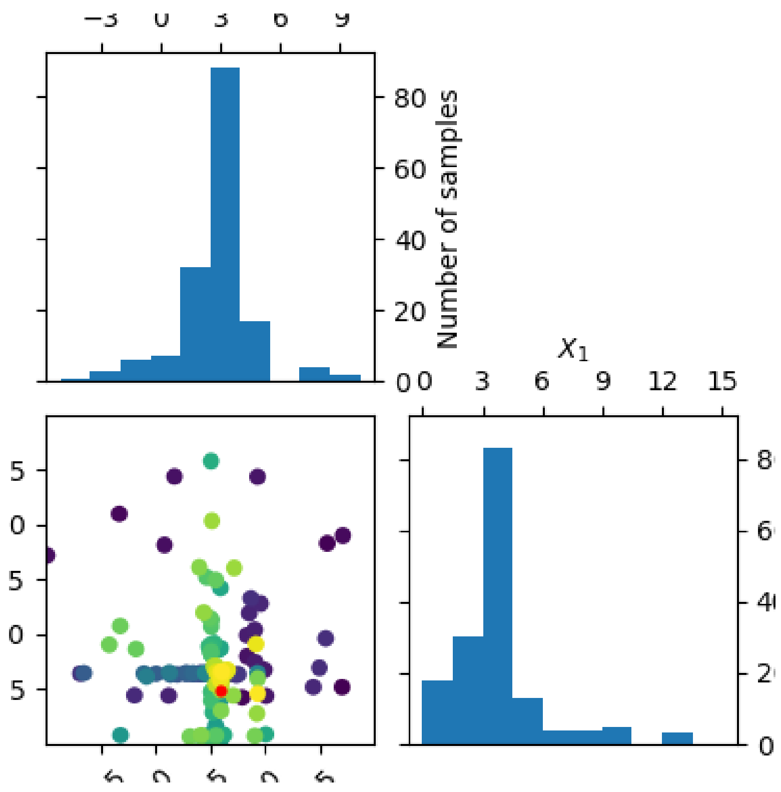
```
    minima = np.array([[-np.pi, 12.275], [+np.pi, 2.275], [9.42478, 2.475]])
    ax.plot(minima[:, 0], minima[:, 1], "r.", markersize=14,
            lw=0, label="Minima")

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

    ax.legend(loc="best", numpoints=1)

    ax.set_xlabel("$X_0$")
    ax.set_xlim([-5, 10])
    ax.set_ylabel("$X_1$")
    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*.

```
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,
                             base_estimator="ET", random_state=4)

_ = plot_evaluations(forest_res, bins=10)
```



plots.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 miminum. The histograms show that the objective is evaluated more often at locations near to one of the three minima.

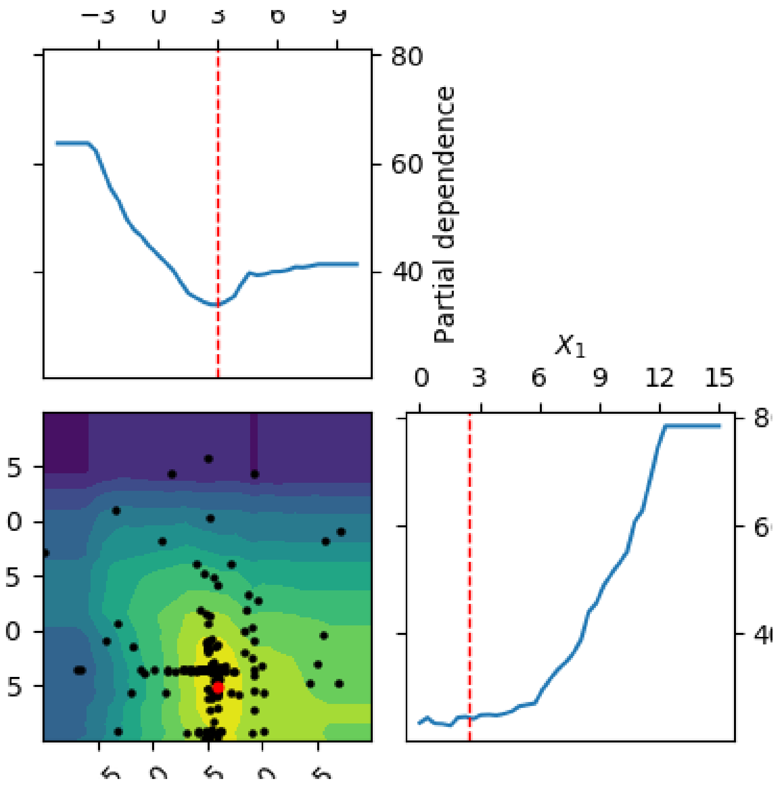
Using *plots.plot_objective* we can visualise the one dimensional partial dependence of the surrogate model for each dimension. The contour plot in the bottom left corner shows the two dimensional partial dependence. In this case this is the same as simply plotting the objective as it only has two dimensions.

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: $\phi\left(x_i\right) = \frac{1}{N}\sum_{j=0}^{N} f\left(x_{1,j}, x_{2,j}, ..., x_i, ..., x_{k,j}\right)$: 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.

```
from skopt.plots import plot_objective

_ = plot_objective(forest_res)
```

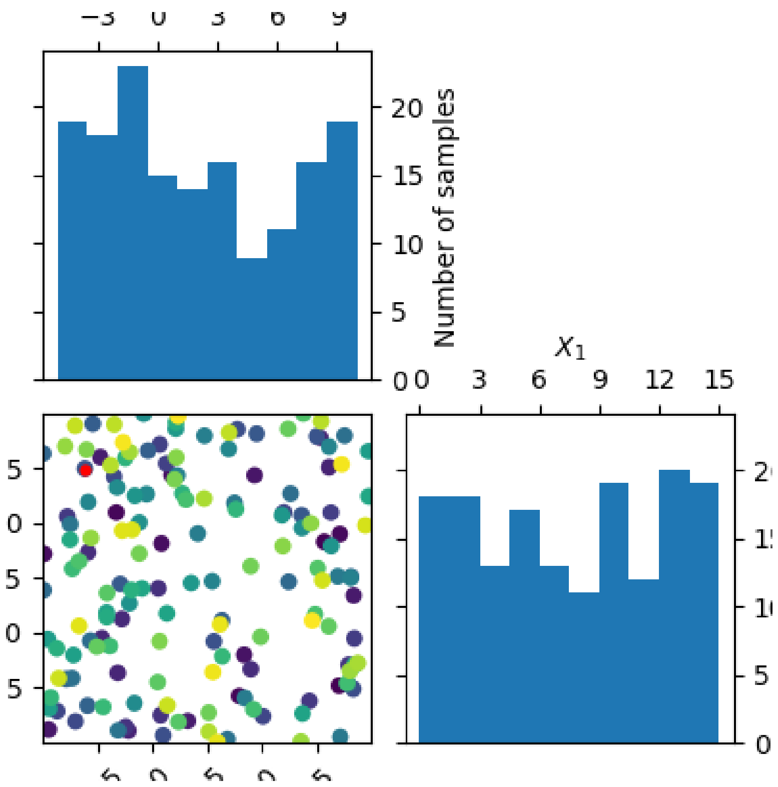


The two dimensional partial dependence plot can look like the true objective but it does not have to. As points at which the objective function is being evaluated are concentrated around the suspected minimum the surrogate model sometimes is not a good representation of the objective far away from the 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.

```
dummy_res = dummy_minimize(branin, bounds, n_calls=n_calls, random_state=4)

_ = plot_evaluations(dummy_res, bins=10)
```



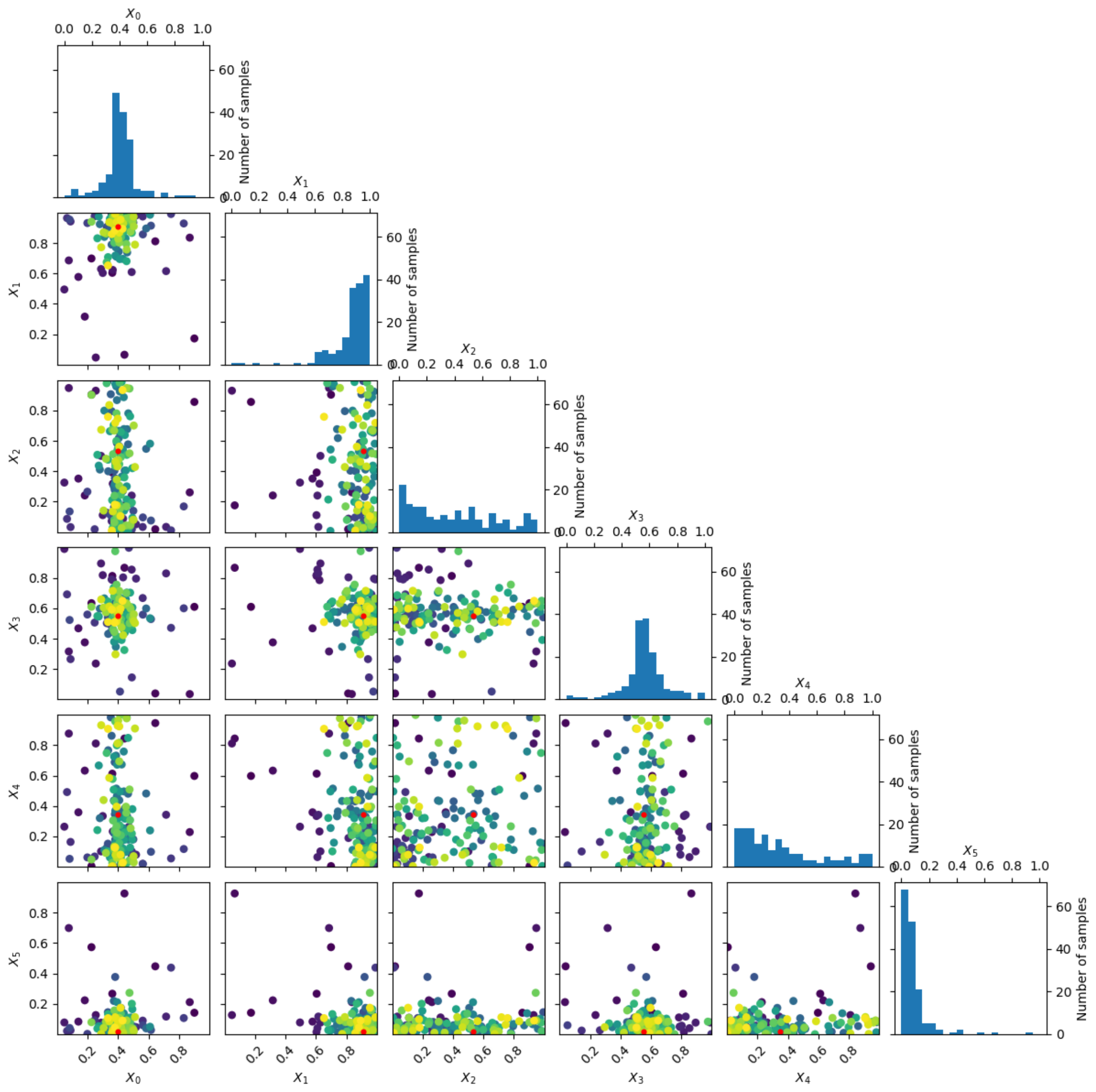
Working in six dimensions

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.

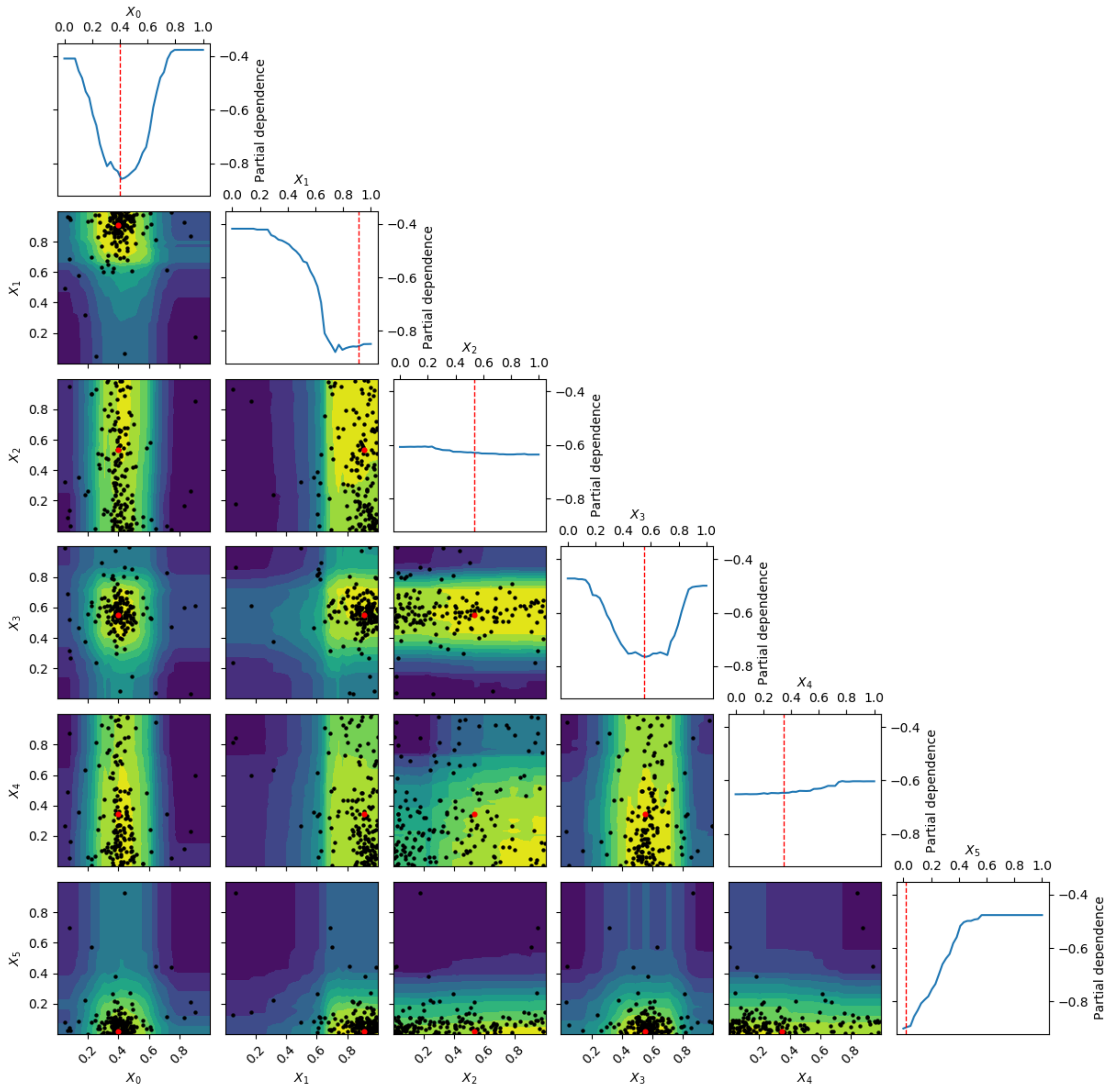
The next example uses class:`benchmarks.hart6` which has six dimensions and shows both `plots.plot_evaluations` and `plots.plot_objective`.

```
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)
```

- 

x_0
x_1
x_2
x_3
x_4
x_5
Number of samples

- 

Going from 6 to 6+2 dimensions

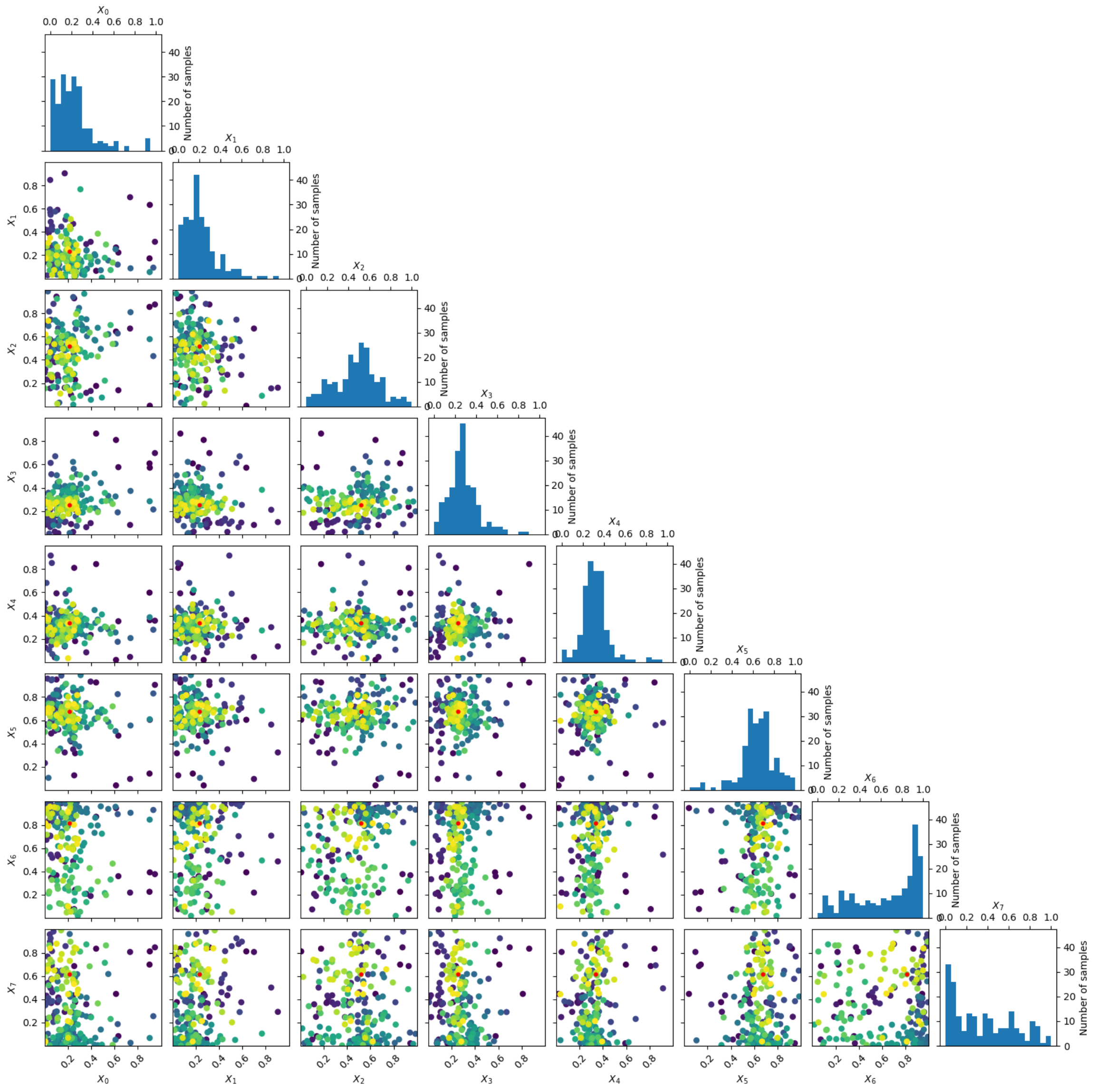
To make things more interesting let's add two dimension to the problem. As `benchmarks.hart6` only depends on six dimensions we know that for this problem the new dimensions will be "flat" or uninformative. This is clearly visible in both the placement of samples and the partial dependence plots.

```
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)
```

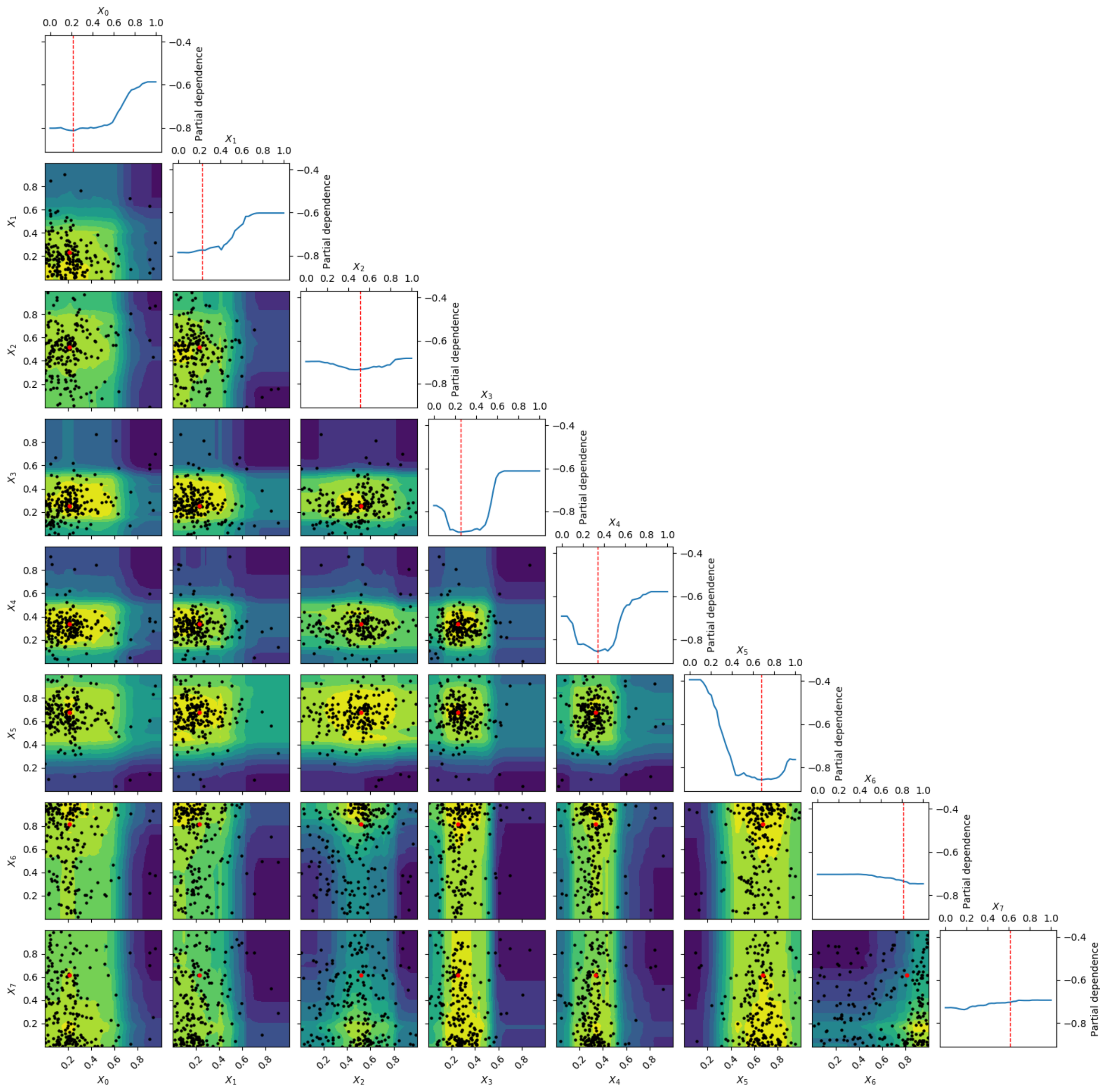
(continues on next page)

(continued from previous page)

```
# .. [Friedman (2001)] `doi:10.1214/aos/1013203451 section 8.2 <http://projecteuclid.
↪org/euclid.aos/1013203451>`
```



•



•

Total running time of the script: (7 minutes 30.177 seconds)

Estimated memory usage: 88 MB

CHAPTER

FIVE

API REFERENCE

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

BayesSearchCV(estimator, search_spaces[, . . .])	Bayesian optimization over hyper parameters.
Optimizer(dimensions[, base_estimator, . . .])	Run bayesian optimisation loop.
Space(dimensions)	Initialize a search space from given specifications.

`skopt`.BayesSearchCV

class skopt.**BayesSearchCV**(*estimator*, *search_spaces*, *optimizer_kwargs=None*, *n_iter=50*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *n_points=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score='raise'*, *return_train_score=False*)

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 (see skopt.Optimizer docs). Represents search space over parameters of the provided estimator. 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'

iid [boolean, default=True] If True, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

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

param_kernel	param_gamma	split0_test_score	…	rank_test_score
'rbf'	0.1	0.8	…	2
'rbf'	0.2	0.9	…	1
'rbf'	0.3	0.7	…	1

will be represented by a `cv_results_` dict of:

```
{
'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.],
'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.

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_).

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

```
>>> 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(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
```

(continues on next page)

(continued from previous page)

```
>>> print(opt.score(X_test, y_test))
0.973...
```

Methods

`decision_function`(self, X)	Call decision_function on the estimator with the best found parameters.
`fit`(self, X[, y, groups, callback])	Run fit on the estimator with randomly drawn parameters.
`get_params`(self[, deep])	Get parameters for this estimator.
`inverse_transform`(self, Xt)	Call inverse_transform on the estimator with the best found params.
`predict`(self, X)	Call predict on the estimator with the best found parameters.
`predict_log_proba`(self, X)	Call predict_log_proba on the estimator with the best found parameters.
`predict_proba`(self, X)	Call predict_proba on the estimator with the best found parameters.
`score`(self, X[, y])	Returns the score on the given data, if the estimator has been refit.
`set_params`(self, **params)	Set the parameters of this estimator.
`transform`(self, X)	Call transform on the estimator with the best found parameters.

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

Initialize self. See help(type(self)) for accurate signature.

`decision_function`(*self, 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.

`fit`(*self, X, y=None, groups=None, callback=None*)

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(*self*, *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 [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *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.

predict(*self*, *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.

predict_log_proba(*self*, *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.

predict_proba(*self*, *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.

score(*self*, *X*, *y=None*)
Returns 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,), optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params (*self*, ***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self [object] 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 (*self*, *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.

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*, *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 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`. Points provided as `x0` count as initialization points. If len(x0) < n_initial_points additional points are sampled at random.

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 $softmax(\eta g_i)$
    - 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=`"auto"`] Method to minimize the acquistion 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.

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

acq_optimizer_kwargs [dict] Additional arguments to be passed to the acquistion 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

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

`__init__`(*self*, *dimensions*, *base_estimator='gp'*, *n_random_starts=None*, *n_initial_points=10*, *acq_func='gp_hedge'*, *acq_optimizer='auto'*, *random_state=None*, *model_queue_size=None*, *acq_func_kwargs=None*, *acq_optimizer_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

`ask`(*self*, *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 (*self*, *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 (*self*)
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 (*self*, *func*, *n_iter=1*)
Execute ask() + tell() `n_iter` times

tell (*self*, *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 (*self*)
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*

`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.

`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_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

`distance`(self, point_a, point_b)	Compute distance between two points in this space.
`from_yaml`(yml_path[, namespace])	Create Space from yaml configuration file
`inverse_transform`(self, Xt)	Inverse transform samples from the warped space back to the
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, X)	Transform samples from the original space into a warped space.

`__init__`(*self*, *dimensions*)
Initialize self. See help(type(self)) for accurate signature.

`property bounds`
The dimension bounds, in the original space.

`distance`(*self*, *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

`inverse_transform`(*self*, *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_dims`

The dimensionality of the original space.

`rvs`(*self*, *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.

`transform`(*self*, *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.

5.1.2 Functions

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

skopt.dummy_minimize

skopt.**dummy_minimize**(*func*, *dimensions*, *n_calls=100*, *x0=None*, *y0=None*, *random_state=None*, *verbose=False*, *callback=None*, *model_queue_size=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.

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.

For more details related to the OptimizeResult object, refer http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

Examples using `skopt.dummy_minimize`

- *Comparing surrogate models*
- *Visualizing optimization results*

`skopt`.dump

`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

`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=10*, *acq_func='EI'*, *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.

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_random_starts` evaluations. Finally, `n_calls - len(x0) - n_random_starts` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_random_starts` evaluations are first made then `n_calls - n_random_starts` subsequent evaluations are made guided by the surrogate model.

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=10] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

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 http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

See also:

functions *`skopt.gp_minimize`*, *`skopt.dummy_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=10*, *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_random_starts` evaluations. Finally, `n_calls - len(x0) - n_random_starts` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_random_starts` evaluations are first made then `n_calls - n_random_starts` subsequent evaluations are made guided by the surrogate model.

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=10] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

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

skopt.gp_minimize

skopt.**gp_minimize**(*func, dimensions, base_estimator=None, n_calls=100, n_random_starts=10, 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_random_starts` evaluations. Finally, `n_calls - len(x0) - n_random_starts` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_random_starts` evaluations are first made then `n_calls - n_random_starts` subsequent evaluations are made guided by the surrogate model.

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=10] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

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 η 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 acquistion 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 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.

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.forest_minimize`*, *`skopt.dummy_minimize`*

Examples using `skopt.gp_minimize`

- *Tuning a scikit-learn estimator with skopt*
- *Store and load skopt optimization results*
- *Comparing surrogate models*
- *Interruptible optimization runs with checkpoints*
- *Bayesian optimization with skopt*
- *Partial Dependence Plots with categorical values*

`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.

`acquisition.gaussian_acquisition_1D`(X, model)	A wrapper around the acquisition function that is called by fmin_l_bfgs_b.
`acquisition.gaussian_ei`(X, model[, y_opt, . . .])	Use the expected improvement to calculate the acquisition values.
`acquisition.gaussian_lcb`(X, model[, kappa, . . .])	Use the lower confidence bound to estimate the acquisition values.
`acquisition.gaussian_pi`(X, model[, y_opt, . . .])	Use the probability of improvement to calculate the acquisition values.

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) | 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.

Examples using `skopt.acquisition.gaussian_ei`

- *Async optimization Loop*
- *Exploration vs exploitation*
- *Bayesian optimization with skopt*
- *Use different base estimators for optimization*

5.2.3 `skopt.acquisition`.gaussian_lcb

`skopt.acquisition.`**`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 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 `skopt.acquisition`.gaussian_pi

`skopt.acquisition.`**`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) | 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

`benchmarks.bench1`(x)	A benchmark function for test purposes.
`benchmarks.bench1_with_time`(x)	Same as bench1 but returns the computation time (constant).

Continued on next page

Table 7 – continued from previous page

benchmarks.bench2(x)	A benchmark function for test purposes.
benchmarks.bench3(x)	A benchmark function for test purposes.
benchmarks.bench4(x)	A benchmark function for test purposes.
benchmarks.bench5(x)	A benchmark function for test purposes.
benchmarks.branin(x[, a, b, c, r, s, t])	Branin-Hoo function is defined on the square $x1 \in [-5, 10], x2 \in [0, 15]$.
benchmarks.hart6([P, A])	The six dimensional Hartmann function is defined on the unit hypercube.

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) = x ** 2 if x < 0 (x-5) ** 2 - 5 otherwise.

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(5*x) * (1 - tanh(x ** 2))

It has a global minima with f(x*) ~= -0.9 at x* ~= -0.3.

skopt.benchmarks.bench4

skopt.benchmarks.**bench4** (*x*)

A benchmark function for test purposes.

f(x) = 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) = 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 $x1 \in [-5, 10], x2 \in [0, 15]$.

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>

Examples using skopt.benchmarks.branin

- *Parallel optimization*
- *Comparing surrogate models*
- *Visualizing optimization results*

skopt.benchmarks.hart6

```
hart6(x, alpha=array([1. , 1.2, 3. , 3.2]), P=array([[0.1312, 0.1696, 0.5569, 0.0124, 0.828
[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=array([[10.  ,  3.  , 17.  ,  3.5 ,
[ 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

- *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.

callbacks.CheckpointSaver(checkpoint_path, ...)	Save current state after each iteration with *skopt.dump*.
callbacks.DeadlineStopper(total_time)	Stop the optimization before running out of a fixed budget of time.
callbacks.DeltaXStopper(delta)	Stop the optimization when `|x1 - x2| < delta`
callbacks.DeltaYStopper(delta[, n_best])	Stop the optimization if the `n_best` minima are within `delta`
callbacks.EarlyStopper	Decide to continue or not given the results so far.
callbacks.TimerCallback()	Log the elapsed time between each iteration of the minimization loop.
callbacks.VerboseCallback(n_total[, n_init, ...])	Callback to control the verbosity.

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
>>> checkpoint_callback = skopt.callbacks.CheckpointSaver("./result.pkl")
>>> skopt.gp_minimize(obj_fun, [(-2, 2)], n_calls=10,
...                   callback=[checkpoint_callback]) # doctest: +SKIP
```

Methods

`__call__`(self, res)	**Parameters**

`__init__`(*self*, *checkpoint_path*, ***dump_options*)

Initialize self. See help(type(self)) for accurate signature.

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__`(self, result)	**Parameters**

`__init__`(*self*, *total_time*)

Initialize self. See help(type(self)) for accurate signature.

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__`(self, result)	**Parameters**

`__init__`(*self*, *delta*)

Initialize self. See help(type(self)) for accurate signature.

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__`(self, result)

Parameters

`__init__`(*self*, *delta*, *n_best=5*)
Initialize self. See help(type(self)) for accurate signature.

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__`(self, result)

Parameters

`__init__`(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

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__`(self, res)

Parameters

`__init__`(*self*)
Initialize self. See help(type(self)) for accurate signature.

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__`(self, res)	**Parameters**

`__init__`(*self*, *n_total*, *n_init=0*, *n_random=0*)
Initialize self. See help(type(self)) for accurate signature.

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:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * 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` 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]] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features when `fit` is performed.

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

[R8d4c5fa7c0c3-1]

Methods

`apply`(self, X)	Apply trees in the forest to X, return leaf indices.
`decision_path`(self, X)	Return the decision path in the forest.
`fit`(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`get_params`(self[, deep])	Get parameters for this estimator.
`predict`(self, X[, return_std])	Predict continuous output for X.

Continued on next page

Table 17 – continued from previous page

`score`(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
`set_params`(self, **params)	Set the parameters of this estimator.

`__init__`(*self*, *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*)
Initialize self. See help(type(self)) for accurate signature.

`apply`(*self*, *X*)
Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or 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 [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

`decision_path`(*self*, *X*)
Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or 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 csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

`property feature_importances_`

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

Returns

feature_importances_ [array, 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`(*self*, *X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like 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]

`get_params`(*self*, *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 [mapping of string to any] Parameter names mapped to their values.

`predict`(*self*, *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`(*self*, *X*, *y*, *sample_weight=None*)

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

The coefficient R^2 is defined as (1 - 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, 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 R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `r2_score()`. This will influence the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `r2_score()` directly or make a custom scorer with `make_scorer()` (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

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

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

Parameters

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

Returns

self [object] 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:

```
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 considerable 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

`fit`(self, X, y)	Fit Gaussian process regression model.
`get_params`(self[, deep])	Get parameters for this estimator.
`log_marginal_likelihood`(self[, theta, …])	Returns log-marginal likelihood of theta for training data.
`predict`(self, X[, return_std, return_cov, …])	Predict output for X.
`sample_y`(self, X[, n_samples, random_state])	Draw samples from Gaussian process and evaluate at X.
`score`(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
`set_params`(self, **params)	Set the parameters of this estimator.

`__init__`(*self*, *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*)
Initialize self. See help(type(self)) for accurate signature.

`fit`(*self*, *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`(*self*, *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 [mapping of string to any] Parameter names mapped to their values.

`log_marginal_likelihood`(*self*, *theta=None*, *eval_gradient=False*, *clone_kernel=True*)
Returns log-marginal likelihood of theta for training data.

Parameters

theta [array-like of shape (n_kernel_params,) or 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.

clone_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 [array, 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(*self*, *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 covariance (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(*self*, *X*, *n_samples=1*, *random_state=0*)

Draw samples from Gaussian process and evaluate at X.

Parameters

X [sequence of length n_samples] Query points where the GP is evaluated. Could either be array-like with shape = (n_samples, n_features) or a list of objects.

n_samples [int, default: 1] The number of samples drawn from the Gaussian process

random_state [int, RandomState instance or None, optional (default=0)] 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`.

Returns

y_samples [array, shape = (n_samples_X, [n_output_dims], n_samples)] Values of n_samples samples drawn from Gaussian process and evaluated at query points.

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

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

The coefficient R^2 is defined as (1 - 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, 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 R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `r2_score()`. This will influence the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `r2_score()` directly or make a custom scorer with `make_scorer()` (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self [object] Estimator instance.

Examples using `skopt.learning.GaussianProcessRegressor`

- *Use different base estimators for optimization*

5.5.3 `skopt.learning`.GradientBoostingQuantileRegressor

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`(self, X, y)	Fit one regressor for each quantile.
`get_params`(self[, deep])	Get parameters for this estimator.
`predict`(self, X[, return_std, return_quantiles])	Predict.
`score`(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
`set_params`(self, **params)	Set the parameters of this estimator.

`__init__`(*self, quantiles=[0.16, 0.5, 0.84], base_estimator=None, n_jobs=1, random_state=None*)

Initialize self. See help(type(self)) for accurate signature.

`fit`(*self, 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`(*self, 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 [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *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 (*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - 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, 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 R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `r2_score()`. This will influence the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `r2_score()` directly or make a custom scorer with `make_scorer()` (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

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

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

Parameters

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

Returns

self [object] Estimator instance.

5.5.4 `skopt.learning`.RandomForestRegressor

class `skopt.learning.`**RandomForestRegressor**(*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*)

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.)] 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:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * 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` 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]] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features when `fit` is performed.

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

[R91c6cd8711c5-1]

Methods

`apply`(self, X)	Apply trees in the forest to X, return leaf indices.
`decision_path`(self, X)	Return the decision path in the forest.
`fit`(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`get_params`(self[, deep])	Get parameters for this estimator.
`predict`(self, X[, return_std])	Predict continuous output for X.
`score`(self, X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
`set_params`(self, **params)	Set the parameters of this estimator.

`__init__`(*self, 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*)
Initialize self. See help(type(self)) for accurate signature.

`apply`(*self, X*)
Apply trees in the forest to X, return leaf indices.

Parameters

X [{array-like or 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 [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

`decision_path`(*self, X*)
Return the decision path in the forest.

New in version 0.18.

Parameters

X [{array-like or 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 csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

`property feature_importances_`

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

Returns

feature_importances_ [array, 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(*self*, *X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape (n_samples, n_features)] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like 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]

get_params(*self*, *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 [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *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] ~= 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(*self*, *X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - 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, 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 R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `r2_score()`. This will influence the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `r2_score()` directly or make a custom scorer with `make_scorer()` (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

`set_params`(*self*, ***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self [object] 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*, *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 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`. Points provided as `x0` count as initialization points. If len(x0) < n_initial_points additional points are sampled at random.

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 $softmax(\eta g_i)$
    - 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=‘”auto”‘] Method to minimize the acquistion 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.

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

acq_optimizer_kwargs [dict] Additional arguments to be passed to the acquistion 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

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

`__init__`(*self*, *dimensions*, *base_estimator='gp'*, *n_random_starts=None*, *n_initial_points=10*, *acq_func='gp_hedge'*, *acq_optimizer='auto'*, *random_state=None*, *model_queue_size=None*, *acq_func_kwargs=None*, *acq_optimizer_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

`ask`(*self*, *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 (*self*, *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 (*self*)
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 (*self*, *func*, *n_iter=1*)
Execute ask() + tell() n_iter times

tell (*self*, *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 (*self*)
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 :param func: Function to minimize.
optimizer.dummy_minimize(func, dimensions[, …])	Random search by uniform sampling within the given bounds.
optimizer.forest_minimize(func, dimensions)	Sequential optimisation using decision trees.

Continued on next page

Table 23 – continued from previous page

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=10, 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 :param func: 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.

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.

- **base_estimator** (*sklearn regressor*) – Should inherit from `sklearn.base.RegressorMixin`. In addition, should have an optional `return_std` argument, which returns `std(Y | x)`` along with `E[Y | x]`.
- **n_calls** (*int, default=100*) – Maximum number of calls to `func`. An objective fucntion will always be evaluated this number of times; Various options to supply initialization points do not affect this value.
- **n_random_starts** (*int, default=10*) – Number of evaluations of `func` with random points before approximating it with `base_estimator`.
- **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 acquistion 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. 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.

For more details related to the OptimizeResult object, refer http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

5.6.3 `skopt.optimizer`.dummy_minimize

`skopt.optimizer.`**`dummy_minimize`**(*func, dimensions, n_calls=100, x0=None, y0=None, random_state=None, verbose=False, callback=None, model_queue_size=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.

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.

For more details related to the OptimizeResult object, refer http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

5.6.4 `skopt.optimizer`.forest_minimize

`skopt.optimizer.`**`forest_minimize`**(*func*, *dimensions*, *base_estimator='ET'*, *n_calls=100*, *n_random_starts=10*, *acq_func='EI'*, *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.

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_random_starts` evaluations. Finally, `n_calls - len(x0) - n_random_starts` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_random_starts` evaluations are first made then `n_calls - n_random_starts` subsequent evaluations are made guided by the surrogate model.

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=10] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

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 http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html

See also:

functions *`skopt.gp_minimize`*, *`skopt.dummy_minimize`*

5.6.5 `skopt.optimizer`.gbrt_minimize

`skopt.optimizer.`**`gbrt_minimize`**(*func*, *dimensions*, *base_estimator=None*, *n_calls=100*, *n_random_starts=10*, *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_random_starts` evaluations. Finally, `n_calls - len(x0) - n_random_starts` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_random_starts` evaluations are first made then `n_calls - n_random_starts` subsequent evaluations are made guided by the surrogate model.

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=10] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

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

5.6.6 `skopt.optimizer`.gp_minimize

`skopt.optimizer.`**`gp_minimize`**(*func*, *dimensions*, *base_estimator=None*, *n_calls=100*, *n_random_starts=10*, *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_random_starts` evaluations. Finally, `n_calls - len(x0) - n_random_starts` evaluations are made guided by the surrogate model. If `x0` and `y0` are both provided then `n_random_starts` evaluations are first made then `n_calls - n_random_starts` subsequent evaluations are made guided by the surrogate model.

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=10] Number of evaluations of `func` with random points before approximating it with `base_estimator`.

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 η 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 acquistion 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 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.

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.forest_minimize*, *skopt.dummy_minimize*

5.7 skopt.plots: Plotting functions.

Plotting functions.

User guide: See the *Plotting tools* section for further details.

plots.partial_dependence(space, model, i[, . . .])	Calculate the partial dependence for dimensions i and j with respect to the objective value, as approximated by model.
plots.plot_convergence(*args, **kwargs)	Plot one or several convergence traces.
plots.plot_evaluations(result[, bins, . . .])	Visualize the order in which points where sampled.
plots.plot_objective(result[, levels, . . .])	Pairwise dependence plot of the objective function.
plots.plot_regret(*args, **kwargs)	Plot one or several cumulative regret traces.

5.7.1 skopt.plots.partial_dependence

skopt.plots.**partial_dependence**(*space*, *model*, *i*, *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_eval is not None, the given values are used instead of random samples. In this case, n_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=None.

sample_points [np.array, shape=(n_points, n_dims), default=None] Only used when x_eval=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_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_points when using partial dependence. Only used when sample_points=None and x_eval=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_eval` is a list of parameter values or None. In case `x_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 `xi`.

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 `(xi, yi)`.

For Categorical variables, the `xi` (and `yi` for 2D) returned are

the indices of the variable in `Dimension.categories`.

5.7.2 `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

ax [`Axes`] The matplotlib axes.

Examples using `skopt.plots.plot_convergence`

- *Tuning a scikit-learn estimator with skopt*
- *Comparing surrogate models*
- *Bayesian optimization with skopt*

5.7.3 `skopt.plots`.plot_evaluations

`skopt.plots.`**`plot_evaluations`**(*result*, *bins=20*, *dimensions=None*)

Visualize the order in which points where sampled.

The scatter plot matrix shows at which points in the search space and in which order samples were evaluated. Pairwise scatter plots are shown on the off-diagonal for each dimension of the search space. The order in which samples were evaluated is encoded in each point's color. The diagonal shows a histogram of sampled values for each dimension. A red point indicates the found minimum.

Parameters

result [`OptimizeResult`] The result for which to create the scatter plot matrix.

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', ..]`.

Returns

ax [`Axes`] The matplotlib axes.

Examples using `skopt.plots.plot_evaluations`

- *Visualizing optimization results*

5.7.4 `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*)

Pairwise dependence plot of the objective function.

The diagonal shows the partial dependence for dimension `i` with respect to the objective function. The off-diagonal shows the partial dependence for dimensions `i` and `j` with respect to the objective function. The objective function is approximated by `result.model.`

Pairwise scatter plots of the points at which the objective function was directly evaluated are shown on the off-diagonal. A red point indicates per default the best observed minimum, but this can be changed by changing argument ´minimum´.

Parameters

result [`OptimizeResult`] The result for which to create the scatter plot matrix.

levels [int, default=10] Number of levels to draw on the contour plot, passed directly to `plt.contour()`.

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', ..]`.

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'.

Returns

ax [`Axes`] The matplotlib axes.

Examples using `skopt.plots.plot_objective`

- *Partial Dependence Plots*
- *Partial Dependence Plots with categorical values*
- *Visualizing optimization results*

5.7.5 `skopt.plots`.plot_regret

`skopt.plots.`**`plot_regret`**`(`**args*, ***kwargs*`)`

Plot one or several cumulative regret traces.

Parameters

args[i] [`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.

`utils.cook_estimator`(base_estimator[, space])	Cook a default estimator.
`utils.dimensions_aslist`(search_space)	Convert a dict representation of a search space into a list of dimensions, ordered by sorted(search_space.keys()).
`utils.expected_minimum`(res[, . . .])	Compute the minimum over the predictions of the last surrogate model.
`utils.expected_minimum_random_sampling`(res)	Minimum search by doing naive random sampling, Returns the parameters that gave the minimum function value.
`utils.dump`(res, filename[, store_objective])	Store an skopt optimization result into a file.
`utils.load`(filename, **kwargs)	Reconstruct a skopt optimization result from a file persisted with skopt.dump.
`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.
`utils.point_aslist`(search_space, point_as_dict)	Convert a dictionary representation of a point from a search space to the list representation.
`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.

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, 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", "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`.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

```
>>> 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')
```

5.8.3 `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.

5.8.4 `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.

5.8.5 `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`.

5.8.6 `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

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

5.8.7 `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

```
>>> 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)])
```

5.8.8 `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

```
>>> 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]
```

5.8.9 `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

```
>>> # 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.
>>> print("Best fitness:", result.fun)
Best fitness: 0.1948080835239698
>>> print("Best parameters:", 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.space.space`: Space

User guide: See the *Space define the optimization 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.

5.9.1 `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”, 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.

- “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

name

prior

size

transformed_bounds

transformed_size

Methods

distance(self, a, b)	Compute distance between category `a` and `b`.
inverse_transform(self, Xt)	Inverse transform samples from the warped space back into the original space.
rvs(self[, n_samples, random_state])	Draw random samples.

Continued on next page

Table 27 – continued from previous page

transform(self, X)	Transform samples form the original space to a warped space.

`__init__`(*self*, *categories*, *prior=None*, *transform=None*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

`distance`(*self*, *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.

Parameters

a [category] First category.

b [category] Second category.

`inverse_transform`(*self*, *Xt*)
Inverse transform samples from the warped space back into the original space.

`rvs`(*self*, *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.

`transform`(*self*, *X*)
Transform samples form the original space to a warped space.

5.9.2 `skopt.space.space`.Dimension

class `skopt.space.space.`**`Dimension`**
Base class for search space dimensions.

Attributes

bounds

name

prior

size

transformed_bounds

transformed_size

Methods

inverse_transform(self, Xt)	Inverse transform samples from the warped space back into the original space.
rvs(self[, n_samples, random_state])	Draw random samples.
transform(self, X)	Transform samples form the original space to a warped space.

`__init__`(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

`inverse_transform`(*self*, *Xt*)
Inverse transform samples from the warped space back into the original space.

`rvs`(*self*, *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`(*self*, *X*)
Transform samples form the original space to a warped space.

5.9.3 `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"`, intgers are sampled uniformly between the lower

and upper bounds.

- If `"log-uniform"`, intgers 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

name

prior

size

transformed_bounds

transformed_size

Methods

`distance`(self, a, b)	Compute distance between point `a` and `b`.
`inverse_transform`(self, Xt)	Inverse transform samples from the warped space back into the original space.
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, X)	Transform samples form the original space to a warped space.

`__init__`(*self, low, high, prior='uniform', base=10, transform=None, name=None, dtype=<class 'numpy.int64'>*)
Initialize self. See help(type(self)) for accurate signature.

`distance`(*self, a, b*)
Compute distance between point `a` and `b`.

Parameters

a [int] First point.

b [int] Second point.

`inverse_transform`(*self, Xt*)
Inverse transform samples from the warped space back into the original space.

`rvs`(*self, 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`(*self, X*)
Transform samples form the original space to a warped space.

5.9.4 `skopt.space.space`.Real

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(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 the dimension, e.g., "learning rate".

dtype [str or dtype, default=np.float] float type which will be used in inverse_transform, can be float.

Attributes

bounds

name

prior

size

transformed_bounds

transformed_size

Methods

`distance`(self, a, b)	Compute distance between point `a` and `b`.
`inverse_transform`(self, Xt)	Inverse transform samples from the warped space back into the original space.
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, X)	Transform samples form the original space to a warped space.

`__init__`(*self*, *low*, *high*, *prior='uniform'*, *base=10*, *transform=None*, *name=None*, *dtype=<class 'float'>*)
Initialize self. See help(type(self)) for accurate signature.

`distance`(*self*, *a*, *b*)
Compute distance between point `a` and `b`.

Parameters

a [float] First point.

b [float] Second point.

`inverse_transform`(*self*, *Xt*)
Inverse transform samples from the warped space back into the original space.

`rvs`(*self*, *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(*self*, *X*)
Transform samples form the original space to a warped space.

5.9.5 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.

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_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

`distance`(self, point_a, point_b)	Compute distance between two points in this space.
`from_yaml`(yml_path[, namespace])	Create Space from yaml configuration file
`inverse_transform`(self, Xt)	Inverse transform samples from the warped space back to the
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, X)	Transform samples from the original space into a warped space.

__init__(*self*, *dimensions*)
Initialize self. See help(type(self)) for accurate signature.

property bounds
The dimension bounds, in the original space.

`distance`(*self*, *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

`inverse_transform`(*self*, *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_dims`
The dimensionality of the original space.

`rvs`(*self*, *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.

transform(*self*, *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.

space.space.check_dimension(dimension[, ...])	Turn a provided dimension description into a dimension object.

5.9.6 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”, “onehot” optional]

- For `Categorical` dimensions, the following transformations are supported.
  - “onehot” (default) one-hot 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.

5.10 `skopt.space.transformers`: transformers

User guide: See the transformers section for further details.

space.transformers.CategoricalEncoder()	OneHotEncoder that can handle categorical variables.
space.transformers.Identity	Identity transform.
space.transformers.LogN(base)	Base N logarithm transform.
space.transformers.Normalize(low, high[, is_int])	Scales each dimension into the interval [0, 1].
space.transformers.Pipeline(transformers)	A lightweight pipeline to chain transformers.
space.transformers.Transformer	Base class for all 1-D transformers.

5.10.1 `skopt.space.transformers`.CategoricalEncoder

class skopt.space.transformers.**CategoricalEncoder**

OneHotEncoder that can handle categorical variables.

Methods

fit(self, X)	Fit a list or array of categories.
inverse_transform(self, Xt)	Inverse transform one-hot encoded categories back to their original
transform(self, X)	Transform an array of categories to a one-hot encoded representation.

__init__(*self*)

Convert labeled categories into one-hot encoded features.

fit(*self*, *X*)

Fit a list or array of categories.

Parameters

X [array-like, shape=(n_categories,)] List of categories.

inverse_transform(*self*, *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(*self*, *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 one-hot encoded categories.

5.10.2 skopt.space.transformers.Identity

class skopt.space.transformers.**Identity**

Identity transform.

Methods

fit	
inverse_transform	
transform	

__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

5.10.3 skopt.space.transformers.LogN

class skopt.space.transformers.**LogN**(*base*)

Base N logarithm transform.

Methods

fit	
inverse_transform	
transform	

__init__(*self*, *base*)

Initialize self. See help(type(self)) for accurate signature.

5.10.4 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=True] Round and cast the return value of `inverse_transform` to integer. Set to `True` when applying this transform to integers.

Methods

fit	
inverse_transform	
transform	

`__init__`(*self*, *low*, *high*, *is_int=False*)
Initialize self. See help(type(self)) for accurate signature.

5.10.5 `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

fit	
inverse_transform	
transform	

`__init__`(*self*, *transformers*)
Initialize self. See help(type(self)) for accurate signature.

5.10.6 `skopt.space.transformers`.Transformer

class `skopt.space.transformers.`**`Transformer`**
Base class for all 1-D transformers.

Methods

fit	
inverse_transform	
transform	

`__init__`(*self*, */*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

BIBLIOGRAPHY

[R8d4c5fa7c0c3-1] L. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

[R91c6cd8711c5-1] L. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

INDEX