HPOlib Documentation

Release 1

Matthias Feurer, Katharina Eggensperger

Sep 20, 2018

Contents

This package is discontinued. We have merged all changes that we have done since the initial release into the master branch, hoping that they are useful for some of you. The current software has several known bugs, which can be found in the issue tracker. In case someone wants to continue working on HPOlib, we're happy to accept and merge pull requests. If you're looking for a set of benchmarks, please use the predecessor HPOlib2. HPOlib2 does not contain any optimization packages. We will add a list of Bayesian optimization packages to the documentation of HPOlib2.

HPOlib is a package which aiming to simplify the development and use of hyperparameter optimization algorithms. It features benchmarks which have been used in papers introducing state-of-the-art hyperparameter optimization tools like spearmint and hyperopt. Furthermore, it provides a common interface to several Bayesian optimization packages as well as the possibility to add your own optimization package.

Contents:

CHAPTER 1

Installation Instructions For HPOlib

First:

```
git clone https://github.com/automl/HPOlib.git
```

1.1 Installing inside an virtualenv

1. Get virtualenv, then load a freshly created virtualenv. (If you are not familiar with virtualenv, you might want to read more about it)

```
pip install virtualenv
virtualenv virtualHPOlib
source virtualHPOlib/bin/activate
```

3. Install `numpy`, `scipy`, `matplotlib`, as this doesn't work through setup.py.

```
pip install numpy
pip install scipy
pip install matplotlib
```

   This may take some time. Afterwards you can verify having those libs installed with:

```
pip freeze
```

4. run setup.py

```
python setup.py install
```

   This will install HPOlib and some requirements (`networkx`, `protobuf`, `pymongo`). Be sure your system is **connected to the internet**, so `setup.py` can download optimizer and runsolver code. Your environment now looks like that

```
pip freeze

    HPOlib==0.0.1
    argparse==1.2.1
    backports.ssl-match-hostname==3.4.0.2
    distribute==0.7.3
    matplotlib==1.3.1
    networkx==1.8.1
    nose==1.3.0
    numpy==1.8.0
    protobuf==2.5.0
    pymongo==2.6.3
    pyparsing==2.0.1
    python-dateutil==2.2
    scipy==0.13.3
    six==1.5.2
    tornado==3.2
    wsgiref==0.1.2
```

and

```
ls optimizers/smac
    smac_2_10_00-dev_parser.py   smac_2_10_00-dev.py   smac_2_10_00-dev_src   ␣
↪smac_2_10_00-devDefault.cfg
```

5. You can now run, e.g. smac with 200 evaluations on the branin function:

```
cd benchmarks/branin
HPOlib-run -o ../../optimizers/smac/smac_2_10_00-dev -s 23
```

This takes depending on your machine ~2 minutes. You can now plot the results of your first experiment:

```
HPOlib-plot FIRSTRUN smac_2_10_00-dev_23_*/smac_*.pkl -s `pwd`/Plots/
```

You can test the other optimizers (spearmint will take quite longer 30min):

```
HPOlib-run -o ../../optimizers/tpe/h -s 23
HPOlib-run -o ../../optimizers/spearmint/spearmint_april2013 -s 23
```

and again:

```
HPOlib-plot SMAC smac_2_10_00-dev_23_*/smac_*.pkl TPE hyperopt_august2013_mod_23_
↪*/hyp*.pkl SPEARMINT spearmint_april2013_mod_23_*/spear*.pkl -s `pwd`/Plots/
```

and to check the general performance on this super complex benchmark:

```
HPOlib-plot BRANIN smac_2_10_00-dev_23_*/smac_*.pkl hyperopt_august2013_mod_23_*/
↪hyp*.pkl spearmint_april2013_mod_23_*/spear*.pkl -s `pwd`/Plots/
```

Problems during installation

`python setup.py` crashes with `ImportError:  cannot import name Feature` during installing pymongo. This happens due to pymongo using a deprecated feature :python:'Feature', which is not available in the setuptools version (>2.2). This error is fixed, but not yet available on PYPI.

Solution: Downgrade `setuptools` with `pip install setuptools==2.2` and try again or install `pymongo` manually.

Updating optimizers We also provide an updated and adjusted version of spearmint. To also install this version do:

```
cd optimizers
rm spearmint_gitfork_mod_src
git clone https://github.com/automl/spearmint.git
mv spearmint spearmint_gitfork_mod_src
```

CHAPTER 2

Algorithms and Datasets

2.1 Benchmarks Overview

To run these algorithms and datasets with hyperparameter optimizers you need to install

1. the **HPOlib** software from *here*
2. the benchmark data: An algorithm and depending on the benchmark a wrapper and/or data

Then the benchmarks can easily be used, as described *here*; Our software allows to integrate your own benchmarks as well. Here is the *HowTo*.

NOTE: For all bechmarks crossvalidation is possible, but not extra listed. Although possible, it obviously makes no sense to do crossvalidation on functions like Branin and pre-computed results like the LDA ongrid. Whether it makes sense to do so is indicated in the column CV.

2.2 Description

2.2.1 Branin, RKHS, Hartmann 6d, Michalewicz and Camelback Function

This benchmark already comes with the basic *HPOlib* bundle.

Dependencies: None **Recommended:** None

Branin, RKHS, Camelback, Michalewicz and the Hartmann 6d function are five simple test functions, which are easy and cheap to evaluate. More test functions can be found here.

Branin has three global minima at (-pi, 12.275), (pi, 2.275), (9.42478, 2.475) where f(x)=0.397887.

RKHS has single global minima at x=0.89235 where f(x)=5.73839.

Camelback has two global minima at (0.0898, -0.7126) and (-0.0898, 0.7126) where f(x) = -1.0316

Hartmann 6d is more difficult with 6 local minima and one global optimum at (0.20169, 0.150011, 0.476874, 0.275332, 0.311652, 0.6573) where f(x)=3.32237.

Michalewicz is usually evaluated on the hypercube x_i [0, pi], for all i = 1, . . . , d. For d=10 its global minima value is f(x) = -9.66015.

2.2.2 LDA ongrid/SVM ongrid

This benchmark already comes with the basic *HPOlib* bundle.

Dependencies: None **Recommended:** None

Online Latent Dirichlet Allocation (LDA) is a very expensive algorithm to evaluate. To make this less time consuming, a 6x6x8 grid of hyperparameter configurations resulting in 288 data points was preevaluated. This grid forms the search space.

Same holds for the Support Vector Machine task, which has 1400 evaluated configurations.

The Online LDA code is written by Hoffman et. al. and the procedure is explained in Online Learning for Latent Dirichlet Allocation. Latent Structured Support Vector Machine code is written by Kevin Mill et. al. and explained in the paper Max-Margin Min-Entropy Models. The grid search was performed by Jasper Snoek and previously used in Practical Bayesian Optimization of Machine Learning Algorithms.

Logistic Regression

Dependencies: theano, scikit-data **Recommended:** CUDA

NOTE: *scikit-data* downloads the dataset from the internet when using the benchmark for the first time. **NOTE:** This benchmarks can use a gpu, but this feature is switched off to run it off-the-shelf. To use a gpu you need to change the THEANO flags in `config.cfg`. See the *HowTo* for changing to gpu and for further information about the THEANO configuration here **NOTE:** In order to run the benchmark you must adjust the paths in the config files.

You can download this benchmark by clicking here or running this command from a shell:

```
wget http://www.automl.org/logreg.tar.gz
tar -xf logistic.tar.gz
```

This benchmark performs a logistic regression to classifiy the popular MNIST dataset. The implementation is Theano based, so that a GPU can be used. The software is written by Jasper Snoek and was first used in the paper Practical Bayesian Optimization of Machine Learning Algorithms.

NOTE: This benchmark comes with the version of hyperopt-nnet which we used for our experiments. There might be a newer version with improvements.

HP-NNet and HP-DBNet

Dependencies: theano, scikit-data **Recommended:** CUDA

NOTE: This benchmark comes with the version of hyperopt-nnet which we used for our experiments. There might be a newer version with improvements. **NOTE:** *scikit-data* downloads the dataset from the internet when using the benchmark for the first time. **NOTE:** In order to run the benchmark you must adjust the paths in the config files.

You can download this benchmark by clicking here or running this command from a shell:

The HP-Nnet (HP-DBNet) is a Theano based implementation of a (deep) neural network. It can be run on a CPU, but is drastically faster on a GPU (please follow the theano flags instructions of the *logistic regression* example). Both of them are written by James Bergstra and were used in the papers Random Search for Hyper-Parameter Optimization and Algorithms for Hyper-Parameter Optimization.

AutoWEKA

NOTE: AutoWEKA is not yet available for download!

CHAPTER 3

Manual

3.1 How to run listed benchmarks

After having succesfully installed the basic **HPOlib** you can download more benchmarks or create your own. Each benchmarks resides in an own directory and consists of an algorithm (+ wrapper if necessary), a configuration file and several hyperparameter configuration descriptions. If you want to use one of the benchmarks listed here, follow these steps:

Let's say you want to run the Reproducing Kernel Hilbert space (RKHS) function:

1. RKHS is located with other benchmarks inside `HPOlib/benchmarks` folder. To run the benchmark first go inside that folder.

   ```
   cd HPOlib/benchmarks/rkhs
   ```

2. Inside this folder you can run one the optimizers (smac, tpe or spearmint) on RKHS function using HPOlib :

   ```
   HPOlib-run -o ../../optimizers/smac/smac_2_10_00-dev -s 23
   HPOlib-run -o ../../optimizers/tpe/h -s 23
   HPOlib-run -o ../../optimizers/spearmint/spearmint_april2013 -s 23
   ```

   Or more generally

   ```
   HPOlib-run /path/to/optimizers/<tpe/hyperopt|smac|spearmint|tpe/random> [-s seed]␣
   ↪[-t title]
   ```

   By default, the optimizers will run 200 evaluations on the function. For smac and tpe this will take about 2 mins but for spearmint it will be longer than 45 mins, so change `number_of_jobs` parameter in `config.cfg` file in same folder to 50 or less.

   ```
   [SMAC]
   p = params.pcs

   [TPE]
   ```

(continues on next page)

(continued from previous page)

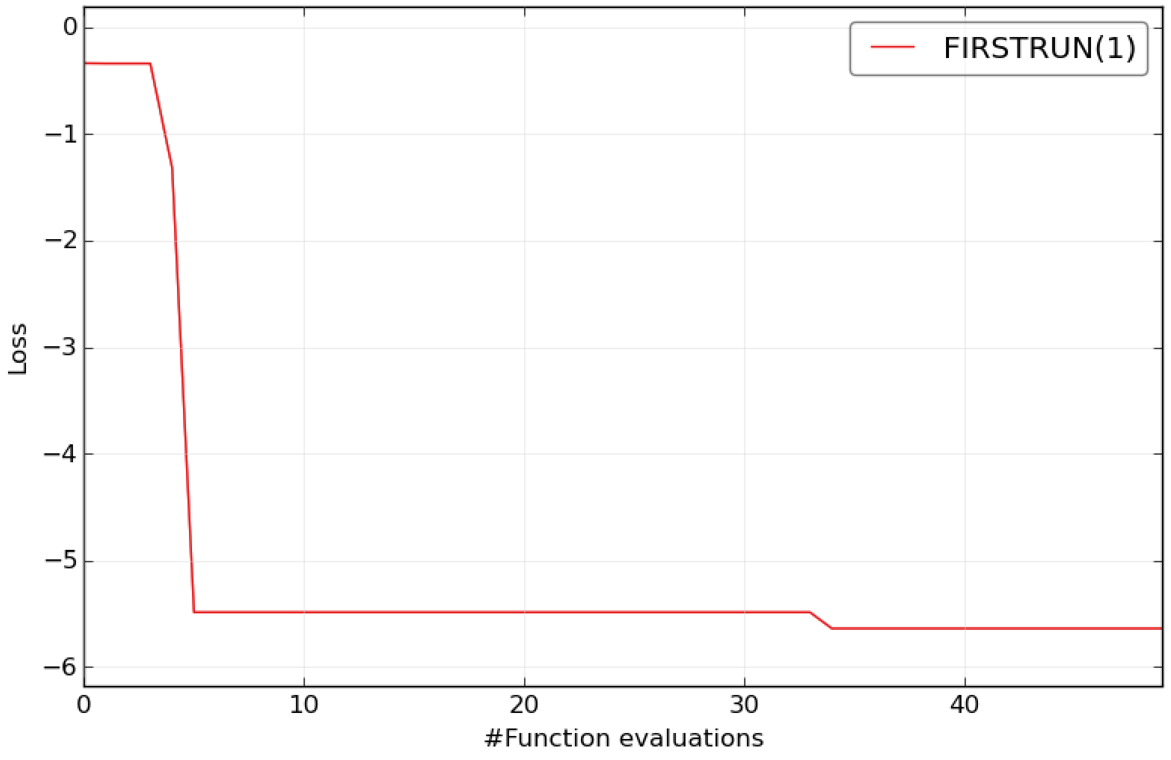
```
space = space.py

[SPEARMINT]
config = config.pb

[HPOLIB]
console_output_delay = 2.0
function = python ../rkhs.py
number_of_jobs = 200 #Change this to 50.
result_on_terminate = 1000
```

3. Now you can plot results for the experiment in different ways:

   Plot the results of only one optimizer:

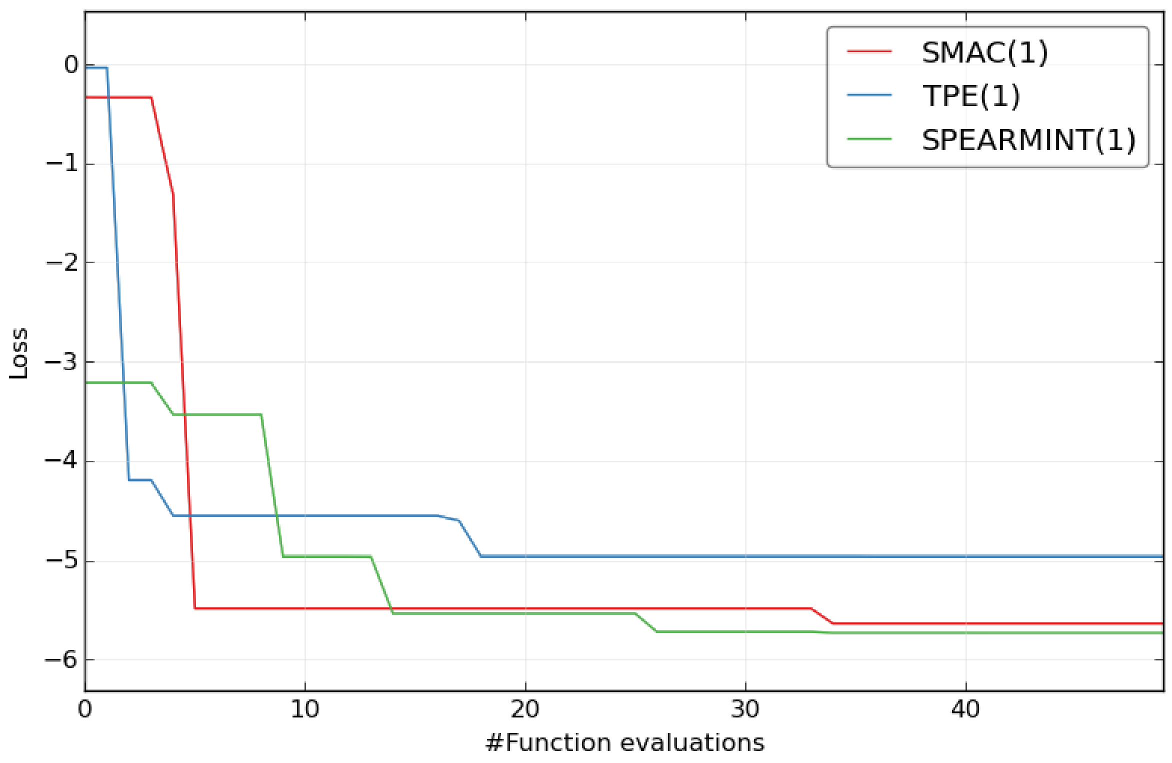
```
HPOlib-plot FIRSTRUN smac_2_10_00-dev_23_*/smac_*.pkl -s `pwd`/Plots/
```

   The Plots can be found inside folder named `Plots` in current working directory (`HPOlib/benchmarks/rkhs`)



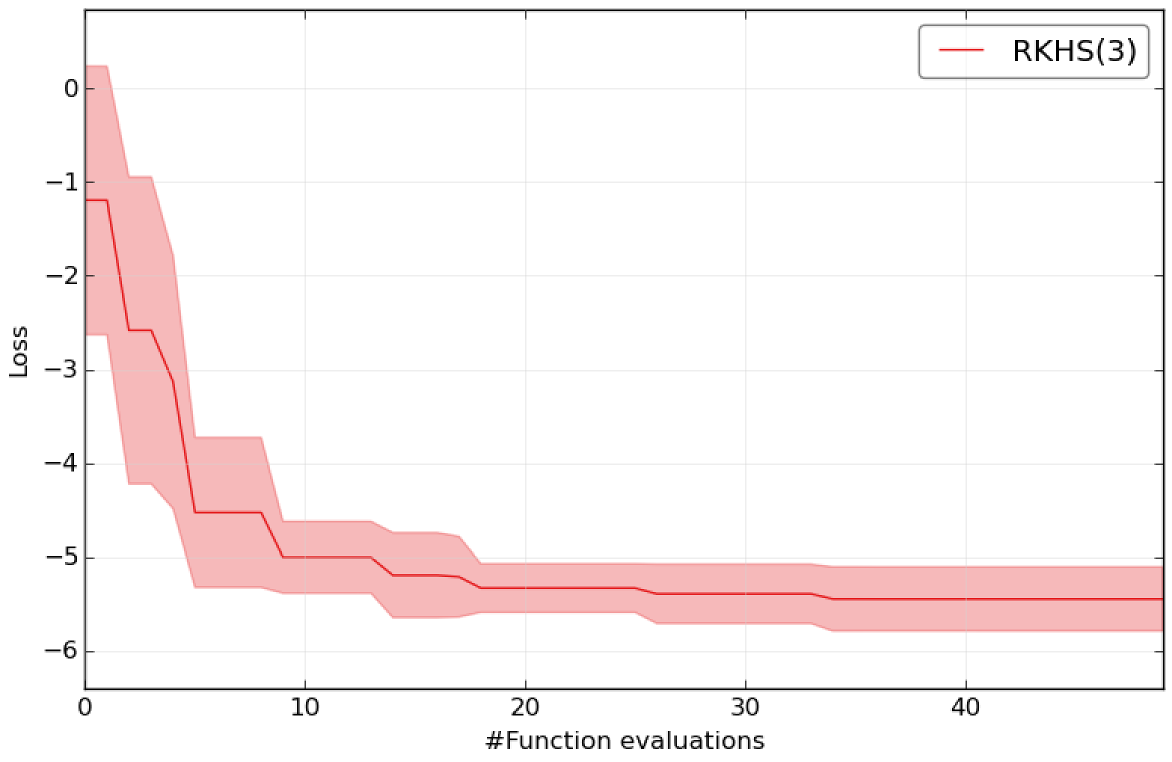
   and if you have run all optimizers and want to compare their results:

```
HPOlib-plot SMAC smac_2_10_00-dev_23_*/smac_*.pkl TPE hyperopt_august2013_mod_23_
↪*/hyp*.pkl SPEARMINT spearmint_april2013_mod_23_*/spear*.pkl -s `pwd`/Plots/
```



and to check the general performance on this super complex benchmark:

```
HPOlib-plot RKHS smac_2_10_00-dev_23_*/smac_*.pkl hyperopt_august2013_mod_23_*/
↪hyp*.pkl spearmint_april2013_mod_23_*/spear*.pkl -s `pwd`/Plots/
```



3.2 How to run your own benchmarks

To run your own benchmark you basically need the software for the benchmark and a search space description for the optimizers smac, spearmint and tpe. In order to work with HPOlib you must put these files into a special directory structure. It is the same directory structure as for the benchmarks which you can download on this website and is explained in the list below. The following lines will guide you through the creation of such a benchmark. Here is a rough guide on what files you need:

- One **directory** having the name of the optimizer for each optimizer you want to use. Currently, these are `hyperopt_august2013_mod`, `random_hyperopt2013_mod`, `smac_2_10_00-dev` and `spearmint_april2013_mod`.
- One **search space** for each optimizer. This must be placed in the directory with the name of the optimizer. You can convert your searchspace to other formats with *HPOlib_convert* from and to all three different optimizers.
- An **executable** which implements the HPOlib interface. Alternatively, this can be a wrapper which parser the command line arguments, calls your target algorithm and returns the result to the HPOlib.
- A **configuration file** *config.cfg*. See the section on *configuring the HPOlib* for details.

3.2.1 Example

First, create a directory `myBenchmark` inside the `HPOlib/benchmarks` directory. The executable `HPOlib/benchmarks/myBenchmark/myAlgo.py` with the target algorithm can be as easy as

```python
import math
import time

import HPOlib.benchmark_util as benchmark_util

def myAlgo(params, **kwargs):
    # Params is a dict that contains the params
    # As the values are forwarded as strings you might want to convert and check them

    if not params.has_key('x'):
        raise ValueError("x is not a valid key in params")

    x = float(params["x"])

    if x < 0 or x > 3.5:
        raise ValueError("x not between 0 and 3.5: %s" % x)

    # **kwargs contains further information, like
    # for crossvalidation
    #    kwargs['folds'] is 1 when no cv
    #    kwargs['fold'] is the current fold. The index is zero-based

    # Run your algorithm and receive a result, you want to minimize
    result = -math.sin(x)

    return result

if __name__ == "__main__":
    starttime = time.time()
    # Use a library function which parses the command line call
    args, params = benchmark_util.parse_cli()
```

(continues on next page)

(continued from previous page)

```
    result = myAlgo(params, **args)
    duration = time.time() - starttime
    print "Result for this algorithm run: %s, %f, 1, %f, %d, %s" % \
        ("SAT", abs(duration), result, -1, str(__file__))
```

As you can see, the script parses command line arguments, calls the target function which is implemented in myAlgo, measures the runtime of the target algorithm and prints a return string to the command line. All relevant information is then extracted by the HPOlib. If you write a new algorithm/wrapper script, you must parse the following call:

```
target_algorithm_executable --fold 0 --folds 1 --params [ [ -param1 value1 ] ]
```

The return string must take the following form:

```
Result for this algorithm run: SAT, <duration>, 1, <result>, -1, <additional␣
↪information>
```

This return string is not yet optimal and exists for historic reasons. It's subject to change in one of the next versions of HPOlib.

Next, create `HPOlib/benchmarks/myBenchmark/config.cfg`, which is the configuration file. It tells the HPOlib everything about the benchmark and looks like this:

```
[TPE]
space = mySpace.py

[HPOLIB]
function = python ../myAlgo.py
number_of_jobs = 200
# worst possible result
result_on_terminate = 0
```

Since the hyperparameter optimization algorithm must know about the variables and their possible values for your target algorithms, the next step is to specify these in a so-called search space. Create a new directory `hyperopt_august2013_mod` inside the `HPOlib/benchmarks/myBenchmark` directory and save these two lines of python in a file called `mySpace.py`. If you look at the `config.cfg`, we already the use of the newly created search space. As problems get more complex, you may want to specify more complex search spaces. It is recommended to do this in the TPE format, then translate it into the SMAC format which can then be translated into the spearmint format. More information on how to write search spaces in the TPE format can be found in this paper and the hyperopt wiki.

```
from hyperopt import hp
space = {'x': hp.uniform('x', 0, 3.5)}
```

Now you can run your benchmark with tpe. The command (which has to be executed from `HPOlib/benchmarks/myBenchmark`) is

```
HPOlib-run -o ../../optimizers/tpe/hyperopt_august2013_mod
```

Further you can run your benchmark with the other optimizers:

```
mkdir smac
python path/to/hpolib/format_converter/TpeSMAC.py tpe/mySpace.py >> smac/params.pcs
python path/to/wrapping.py smac
mkdir spearmint
python path/to/hpolib/format_converter/SMACSpearmint.py >> spearmint/config.pb
python path/to/wrapping.py spearmint
```

3.3 Configure the HPOlib

The *config.cfg* is a file, which contains necessary settings about your experiment. It is designed such that as little as possible information needs to be given. This means all values for optimizers and the wrapping software are set to the default values, except you want to change them. Default values are stored in a file called `config_parser/generalDefault.cfg`. The following table describes the values you must provide: The file is divided into sections. You only need to fill in values for the [HPOLIB] section.

Key	Description
function	The executeable for the target algorithm. The path can either be either absolute or relative to an optimizer directory in your benchmark folder (if the executeable is not found you can try to prepend the parent directory to the path)
number_of_jobs	number of evaluations that are performed by the optimizers. **NOTE**:When using k-fold-crossvalidation, SMAC will use `k * number_of_jobs evaluations`
result_on_terminate	If your algorithms crashes, is killed, takes too long etc. This result is given to the optimizer. Should be the worst possible, but realistic result for a problem

An example can be found in the section [adding your own benchmark](manual.html#config_example). The following parameters can be specified:

Section	Parameter	Default value	Description
HPOLIB	number_cv_folds	`1`	number of folds for a crossvalidation
HPOLIB	max_crash_per_cv	`3`	If some runs of the crossvalidation fail, stop the crossvalidation for this configuration after max_crash_per_cv failed folds.
HPOLIB	remove_target_algorithm_output	`True`	Per default, the target algorithm output is deleted. Set to False to keep the output. This is useful for debugging.
HPOLIB	console_output_delay	`1.0`	HPOlib reads the experiment pickle periodically to print the current status to the command line interface. Doing this often can inhibit performance of your hard-drive (espacially if perform a lot of HPOlib experiments in parallel) so you might want to increase this number if you experience delay when accessing your hard drive.
HPOLIB	runsolver_time_limit, memory_limit, cpu_limit		Enforce resource limits to a target algorithm run. If these limits are exceeded, the target algorithm will be killed by the runsolver. This can be used to ensure e.g. a runtime per algorithm or make sure an algorithm does not use too much space on a computing cluster.
HPOLIB	total_time_limit		Enforce a total time limit on the hyperparameter optimization.
HPOLIB	leading_runsolver_info		Important when using THEANO and CUDA, see *Configure theano for gpu and openBlas usage*
HPOLIB	use_HPOlib_time_measurement	`True`	When set to True (the default), the runsolver time measurement is saved. Otherwise, the time measured by the target algorithm is saved.
HPOLIB	number_of_concurrent_jobs	`1`	WARNING: this only works for spearmint and SMAC and is not tested!
HPOLIB	function_setup		An executable which is called before the first target algorithm call. This can be for example check if everything is installed properly.
HPOLIB	function_teardown		An executable which is called after the last target algorithm call. This can be for example delete temporary directories.
HPOLIB	experiment_directory_prefix		Adds a prefix to the automatically generated experiment directory. Can be useful if one experiments is run several times with different parameter settings.
HPOLIB	handles_cv		This flag determines whether optimization_interceptor or the optimizer handles cross validation. This is only set to 1 for SMAC and must only be used by optimization algorithm developers.

The following keys change the behaviour of the integrated hyperparameter optimization packages:

Section	Parameter	Default value Description	
TPE	space	`space.py`	Name of the search space for tpe
TPE	path_to_optimizer	`./hyperopt_august2013_mod_src`	Please consult the SMAC documentation.
SMAC	p	`smac/params.pcs`	Please consult the SMAC documentation.
SMAC	run_obj	`QUALITY`	Please consult the SMAC documentation.
SMAC	intra_instance_obj	`MEAN`	Please consult the SMAC documentation.
SMAC	rf_full_tree_bootstrap	`False`	Please consult the SMAC documentation.
SMAC	rf_split_min	`10`	Please consult the SMAC documentation.
SMAC	adaptive_capping	`false`	Please consult the SMAC documentation.
SMAC	max_incumbent_runs	`2000`	Please consult the SMAC documentation.
SMAC	num_iterations	`2147483647`	Please consult the SMAC documentation.
SMAC	deterministic	`True`	Please consult the SMAC documentation.
SMAC	retry_target_algorithm_run_count	`0`	Please consult the SMAC documentation.
SMAC	intensification_percentage	`0`	Please consult the SMAC documentation.
SMAC	validation	`false`	Please consult the SMAC documentation.
SMAC	path_to_optimizer	`./smac_2_06_01-dev_src`	Please consult the SMAC documentation.
SPEARMINT	config	`config.pb`	
SPEARMINT	method	`GPEIOptChooser`	The spearmint chooser to be used. Please consult the spearmint documentation for possible choices. WARNING: Only the GPEIOptChooser is tested!
SPEARMINT	method_args		Pass arguments to the chooser method. Please consult the spearmint documentation for possible choices.
SPEARMINT	grid_size	`20000`	Length of the Sobol sequence spearmint uses to optimize the Expected Improvement.
SPEARMINT	spearmint_polling_time	`3.0`	Spearmint reads its experiment pickle and checks for finished jobs periodically to find out whether a new job has to be started. For very short functions evaluations, this value can be decreased. Bear in mind that this puts load on your hard drive and can slow down your system if the experiment pickle becomes large (e.g. for the AutoWeka benchmark) or you run a lot of parallel jobs (>100).
SPEARMINT	path_to_optimizer	`./spearmint_april2013_mod_src`	

The config parameters can also be set via the command line. A use case for this feature is to run the same experiment multiple times, but with different parameters. The syntax is:

```
HPOlib-run -o spearmint/spearmint_april2013_mod --SECTION:argument value
```

To set for example the spearmint grid size to 40000, use the following call

```
HPOlib-run -o spearmint/spearmint_april2013_mod --SPEARMINT:grid_size 40000
```

If your target algorithm is a python script, you can also load the config file from within your target algorithm. This allows you to specify extra parameters for your target algorithm in the config file. Simply import `HPOlib.wrapping_util` in your python script and call `HPOlib.wrapping_util.load_experiment_config_file()`. The return value is a python config parser object.

3.3.1 Configure theano for gpu and openBlas usage

The THEANO-based benchmarks can be speed-up by either running them on a nvidia GPU or with an optimized BLAS library. Theano is either configured with theano flags, by changing the value of a variable in the target program (not recommended as you have to change source code) or by using a `.theanorc` file. The `.theanorc` file is good for global configurations and you can find more information on how to use it on the theano config page. For a more fine-grained control of theano you have to use theano flags.

Unfortunately, setting them in the shell before invoking `HPOlib-run` does not work and therefore these parameters have to be added set via the config variable `leading_runsolver_info`. This is already set to a reasonable default for the respective benchmarks but has to be changed in order to speed up calculations.

For openBlas, change the paths in the following paragraph and replace the value of the config variable `leading_runsolver_info`. In case you want to change more of the theano behaviour (e.g. the compile directory) you must append these flags to the config variable.

```
OPENBLAS_NUM_THREADS=2 LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/the/openBLAS/lib␣
↪LIBRARY_PATH=$LIBRARY_PATH:/path/to/the/openBLAS/lib THEANO_FLAGS=floatX=float32,
↪device=cpu,blas.ldflags=-lopenblas
```

If you want to use CUDA on your nvidia GPU, you have to change `device=cpu` to `device=gpu` and add `cuda.root=/usr/local/cuda` to the THEANO flags. Change `cuda.root` to your cuda installation directory if you did not install cuda to the default location. For that, replace the path `cuda.root=/usr/local/cuda` with the path to your CUDA installation.

3.4 How to run your own optimizer

Before you integrate your own optimization algorithm, make sure that you know how the HPOlib is structured and read the section *The HPOlib Structure*. The interface to include your own optimizer is straight-forward. Let's assume that you have written a hyperparameter optimization package called BayesOpt2. You tell the HPOlib to use your software with the command line argument `-o` or `--optimizer`. A call to `HPOlib-run -o /path/to/BayesOpt2` should the run an experiment with your newly written software.

But so far, the HPOlib does not know how to call your software. To let the HPOlib know about the interface to your optimizer, you need to create the three following files (replace BayesOpt2 if your optimization package has a different name):

- **BayesOpt2.py: will create all files your optimization package needs in order** to run
- **BayesOpt2_parser.py: a parser which can change the configuration of your** optimization algorithm based
  on HPOlib defaults
- BayesOpt2Default.cfg: default configuration for your optimization algorithm

Moreover, your algorithm has to call a script of the HPOlib namely `optimization_interceptor.py`, which does bookkeeping and manages a potential cross validation. The rest of this section will explain how to call `optimization_interceptor.py` and the interface your scripts must provide and the functionality which they must perform.

3.4.1 Calling `optimization_interceptor.py`

3.4.2 BayesOpt2.py

To run BayesOpt2, HPOlib will call the main function of the script `bayesopt2.py`. The function signature is as follows:

```
(call_string, directory) = optimizer_module.main(config=config, options=args,␣
↪experiment_dir=experiment_dir, experiment_directory_prefix=experiment_directory_
↪prefix)
```

Argument `config` is of type ConfigParser, `options` of type ArgumentParser and `experiment_dir` is a string to the experiment directory. The return value is a tuple `(call_string, directory)`. `call_string` must be a valid (bash) shell command which calls your hyperparameter optimization package in the way you intend. You can construct the call string based on the information in the config and the options you are provided with. `directory` must be a new directory in which all experiment output will be stored. `HPOlib-run` will the change in to the output directory which your function returned and execute the call string. Your script must therefore do the following in the `main` function:

1. Set up an experiment directory and return the path to the experiment directory. It is highly recommended to create a directory with the following name:

   ```
   <experiment_directory_prefix><bayesopt2><time_string>
   ```

2. Return a valid bash shell command, which will be used to call your optimizer from the command line interface. The target algorithm you want to optimize is mostly called `optimization_interceptor.py`, except for SMAC which handles crossvalidation on its own. Calling `optimization_interceptor.py` allows optimizer independend bookkeeping. The actual function call is the invoked by the HPOlib. Its interface is

   ```
   python optimization_interceptor.py -param_name1 'param_value' -x '5' -y '3.0'`
   ```

   etc... The function simply prints the loss to the command line. If your hyperparameter optimization package is written in python, you can also directly call the method `doForTPE(params)`, where the params argument is a dictionary with all parameter values (both key and value being strings).

Have a look at the bundled scripts `smac_2_06_01-dev.py`, `spearmint_april2013_mod.py` and `hyperopt_august2013_mod.py` to get an idea what can/must be done.

3.4.3 BayesOpt2_parser.py

The parser file implements a simple interface which only allows the manipulation of the config file:

```
config = manipulate_config(config)
```

See the python documentation for the documentation of the config object. Common usage of `manipulate_config` is to check if mandatory arguments are provided. This is also the recommended place to convert values from the HPOLIB section to the appropriate values of the optimization package.

3.4.4 BayesOpt2Default.cfg

A configuration file for your optimization package as described in the *configuration section*.

3.5 Convert Search Spaces

3.6 Test/Validate the Best Configuration(s)

To get an unbiased performance estimate of the best configuration(s) found, HPOlib offers a script to run a test function with these configurations. The scripts is called like:

```
HPOlib-testbest --all|--best|--trajectory --cwd path/to/the/optimization/directory
```

HPOlib-testbest will open the experiment pickle file which is used for HPOlib bookkeeping, extract the hyperparameters for the best configuration and call the test function specified in the configuration file. The result of the test function is then stored in the experiment pickle and can be further processed. The first argument (either `--all`, `--best` or `--trajectory` determines for which configurations the HPOlib will call the test script.

- `--all`: Will call the test-script for all configurations. This is can be very expensive.
- `--best`: Call the test-script only for the best configuration.
- `--trajectory`: Not yet implemented!

The second argument `--cwd` tells HPOlib in which experiment directory it should run test the configurations. As an example, consider the usecase that we ran SMAC to optimize the *logistic regression* and want to get the test performance for the best configuration.

```
HPOlib-testbest --best --cwd logreg/nocv/smac_2_08_00-master_2000_2014-11-7--16-49-28-
↪166127/
```

Further options are:

- `--redo-runs`: If argument is given, previous runs will be executed again and the previous results will be overwritten.
- `--n-jobs`: Number of parallel function evaluations. You should not set this number higher than the number of cores in your computer.

3.7 Dispatchers: Different ways to invoke the Target Algorithm

3.7.1 Runsolver Wrapper

3.7.2 Python Function

CHAPTER 4

Optimization algorithms

HPOlib ships several optimization packages by default. These are:

- *ConfigurationRunner* Executes configurations which are saved in a csv file.
- **SMAC v2.06.01** Includes the ROAR and SMAC algorithm (Hutter et al., 2011).
- **SMAC v2.08.00** Includes the ROAR and SMAC algorithm (Hutter et al., 2011).
- **SMAC v2.10.00** Includes the ROAR and SMAC algorithm (Hutter et al., 2011).
- **Spearmint (github clone from april 2013)** Performs Bayesian optimization with Gaussian Processes as described in Snoek et al. (2012).
- **Hyperopt (github clone from august 2013)** Includes random search (Bergstra and Bengio, 2012) and the Tree Parzen Estimator (Bergstra et al., 2011)

4.1 Configuration Runner

The *ConfigurationRunner* is an optimizer which runs configurations saved in a csv file. It is useful to evaluate configurations which do not come from an optimization algorithm and still benefit from HPOlib's functionality.

By default, it expects a csv file called *configurations* as input. The first line determines the names of the hyperparameters, every following line determines a single configuration.

The following is an example file for the branin function:

```
x,y
0,0
1,1
2,2
3,3
4,4
5,5
6,6
7,7
```

(continues on next page)

(continued from previous page)

```
8,8
9,9
10,10
```

WARNING: *ConfigurationRunner* does not check if the configurations adhere to any configuration space. This must be done by the user.

Furthermore, *ConfigurationRunner* can execute the function evaluations in parallel. This is governed by the argument *n_jobs* and only useful if the target machine has enough processors/cores or the jobs are distributed across several machines.

CHAPTER 5

Plotting results

5.1 Exporting results

To process results with programming languages different than python we provide a script called `HPOlib-export`, which can convert HPOlib experiment pickles into different formats:

```
HPOlib-export input output [-t|--type output_type]
```

5.1.1 Example

```
HPOlib-export benchmarks/branin/smac_2_06_01-dev_1_2014-11-24--16-6-19-290280/smac_2_
↪06_01-dev.pkl output/smac_branin_seed1 -t json
```

The output looks something like this:

```
{"instance_order": [[0, 0], [1, 0], [2, 0], [3, 0], [4, 0], [5, 0], [6, 0], [7, 0],␣
↪[8, 0], [9, 0]], "cv_endtime": [1416846588.037684, 1416846588.714215, 1416846589.
↪185275, 1416846589.71545, 1416846590.240511, 1416846590.645061, 1416846591.157578,␣
↪1416846591.588725, 1416846592.075068, 1416846592.565032], "optimizer_time": [],
↪"title": null, "folds": 1, "total_wallclock_time": 89.56732000000001, "trials": [{
↪"status": 3, "std": 0.0, "test_additional_data": {"0": "../logreg.py"}, "test_
↪duration": 3.9904310000000001, "instance_results": [0.0906], "test_std": 0.0,
↪"additional_data": {"0": "../logreg.py"}, "test_instance_durations": [3.
↪9904310000000001], "params": {"batchsize": "0", "l2_reg": "0", "lrate": "0", "n_
↪epochs": "0"}, "result": 0.0906, "test_instance_status": [3], "duration": 3.
↪9904310000000001, "test_status": 3, "test_result": 0.0906, "test_instance_results":␣
↪[0.0906], "instance_status": [3], "instance_durations": [3.9904310000000001]}, {
↪"status": 3, "std": 0.0, "test_additional_data": {"0": "../logreg.py"}, "test_
↪duration": 2.6245590000000001, "instance_results": [0.20121], "test_std": 0.0,
↪"additional_data": {"0": "../logreg.py"}, "test_instance_durations": [2.
↪6245590000000001], "params": {"batchsize": "4", "l2_reg": "6", "lrate": "3", "n_
↪epochs": "0"}, "result": 0.20121, "test_instance_status": [3], "duration": 2.
↪6245590000000001, "test_status": 3, "test_result": 0.20121, "test_instance_results
↪": [0.20121], "instance_status": [3], "instance_durations": [2.6245590000000001]}, {
↪"status": 3, "std": 0.0, "test_additional_data": {"0": "../logreg.py"}, "test_
↪duration": 3.3856489999999999, "instance_results": [0.15843699999999999], "test_std
↪": 0.0, "additional_data": {"0": "../logreg.py"}, "test_instance_durations": [3.
↪3856489999999999], "params": {"batchsize": "5", "l2_reg": "2", "lrate": "3", "n_
↪epochs": "1"}, "result": 0.15843699999999999, "test_instance_status": [3], "duration
↪": 3.3856489999999999, "test_status": 3, "test_result": 0.15843699999999999, "test
```

(continues on next page)

(continued from previous page)

```
```

Currently supported output types/formats are:

- json

CHAPTER 6

The HPOlib Structure

To be written…

CHAPTER 7

Citing the HPOlib

If you use the HPOlib for your research, please cite our paper introducing the HPOlib:

Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters[pdf] [poster]

NIPS Workshop on Bayesian Optimization in Theory and Practice (BayesOpt '13)

with the following Bibtex file:

```
@inproceedings{eggensperger2013,
    title = {Towards an Empirical Foundation for Assessing Bayesian Optimization of␣
↪Hyperparameters},
    booktitle = {{NIPS} workshop on Bayesian Optimization in Theory and Practice},
    author = {Eggensperger, K. and Feurer, M. and Hutter, F. and Bergstra, J. and␣
↪Snoek, J. and Hoos, H. and Leyton-Brown, K.},
    year = {2013}
}
```

CHAPTER 8

Indices and tables

- search