BO Benchmark Documentation

Uber AI Labs

Sep 10, 2020

CONTENTS

Contents:

CHAPTER

ONE

INSTALLATION

This project provides a benchmark framework to easily compare Bayesian optimization methods on real machine learning tasks.

This project is experimental and the APIs are not considered stable.

This Bayesian optimization (BO) benchmark framework requires a few easy steps for setup. It can be run either on a local machine (in serial) or prepare a *commands file* to run on a cluster as parallel experiments (dry run mode).

Only `Python>=3.6` is officially supported, but older versions of Python likely work as well.

The core package itself can be installed with:

```
pip install bayesmark
```

However, to also require installation of all the "built in" optimizers for evaluation, run:

```
pip install bayesmark[optimizers]
```

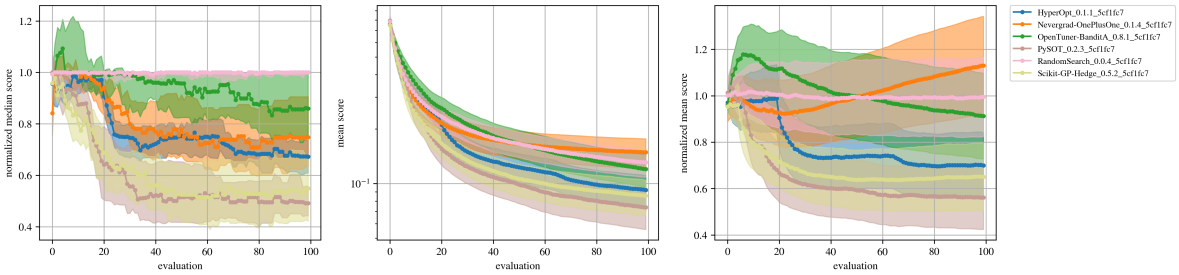
It is also possible to use the same pinned dependencies we used in testing by *installing from the repo*.

Building an environment to run the included notebooks can be done with:

```
pip install bayesmark[notebooks]
```

Or, `bayesmark[optimizers,notebooks]` can be used.

A quick example of running the benchmark is *here*. The instructions are used to generate results as below:



1.1 Non-pip dependencies

To be able to install `opentuner` some system level (non-pip) dependencies must be installed. This can be done with:

```
sudo apt-get install libsqlite3-0
sudo apt-get install libsqlite3-dev
```

On Ubuntu, this results in:

```
> dpkg -l | grep libsqlite
ii  libsqlite3-0:amd64    3.11.0-1ubuntu1  amd64  SQLite 3 shared library
ii  libsqlite3-dev:amd64  3.11.0-1ubuntu1  amd64  SQLite 3 development files
```

The environment should now all be setup to run the BO benchmark.

CHAPTER

TWO

RUNNING

Now we can run each step of the experiments. First, we run all combinations and then run some quick commands to analyze the output.

2.1 Launch the experiments

The experiments are run using the experiment launcher, which has the following interface:

```
usage: bayesmark-launch [-h] [-dir DB_ROOT] [-odir OPTIMIZER_ROOT] [-v] [-u UUID]
                        [-dr DATA_ROOT] [-b DB] [-o OPTIMIZER [OPTIMIZER ...]]
                        [-d DATA [DATA ...]]
                        [-c [{DT,MLP-adam,MLP-sgd,RF,SVM,ada,kNN,lasso,linear} ...]]
                        [-m [{acc,mae,mse,nll} ...]] [-n N_CALLS]
                        [-p N_SUGGEST] [-r N_REPEAT] [-nj N_JOBS] [-ofile JOBS_FILE]
```

The arguments are:

```
-h, --help            show this help message and exit
-dir DB_ROOT, -db-root DB_ROOT
                      root directory for all benchmark experiments output
-odir OPTIMIZER_ROOT, --opt-root OPTIMIZER_ROOT
                      Directory with optimization wrappers
-v, --verbose         print the study logs to console
-u UUID, --uuid UUID  length 32 hex UUID for this experiment
-dr DATA_ROOT, --data-root DATA_ROOT
                      root directory for all custom csv files
-b DB, --db DB        database ID of this benchmark experiment
-o OPTIMIZER [OPTIMIZER ...], --opt OPTIMIZER [OPTIMIZER ...]
                      optimizers to use
-d DATA [DATA ...], --data DATA [DATA ...]
                      data sets to use
-c, --classifier [{DT,MLP-adam,MLP-sgd,RF,SVM,ada,kNN,lasso,linear} ...]
                      classifiers to use
-m, --metric [{acc,mae,mse,nll} ...]
                      scoring metric to use
-n N_CALLS, --calls N_CALLS
                      number of function evaluations
-p N_SUGGEST, --suggestions N_SUGGEST
                      number of suggestions to provide in parallel
-r N_REPEAT, --repeat N_REPEAT
                      number of repetitions of each study
-nj N_JOBS, --num-jobs N_JOBS
                      number of jobs to put in the dry run file, the default
```

(continues on next page)

(continued from previous page)

```
                        0 value disables dry run (real run)
-ofile JOBS_FILE, --jobs-file JOBS_FILE
                        a jobs file with all commands to be run
```

The output files will be placed in `[DB_ROOT]/[DBID]`. If `DBID` is not specified, it will be a randomly created subdirectory with a new name to avoid overwriting previous experiments. The path to `DBID` is shown at the beginning of `stdout` when running `bayesmark-launch`. In general, let the launcher create and setup `DBID` unless you are appending to a previous experiment, in which case, specify the existing `DBID`.

The launcher's sequence of commands can be accessed programmatically via *experiment_launcher.gen_commands()*. The individual experiments can be launched programmatically via *experiment.run_sklearn_study()*.

2.1.1 Selecting the experiments

A list of optimizers, classifiers, data sets, and metrics can be listed using the `-o`/`-c`/`-d`/`-m` commands, respectively. If not specified, the program launches all possible options.

2.1.2 Selecting the optimizer

A few different open source optimizers have been included as an example and are considered the "built-in" optimizers. The original repos are shown in the *Links*.

The data argument `-o` allows a list containing the "built-in" optimizers:

```
"HyperOpt", "Nevergrad-OnePlusOne", "OpenTuner-BanditA", "OpenTuner-GA", "OpenTuner-
↪GA-DE", "PySOT", "RandomSearch", "Scikit-GBRT-Hedge", "Scikit-GP-Hedge", "Scikit-GP-
↪LCB"
```

or, one can specify a user-defined optimizer. The class containing an optimizer conforming to the API must be found in in the folder specified by `--opt-root`. Additionally, a configuration defining each optimizer must be defined in `[OPT_ROOT]/config.json`. The `--opt-root` and `config.json` may be omitted if only built-in optimizers are used.

Additional details for providing a new optimizer are found in *adding a new optimizer*.

2.1.3 Selecting the data set

By default, this benchmark uses the sklearn example data sets as the "built-in" data sets for use in ML model tuning problems.

The data argument `-d` allows a list containing the "built-in" data sets:

```
"breast", "digits", "iris", "wine", "boston", "diabetes"
```

or, it can refer to a custom `csv` file, which is the name of file in the folder specified by `--data-root`. It also follows the convention that regression data sets start with `reg-` and classification data sets start with `clf-`. For example, the classification data set in `[DATA_ROOT]/clf-foo.csv` is specified with `-d clf-foo`.

The `csv` file can be anything readable by pandas, but we assume the final column is the target and all other columns are features. The target column should be integer for classification data and float for regression. The features should float (or `str` for categorical variable columns). See `bayesmark.data.load_data` for more information.

2.1.4 Dry run for cluster jobs

It is also possible to do a "dry run" of the launcher by specifying a value for `--num-jobs` greater than zero. For example, if `--num-jobs 50` is provided, a text file listing 50 commands to run is produced, with one command (job) per line. This is useful when preparing a list of commands to run later on a cluster.

A dry run will generate a command file (e.g., `jobs.txt`) like the following (with a meta-data header). Each line corresponds to a command that can be used as a job on a different worker:

```
# running: {'--uuid': None, '-db-root': '/foo', '--opt-root': '/example_opt_root', '--
↪data-root': None, '--db': 'bo_example_folder', '--opt': ['RandomSearch', 'PySOT'],
↪'--data': None, '--classifier': ['SVM', 'DT'], '--metric': None, '--calls': 15, '--
↪suggestions': 1, '--repeat': 3, '--num-jobs': 50, '--jobs-file': '/jobs.txt', '--
↪verbose': False, 'dry_run': True, 'rev': '9a14ef2', 'opt_rev': None}
# cmd: python bayesmark-launch -n 15 -r 3 -dir foo -o RandomSearch PySOT -c SVM DT -
↪nj 50 -b bo_example_folder
job_e2b63a9_00 bayesmark-exp -c SVM -d diabetes -o PySOT -u␣
↪079a155f03095d2ba414a5d2cedde08c -m mse -n 15 -p 1 -dir foo -b bo_example_folder &&␣
↪bayesmark-exp -c SVM -d boston -o RandomSearch -u 400e4c0be8295ad59db22d9b5f31d153 -
↪m mse -n 15 -p 1 -dir foo -b bo_example_folder && bayesmark-exp -c SVM -d digits -o␣
↪RandomSearch -u fe73a2aa960a5e3f8d78bfc4bcf51428 -m acc -n 15 -p 1 -dir foo -b bo_
↪example_folder
job_e2b63a9_01 bayesmark-exp -c DT -d diabetes -o PySOT -u␣
↪db1d9297948554e096006c172a0486fb -m mse -n 15 -p 1 -dir foo -b bo_example_folder &&␣
↪bayesmark-exp -c SVM -d boston -o RandomSearch -u 7148f690ed6a543890639cc59db8320b -
↪m mse -n 15 -p 1 -dir foo -b bo_example_folder && bayesmark-exp -c SVM -d breast -o␣
↪PySOT -u 72c104ba1b6d5bb8a546b0064a7c52b1 -m nll -n 15 -p 1 -dir foo -b bo_example_
↪folder
job_e2b63a9_02 bayesmark-exp -c SVM -d iris -o PySOT -u␣
↪cc63b2c1e4315a9aac0f5f7b496bfb0f -m nll -n 15 -p 1 -dir foo -b bo_example_folder &&␣
↪bayesmark-exp -c DT -d breast -o RandomSearch -u aec62e1c8b5552e6b12836f0c59c1681 -
↪m nll -n 15 -p 1 -dir foo -b bo_example_folder && bayesmark-exp -c DT -d digits -o␣
↪RandomSearch -u 4d0a175d56105b6bb3055c3b62937b2d -m acc -n 15 -p 1 -dir foo -b bo_
↪example_folder
...
```

This package does not have built in support for deploying these jobs on a cluster or cloud environment (.e.g., AWS).

2.1.5 The UUID argument

The `UUID` is a 32-char hex string used as a master random seed which we use to draw random seeds for the experiments. If `UUID` is not specified a version 4 UUID is generated. The used UUID is displayed at the beginning of `stdout`. In general, the `UUID` should not specified/re-used except for debugging because it violates the assumption that the experiment UUIDs are unique.

2.2 Aggregate results

Next to aggregate all the experiment files into combined (json) files we need to run the aggregation command:

```
usage: bayesmark-agg [-h] [-dir DB_ROOT] [-odir OPTIMIZER_ROOT] [-v] -b DB [-rv]
```

The arguments are:

```
-h, --help            show this help message and exit
-dir DB_ROOT, -db-root DB_ROOT
                      root directory for all benchmark experiments output
-odir OPTIMIZER_ROOT, --opt-root OPTIMIZER_ROOT
                      Directory with optimization wrappers
-v, --verbose         print the study logs to console
-b DB, --db DB        database ID of this benchmark experiment
-rv, --ravel          ravel all studies to store batch suggestions as if
                      they were serial
```

The `DB_ROOT` must match the folder from the launcher `bayesmark-launch`, and `DBID` must match that displayed from the launcher as well. The aggregate files are found in `[DB_ROOT]/[DBID]/derived`.

The result aggregation can be done programmatically via *experiment_aggregate.concat_experiments()*.

2.3 Analyze and summarize results

Finally, to run a statistical analysis presenting a summary of the experiments we run

```
usage: bayesmark-anal [-h] [-dir DB_ROOT] [-odir OPTIMIZER_ROOT] [-v] -b DB
```

The arguments are:

```
-h, --help            show this help message and exit
-dir DB_ROOT, -db-root DB_ROOT
                      root directory for all benchmark experiments output
-odir OPTIMIZER_ROOT, --opt-root OPTIMIZER_ROOT
                      Directory with optimization wrappers
-v, --verbose         print the study logs to console
-b DB, --db DB        database ID of this benchmark experiment
```

The `DB_ROOT` must match the folder from the launcher `bayesmark-launch`, and `DBID` must match that displayed from the launcher as well. The aggregate files are found in `[DB_ROOT]/[DBID]/derived`.

The `bayesmark-anal` command looks for a `baseline.json` file in `[DB_ROOT]/[DBID]/derived`, which states the best possible and random search performance. If no such file is present, `bayesmark-anal` automatically calls `bayesmark-baseline` to build it. The baselines are inferred from the random search performance in the logs. The baseline values are considered fixed (not random) quantities when `bayesmark-anal` builds confidence intervals. Therefore, we allow the user to leave them fixed and do not rebuild them when `bayesmark-anal` is called if a baselines file is already present.

The result analysis can be done programmatically via *experiment_analysis.compute_aggregates()*, and the baseline computation via *experiment_baseline.compute_baseline()*.

See *How scoring works* for more information on how the scores are computed and aggregated.

2.4 Example

After finishing the setup (environment) a small-scale serial can be run as follows:

```
> # setup
> DB_ROOT=./notebooks  # path/to/where/you/put/results
```

(continues on next page)

(continued from previous page)

```
> DBID=bo_example_folder
> mkdir $DB_ROOT
> # experiments
> bayesmark-launch -n 15 -r 3 -dir $DB_ROOT -b $DBID -o RandomSearch PySOT -c SVM DT -
↪v
Supply --uuid 3adc3182635e44ea96969d267591f034 to reproduce this run.
Supply --dbid bo_example_folder to append to this experiment or reproduce jobs file.
User must ensure equal reps of each optimizer for unbiased results
-c DT -d boston -o PySOT -u a1b287b450385ad09b2abd7582f404a2 -m mae -n 15 -p 1 -dir /
↪notebooks -b bo_example_folder
-c DT -d boston -o PySOT -u 63746599ae3f5111a96942d930ba1898 -m mse -n 15 -p 1 -dir /
↪notebooks -b bo_example_folder
-c DT -d boston -o RandomSearch -u 8ba16c880ef45b27ba0909199ab7aa8a -m mae -n 15 -p 1␣
↪-dir /notebooks -b bo_example_folder
...
0 failures of benchmark script after 144 studies.
done
> # aggregate
> bayesmark-agg -dir $DB_ROOT -b $DBID
> # analyze
> bayesmark-anal -dir $DB_ROOT -b $DBID -v
...
median score @ 15:
optimizer
PySOT_0.2.3_9b766b6           0.330404
RandomSearch_0.0.1_9b766b6    0.961829
mean score @ 15:
optimizer
PySOT_0.2.3_9b766b6           0.124262
RandomSearch_0.0.1_9b766b6    0.256422
normed mean score @ 15:
optimizer
PySOT_0.2.3_9b766b6           0.475775
RandomSearch_0.0.1_9b766b6    0.981787
done
```

The aggregate result files (i.e., `summary.json`) will now be available in `$DB_ROOT/$DBID/derived`. However, this will be high variance since it was from only 3 trials and only to 15 function evaluations.

2.5 Plotting and notebooks

Plotting the quantitative results found in `$DB_ROOT/$DBID/derived` can be done using the notebooks found in the `notebooks/` folder of the git repository. The notebook `plot_mean_score.ipynb` generates plots for aggregate scores averaging over all problems. The notebook `plot_test_case.ipynb` generates plots for each test problem.

To use the notebooks, first copy over the `notebooks/` folder from git repository.

To setup the kernel for running the notebooks use:

```
virtualenv bobm_ipynb --python=python3.6
source ./bobm_ipynb/bin/activate
pip install bayesmark[notebooks]
python -m ipykernel install --name=bobm_ipynb --user
```

Now, the notebooks for plotting can be run with the command `jupyter notebook` and selecting the kernel `bobm_ipynb`.

It is also possible to convert the notebooks to an HTML report at the command line using `nbconvert`. For example, use the command:

```
jupyter nbconvert --to html --execute notebooks/plot_mean_score.ipynb
```

The output file will be in `./notebooks/plot_mean_score.html`. Here is an example export. See the `nbconvert` documentation page for more output formats. By default, the notebooks look in `./notebooks/bo_example_folder/` for the `summary.json` from `bayesmark-anal`.

To run `plot_test_case.ipynb` use the command:

```
jupyter nbconvert --to html --execute notebooks/plot_test_case.ipynb --
↪ExecutePreprocessor.timeout=600
```

The `--ExecutePreprocessor.timeout=600` timeout increase is needed due to the large number of plots being generated. The output will be in `./notebooks/plot_test_case.html`.

CHAPTER

THREE

ADDING A NEW OPTIMIZER

All optimizers in this benchmark are required to follow the interface specified of the `AbstractOptimizer` class in `bayesmark.abstract_optimizer`. In general, this requires creating a wrapper class around the new optimizer. The wrapper classes must all be placed in a folder referred to by the `--opt-root` argument. This folder must also contain the `config.json` folder.

The interface is simple, one must merely implement the `suggest` and `observe` functions. The `suggest` function generates new guesses for evaluating the function. Once evaluated, the function evaluations are passed to the `observe` function. The objective function is *not* evaluated by the optimizer class. The objective function is evaluated on outside and results are passed to `observe`. This is the correct setup for Bayesian optimization because:

- We can observe/try inputs that were never suggested
- We can ignore suggestions
- The objective function may not be something as simple as a Python function

So passing the function as an argument as is done in `scipy.optimization` is artificially restrictive.

The implementation of the wrapper will look like the following:

```
from bayesmark.abstract_optimizer import AbstractOptimizer
from bayesmark.experiment import experiment_main


class NewOptimizerName(AbstractOptimizer):
    # Used for determining the version number of package used
    primary_import = "name of import used e.g, opentuner"

    def __init__(self, api_config, optional_arg_foo=None, optional_arg_bar=None):
        """Build wrapper class to use optimizer in benchmark.

        Parameters
        ----------
        api_config : dict-like of dict-like
            Configuration of the optimization variables. See API description.
        """
        AbstractOptimizer.__init__(self, api_config)
        # Do whatever other setup is needed
        # ...

    def suggest(self, n_suggestions=1):
        """Get suggestion from the optimizer.

        Parameters
        ----------
        n_suggestions : int
```

(continues on next page)

(continued from previous page)

```
            Desired number of parallel suggestions in the output

        Returns
        -------
        next_guess : list of dict
            List of `n_suggestions` suggestions to evaluate the objective
            function. Each suggestion is a dictionary where each key
            corresponds to a parameter being optimized.
        """
        # Do whatever is needed to get the parallel guesses
        # ...
        return x_guess

    def observe(self, X, y):
        """Feed an observation back.

        Parameters
        ----------
        X : list of dict-like
            Places where the objective function has already been evaluated.
            Each suggestion is a dictionary where each key corresponds to a
            parameter being optimized.
        y : array-like, shape (n,)
            Corresponding values where objective has been evaluated
        """
        # Update the model with new objective function observations
        # ...
        # No return statement needed


if __name__ == "__main__":
    # This is the entry point for experiments, so pass the class to experiment_main␣
↪to use this optimizer.
    # This statement must be included in the wrapper class file:
    experiment_main(NewOptimizerName)
```

Depending on the API of the optimizer being wrapped, building this wrapper class may only or require a few lines of code, or be a total pain.

3.1 The config file

Note: A config file is now optional. If no `config.json` is provided, the experiment launcher will look for all folders with an *optimizer.py* in the `--opt-root` directory.

Each optimizer wrapper can have multiple configurations, which is each referred to as a different optimizer in the benchmark. For example, the JSON config file will have entries as follows:

```
{
    "OpenTuner-BanditA-New": [
        "opentuner_optimizer.py",
        {"techniques": ["AUCBanditMetaTechniqueA"]}
    ],
    "OpenTuner-GA-DE-New": [
        "opentuner_optimizer.py",
```

(continues on next page)

(continued from previous page)

```
        {"techniques": ["PSO_GA_DE"]}
    ],
    "OpenTuner-GA-New": [
        "opentuner_optimizer.py",
        {"techniques": ["PSO_GA_Bandit"]}
    ]
}
```

Basically, the entries are `"name_of_strategy": ["file_with_class", {kwargs_for_the_constructor}]`. Here, `OpenTuner-BanditA`, `OpenTuner-GA-DE`, and `OpenTuner-GA` are all treated as different optimizers by the benchmark even though the all use the same class from `opentuner_optimizer.py`.

This `config.json` must be in the same folder as the optimizer classes (e.g., `opentuner_optimizer.py`).

3.2 Running with a new optimizer

To run the benchmarks using a new optimizer, simply provide its name (from `config.json`) in the `-o` list. The `--opt-root` argument must be specified in this case. For example, the launch command from the *example* becomes:

```
bayesmark-launch -n 15 -r 3 -dir $DB_ROOT -b $DBID -o RandomSearch PySOT-New -c SVM␣
↪DT --opt-root ./example_opt_root -v
```

Here, we are using the example `PySOT-New` wrapper from the `example_opt_root` folder in the git repo. It is equivalent to the builtin `PySOT`, but gives an example of how to provide a new custom optimizer.

CHAPTER

FOUR

CONTRIBUTING

The following instructions have been tested with Python 3.6.8 on Ubuntu (16.04.5 LTS).

4.1 Install in editable mode

First, define the variables for the paths we will use:

```
GIT=/path/to/where/you/put/repos
ENVS=/path/to/where/you/put/virtualenvs
```

Then clone the repo in your git directory `$GIT`:

```
cd $GIT
git clone https://github.com/uber/bayesmark.git
```

Inside your virtual environments folder `$ENVS`, make the environment:

```
cd $ENVS
virtualenv bayesmark --python=python3.6
source $ENVS/bayesmark/bin/activate
```

Now we can install the pip dependencies. Move back into your git directory and run

```
cd $GIT/bayesmark
pip install -r requirements/base.txt
pip install -r requirements/optimizers.txt
pip install -e .  # Install the benchmark itself
```

You may want to run `pip install -U pip` first if you have an old version of `pip`. The file `optimizers.txt` contains the dependencies for all the optimizers used in the benchmark. The analysis and aggregation programs can be run using only the requirements in `base.txt`.

4.2 Contributor tools

First, we need to setup some needed tools:

```
cd $ENVS
virtualenv bayesmark_tools --python=python3.6
source $ENVS/bayesmark_tools/bin/activate
pip install -r $GIT/bayesmark/requirements/tools.txt
```

To install the pre-commit hooks for contributing run (in the `bayesmark_tools` environment):

```
cd $GIT/bayesmark
pre-commit install
```

To rebuild the requirements, we can run:

```
cd $GIT/bayesmark
# Get py files from notebooks to analyze
jupyter nbconvert --to script notebooks/*.ipynb
# Generate the .in files (but pins to latest, which we might not want)
pipreqs bayesmark/ --ignore bayesmark/builtin_opt/ --savepath requirements/base.in
pipreqs test/ --savepath requirements/test.in
pipreqs bayesmark/builtin_opt/ --savepath requirements/optimizers.in
pipreqs notebooks/ --savepath requirements/ipynb.in
pipreqs docs/ --savepath requirements/docs.in
# Regenerate the .txt files from .in files
pip-compile-multi --no-upgrade
```

4.3 Generating the documentation

First setup the environment for building with `Sphinx`:

```
cd $ENVS
virtualenv bayesmark_docs --python=python3.6
source $ENVS/bayesmark_docs/bin/activate
pip install -r $GIT/bayesmark/requirements/docs.txt
```

Then we can do the build:

```
cd $GIT/bayesmark/docs
make all
open _build/html/index.html
```

Documentation will be available in all formats in `Makefile`. Use `make html` to only generate the HTML documentation.

4.4 Running the tests

The tests for this package can be run with:

```
cd $GIT/bayesmark
./test.sh
```

The script creates a conda environment using the requirements found in `requirements/test.txt`.

The `test.sh` script *must* be run from a *clean* git repo.

Or if we only want to run the unit tests and not check the adequacy of the requirements files, one can use

```
# Setup environment
cd $ENVS
virtualenv bayesmark_test --python=python3.6
source $ENVS/bayesmark_test/bin/activate
```

(continues on next page)

(continued from previous page)

```
pip install -r $GIT/bayesmark/requirements/test.txt
pip install -e $GIT/bayesmark
# Now run tests
cd $GIT/bayesmark/
pytest test/ -s -v --hypothesis-seed=0 --disable-pytest-warnings --cov=bayesmark --
↪cov-report html
```

A code coverage report will also be produced in `$GIT/bayesmark/htmlcov/index.html`.

4.5 Deployment

The wheel (tar ball) for deployment as a pip installable package can be built using the script:

```
cd $GIT/bayesmark/
./build_wheel.sh
```

CHAPTER

FIVE

LINKS

The source is hosted on GitHub.

The documentation is hosted at Read the Docs.

Installable from PyPI.

The builtin optimizers are wrappers on the following projects:

- HyperOpt
- Nevergrad
- OpenTuner
- PySOT
- Scikit-optimize

CHAPTER

SIX

LICENSE

This project is licensed under the Apache 2 License - see the LICENSE file for details.

CHAPTER

SEVEN

HOW SCORING WORKS

The scoring system is about aggregating the function evaluations of the optimizers. We represent F_{pmtn} as the function evaluation of objective function p (`TEST_CASE`) from the suggestion of method m (`METHOD`) at batch t (`ITER`) under repeated trial n (`TRIAL`). In the case of batch sizes greater than 1, F_{pmtn} is the minimum function evaluation across the suggestions in batch t. The first transformation is that we consider the *cumulative minimum* over batches t as the performance of the optimizer on a particular trial:

$$S_{pmtn} = \text{cumm-min}_t F_{pmtn} \, .$$

All of the aggregate quantities described here are computed by *experiment_analysis.compute_aggregates()* (which is called by *bayesmark-anal*) in either the `agg_result` or `summary` xarray datasets. Additionally, the baseline performances are in the xarray dataset `baseline_ds` from *experiment_baseline.compute_baseline()*. The baseline dataset can be generated via the `bayesmark-baseline` command, but it is called automatically by `bayesmark-anal` if needed.

7.1 Median scores

The more robust, but less decision-theoretically appealing method for aggregation is to look at median scores. On a per problem basis we simply consider the median (`agg_result[PERF_MED]`):

$$\text{med-perf}_{pmt} = \text{median}_n \, S_{pmtn} \, .$$

However, this score is not very comparable across different problems as the objectives are all on different scales with possible different units. Therefore, we decide the *normalized score* (`agg_result[NORMED_MED]`) in a way that is *invariant* to linear transformation of the objective function:

$$\text{norm-med-perf}_{pmt} = \frac{\text{med-perf}_{pmt} - \text{opt}_p}{\text{rand-med-perf}_{pt} - \text{opt}_p} \, ,$$

where opt_p (`baseline_ds[PERF_BEST]`) is an estimate of the global minimum of objective function p; and $\text{rand-med-perf}_{pt}$ is the median performance of random search at batch t on objective function p. This means that, on any objective, an optimizer has score 0 after converging to the global minimum; and random search performs as a straight line at 1 for all t. Conceptually, the median random search performance (`baseline_ds[PERF_MED]`) is computed as:

$$\text{rand-med-perf}_{pt} = \text{median}_n \, S_{pmtn} \, ,$$

with $m =$ random search. However, every observation of F_{pmtn} is iid in the case of random search. There is no reason to break the samples apart into trials n. Instead, we use the function *quantiles.min_quantile_CI()* to compute a more statistically efficient pooled estimator using the pooled random search samples over t and n. This pooled method is a nonparametric estimator of the quantiles of the minimum over a batch of samples, which is distribution free.

To further aggregate the performance over all objectives for a single optimizer we can consider the median-of-medians (`summary[PERF_MED]`):

$$\text{med-perf}_{mt} = \text{median}_p \, \text{norm-med-perf}_{pmt} \, .$$

Combining scores across different problems is sensible here because we have transformed them all onto the same scale.

7.2 Mean scores

From a decision theoretical perspective it is more sensible to consider the mean (possible warped) score. The median score can hide a high percentage of runs that completely fail. However, when we look at the mean score we first take the clipped score with a baseline value:

$$S'_{pmtn} = \min(S_{pmtn}, \text{clip}_p) \, .$$

This is largely because there may be a non-zero probably of $F = \infty$ (as in when the objective function crashes), which means that mean random search performance is infinite loss. We set clip_p (`baseline_ds[PERF_CLIP]`) to the median score after a single function evaluation, which is $\text{rand-med-perf}_{p0}$ for a batch size of 1. The mean performance on a single problem (`agg_result[PERF_MEAN]`) then becomes:

$$\text{mean-perf}_{pmt} = \text{mean}_n \, S'_{pmtn} \, .$$

Which then becomes a normalized performance (`agg_result[NORMED_MEAN]`) of:

$$\text{norm-mean-perf}_{pmt} = \frac{\text{mean-perf}_{pmt} - \text{opt}_p}{\text{clip}_p - \text{opt}_p} \, .$$

Note there that the random search performance is only 1 at the first batch unlike for $\text{norm-med-perf}_{pmt}$.

Again we can aggregate this into all objective function performance with (`summary[PERF_MEAN]`):

$$\text{mean-perf}_{mt} = \text{mean}_p \, \text{norm-mean-perf}_{pmt} \, ,$$

which is a mean-of-means (or *grand mean*), which is much more sensible in general than a median-of-medians. We can again obtain the property of random search having a constant performance of 1 for all t using (`summary[NORMED_MEAN]`):

$$\text{norm-mean-perf}_{mt} = \frac{\text{mean-perf}_{mt}}{\text{rand-mean-perf}_t} \, ,$$

where the random search baseline has been determined with the same sequence of equations as the other methods. These all collapse down to:

$$\text{rand-mean-perf}_t = \text{mean}_p \, \frac{\text{rand-mean-perf}_{pt} - \text{opt}_p}{\text{clip}_p - \text{opt}_p} \, .$$

Conceptually, we compute this random search baseline (`baseline_ds[PERF_MEAN]`) as:

$$\text{rand-mean-perf}_{pt} = \text{mean}_n \, S'_{pmtn} \, ,$$

with $m =$ random search. However, because all function evaluations for random search are iid across t, we can use a more statistically efficient pooled estimator *`expected_max.expected_min()`*, which is an unbiased distribution free estimator on the expected minimum of m samples from a distribution.

Note that $\text{norm-mean-perf}_{mt}$ is, in aggregate, a linear transformation on the expected loss S'. This makes it more justified in a decision theory framework than the median score. However, to view it as a linear transformation we are considering the values in `baseline_ds` to be fixed reference losses values and not the output from the experiment.

7.3 Error bars

The datasets `agg_result` and `summary` also compute error bars in the form of `LB_` and `UB_` variables. These error bars do not consider the random variation in the baseline quantities from `baseline_ds` like `opt` and `clip`. They are instead treated as fixed constant reference points. Therefore, they are computed by a different command `bayesmark-baseline`. The user can generate the baselines when they want, but since they are not considered a random quantity in the statistics they are not automatically generated from the experimental data (unless the baseline file `derived/baseline.json` is missing).

Additionally, the error bars on the grand mean (`summary[PERF_MEAN]`) are computed by simply using t-statistic based error bars on the individual means. Under a "random effects" model, this does not actually lose any statistical power. However, this is computing the mean on the loss over sampling from new problems under the "same distribution" of benchmark problems. These error bars will be wider than if we computed the error bars on the grand mean over this particular set of benchmark problems.

CHAPTER

EIGHT

CODE OVERVIEW

8.1 Data

Module to deal with all matters relating to loading example data sets, which we tune ML models to.

class bayesmark.data.**ProblemType**
The different problem types we consider. Currently, just regression (*reg*) and classification (*clf*).

bayesmark.data.**get_problem_type**(*dataset_name*)
Determine if this dataset is a regression of classification problem.

Parameters **dataset** (*str*) – Which data set to use, must be key in *DATA_LOADERS* dict, or name of custom csv file.

Returns **problem_type** – *Enum* to indicate if regression of classification data set.

Return type *ProblemType*

bayesmark.data.**load_data**(*dataset_name*, *data_root=None*)
Load a data set and return it in, pre-processed into numpy arrays.

Parameters

- **dataset** (*str*) – Which data set to use, must be key in *DATA_LOADERS* dict, or name of custom csv file.
- **data_root** (*str*) – Root directory to look for all custom csv files. May be `None` for sklearn data sets.

Returns

- **data** (`numpy.ndarray` of shape (n, d)) – The feature matrix of the data set. It will be *float* array.
- **target** (`numpy.ndarray` of shape (n,)) – The target vector for the problem, which is *int* for classification and *float* for regression.
- **problem_type** (*bayesmark.data.ProblemType*) – *Enum* to indicate if regression of classification data set.

8.2 Expected Max Estimation

Compute expected maximum or minimum from iid samples.

bayesmark.expected_max.**expected_max**(*x*, *m*)
Compute unbiased estimator of expected `max(x[1:m])` on a data set.

Parameters

- **x** (numpy.ndarray of shape (n,)) – Data set we would like expected max(x[1:m]) on.
- **m** (*int* or numpy.ndarray with dtype *int*) – This function is for estimating the expected maximum over *m* iid draws. Require m >= 1. This can be broadcasted. If m > n, the weights will be nan, because there is no way to get unbiased estimate in that case.

Returns **E_max_x** – Unbiased estimate of mean max of *m* draws from distribution on *x*.

Return type float

bayesmark.expected_max.**expected_min**(*x*, *m*)

Compute unbiased estimator of expected min(x[1:m]) on a data set.

Parameters

- **x** (numpy.ndarray of shape (n,)) – Data set we would like expected min(x[1:m]) on. Require len(x) >= 1.
- **m** (*int* or numpy.ndarray with dtype *int*) – This function is for estimating the expected minimum over *m* iid draws. Require m >= 1. This can be broadcasted. If m > n, the weights will be nan, because there is no way to get unbiased estimate in that case.

Returns **E_min_x** – Unbiased estimate of mean min of *m* draws from distribution on *x*.

Return type float

bayesmark.expected_max.**get_expected_max_weights**(*n*, *m*)

Get the L-estimator weights for computing unbiased estimator of expected max(x[1:m]) on a data set.

Parameters

- **n** (*int*) – Number of data points in data set len(x). Must be >= 1.
- **m** (*int* or numpy.ndarray with dtype *int*) – This function is for estimating the expected maximum over *m* iid draws. Require m >= 1. This can be broadcasted. If m > n, the weights will be nan, because there is no way to get unbiased estimate in that case.

Returns **pdf** – The weights for L-estimator. Will be positive and sum to one.

Return type numpy.ndarray, shape (n,)

8.3 Experiment Aggregation

Aggregate the results of many studies to prepare analysis.

bayesmark.experiment_aggregate.**concat_experiments**(*all_experiments*, *ravel=False*)

Aggregate the Datasets from a series of experiments into combined Dataset.

Parameters

- **all_experiments** (*typing.Iterable*) – Iterable (possible from a generator) with the Datasets from each experiment. Each item in *all_experiments* is a pair containing (meta_data, data). See *load_experiments* for details on these variables,
- **ravel** (*bool*) – If true, ravel all studies to store batch suggestions as if they were serial.

Returns

- **all_perf** (xarray.Dataset) – DataArray containing all of the *perf_da* from the experiments. The meta-data from the experiments are included as extra dimensions. *all_perf* has dimensions (ITER, SUGGEST, TEST_CASE, METHOD, TRIAL). To convert

the *uuid* to a trial, there must be an equal number of repetition in the experiments for each *TEST_CASE*, *METHOD* combination. Likewise, all of the experiments need an equal number of *ITER* and *SUGGEST*. If *ravel* is true, then the *SUGGEST* is singleton.

- **all_time** (xarray.Dataset) – Dataset containing all of the *time_ds* from the experiments. The new dimensions are `(ITER, TEST_CASE, METHOD, TRIAL)`. It has the same variables as *time_ds*.
- **all_suggest** (xarray.Dataset) – DataArray containing all of the *suggest_ds* from the experiments. It has dimensions `(ITER, SUGGEST, TEST_CASE, METHOD, TRIAL)`.
- **all_sigs** (*dict(str, list(list(float)))*) – Aggregate of all experiment signatures.

bayesmark.experiment_aggregate.**load_experiments**(*uuid_list*, *db_root*, *dbid*)

Generator to load the results of the experiments.

Parameters

- **uuid_list** (*list(uuid.UUID)*) – List of UUIDs corresponding to experiments to load.
- **db_root** (*str*) – Root location for data store as requested by the serializer used.
- **dbid** (*str*) – Name of the data store as requested by the serializer used.

Yields

- **meta_data** (*(str, str, str)*) – The *meta_data* contains a *tuple* of *str* with `test_case, optimizer, uuid`.
- **data** ((xarray.Dataset, xarray.Dataset, xarray.Dataset list(float))) – The *data* contains a tuple of `(perf_ds, time_ds, suggest_ds, sig)`. The *perf_ds* is a xarray.Dataset containing the evaluation results with dimensions `(ITER, SUGGEST)`, each variable is an objective. The *time_ds* is an xarray.Dataset containing the timing results of the form accepted by *summarize_time*. The coordinates must be compatible with *perf_ds*. The suggest_ds is a xarray.Dataset containing the inputs to the function evaluations. Each variable is a function input. Finally, *sig* contains the *test_case* signature and must be *list(float)*.

bayesmark.experiment_aggregate.**summarize_time**(*all_time*)

Transform a single timing dataset from an experiment into a form better for aggregation.

Parameters **all_time** (xarray.Dataset) – Dataset with variables `(SUGGEST_PHASE, EVAL_PHASE, OBS_PHASE)` which have dimensions `(ITER,)`, `(ITER, SUGGEST)`, and `(ITER,)`, respectively. The variable *EVAL_PHASE* has the function evaluation time for each parallel suggestion.

Returns time_summary – Dataset with variables `(SUGGEST_PHASE, OBS_PHASE, EVAL_PHASE_MAX, EVAL_PHASE_SUM)` which all have dimensions `(ITER,)`. The maximum *EVAL_PHASE_MAX* is relevant for wall clock time, while *EVAL_PHASE_SUM* is relevant for CPU time.

Return type xarray.Dataset

bayesmark.experiment_aggregate.**validate_agg_perf**(*perf_da*, *min_trial=1*)

Validate the aggregated eval data set.

bayesmark.experiment_aggregate.**validate_perf**(*perf_da*)

Validate the input eval data arrays.

bayesmark.experiment_aggregate.**validate_time**(*all_time*)

Validate the aggregated time data set.

8.4 Experiment Analysis

Perform analysis to compare different optimizers across problems.

bayesmark.experiment_analysis.**compute_aggregates**(*perf_da*, *baseline_ds*, *visible_perf_da=None*)

Aggregate function evaluations in the experiments to get performance summaries of each method.

Parameters

- **perf_da** (xarray.DataArray) – Aggregate experimental results with each function evaluation in the experiments according to true loss (e.g., generalization). *perf_da* has dimensions `(ITER, SUGGEST, TEST_CASE, METHOD, TRIAL)` as is assumed to have no nan values.
- **baseline_ds** (xarray.Dataset) – Dataset with baseline performance. It was variables `(PERF_MED, PERF_MEAN, PERF_CLIP, PERF_BEST)` with dimensions `(ITER, TEST_CASE)`, `(ITER, TEST_CASE)`, `(TEST_CASE,)`, and `(TEST_CASE,)`, respectively. *PERF_MED* is a baseline of performance based on random search when using medians to summarize performance. Likewise, *PERF_MEAN* is for means. *PERF_CLIP* is an upperbound to clip poor performance when using the mean. *PERF_BEST* is an estimate on the global minimum.
- **visible_perf_da** (xarray.DataArray) – Aggregate experimental results with each function evaluation in the experiments according to visible loss (e.g., validation). *visible_perf_da* has dimensions `(ITER, SUGGEST, TEST_CASE, METHOD, TRIAL)` as is assumed to have no nan values. If *None*, we set `visible_perf_da = perf_da`.

Returns

- **agg_result** (xarray.Dataset) – Dataset with summary of performance for each method and test case combination. Contains variables: `(PERF_MED, LB_MED, UB_MED, NORMED_MED, PERF_MEAN, LB_MEAN, UB_MEAN, NORMED_MEAN)` each with dimensions `(ITER, METHOD, TEST_CASE)`. *PERF_MED* is a median summary of performance with *LB_MED* and *UB_MED* as error bars. *NORMED_MED* is a rescaled *PERF_MED* so we expect the optimal performance is 0, and random search gives 1 at all *ITER*. Likewise, *PERF_MEAN*, *LB_MEAN*, *UB_MEAN*, *NORMED_MEAN* are for mean performance.
- **summary** (xarray.Dataset) – Dataset with overall summary of performance of each method. Contains variables `(PERF_MED, LB_MED, UB_MED, PERF_MEAN, LB_MEAN, UB_MEAN)` each with dimensions `(ITER, METHOD)`.

bayesmark.experiment_analysis.**get_perf_array**(*evals*, *evals_visible*)

Get the actual (e.g., generalization loss) over iterations.

Parameters

- **evals** (numpy.ndarray of shape (n_iter, n_batch, n_trials)) – The actual loss (e.g., generalization) for a given experiment.
- **evals_visible** (numpy.ndarray of shape (n_iter, n_batch, n_trials)) – The observable loss (e.g., validation) for a given experiment.

Returns **perf_array** – The best performance so far at iteration i from *evals*. Where the best has been selected according to *evals_visible*.

Return type numpy.ndarray of shape (n_iter, n_trials)

8.5 Experiment Baseline

Build performance baselines from aggregate results to prepare analysis.

bayesmark.experiment_baseline.**compute_baseline**(*perf_da*)

Compute a performance baseline of base and best performance from the aggregate experimental results.

Parameters **perf_da** (xarray.DataArray) – Aggregate experimental results with each function evaluation in the experiments. *all_perf* has dimensions `(ITER, SUGGEST, TEST_CASE, METHOD, TRIAL)` as is assumed to have no nan values.

Returns **baseline_ds** – Dataset with baseline performance. It was variables `(PERF_MED, PERF_MEAN, PERF_CLIP, PERF_BEST)` with dimensions `(ITER, TEST_CASE)`, `(ITER, TEST_CASE)`, `(TEST_CASE,)`, and `(TEST_CASE,)`, respectively. *PERF_MED* is a baseline of performance based on random search when using medians to summarize performance. Likewise, *PERF_MEAN* is for means. *PERF_CLIP* is an upperbound to clip poor performance when using the mean. *PERF_BEST* is an estimate on the global minimum.

Return type xarray.Dataset

bayesmark.experiment_baseline.**validate**(*baseline_ds*)

Perform same tracks as will happen in analysis.

8.6 Experiment Launcher

Launch studies in separate studies or do dry run to build jobs file with lists of commands to run.

bayesmark.experiment_launcher.**arg_safe_str**(*val*)

Cast value as *str*, raise error if not safe as argument to *argparse*.

bayesmark.experiment_launcher.**dry_run**(*args*, *opt_file_lookup*, *run_uuid*, *fp*, *random=<mtrand.RandomState object>*)

Write to buffer description of commands for running all experiments.

This function is almost pure by writing to a buffer, but it could be switched to a generator.

Parameters

- **args** (*dict(CmdArgs, [int, str])*) – Arguments of options to pass to the experiments being launched. The keys corresponds to the same arguments passed to this program.
- **opt_file_lookup** (*dict(str, str)*) – Mapping from method name to filename containing wrapper class for the method.
- **run_uuid** (*uuid.UUID*) – UUID for this launcher run. Needed to generate different experiments UUIDs on each call. This function is deterministic provided the same *run_uuid*.
- **fp** (*writable buffer*) – File handle to write out sequence of commands to execute (broken into jobs on each line) to execute all the experiments (possibly each job in parallel).
- **random** (*RandomState*) – Random stream to use for reproducibility.

bayesmark.experiment_launcher.**gen_commands**(*args*, *opt_file_lookup*, *run_uuid*)

Generator providing commands to launch processes for experiments.

Parameters

- **args** (*dict(CmdArgs, [int, str])*) – Arguments of options to pass to the experiments being launched. The keys corresponds to the same arguments passed to this program.

- **opt_file_lookup** (*dict(str, str)*) – Mapping from method name to filename containing wrapper class for the method.
- **run_uuid** (*uuid.UUID*) – UUID for this launcher run. Needed to generate different experiments UUIDs on each call. This function is deterministic provided the same *run_uuid*.

Yields

- **iteration_key** (*(str, str, str, str)*) – Tuple containing `(trial, classifier, data, optimizer)` to index the experiment.
- **full_cmd** (*tuple(str)*) – Strings containing command and arguments to run a process with experiment. Join with whitespace or use *util.shell_join()* to get string with executable command. The command omits `--opt-root` which means it will default to `.` if the command is executed. As such, the command assumes it is executed with `--opt-root` as the working directory.

bayesmark.experiment_launcher.**real_run**(*args*, *opt_file_lookup*, *run_uuid*, *timeout=None*)

Run sequence of independent experiments to fully run the benchmark.

This uses *subprocess* to launch a separate process (in serial) for each experiment.

Parameters

- **args** (*dict(CmdArgs, [int, str])*) – Arguments of options to pass to the experiments being launched. The keys corresponds to the same arguments passed to this program.
- **opt_file_lookup** (*dict(str, str)*) – Mapping from method name to filename containing wrapper class for the method.
- **run_uuid** (*uuid.UUID*) – UUID for this launcher run. Needed to generate different experiments UUIDs on each call. This function is deterministic provided the same *run_uuid*.
- **timeout** (*int*) – Max seconds per experiment

8.7 Experiment

Perform a study.

bayesmark.experiment.**build_eval_ds**(*function_evals*, *objective_names*)

Convert `numpy.ndarray` with function evaluations to `xarray.Dataset`.

This function is a data cleanup routine after running an experiment, before serializing the data to end the study.

Parameters

- **function_evals** (`numpy.ndarray` of shape (n_calls, n_suggestions, n_obj)) – Value of objective for each evaluation.
- **objective_names** (*list(str) of shape (n_obj,)*) – The names of each objective.

Returns **eval_ds** – `xarray.Dataset` containing one variable for each objective with the objective function evaluations. It has dimensions `(ITER, SUGGEST)`.

Return type `xarray.Dataset`

bayesmark.experiment.**build_suggest_ds**(*suggest_log*)

Convert `numpy.ndarray` with function evaluation inputs to `xarray.Dataset`.

This function is a data cleanup routine after running an experiment, before serializing the data to end the study.

Parameters **suggest_log** (*list(list(dict(str, object)))*) – Log of the suggestions. It has shape *(n_call, n_suggest)*.

Returns **suggest_ds** – `xarray.Dataset` containing one variable for each input with the objective function evaluations. It has dimensions `(ITER, SUGGEST)`.

Return type `xarray.Dataset`

`bayesmark.experiment.`**`build_timing_ds`**(*suggest_time*, *eval_time*, *observe_time*)

Convert `numpy.ndarray` with timing evaluations to `xarray.Dataset`.

This function is a data cleanup routine after running an experiment, before serializing the data to end the study.

Parameters

- **suggest_time** (`numpy.ndarray` of shape (n_calls,)) – The time to make each (batch) suggestion.
- **eval_time** (`numpy.ndarray` of shape (n_calls, n_suggestions)) – The time for each evaluation of the objective function.
- **observe_time** (`numpy.ndarray` of shape (n_calls,)) – The time for each (batch) evaluation of the objective function, and the time to make an observe call.

Returns **time_ds** – Dataset with variables `(SUGGEST_PHASE, EVAL_PHASE, OBS_PHASE)` which have dimensions `(ITER,)`, `(ITER, SUGGEST)`, and `(ITER,)`, respectively. The variable *EVAL_PHASE* has the function evaluation time for each parallel suggestion.

Return type `xarray.Dataset`

`bayesmark.experiment.`**`get_objective_signature`**(*model_name*, *dataset*, *scorer*, *data_root=None*)

Get signature of an objective function specified by an sklearn model and dataset.

This routine specializes *signatures.get_func_signature()* for the *sklearn* study case.

Parameters

- **model_name** (*str*) – Which sklearn model we are attempting to tune, must be an element of *constants.MODEL_NAMES*.
- **dataset** (*str*) – Which data set the model is being tuned to, which must be either a) an element of *constants.DATA_LOADER_NAMES*, or b) the name of a csv file in the *data_root* folder for a custom data set.
- **scorer** (*str*) – Which metric to use when evaluating the model. This must be an element of *sklearn_funcs.SCORERS_CLF* for classification models, or *sklearn_funcs.SCORERS_REG* for regression models.
- **data_root** (*str*) – Absolute path to folder containing custom data sets. This may be `None` if no custom data sets are used.‘‘

Returns **signature** – The signature of this test function.

Return type list(str)

`bayesmark.experiment.`**`load_optimizer_kwargs`**(*optimizer_name*, *opt_root*)

Load the kwarg options for this optimizer being tested.

This is part of the general experiment setup before a study.

Parameters

- **optimizer_name** (*str*) – Name of the optimizer being tested. This optimizer name must be present in optimizer config file.

- **opt_root** (*str*) – Absolute path to folder containing the config file.

Returns **kwargs** – The kwargs setting to pass into the optimizer wrapper constructor.

Return type dict(str, object)

bayesmark.experiment.**main**()

This is where experiments happen. Usually called by the experiment launcher.

bayesmark.experiment.**run_sklearn_study**(*opt_class*, *opt_kwargs*, *model_name*, *dataset*, *scorer*, *n_calls*, *n_suggestions*, *data_root=None*, *callback=None*)

Run a study for a single optimizer on a single *sklearn* model/data set combination.

This routine is meant for benchmarking when tuning *sklearn* models, as opposed to the more general *run_study()*.

Parameters

- **opt_class** (abstract_optimizer.AbstractOptimizer) – Type of wrapper optimizer must be subclass of abstract_optimizer.AbstractOptimizer.
- **opt_kwargs** (*kwargs*) – *kwargs* to use when instantiating the wrapper class.
- **model_name** (*str*) – Which sklearn model we are attempting to tune, must be an element of *constants.MODEL_NAMES*.
- **dataset** (*str*) – Which data set the model is being tuned to, which must be either a) an element of *constants.DATA_LOADER_NAMES*, or b) the name of a csv file in the *data_root* folder for a custom data set.
- **scorer** (*str*) – Which metric to use when evaluating the model. This must be an element of *sklearn_funcs.SCORERS_CLF* for classification models, or *sklearn_funcs.SCORERS_REG* for regression models.
- **n_calls** (*int*) – How many iterations of minimization to run.
- **n_suggestions** (*int*) – How many parallel evaluation we run each iteration. Must be >= 1.
- **data_root** (*str*) – Absolute path to folder containing custom data sets. This may be None if no custom data sets are used.‘‘
- **callback** (*callable*) – Optional callback taking the current best function evaluation, and the number of iterations finished. Takes array of shape *(n_obj,)*.

Returns

- **function_evals** (numpy.ndarray of shape (n_calls, n_suggestions, n_obj)) – Value of objective for each evaluation.
- **timing_evals** ((numpy.ndarray, numpy.ndarray, numpy.ndarray)) – Tuple of 3 timing results: (suggest_time, eval_time, observe_time) with shapes (n_calls,), (n_calls, n_suggestions), and (n_calls,). These are the time to make each suggestion, the time for each evaluation of the objective function, and the time to make an observe call.
- **suggest_log** (*list(list(dict(str, object)))*) – Log of the suggestions corresponding to the *function_evals*.

bayesmark.experiment.**run_study**(*optimizer*, *test_problem*, *n_calls*, *n_suggestions*, *n_obj=1*, *callback=None*)

Run a study for a single optimizer on a single test problem.

This function can be used for benchmarking on general stateless objectives (not just *sklearn*).

Parameters

- **optimizer** (abstract_optimizer.AbstractOptimizer) – Instance of one of the wrapper optimizers.
- **test_problem** (*sklearn_funcs.TestFunction*) – Instance of test function to attempt to minimize.
- **n_calls** (*int*) – How many iterations of minimization to run.
- **n_suggestions** (*int*) – How many parallel evaluation we run each iteration. Must be >= 1.
- **n_obj** (*int*) – Number of different objectives measured, only objective 0 is seen by optimizer. Must be >= 1.
- **callback** (*callable*) – Optional callback taking the current best function evaluation, and the number of iterations finished. Takes array of shape *(n_obj,)*.

Returns

- **function_evals** (numpy.ndarray of shape (n_calls, n_suggestions, n_obj)) – Value of objective for each evaluation.
- **timing_evals** ((numpy.ndarray, numpy.ndarray, numpy.ndarray)) – Tuple of 3 timing results: (suggest_time, eval_time, observe_time) with shapes (n_calls,), (n_calls, n_suggestions), and (n_calls,). These are the time to make each suggestion, the time for each evaluation of the objective function, and the time to make an observe call.
- **suggest_log** (*list(list(dict(str, object)))*) – Log of the suggestions corresponding to the *function_evals*.

8.8 Function Signatures

Routines to compute and compare the "signatures" of objective functions. These are useful to make sure two different studies were actually optimizing the same objective function (even if they say the same test case in the meta-data).

bayesmark.signatures.**analyze_signature_pair**(*signatures*, *signatures_ref*)

Analyze a pair of signatures (often from two sets of experiments) and return the error between them.

Parameters

- **signatures** (*dict(str, list(float))*) – Signatures from set of experiments. The signatures must all be the same length, so it should be 2D array like.
- **signatures_ref** (*dict(str, list(float))*) – The signatures from a reference set of experiments. The keys in *signatures* must be a subset of the signatures in *signatures_ref*.

Returns

- **sig_errs** (pandas.DataFrame) – rows are test cases, columns are test points.
- **signatures_median** (*dict(str, list(float))*) – Median signature across all repetition per test case.

bayesmark.signatures.**analyze_signatures**(*signatures*)

Analyze function signatures from the experiment.

Parameters **signatures** (*dict(str, list(list(float)))*) – The signatures should all be the same length, so it should be 2D array like.

Returns

- **sig_errs** (pandas.DataFrame) – rows are test cases, columns are test points.
- **signatures_median** (*dict(str, list(float))*) – Median signature across all repetition per test case.

bayesmark.signatures.**get_func_signature**(*f*, *api_config*)

Get the function signature for an objective function in an experiment.

Parameters

- **f** (*typing.Callable*) – The objective function we want to compute the signature of. This function must take inputs in the form of `dict(str, object)` with one dictionary key per variable, and provide *float* as the output.
- **api_config** (*dict(str, dict)*) – Configuration of the optimization variables. See API description.

Returns

- **signature_x** (*list(dict(str, object)) of shape (n_suggest,)*) – The input locations probed on signature call.
- **signature_y** (*list(float) of shape (n_suggest,)*) – The objective function values at the inputs points. This is the real signature.

8.9 Numpy Util

Utilities to that could be included in *numpy* but aren't.

bayesmark.np_util.**argmin_2d**(*X*)

Take the arg minimum of a 2D array.

bayesmark.np_util.**clip_chk**(*x*, *lb*, *ub*, *allow_nan=False*)

Clip all element of *x* to be between *lb* and *ub* like `numpy.clip()`, but also check `numpy.isclose()`.

Shapes of all input variables must be broadcast compatible.

Parameters

- **x** (numpy.ndarray) – Array containing elements to clip.
- **lb** (numpy.ndarray) – Lower limit in clip.
- **ub** (numpy.ndarray) – Upper limit in clip.
- **allow_nan** (*bool*) – If true, we allow `nan` to be present in *x* without out raising an error.

Returns **x** – An array with the elements of *x*, but where values < *lb* are replaced with *lb*, and those > *ub* with *ub*.

Return type numpy.ndarray

bayesmark.np_util.**cummin**(*x_val*, *x_key*)

Get the cumulative minimum of *x_val* when ranked according to *x_key*.

Parameters

- **x_val** (numpy.ndarray of shape (n, d)) – The array to get the cumulative minimum of along axis 0.
- **x_key** (numpy.ndarray of shape (n, d)) – The array for ranking elements as to what is the minimum.

Returns **c_min** – The cumulative minimum array.

Return type numpy.ndarray of shape (n, d)

bayesmark.np_util.**isclose_lte**(*x*, *y*)
Check that less than or equal to (lte, `x <= y`) is approximately true between all elements of *x* and *y*.

This is similar to `numpy.allclose()` for equality. Shapes of all input variables must be broadcast compatible.

Parameters

- **x** (numpy.ndarray) – Lower limit in <= check.
- **y** (numpy.ndarray) – Upper limit in <= check.

Returns **lte** – True if `x <= y` is approximately true element-wise.

Return type bool

bayesmark.np_util.**linear_rescale**(*X*, *lb0*, *ub0*, *lb1*, *ub1*, *enforce_bounds=True*)
Linearly transform all elements of *X*, bounded between *lb0* and *ub0*, to be between *lb1* and *ub1*.

Shapes of all input variables must be broadcast compatible.

Parameters

- **X** (numpy.ndarray) – Array containing elements to rescale.
- **lb0** (numpy.ndarray) – Current lower bound of *X*.
- **ub0** (numpy.ndarray) – Current upper bound of *X*.
- **lb1** (numpy.ndarray) – Desired lower bound of *X*.
- **ub1** (numpy.ndarray) – Desired upper bound of *X*.
- **enforce_bounds** (*bool*) – If True, perform input bounds check (and clipping if slight violation) on the input *X* and again on the output. This argument is not meant to be vectorized like the other input variables.

Returns **X** – Elements of input *X* after linear rescaling.

Return type numpy.ndarray

bayesmark.np_util.**random_seed**(*random=<mtrand.RandomState object>*)
Draw a random seed compatible with `numpy.random.RandomState`.

Parameters **random** (numpy.random.RandomState) – Random stream to use to draw the random seed.

Returns **seed** – Seed for a new random stream in `[0, 2**32-1)`.

Return type int

bayesmark.np_util.**shuffle_2d**(*X*, *random=<mtrand.RandomState object>*)
Generalization of `numpy.random.shuffle()` of 2D array.

Performs in-place shuffling of *X*. So, it has no return value.

Parameters

- **X** (numpy.ndarray of shape (n, m)) – Array-like 2D data to shuffle in place. Shuffles order of rows and order of elements within a row.
- **random** (numpy.random.RandomState) – Random stream to use to draw the random seed.

bayesmark.np_util.**snap_to**(*x*, *fixed_val=None*)

Snap input *x* to the *fixed_val* unless *fixed_val* is *None*, where *x* is returned.

Parameters

- **x** (numpy.ndarray) – Array containing elements to snap.
- **fixed_val** (numpy.ndarray or None) – Values to be returned if *x* is close, otherwise an error is raised. If *fixed_val* is *None*, *x* is returned.

Returns **fixed_val** – Snapped to value of *x*.

Return type numpy.ndarray

bayesmark.np_util.**strat_split**(*X*, *n_splits*, *inplace=False*, *random=<mtrand.RandomState object>*)

Make a stratified random split of items.

Parameters

- **X** (numpy.ndarray of shape (n, m)) – Data we would like to split randomly into groups. We should get the same number +/-1 of elements from each row in each group.
- **n_splits** (*int*) – How many groups we want to split into.
- **inplace** (*bool*) – If true, this function will cause in place modifications to *X*.
- **random** (numpy.random.RandomState) – Random stream to use for reproducibility.

Returns **Y** – Stratified split of *X* where each row of *Y* contains the same number +/-1 of elements from each row of *X*. Must be a list of arrays since each row may have a different length.

Return type list(numpy.ndarray)

8.10 Path Util

Utilities handy for manipulating paths that have extra checks not included in *os.path*.

bayesmark.path_util.**absopen**(*path*, *mode*)

Safe version of the built in open() that only opens absolute paths.

Parameters

- **path** (*str*) – Absolute path. An assertion failure is raised if it is not absolute.
- **mode** (*str*) – Open mode, any mode understood by the built in open(), e.g., "r" or "w".

Returns **f** – File handle open to use.

Return type file handle

bayesmark.path_util.**abspath**(*path*, *verify=True*)

Combo of os.path.abspath() and os.path.expanduser() that will also check existence of directory.

Parameters

- **path** (*str*) – Relative path string that can also contain home directories, e.g., "~/git/".
- **verify** (*bool*) – If true, verifies that the directory exists. Raises an assertion failure if it does not exist.

Returns **path** – Absolute version of input path.

Return type str

bayesmark.path_util.**join_safe_r**(**args*)

Safe version of `os.path.join()` that checks resulting path is absolute and the file exists for reading.

Parameters ***args** (*str*) – varargs for parts of path to combine. The last argument must be a file name.

Returns **fname** – Absolute path to filename.

Return type str

bayesmark.path_util.**join_safe_w**(**args*)

Safe version of `os.path.join()` that checks resulting path is absolute.

Because this routine is for writing, if the file already exists, a warning is raised.

Parameters ***args** (*str*) – varargs for parts of path to combine. The last argument must be a file name.

Returns **fname** – Absolute path to filename.

Return type str

8.11 Quantile Estimation

Compute quantiles and confidence intervals.

bayesmark.quantiles.**max_quantile_CI**(*X*, *q*, *m*, *alpha=0.05*)

Calculate CI on *q* quantile of distribution on max of *m* iid samples using a data set *X*.

This uses nonparametric estimation from order statistics and will have alpha level of at most *alpha* due to the discrete nature of order statistics.

Parameters

- **X** (`numpy.ndarray` of shape (n,)) – Data for quantile estimation. Can be vectorized. Must be sortable data type (which is almost everything).
- **q** (*float*) – Quantile to compute, must be in (0, 1). Can be vectorized.
- **m** (*int*) – Compute statistics for distribution on max over *m* samples. Must be `>= 1`. Can be vectorized.
- **alpha** (*float*) – False positive rate we allow for CI, must be in (0, 1). Can be vectorized.

Returns

- **estimate** (dtype of *X*, scalar) – Best estimate on *q* quantile on max over *m* iid samples.
- **LB** (dtype of *X*, scalar) – Lower end on CI
- **UB** (dtype of *X*, scalar) – Upper end on CI

bayesmark.quantiles.**min_quantile_CI**(*X*, *q*, *m*, *alpha=0.05*)

Calculate confidence interval on *q* quantile of distribution on min of *m* iid samples using a data set *X*.

This uses nonparametric estimation from order statistics and will have alpha level of at most *alpha* due to the discrete nature of order statistics.

Parameters

- **X** (`numpy.ndarray` of shape (n,)) – Data for quantile estimation. Can be vectorized. Must be sortable data type (which is almost everything).

- **q** (*float*) – Quantile to compute, must be in (0, 1). Can be vectorized.
- **m** (*int*) – Compute statistics for distribution on min over *m* samples. Must be `>= 1`. Can be vectorized.
- **alpha** (*float*) – False positive rate we allow for CI, must be in (0, 1). Can be vectorized.

Returns

- **estimate** (dtype of *X*, scalar) – Best estimate on *q* quantile on min over *m* iid samples.
- **LB** (dtype of *X*, scalar) – Lower end on CI
- **UB** (dtype of *X*, scalar) – Upper end on CI

bayesmark.quantiles.**order_stats**(*X*)

Compute order statistics on sample *X*.

Follows convention that order statistic 1 is minimum and statistic n is maximum. Therefore, array elements `0` and `n+1` are `-inf` and `+inf`.

Parameters **X** (`numpy.ndarray` of shape (n,)) – Data for order statistics. Can be vectorized. Must be sortable data type (which is almost everything).

Returns **o_stats** – Order statistics on *X*.

Return type `numpy.ndarray` of shape (n+2,)

bayesmark.quantiles.**quantile**(*X*, *q*)

Computes *q* th quantile of *X*.

Similar to `numpy.percentile()` except that it matches the mathematical definition of a quantile *and q* is scaled in (0,1) rather than (0,100).

Parameters

- **X** (`numpy.ndarray` of shape (n,)) – Data for quantile estimation. Can be vectorized. Must be sortable data type (which is almost everything).
- **q** (*float*) – Quantile to compute, must be in (0, 1). Can be vectorized.

Returns **estimate** – Empirical *q* quantile from sample *X*.

Return type dtype of *X*, scalar

bayesmark.quantiles.**quantile_CI**(*X*, *q*, *alpha=0.05*)

Calculate CI on *q* quantile from same *X* using nonparametric estimation from order statistics.

This will have alpha level of at most *alpha* due to the discrete nature of order statistics.

Parameters

- **X** (`numpy.ndarray` of shape (n,)) – Data for quantile estimation. Can be vectorized. Must be sortable data type (which is almost everything).
- **q** (*float*) – Quantile to compute, must be in (0, 1). Can be vectorized.
- **alpha** (*float*) – False positive rate we allow for CI, must be in (0, 1). Can be vectorized.

Returns

- **LB** (dtype of *X*, scalar) – Lower end on CI
- **UB** (dtype of *X*, scalar) – Upper end on CI

bayesmark.quantiles.**quantile_and_CI**(*X*, *q*, *alpha=0.05*)

Calculate CI on *q* quantile from same *X* using nonparametric estimation from order statistics.

This will have alpha level of at most *alpha* due to the discrete nature of order statistics.

Parameters

- **X** (numpy.ndarray of shape (n,)) – Data for quantile estimation. Can be vectorized. Must be sortable data type (which is almost everything).
- **q** (*float*) – Quantile to compute, must be in (0, 1). Can be vectorized.
- **alpha** (*float*) – False positive rate we allow for CI, must be in (0, 1). Can be vectorized.

Returns

- **estimate** (dtype of *X*, scalar) – Empirical *q* quantile from sample *X*.
- **LB** (dtype of *X*, scalar) – Lower end on CI
- **UB** (dtype of *X*, scalar) – Upper end on CI

8.12 Random Search

A baseline random search in our standardized optimizer interface. Useful for baselines.

bayesmark.random_search.**suggest_dict**(*X*, *y*, *meta*, *n_suggestions=1*, *random=<mtrand.RandomState object>*)

Stateless function to create suggestions for next query point in random search optimization.

This implements the API for general structures of different data types.

Parameters

- **X** (*list(dict)*) – Places where the objective function has already been evaluated. Not actually used in random search.
- **y** (numpy.ndarray, shape (n,)) – Corresponding values where objective has been evaluated. Not actually used in random search.
- **meta** (*dict(str, dict)*) – Configuration of the optimization variables. See API description.
- **n_suggestions** (*int*) – Desired number of parallel suggestions in the output
- **random** (numpy.random.RandomState) – Optionally pass in random stream for reproducibility.

Returns **next_guess** – List of *n_suggestions* suggestions to evaluate the objective function. Each suggestion is a dictionary where each key corresponds to a parameter being optimized.

Return type list(dict)

8.13 Serialization

A serialization abstraction layer (SAL) to save and load experimental results. All IO of experimental results should go through this module. This makes changing the backend (between different databases) transparent to the benchmark code.

class bayesmark.serialize.**XRSerializer**

Serialization layer when saving and loading *xarray* datasets (currently) as *json*.

get_derived_keys(*db*)

List the derived keys currently available in the database.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.

Returns **keys** – The variable names (or keys) in the database for derived data.

Return type list(str)

get_keys(*db*)

List the non-derived keys available in the database.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.

Returns **keys** – The variable names (or keys) in the database for non-derived data.

Return type list(str)

get_uuids(*db*, *key*)

List the UUIDs for the versions of a variable (non-derived key) available in the database.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.
- **keys** (*str*) – The variable name in the database for non-derived data.

Returns **uuids** – The UUIDs for the versions of this key.

Return type list(uuid.UUID)

init_db(*keys*, *db=None*, *exist_ok=True*)

Initialize a "database" for storing data at the specified location.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **keys** (*list(str)*) – The variable names (or keys) we will store in the database for non-derived data.
- **db** (*str*) – The name of the database. If `None`, a non-conflicting name will be generated.
- **exist_ok** (*bool*) – If true, do not raise an error if this database already exists.

Returns **db** – The name of the database.

Return type str

init_db_manual(*keys*, *db*)

Instruction for how one would manually initialize the "database" on another system.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **keys** (*list(str)*) – The variable names (or keys) we will store in the database for non-derived data.
- **db** (*str*) – The name of the database.

Returns **manual_setup_info** – The setup instructions.

Return type str

load(*db*, *key*, *uuid_*)
Load a dataset under a key name in the database. This is the inverse of `save()`.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.
- **key** (*str*) – The variable name in the database for the data.
- **uuid_** (*uuid.UUID*) – The UUID to represent the version of this variable we want to load.

Returns

- **data** (`xarray.Dataset`) – An `xarray.Dataset` variable for the non-derived data from an experiment.
- **meta** (*json-serializable*) – Associated meta-data with the experiment. This can be anything json serializable.

load_derived(*db*, *key*)
Load a dataset under a key name in the database as derived data. This is the inverse of `save_derived()`.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.
- **key** (*str*) – The variable name in the database for the data.

Returns

- **data** (`xarray.Dataset`) – An `xarray.Dataset` variable for the derived data from experiments.
- **meta** (*json-serializable*) – Associated meta-data with the experiments. This can be anything json serializable.

logging_path(*db*, *uuid_*)
Get an absolute path for logging from an experiment given its UUID.

Parameters

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.
- **uuid_** (*uuid.UUID*) – The UUID to represent this experiment.

Returns **logfile** – Absolute path suitable for logging in this experiment.

Return type str

save(*meta*, *db_root*, *db*, *key*, *uuid_*)
Save a dataset under a key name in the database.

Parameters

- **data** (`xarray.Dataset`) – An `xarray.Dataset` variable we would like to store as non-derived data from an experiment.
- **meta** (*json-serializable*) – Associated meta-data with the experiment. This can be anything json serializable.

- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.
- **key** (*str*) – The variable name in the database for the data.
- **uuid_** (*uuid.UUID*) – The UUID to represent the version of this variable we are storing.

save_derived(*meta*, *db_root*, *db*, *key*)
Save a dataset under a key name in the database as derived data.

Parameters

- **data** (xarray.Dataset) – An xarray.Dataset variable we would like to store as derived data from experiments.
- **meta** (*json-serializable*) – Associated meta-data with the experiments. This can be anything json serializable.
- **db_root** (*str*) – Absolute path to the database.
- **db** (*str*) – The name of the database.
- **key** (*str*) – The variable name in the database for the data.

8.14 Sklearn Tuning

Routines to build a standardized interface to make *sklearn* hyper-parameter tuning problems look like an objective function.

This file mostly contains a dictionary collection of all sklearn test funcs.

The format of each element in *MODELS* is: model_name: (model_class, fixed_param_dict, search_param_api_dict) *model_name* is an arbitrary name to refer to a certain strategy. At usage time, the optimizer instance is created using: `model_class(**kwarg_dict)` The kwarg dict is *fixed_param_dict* + *search_param_dict*. The *search_param_dict* comes from a optimizer which is configured using the *search_param_api_dict*. See the API description for information on setting up the *search_param_api_dict*.

class bayesmark.sklearn_funcs.**SklearnModel**(*model*, *dataset*, *metric*, *shuffle_seed=0*, *data_root=None*)
Test class for sklearn classifier/regressor CV score objective functions.

evaluate(*params*)
Evaluate the sklearn CV objective at a particular parameter setting.

Parameters **params** (*dict(str, object)*) – The varying (non-fixed) parameter dict to the sklearn model.

Returns **cv_loss** – Average loss over CV splits for sklearn model when tested using the settings in params.

Return type float

static inverse_test_case_str(*test_case*)
Inverse of *test_case_str*.

static test_case_str(*model*, *dataset*, *scorer*)
Generate the combined test case string from model, dataset, and scorer combination.

class bayesmark.sklearn_funcs.**SklearnSurrogate**(*model*, *dataset*, *scorer*, *path*)
Test class for sklearn classifier/regressor CV score objective function surrogates.

evaluate(*params*)
Evaluate the sklearn CV objective at a particular parameter setting.

Parameters **params** (*dict(str, object)*) – The varying (non-fixed) parameter dict to the sklearn model.

Returns **overall_loss** – Average loss over CV splits for sklearn model when tested using the settings in params.

Return type float

class bayesmark.sklearn_funcs.**TestFunction**
Abstract base class for test functions in the benchmark. These do not need to be ML hyper-parameter tuning.

abstract evaluate(*params*)
Abstract method to evaluate the function at a parameter setting.

get_api_config()
Get the API config for this test problem.

Returns **api_config** – The API config for the used model. See README for API description.

Return type dict(str, dict(str, object))

8.15 Space

Do the conversion of search spaces into a normalized cartesian space.

class bayesmark.space.**Boolean**(*warp=None*, *values=None*, *range_=None*)
Space for transforming Boolean variables to continuous normalized space.

class bayesmark.space.**Categorical**(*warp=None*, *values=None*, *range_=None*)
Space for transforming categorical variables to continuous normalized space.

unwarp(*X_w*)
Inverse of *warp* function.

Parameters **X_w** (numpy.ndarray of shape (..., m)) – Warped version of input space. The warped space has a one-hot encoding and therefore *m* is the number of possible values in the space. *X_w* will have a *float* type. Non-zero/one values are allowed in *X_w*. The maximal element in the vector is taken as the encoded value.

Returns **X** – Unwarped version of *X_w*. *X* will have same type code as the *Categorical* class, which is unicode ('U').

Return type numpy.ndarray of shape (...)

warp(*X*)
Warp inputs to a continuous space.

Parameters **X** (numpy.ndarray of shape (...)) – Input variables to warp. This is vectorized to work in any dimension, but it must have the same type code as the class, which is unicode ('U') for the *Categorical* space.

Returns **X_w** – Warped version of input space. By convention there is an extra dimension on warped array. The warped space has a one-hot encoding and therefore *m* is the number of possible values in the space. *X_w* will have a *float* type.

Return type numpy.ndarray of shape (..., m)

class bayesmark.space.**Integer**(*warp='linear'*, *values=None*, *range_=None*)
Space for transforming integer variables to continuous normalized space.

class bayesmark.space.**JointSpace**(*meta*)

Combination of multiple *Space* objectives to transform multiple variables at the same time (jointly).

get_bounds()

Get bounds of the warped joint space.

Returns **bounds** – Bounds in the warped space. First column is the lower bound and the second column is the upper bound. `bounds.tolist()` gives the bounds in the standard form expected by scipy optimizers: `[(lower_1, upper_1), ..., (lower_n, upper_n)]`.

Return type `numpy.ndarray` of shape (m, 2)

grid(*max_interp=8*)

Return grid spanning the original (unwarped) space.

Parameters **max_interp** (*int*) – The number of points to use in grid space when a range and not values are used to define the space. Must be `>= 0`.

Returns **axes** – Grids spanning the original spaces of each variable. For each variable, this is simply `self.values` if a grid has already been specified, otherwise it is just grid across the range.

Return type dict(str, list)

unwarp(*X_w*, *fixed_vals={}*)

Inverse of `warp()`.

Parameters

- **X_w** (`numpy.ndarray` of shape (n, m)) – Warped version of input space. Must be 2D *float* `numpy.ndarray`. *n* is the number of separate points in the warped joint space. *m* is the size of the joint warped space, which can be inferred in advance by calling `get_bounds()`.
- **fixed_vals** (*dict*) – Subset of variables we want to keep fixed in X. Unwarp checks that the unwarped version of *X_w* matches *fixed_vals* up to numerical error. Otherwise, an error is raised.

Returns **X** – List of *n* points in the joint space to warp. Each list element is a dictionary where each key corresponds to a variable in the joint space.

Return type list(dict(str, object)) of shape (n,)

validate(*X*)

Raise *ValueError* if X does not match the format expected for a joint space.

warp(*X*)

Warp inputs to a continuous space.

Parameters **X** (*list(dict(str, object)) of shape (n,)*) – List of *n* points in the joint space to warp. Each list element is a dictionary where each key corresponds to a variable in the joint space. Keys can be be missing in the records and the according warped variables will be `nan`.

Returns **X_w** – Warped version of input space. Result is 2D *float* np array. *n* is the number of input points, length of *X*. *m* is the size of the joint warped space, which can be inferred by calling `get_bounds()`.

Return type `numpy.ndarray` of shape (n, m)

class bayesmark.space.**Real**(*warp='linear'*, *values=None*, *range_=None*)

Space for transforming real variables to normalized space (after warping).

class bayesmark.space.**Space**(*dtype*, *default_round*, *warp='linear'*, *values=None*, *range_=None*)
Base class for all types of variables.

get_bounds()
Get bounds of the warped space.

Returns **bounds** – Bounds in the warped space. First column is the lower bound and the second column is the upper bound. Calling `bounds.tolist()` gives the bounds in the standard form expected by *scipy* optimizers: `[(lower_1, upper_1), ..., (lower_n, upper_n)]`.

Return type `numpy.ndarray` of shape (D, 2)

grid(*max_interp=8*)
Return grid spanning the original (unwarped) space.

Parameters **max_interp** (*int*) – The number of points to use in grid space when a range and not values are used to define the space. Must be `>= 0`.

Returns **values** – Grid spanning the original space. This is simply *self.values* if a grid has already been specified, otherwise it is just grid across the range.

Return type list

unwarp(*X_w*)
Inverse of *warp* function.

Parameters **X_w** (`numpy.ndarray` of shape (..., m)) – Warped version of input space. This is vectorized to work in any dimension. But, by convention, there is an extra dimension on the warped array. Currently, the last dimension `m=1` for all warpers. *X_w* must be of a *float* type.

Returns **X** – Unwarped version of *X_w*. *X* will have the same type code as the class, which is in *self.type_code*.

Return type `numpy.ndarray` of shape (...)

validate(*X*, *pre=False*)
Routine to validate inputs to warp.

This routine does not perform any checking on the dimensionality of *X* and is fully vectorized.

validate_warped(*X*, *pre=False*)
Routine to validate inputs to unwarp. This routine is vectorized, but *X* must have at least 1-dimension.

warp(*X*)
Warp inputs to a continuous space.

Parameters **X** (`numpy.ndarray` of shape (...)) – Input variables to warp. This is vectorized to work in any dimension, but it must have the same type code as the class, which is in *self.type_code*.

Returns **X_w** – Warped version of input space. By convention there is an extra dimension on warped array. Currently, `m=1` for all warpers. *X_w* will have a *float* type.

Return type `numpy.ndarray` of shape (..., m)

bayesmark.space.**biexp**(*x*)
Inverse of *bilog()* function.

Parameters **x** (*scalar*) – Input variable in linear space. Can be any numeric type and is vectorizable.

Returns **y** – The biexp of *x*.

Return type float

bayesmark.space.**bilog**(*x*)
Bilog warping function. Extension of log to work with negative numbers.

`Bilog(x) ~= log(x)` for large *x* or `-log(abs(x))` if *x* is negative. However, the bias term ensures good behavior near 0 and `bilog(0) = 0`.

Parameters **x** (*scalar*) – Input variable in linear space. Can be any numeric type and is vectorizable.

Returns **y** – The bilog of *x*.

Return type float

bayesmark.space.**decode**(*Y*, *labels*, *assume_sorted=False*)
Perform inverse of one-hot encoder *encode*.

Parameters

- **Y** (numpy.ndarray of shape (..., n)) – One-hot encoding of categorical data *X*. Extra dimension is appended at end for the one-hot vector. Maximum element is taken if there is more than one non-zero entry in one-hot vector.
- **labels** (numpy.ndarray of shape (n,)) – Complete list of all possible labels. List is flattened if it is not already 1-dimensional.
- **assume_sorted** (*bool*) – If true, assume labels is already sorted and unique. This saves the computational cost of calling numpy.unique().

Returns **X** – Categorical values corresponding to one-hot encoded *Y*.

Return type numpy.ndarray of shape (...)

bayesmark.space.**encode**(*X*, *labels*, *assume_sorted=False*, *dtype=<class 'bool'>*, *assume_valid=False*)
Perform one hot encoding of categorical data in numpy.ndarray variable *X* of any dimension.

Parameters

- **X** (numpy.ndarray of shape (...)) – Categorical values of any standard type. Vectorized to work for any dimensional *X*.
- **labels** (numpy.ndarray of shape (n,)) – Complete list of all possible labels. List is flattened if it is not already 1 dimensional.
- **assume_sorted** (*bool*) – If true, assume labels is already sorted and unique. This saves the computational cost of calling numpy.unique().
- **dtype** (*type*) – Desired data of feature array. One-hot is most logically *bool*, but feature matrices are usually *float*.
- **assume_valid** (*bool*) – If true, assume all element of *X* are in the list *labels*. This saves the computational cost of verifying *X* are in *labels*. If true and a non-label *X* occurs this routine will silently give bogus result.

Returns **Y** – One-hot encoding of *X*. Extra dimension is appended at end for the one-hot vector. It has data type *dtype*.

Return type numpy.ndarray of shape (..., n)

bayesmark.space.**identity**(*x*)
Helper function that perform warping in linear space. Sort of a no-op.

Parameters **x** (*scalar*) – Input variable in linear space. Can be any numeric type and is vectorizable.

Returns **y** – Same as input *x*.

Return type scalar

8.16 Stats

General statistic tools useful in the benchmark.

bayesmark.stats.**robust_standardize**(*X*, *q_level=0.5*)
Perform robust standardization of data matrix *X* over axis 0.

Similar to `sklearn.preprocessing.robust_scale()` except also does a Gaussian adjustment rescaling so that if Gaussian data is passed in the transformed data will, in large *n*, be distributed as N(0,1). See sklearn feature request #10139 on github.

Parameters

- **X** (`numpy.ndarray` of shape (n, ...)) – Array containing elements standardize. Require `n >= 2`.
- **q_level** (*scalar*) – Must be in [0, 1]. Inter-quartile range to use for scale estimation.

Returns **X** – Elements of input *X* standardization.

Return type `numpy.ndarray` of shape (n, ...)

bayesmark.stats.**t_EB**(*x*, *alpha=0.05*, *axis=-1*)
Get t-statistic based error bars on mean of *x*.

Parameters

- **x** (`numpy.ndarray` of shape (n_samples,)) – Data points to estimate mean. Must not be empty or contain `NaN`.
- **alpha** (*float*) – The alpha level (`1-confidence`) probability (in (0, 1)) to construct confidence interval from t-statistic.
- **axis** (*int*) – The axis on *x* where we compute the t-statistics. The function is vectorized over all other dimensions.

Returns **EB** – Size of error bar on mean (`>= 0`). The confidence interval is `[mean(x) - EB, mean(x) + EB]`. *EB* is `inf` when `len(x) <= 1`. Will be `NaN` if there are any infinite values in *x*.

Return type float

8.17 Util (General)

General utilities that should arguably be included in Python.

bayesmark.util.**all_unique**(*L*)
Check if all elements in a list are unique.

Parameters **L** (*list*) – List we would like to check for uniqueness.

Returns **uniq** – True if all elements in *L* are unique.

Return type bool

bayesmark.util.**chomp**(*str_val*, *ext='\n'*)
Chomp a suffix off a string.

Parameters

- **str_val** (*str*) – String we want to chomp off a suffix, e.g., `"foo.log"`, and we want to chomp the file extension.
- **ext** (*str*) – The suffix we want to chomp. An error is raised if *str_val* doesn't end in *ext*.

Returns **chomped** – Version of *str_val* with *ext* removed from the end.

Return type str

bayesmark.util.**in_or_none**(*x*, *L*)

Check if item is in list of list is None.

bayesmark.util.**preimage_func**(*f*, *x*)

Pre-image a funcation at a set of input points.

Parameters

- **f** (*typing.Callable*) – The function we would like to pre-image. The output type must be hashable.
- **x** (*typing.Iterable*) – Input points we would like to evaluate *f*. *x* must be of a type acceptable by *f*.

Returns **D** – This dictionary maps the output of *f* to the list of *x* values that produce it.

Return type dict(object, list(object))

bayesmark.util.**range_str**(*stop*)

Version of `range(stop)` that instead returns strings that are zero padded so the entire iteration is of the same length.

Parameters **stop** (*int*) – Stop value equivalent to `range(stop)`.

Yields **x** (*str*) – String representation of integer zero padded so all items from this generator have the same `len(x)`.

bayesmark.util.**shell_join**(*argv*, *delim=' '*)

Join strings together in a way that is an inverse of *shlex* shell parsing into *argv*.

Basically, if the resulting string is passed as a command line argument then *sys.argv* will equal *argv*.

Parameters

- **argv** (*list(str)*) – List of arguments to collect into command line string. It will be escaped accordingly.
- **delim** (*str*) – Whitespace delimiter to join the strings.

Returns **cmd** – Properly escaped and joined command line string.

Return type str

bayesmark.util.**str_join_safe**(*delim*, *str_vec*, *append=False*)

Version of *str.join* that is guaranteed to be invertible.

Parameters

- **delim** (*str*) – Delimiter to join the strings.
- **str_vec** (*list(str)*) – List of strings to join. A *ValueError* is raised if *delim* is present in any of these strings.
- **append** (*bool*) – If true, assume the first element is already joined and we are appending to it. So, *str_vec[0]* can contain *delim*.

Returns **joined_str** – Joined version of *str_vec*, which is always recoverable with `joined_str.split(delim)`.

Return type str

Examples

Append is required because,

```
ss = str_join_safe('_', ('foo', 'bar'))
str_join_safe('_', (ss, 'baz', 'qux'))
```

would fail because we are appending `'baz'` and `'qux'` to the already joined string `ss = 'foo_bar'`.

In this case, we use

```
ss = str_join_safe('_', ('foo', 'bar'))
str_join_safe('_', (ss, 'baz', 'qux'), append=True)
```

bayesmark.util.**strict_sorted**(*L*)

Return a strictly sorted version of *L*. Therefore, this raises an error if *L* contains duplicates.

Parameters **L** (*list*) – List we would like to sort.

Returns **S** – Strictly sorted version of *L*.

Return type list

8.18 Xarray Util

General utilities for *xarray* that should be included in *xarray*.

bayesmark.xr_util.**coord_compat**(*da_seq*, *dims*)

Check if a sequence of `xarray.DataArray` have compatible coordinates.

Parameters

- **da_seq** (list(`xarray.DataArray`)) – Sequence of `xarray.DataArray` we would like to check for compatibility. `xarray.Dataset` work too.
- **dims** (*list*) – Subset of all dimensions in the `xarray.DataArray` we are concerned with for compatibility.

Returns **compat** – True if all the `xarray.DataArray` have compatible coordinates.

Return type bool

bayesmark.xr_util.**da_concat**(*da_dict*, *dims*)

Concatenate a dictionary of `xarray.DataArray` similar to `pandas.concat()`.

Parameters

- **da_dict** (dict(tuple(str), `xarray.DataArray`)) – Dictionary of `xarray.DataArray` to combine. The keys are tuples of index values. The `xarray.DataArray` must have compatible coordinates.
- **dims** (*list(str)*) – The names of the new dimensions we create for the dictionary keys. This must be of the same length as the key tuples in *da_dict*.

Returns **da** – Combined data array. The new dimensions will be `input_da.dims + dims`.

Return type xarray.DataArray

bayesmark.xr_util.**da_to_string**(*da*)
Generate a human readable version of a 1D xarray.DataArray.

Parameters **da** (xarray.DataArray) – The xarray.DataArray to display. Must only have one dimension.

Returns **str_val** – String with human readable version of *da*.

Return type str

bayesmark.xr_util.**ds_concat**(*ds_dict*, *dims*)
Concatenate a dictionary of xarray.Dataset similar to pandas.concat(), and a generalization of *da_concat()*.

Parameters

- **ds_dict** (dict(tuple(str), xarray.DataArray)) – Dictionary of xarray.Dataset to combine. The keys are tuples of index values. The xarray.Dataset must have compatible coordinates, and all have the same variables.
- **dims** (*list(str)*) – The names of the new dimensions we create for the dictionary keys. This must be of the same length as the key tuples in *ds_dict*.

Returns **ds** – Combined dataset. For each variable *var*, the new dimensions will be `input_ds[var].dims + dims`.

Return type xarray.Dataset

bayesmark.xr_util.**ds_like**(*ref*, *vars_*, *dims*, *fill=nan*)
Produce a blank xarray.Dataset copying some coordinates from another xarray.Dataset.

Parameters

- **ref** (xarray.Dataset) – The reference dataset we want to copy coordinates from.
- **vars_** (*typing.Iterable*) – List of variable names we want in the new dataset.
- **dims** (*list*) – List of dimensions we want to copy over from *ref*. These are the dimensions of the output.
- **fill** (*scalar*) – Scalar value to fill the blank dataset. The *dtype* will be determined from the *fill* value.

Returns **ds** – A new dataset with variables *vars_* and dimensions *dims* where the coordinates have been copied from *ref*. All values are filled with *fill*.

Return type xarray.Dataset

bayesmark.xr_util.**ds_like_mixed**(*ref*, *vars_*, *dims*, *fill=nan*)
The same as *ds_like* but allow different dimensions for each variable.

Parameters

- **ref** (xarray.Dataset) – The reference dataset we want to copy coordinates from.
- **vars_** (*typing.Iterable*) – List of (variable names, dimension) pairs we want in the new dataset. The dimensions for each variable must be a subset of *dims*.
- **dims** (*list*) – List of all dimensions we want to copy over from *ref*.
- **fill** (*scalar*) – Scalar value to fill the blank dataset. The *dtype* will be determined from the *fill* value.

Returns **ds** – A new dataset with variables *vars_* and dimensions *dims* where the coordinates have been copied from *ref*. All values are filled with *fill*.

Return type xarray.Dataset

bayesmark.xr_util.**is_simple_coords**(*coords*, *min_side=0*, *dims=None*)

Check if all xr coordinates are "simple". That is, equals to `np.arange(n)`.

Parameters

- **coords** (*dict-like of coordinates*) – The coordinates we would like to check, e.g. from `DataArray.coords`.
- **min_side** (*int*) – The minimum side requirement. We can set this `min_side=1` and have empty coordinates result in a return value of `False`.
- **dims** (*None or list of dimension names*) – Dimensions we want to check for simplicity. If `None`, check all dimensions.

Returns **simple** – True when all coordinates are simple.

Return type bool

bayesmark.xr_util.**only_dataarray**(*ds*)

Convert a xarray.Dataset to a xarray.DataArray. If the xarray.Dataset has more than one variable, an error is raised.

Parameters **ds** (xarray.Dataset) – xarray.Dataset we would like to convert to a xarray.DataArray. This must contain only one variable.

Returns **da** – The xarray.DataArray extracted from *ds*.

Return type xarray.DataArray

CHAPTER

NINE

CREDITS

9.1 Development lead

Ryan Turner (rdturnermtl)

9.2 Contributors

- David Eriksson (dme65)

PYTHON MODULE INDEX

b

INDEX

I

J

L

M

O

P

Q

R

S

T

U

V

W

X