profit documentation

Release 0.7.dev3+g2ad6323

The RedMod Team

May 04, 2023

CONTENTS:

CHAPTER

ONE

PROBABILISTIC RESPONSE MODEL FITTING WITH INTERACTIVE TOOLS

This is a collection of tools for studying parametric dependencies of black-box simulation codes or experiments and construction of reduced order response models over input parameter space.

proFit can be fed with a number of data points consisting of different input parameter combinations and the resulting output of the simulation under investigation. It then fits a response-surface through the point cloud using Gaussian process regression (GPR) models. This probabilistic response model allows to predict ("interpolate") the output at yet unexplored parameter combinations including uncertainty estimates. It can also tell you where to put more training points to gain maximum new information (experimental design) and automatically generate and start new simulation runs locally or on a cluster. Results can be explored and checked visually in a web frontend.

Telling proFit how to interact with your existing simulations is easy and requires no changes in your existing code. Current functionality covers starting simulations locally or on a cluster via Slurm, subsequent surrogate modelling using GPy, scikit-learn, as well as an active learning algorithm to iteratively sample at interesting points and a Markov-Chain-Monte-Carlo (MCMC) algorithm. The web frontend to interactively explore the point cloud and surrogate is based on plotly/dash.

1.1 Features

- Compute evaluation points (e.g. from a random distribution) to run simulation
- Template replacement and automatic generation of run directories
- Starting parallel runs locally or on the cluster (SLURM)
- Collection of result output and postprocessing
- Response-model fitting using Gaussian Process Regression and Linear Regression
- Active learning to reduce number of samples needed
- MCMC to find a posterior parameter distribution (similar to active learning)
- Graphical user interface to explore the results

1.2 Installation

Currently, the code is under heavy development, so it should be cloned from GitHub via Git and pulled regularly.

1.2.1 Requirements

```
sudo apt install python3-dev build-essential
```

To enable compilation of the fortran modules the following is needed:

```
sudo apt install gfortran
```

1.2.2 Dependencies

- numpy, scipy, matplotlib, sympy, pandas
- ChaosPy
- GPy
- scikit-learn
- h5py
- plotly/dash - for the UI
- ZeroMQ - for messaging
- sphinx - for documentation, only needed when `docs` is specified
- torch, GPyTorch - only needed when `gpu` is specified

All dependencies are configured in `setup.cfg` and should be installed automatically when using `pip`.

Automatic tests use `pytest`.

1.2.3 Windows 10

To install proFit under Windows 10 we recommend using *Windows Subsystem for Linux (WSL2)* with the Ubuntu 20.04 LTS distribution (install guide).

After the installation of WSL2 execute the following steps in your Linux terminal (when asked press `y` to continue):

Make sure you have the right version of Python installed and the basic developer toolset available

```
sudo apt update
sudo apt install python3 python3-pip python3-dev build-essential
```

To install proFit from Git (see below), make sure that the project is located in the Linux file system not the Windows system.

To configure the Python interpreter available in your Linux distribution in pycharm (tested with professional edition) follow this guide.

1.2.4 Installation from PyPI

To install the latest stable version of proFit, use

```
pip install profit
```

For the latest pre-release, use

```
pip install --pre profit
```

1.2.5 Installation from Git

To install proFit for the current user (`--user`) in development-mode (`-e`) use:

```
git clone https://github.com/redmod-team/profit.git
cd profit
pip install -e . --user
```

1.2.6 Fortran

Certain surrogates require a compiled Fortran backend. To enable compilation of the fortran modules during install:

```
USE_FORTRAN=1 pip install .
```

1.2.7 Troubleshooting installation problems

1. Make sure you have all the requirements mentioned above installed.
2. If `pip` is not recognized try the following:

```
python3 -m pip install -e . --user
```

1. If pip warns you about PATH or proFit is not found close and reopen the terminal and type `profit --help` to check if the installation was successful.

1.2.8 Documentation using *Sphinx*

Install requirements for building the documentation using `sphinx`

```
pip install .[docs]
```

Additionally `pandoc` is required on a system level:

```
sudo apt install pandoc
```

1.3 HowTo

Examples for different model codes are available under `examples/`:

- `fit`: Simple fit via python interface.
- `mockup`: Simple model called by console command based on template directory.

Also, the integration tests under `tests/integration_tests/` may be informative examples:

- `active_learning`:
  - 1D: One dimensional mockup with active learning
  - 2D: Two dimensional mockup with active learning
  - Log: Active learning with logarithmic search space
  - MCMC: Markov-Chain-Monte-Carlo application to mockup experimental data
- `mockup`:
  - 1D
  - 2D
  - Custom postprocessor: Instead of the prebuilt postprocessor, a user-built class is used.
  - Custom worker: A user-built worker function is used.
  - Independent: Output with an independent (linear) variable additional to input parameters: f(t; u, v).
  - KarhunenLoeve: Multi output surrogate model with Karhunen-Loeve encoder.
  - Multi output: Multi output surrogate with two different output variables.

1.3.1 Steps

1. Create and enter a directory (e.g. `study`) containing `profit.yaml` for your run. If your code is based on text configuration files for each run, copy the according directory to `template` and replace values of parameters to be varied within UQ/surrogate models by placeholders `{param}`.
2. Running the simulations:

   ```
   profit run
   ```

   to start simulations at all the points. Per default the generated input variables are written to `input.txt` and the output data is collected in `output.txt`.

   For each run of the simulation, proFit creates a run directory, fills the templates with the generated input data and collects the results. Each step can be customized with the configuration file.
3. To fit the model:

   ```
   profit fit
   ```

   Customization can be done with `profit.yaml` again.
4. Explore data graphically:

   ```
   profit ui
   ```

   starts a Dash-based browser UI

The figure below gives a graphical representation of the typical profit workflow described above. The boxes in red describe user actions while the boxes in blue are conducted by profit.

1.3.2 Cluster

proFit supports scheduling the runs on a cluster using *slurm*. This is done entirely via the configuration files and the usage doesn't change.

`profit ui` starts a *dash* server and it is possible to remotely connect to it (e.g. via *ssh port forwarding*)

1.4 User-supplied files

- a configuration file: (default: `profit.yaml`)
  - Add parameters and their distributions via `variables`
  - Set paths and filenames
  - Configure the run backend (how to interact with the simulation)
  - Configure the fit / surrogate model
- the `template` directory
  - containing everything a simulation run needs (scripts, links to executables, input files, etc)
  - input files use a template format where `{variable_name}` is substituted with the generated values
- a custom *Postprocessor* (optional)
  - if the default postprocessors don't work with the simulation a custom one can be specified using the `include` parameter in the configuration.

Example directory structure:

CHAPTER

TWO

GETTING STARTED

This guide aims to give a rough overview over a full proFit workflow. Further examples of the configuration can be found in `examples/` and in `tests/integration_tests/`. If you want to just use a few features have a look at `examples/api/`, `tests` and the API reference.

2.1 Example: Run System

Showcases the API Usage of the run system proFit v0.5.dev

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

2.1.1 Setup

The Worker

The simulation we want to run is wrapped in a *Worker*. We need to name the output variables, the input variables can be inferred from the simulations's argument list.

As an alternative, the *CommandWorker* can be configured to run any command/executable. It will prepare a directory for each simulation, filling the input values according to a template and read the output files to transmit them back.

Note: for the *LocalRunner* and *SlurmRunner* it is only possible to use a custom *Worker* which is *importable* (e.g. defined in a separate file which is specified in the configuration). Custom *Workers* defined in a script or notebook can only be used with the *ForkRunner* for now.

[2]:
```
from profit.run import Worker


@Worker.wrap("simulation")
def simulation(u, v) -> "f":
    return u * np.cos(v)
```

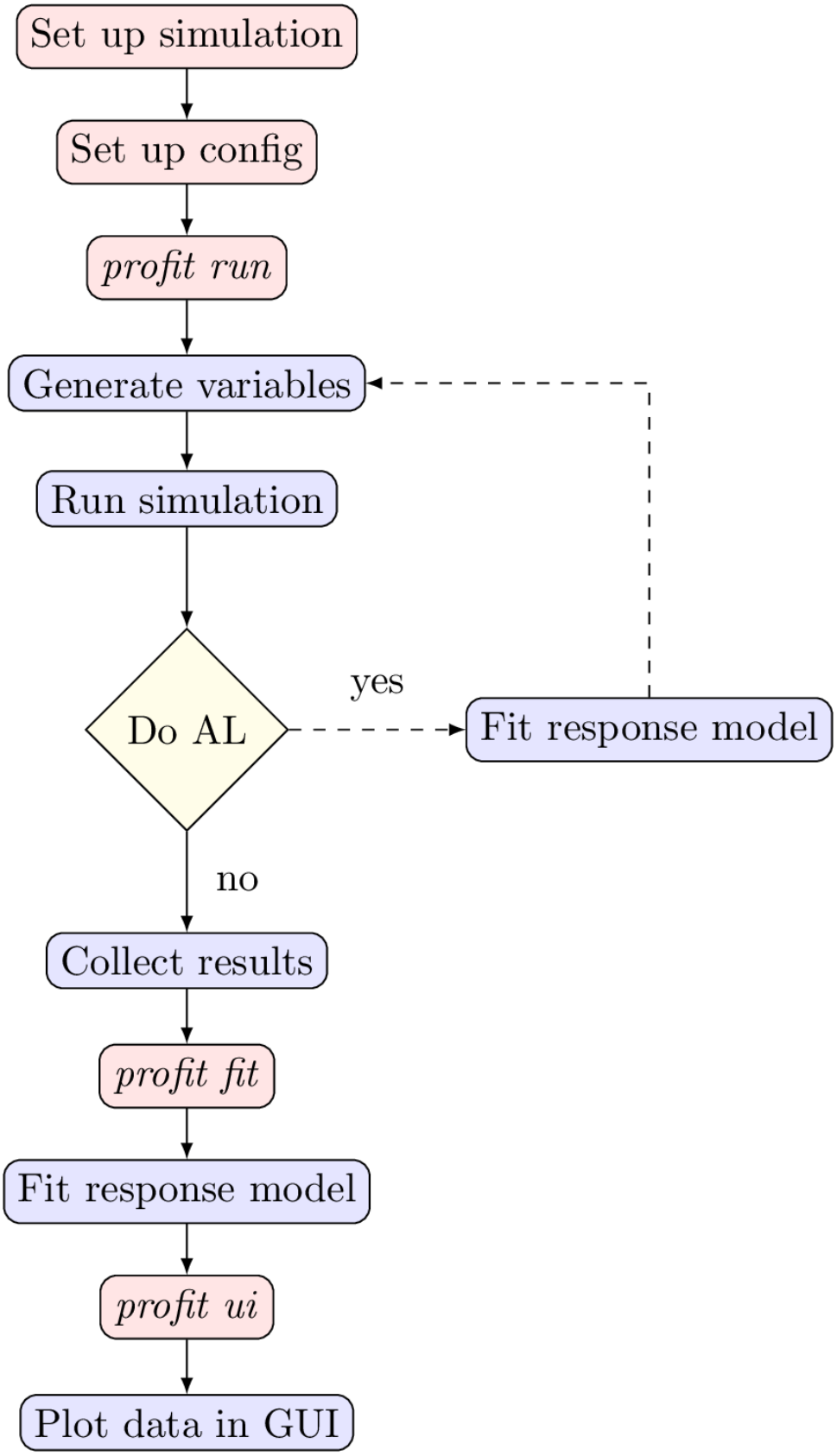


Fig. 1: The typical profit workflow

The Interface

To manage the flow of data between the *Workers* and the *Runner*, we choose and configure an *Interface*.

Memorymap Interface (`memmap`)	ZeroMQ Interface (`zeromq`)
single memory mapped file	messages via a protocol (tcp)
only local, but fast	local and distributed (HPC)

[3]:
```
from profit.run import RunnerInterface

interface = RunnerInterface["memmap"](
    size=10,
    input_config={"u": {"dtype": float}, "v": {"dtype": float}},
    output_config={"f": {"dtype": float, "size": (1, 1)}},
)
```

The Runner

The *Runner* is the central components of the run system which brings everything together. The *Runner* is also responsible for starting and distributing the individual *Workers*.

Fork Runner (`fork`)	Local Runner (`local`)	Slurm Runner (`slurm`)
forking / `Process`	via the shell / `subprocess`	via the *Slurm* scheduler
fastest, supports temporary *Workers*		submits *Slurm* jobs (HPC)

[4]:
```
from profit.run import Runner

runner = Runner["fork"](
    interface=interface,
    worker="simulation",  # don't require the Worker, just it's label or config dictionary
)
```

2.1.2 Running

[5]:
```
runner.next_run_id = 0  # reset Runner
runner.spawn({"u": 1.2, "v": 2})

U = np.random.random(9)
V = np.linspace(0, 2, 9)
parameters = [{"u": u, "v": v} for u, v in zip(U, V)]
runner.spawn_array(parameters, progress=True, wait=True)
```

```
submitted: 100%|| 9/9 [00:00<00:00, 19.73it/s]
finished : 100%|| 10/10 [00:00<00:00, 98.49it/s]
```

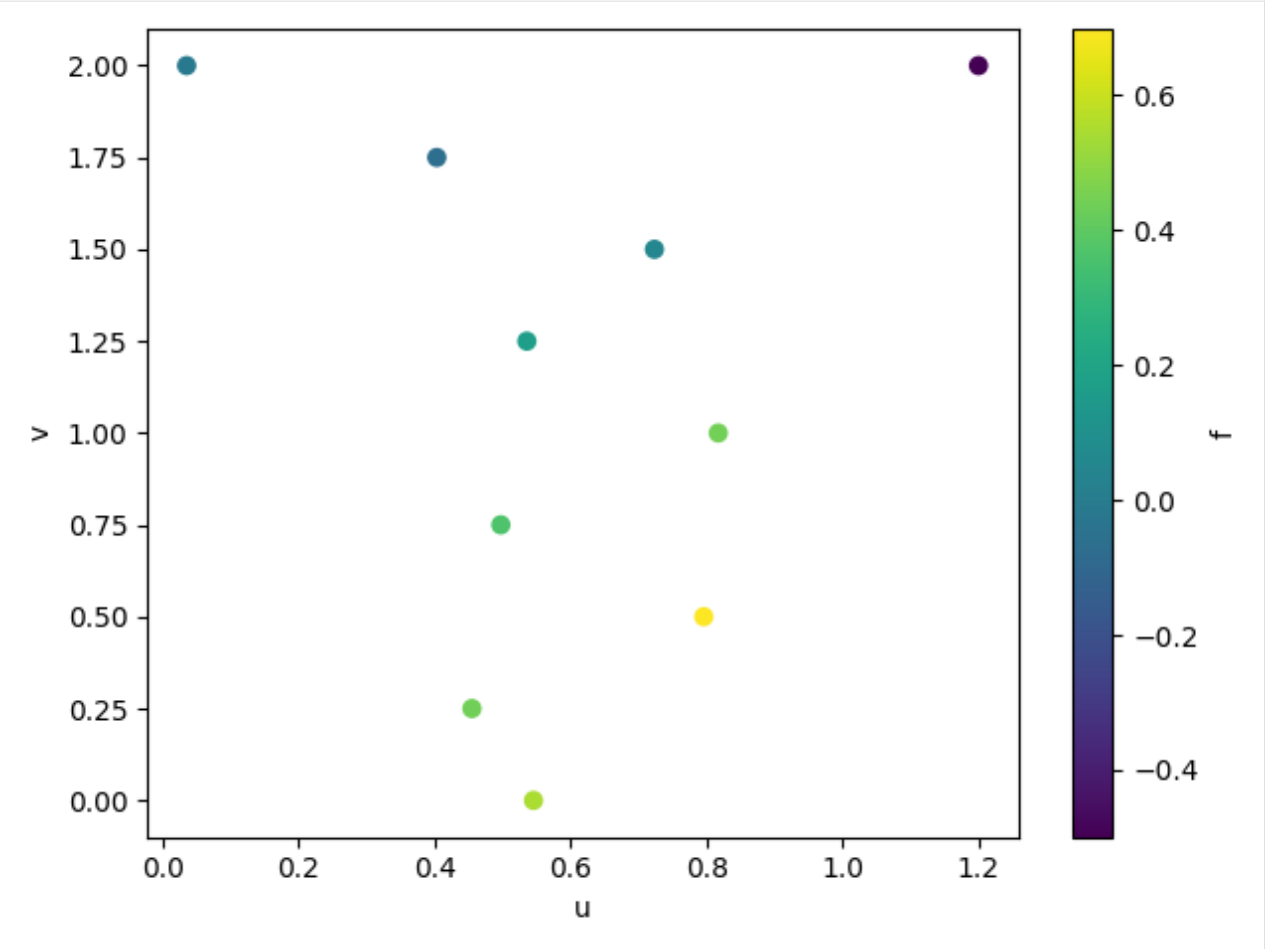
2.1.3 Results

[6]:
```
import pandas as pd

pd.DataFrame(runner.input_data).join(pd.DataFrame(runner.output_data))
```

[6]:

	u	v	f
0	1.200000	2.00	-0.499376
1	0.545389	0.00	0.545389
2	0.454843	0.25	0.440703
3	0.795911	0.50	0.698477
4	0.497352	0.75	0.363907
5	0.817201	1.00	0.441536
6	0.535651	1.25	0.168903
7	0.723296	1.50	0.051164
8	0.403129	1.75	-0.071856
9	0.035142	2.00	-0.014624

[7]:
```
fig, ax = plt.subplots(tight_layout=True)
sc = ax.scatter(
    runner.input_data["u"], runner.input_data["v"], c=runner.output_data["f"]
)
plt.colorbar(sc, ax=ax, label="f")
ax.set(
    xlabel="u",
    ylabel="v",
)
None
```



2.1.4 Clean Runner & Interface

[8]:
```
runner.clean()
```

2.2 Example: Surrogates

Showcases the API usage of GP and LinReg Surrogates in 1D and 2D

Note: these examples use mostly the default values. Better results can be achieved if hyperparameters are set manually to known values (e.g. if the uncertainty is known).

[1]:
```
# import libraries
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng()
```

[2]:
```
# import the Surrogate
from profit.sur import Surrogate
```

(continues on next page)

(continued from previous page)

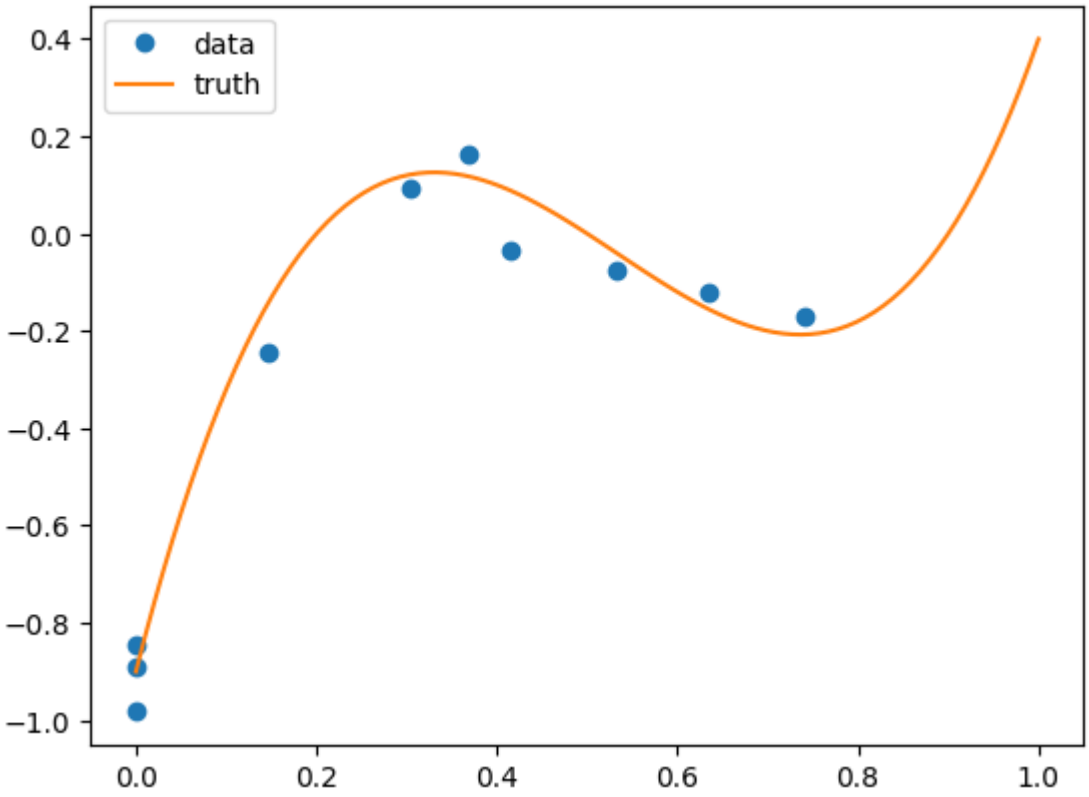
```
# ensure GP and LinReg Surrogates are available
import profit.sur.gp
import profit.sur.linreg
```

[3]:
```
# available surrogates:
Surrogate.labels
```

[3]:
```
{'GPy': profit.sur.gp.gpy_surrogate.GPySurrogate,
 'CoregionalizedGPy': profit.sur.gp.gpy_surrogate.CoregionalizedGPySurrogate,
 'Custom': profit.sur.gp.custom_surrogate.GPSurrogate,
 'CustomMultiOutputGP': profit.sur.gp.custom_surrogate.MultiOutputGPSurrogate,
 'Sklearn': profit.sur.gp.sklearn_surrogate.SklearnGPSurrogate,
 'ChaospyLinReg': profit.sur.linreg.chaospy_linreg.ChaospyLinReg}
```

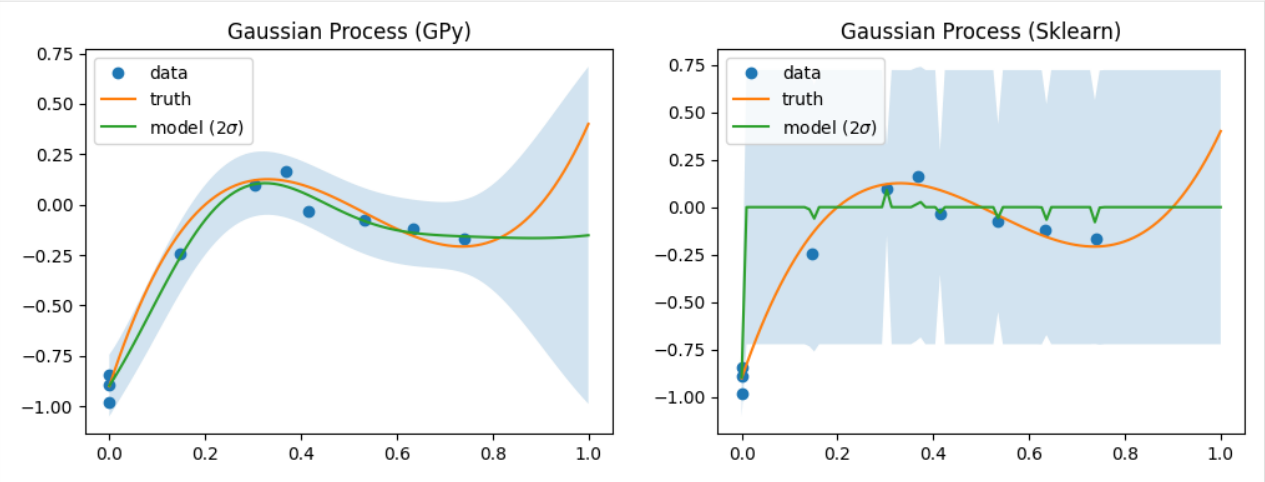
2.2.1 1D

[4]:
```
# generate mockup data
n = 10
f = lambda x: 10 * (x - 0.2) * (x - 0.9) * (x - 0.5)

x = np.maximum(np.minimum(rng.normal(0.5, 0.3, n), 1), 0)
y = f(x) + rng.normal(0, 0.05, n)

xx = np.linspace(0, 1, 100)
yy = f(xx)

ax = plt.subplot()
ax.plot(x, y, "o", label="data")
ax.plot(xx, yy, label="truth")
ax.legend()
None
```

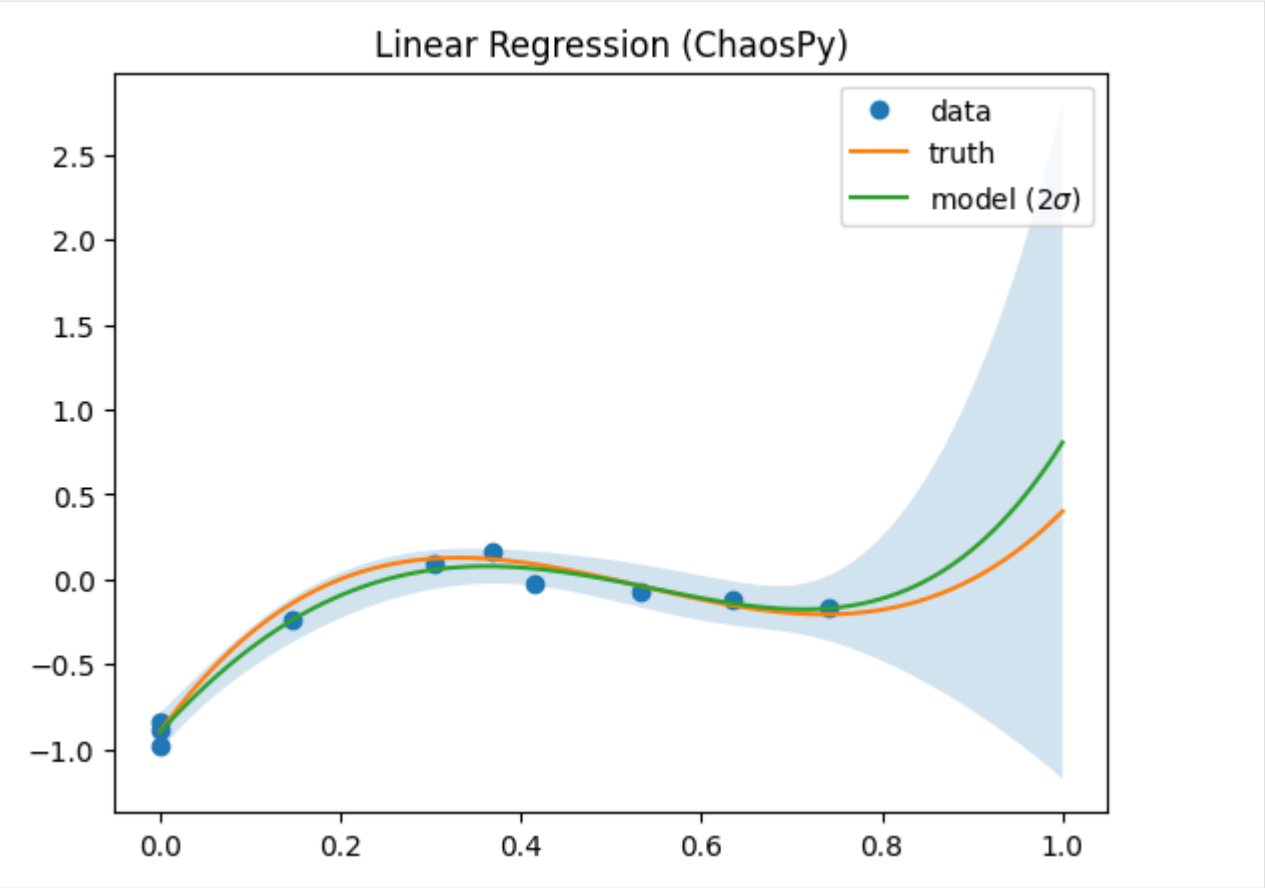


[5]:
```
# fit two GPs (different implementations)
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
for ax, label in zip(axs, ["GPy", "Sklearn"]):

    sur = Surrogate[label]()
    sur.train(x, y)
    mean, var = sur.predict(xx)

    mean = mean.flatten()
    std = np.sqrt(var.flatten())

    ax.plot(x, y, "o", label="data")
    ax.plot(xx, yy, label="truth")
    ax.fill_between(xx, mean + 2 * std, mean - 2 * std, alpha=0.2)
    ax.plot(xx, mean, label="model ($2 \\sigma$)")
    ax.legend()
    ax.set(title=f"Gaussian Process ({label})")
```

```
warning in stationary: failed to import cython module: falling back to numpy
warning in coregionalize: failed to import cython module: falling back to numpy
warning in choleskies: failed to import cython module: falling back to numpy
```

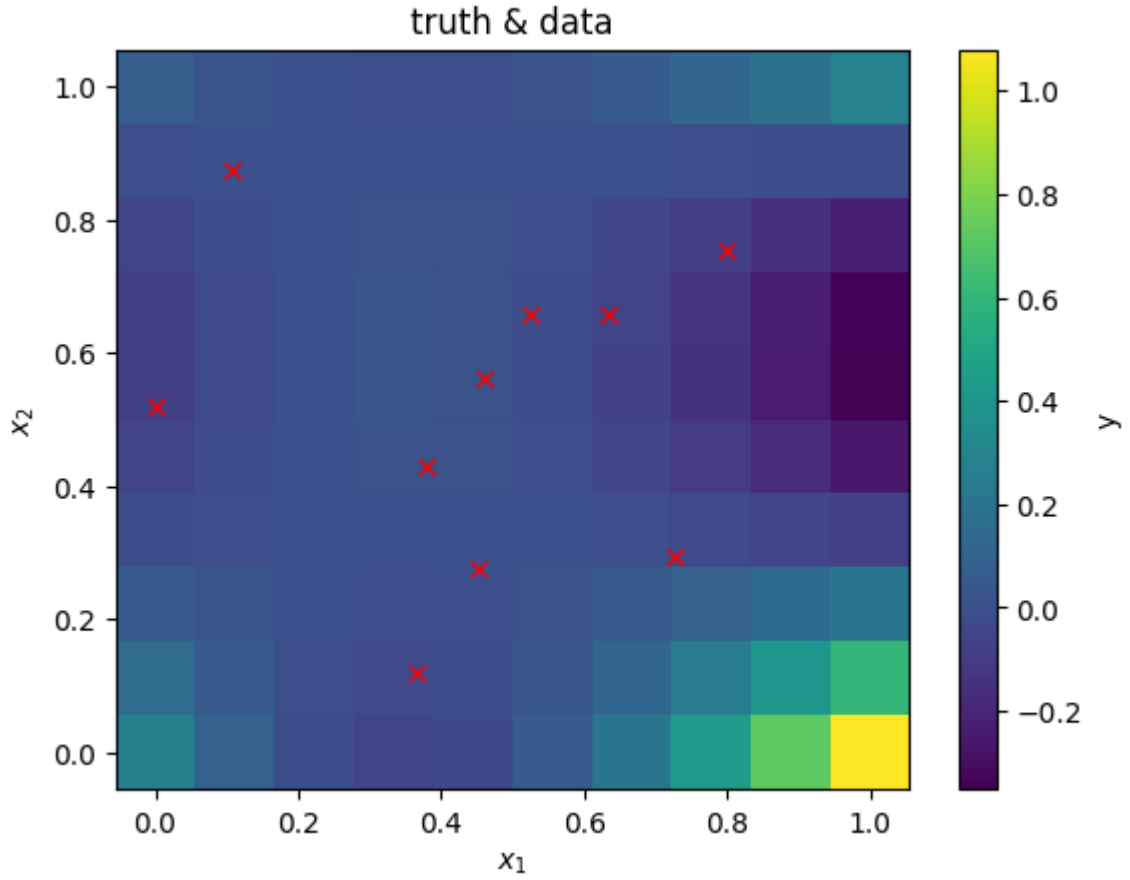


[6]:
```
# fit with Linear Regression
sur = Surrogate["ChaospyLinReg"](sigma_n=0.1, sigma_p=10, model="monomial", order=5)
sur.train(x, y)
mean, var = sur.predict(xx)

mean = mean.flatten()
std = np.sqrt(var.flatten())

ax = plt.subplot()
ax.plot(x, y, "o", label="data")
ax.plot(xx, yy, label="truth")
ax.fill_between(xx, mean + 2 * std, mean - 2 * std, alpha=0.2)
ax.plot(xx, mean, label="model ($2 \\sigma$)")
ax.legend()
ax.set(title=f"Linear Regression (ChaosPy)")
None
```

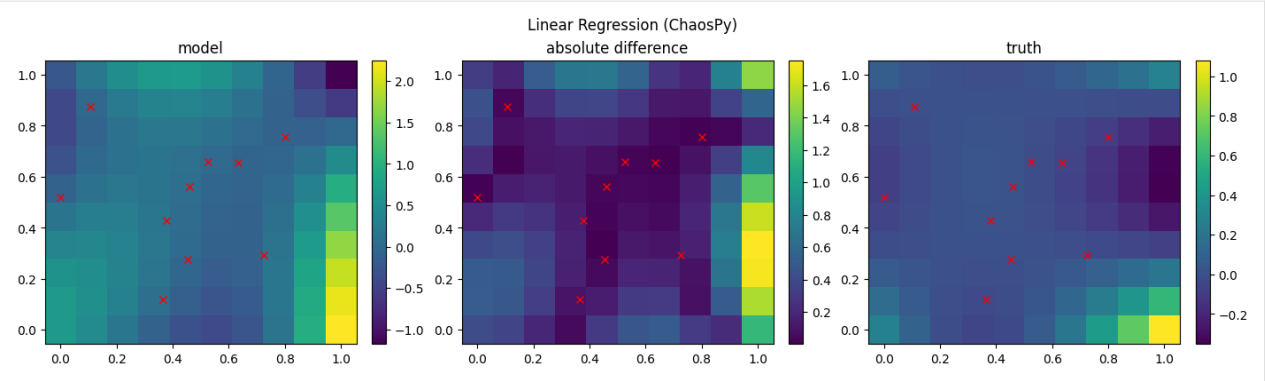


2.2.2 2D

[7]:
```
# generate mockup data
n = 10
f = lambda x, y: 10 * (x - 0.2) * (y - 0.9) * (x - 0.5) * (y - 0.3)

x = np.maximum(np.minimum(rng.normal(0.5, 0.25, (n, 2)), 1), 0)
y = f(x[:, 0], x[:, 1]) + rng.normal(0, 0.05, n)

xx1 = np.linspace(0, 1, 10)
xx2 = np.linspace(0, 1, 10)
xx1, xx2 = np.meshgrid(xx1, xx2)
xx = np.vstack([xx1.flatten(), xx2.flatten()]).T
yy = f(xx1, xx2)

fig, ax = plt.subplots()
ax.plot(x[:, 0], x[:, 1], "rx")
c = ax.pcolormesh(xx1, xx2, yy, shading="auto")
fig.colorbar(c, label="y")
ax.set(xlabel="$x_1$", ylabel="$x_2$", title="truth & data")
None
```



[8]:
```
# fit with Linear Regression
sur = Surrogate["ChaospyLinReg"](sigma_n=0.1, sigma_p=10, model="hermite", order=4)
sur.train(x, y)
mean, var = sur.predict(xx)

mean = mean.flatten().reshape(xx1.shape)
std = np.sqrt(var.flatten())

fig, axs = plt.subplots(1, 3, figsize=(14, 4), constrained_layout=True)
c = axs[0].pcolormesh(xx1, xx2, mean, shading="auto")
fig.colorbar(c, ax=axs[0])
c = axs[1].pcolormesh(xx1, xx2, np.abs(yy - mean), shading="auto")
fig.colorbar(c, ax=axs[1])
c = axs[2].pcolormesh(xx1, xx2, yy, shading="auto")
fig.colorbar(c, ax=axs[2])
axs[0].set(title=f"model")
axs[1].set(title=f"absolute difference")
axs[2].set(title=f"truth")
for i in range(3):
    axs[i].plot(x[:, 0], x[:, 1], "rx")
fig.suptitle(f"Linear Regression (ChaosPy)")
None
```



2.3 The Configuration File

proFit is controlled almost entirely via a single configuration file which is usually called `profit.yaml`. Other filenames are permitted and the respective path can be given as an argument to the profit commands. The configuration is typically written using the YAML file format.

The configuration uses mostly a hierachical structure closely mirroring the structure of the different components. Different implementations of the same component are usually selected using the `class` attribute. A shorter notation uses the label of the component directly and uses default values for all it's parameters. The following two snippets therefore are the same:

```
run:
  interface:
    class: zeromq
```

```
run:
  interface: zeromq
```

There are also shorthand notations for Variables and Encoders.

2.4 Set up simulation

proFit currently distinguishes two types of simulations:

- **executables with input and output files**

  the simulation reads its input parameters from a file, proFit prepares the file using a `Preprocessor`
  the simulation writes its ouput values to a file, proFit read the file using a `Postprocessor`

- **Python simulations which are called from a function**

  the input parameters are the arguments of the function
  the output parameters are the return values
  proFit can wrap this function using a custom `Worker`

For executable simulations, the first step is usually ensuring that they are installed properly and run within a dedicated working directory. Find out which environment variables need to be set and link all relevant files. This directory

will become the template directory which proFit will (usually) copy for each run of the simulation. Furthermore the simulation will just inherit the environment proFit was started in. A typical directory structure could look like this:

```
study/
  profit.yaml
  template/
    simulation.x  ->  /path/to/simulation/executable.x
    params.txt
```

Function simulations can be run without any run directories.

2.5 Variables

The configuration file usually begins with `ntrain:` which set the desired number of data points. The next section is usually `variables`. In it the different input parameters and output values are defined. For the parameters a suitable distribution from which the samples will be taken is selected. Additionally independent variables allow the output values to be vectors and constant values can also be set directly from proFit. More details can be found in *Variables*.

```yaml
ntrain: 100
variables:
  u: Uniform(4.7, 5.3)
  v: Uniform(0.55, 0.6)
  n: 10000
  f: Output
```

2.6 Pre- & Postprocessor

An executable simulation needs a `Preprocessor` and `Postprocessor` to prepare the input parameters and collect the results.

The recommended Preprocessor is the `profit.run.default.TemplatePreprocessor` which will fill placeholders in the template directory with the corresponding values.

For the output values proFit currently supports JSON, HDF5 and CSV/TSV via three different Postprocessors. In addition it is easy to add a custom Postprocessor, see *Custom extensions*. All the configuration options are given in *Configuration*. A possible configuration is given below:

```yaml
run:
  worker:
    class: command
    command: ./simulation
    pre:
      class: template
      path: ./path/to/template_directory  # relative to the base directory
      param_files:
        - params.txt
    post: json
```

The contents of `params.txt` could look like this:

```
# just a plain csv
# u, v, n, m
{u}, {v}, {n}, 10
```

2.7 Python Simulation

For a simulation which can be called from python directly, the recommended configuration is different and uses *Custom extensions* instead. A python function `simulation` which takes the input parameters as arguments and returns the output values can be registered with proFit in the following way:

```
from profit.run import Worker

@Worker.wrap("my_name")
def simulation(u, v) -> "f":
    ...
```

The type annotation is used to tell proFit which return value belongs to which return value if there are several. The configuration is then:

```
include: path_to_simulation.py
run:
  worker: my_name
```

2.8 Interface & Runner

The other two main components of the run system are the `Interface` and the `Runner` itself. These play a vital role if the simulation should be scheduled on a cluster, rather than run locally. For more information see *Cluster Support*.

2.9 Next steps

Everything should be ready to run now:

- calling `profit run` will start start `ntrain` simulations with different parameters and collect their results
- calling `profit fit` will then use these results to fit a surrogate model which is configured in the `fit` section (see *Surrogate models*)
- with *Active Learning* enabled, the fit will already happen during the run step. Active Learning optimizes the paramters at which the simulation is run to gain as much value per simulation run as possible
- finally the results can be explored interactively in a browser after starting a `plotly/dash` server using `profit ui` (see *User Interface*)

There is a wide variety of configuration options to customize the run system, the surrogate fitting and the active learning algorithm. Please have a look at the documentation on the *Configuration* and don't hestitate to contact the developers if you encounter any bugs.

CHAPTER

THREE

CONFIGURATION

The entry point for the user is the configuration file, by default `profit.yaml` which is located in the base directory. Here, parameters, paths, variable names, etc. are stored. For more information on the single modules, see the corresponding section in *Components*.

The configuration file, `profit.yaml` (or a `.py` file containing python dictionaries of the parameters) contains all parameters for running the simulation, fitting and active learning. Examples and a full list of available options, as well as the default values, which are located in the file `profit/defaults.py`, are shown below. For the run system, the default values are documented in the individual classes, see the API reference.

3.1 Structure

The structure of the configuration represents the different modes in which proFit can be executed.

- **Base config**
  Declares general parameters, like number of samples, inclusion of external files and variables.
- **Run config**
  Defines runner, interface and worker, as well as pre- and postprocessing steps.
- **Fit config**
  Sets parameters for the surrogate model.
- **Active learning config**
  AL has a separate configuration, since it can be extensive. Includes choice of algorithm, acquisition function, number of warmup runs, etc.
- **UI config**
  For now, it implements only one option, i.e. to show directly show a figure after calling `profit fit`. It is planned to extend this section to set specific parameters inside the GUI.

All configs set the default values at first, which are then overwritten by the user inputs. Thereafter, inside the method `process_entries`, the user inputs are standardized (e.g. convert relative paths to absolute, convert strings to floats if possible, etc.).

Some parameters are themselves sub configurations, e.g. the runner (local or slurm) or active learning algorithms (standard AL or MCMC). These have themselves again different parameters.

The code structure is similar to the other modules of proFit: a hierarchical class structure, where custom configurations can be registered (see also *Custom extensions*). In the case custom components are implemented, but have no corresponding configuration, a `DefaultConfig` is used, which just returns the user parameters without modifications.

3.2 Examples

Minimal configuration

Parameters that are mentioned in the following description but do not occur in the configuration file are set by default values:

- The configuration executes 10 (`ntrain`) runs of a script (`simulate.x`) locally on all available CPUs when providing the command `profit run`, with one input variable (`x`) drawn from a uniform random distribution on the interval [0, 1] (for further information on variables, see *Variables*. For the run system, see *The Run System*).
- The input file containing the variable x is found in the `template` directory.
- The script writes the output in `json` format to `stdout`. After all runs are finished, the total input and output data is saved to `input.txt` and `output.txt`, respectively.
- Using the command `profit fit`, the default `GPySurrogate` is used to fit the data with initial fit hyperparameters incurred from the data directly and the model is saved to the file `model_GPy.hdf5` (for further information on surrogate models, see *Surrogate models*).
- Thereafter, the data and fit can be viewed in a graphical user interface using `profit ui` (For more information on the UI, see *User Interface*).

```
ntrain: 10
variables:
    x: Uniform()
    f: Output
run:
    command: ./simulate.x
```

Run on cluster

Example for executing a simulation with GORILLA. See *Cluster Support* for more details.

```
ntrain: 100
variables:
    # normalized collisionality
    nu_star: LogUniform(1e-3, 1e-1)
    # mach number
    v_E: Normal(0, 2e-4)
    # Energy in eV
    E: 3000
    # particle species (1 = electrons, 2 = deuterium ions)
    species: 1
    # number of particles (for the monte carlo simulation)
    n_particles: 10000
    # mono energetic radial diffusion coefficient
    D11: Output
    D11_std: Output

run:
    runner:
        class: slurm
        OpenMP: True
        cpus: all
        options:
```

(continues on next page)

(continued from previous page)

```
        job-name: profit-example
        partition: compute
        time: 24:00:00
interface:
    class: zeromq
    port: 9100
worker:
    class: command
    command: ./mono_energetic_transp_main.x
    pre:
        class: template
        path: ./template
        param_files: [mono_energetic_transp_coef.inp, gorilla.inp]
    post:
        class: numpytxt
        path: nustar_diffcoef_std.dat
        names: "IGNORE D11 D11_std"
```

3.3 Full list of options

Below all available options with their respective default values are shown.

3.3.1 Base config

```
base_dir: Current working directory  # Directory where the `profit.yaml` file␣
↪is located.
run_dir: Current working directory  # Directory where the single runs are␣
↪generated.
config_file: profit.yaml  # Name of this file.
include: []  # Paths to external files (e.g. custom components), which are␣
↪loaded in the beginning.
files:
    input: input.txt  # Input variables of all runs.
    output: output.txt  # Collected output of all runs.
ntrain: 10  # Number of training runs.
variables: {}  # Definition of variables.
```

3.3.2 Run config

```
run:
    runner: fork  # Local runner with its default parameters (see below).
    interface: memmap  # Numpy memmap interface with its default parameters.
    worker: command  # Command worker with its default parameters
    debug: false  # override debug for Worker & Runner
```

All runners

```
runner:
    debug: false
    parallel: 0  # maximum number of parallel Workers. 0 means no limit
    sleep: 0.1  # sleep time in s between polling
    logfile: runner.log
```

profit.run.Runner

Fork runner

```
runner:
    class: fork  # For fast local execution
    parallel: all  # Number of CPUs used. 'all' infers the number of␣
↪available CPUs
```

profit.run.local.ForkRunner

Local runner

```
runner:
    class: local  # For local execution.
    parallel: all  # Number of CPUs used. 'all' infers the number of␣
↪available CPUs
    command: profit-worker  # override command to start the Worker
```

profit.run.local.LocalRunner

Slurm runner

```
runner:
    class: slurm  # For clusters with SLURM interface.
    path: slurm.bash  # Path to SLURM script which is generated.
    custom: False  # Use a custom script instead.
    openmp: False  # Insert OpenMP options in SLURM script.
    cpus: 1  # Number of CPUs to allocate per Worker
    options:  # SLURM options.
        job-name: profit
```

profit.run.slurm.SlurmRunner

Memmap interface

```
interface:
    class: memmap  # Using a memory mapped array (with numpy memmap).
    path: interface.npy  # Path to interface file.
```

profit.run.local.MemmapRunnerInterface
profit.run.local.MemmapWorkerInterface

ZeroMQ interface

```
interface:
    class: zeromq  # Using a lightweight message queue (with ZeroMQ).
    transport: tcp  # ZeroMQ transport protocol
    port: 9000  # port of the Runner Interface
    timeout: 4  # connection timeout when waiting for an answer in seconds␣
↪(Worker)
    retries: 3  # number of tries to establish a connection (Worker)
    retry_sleep: 1  # sleep time in seconds between each retry (Worker)
    address: ~  # override ip address or hostname of the Runner Interface␣
↪(default: localhost, automatic with Slurm)
    connection: ~  # override for the ZeroMQ connection spec (Worker side)
    bind: ~  # override for the ZeroMQ bind spec (Runner side)
```

profit.run.zeromq.ZeroMQRunnerInterface
profit.run.zeromq.ZeroMQWorkerInterface

Command Worker

```
worker:
    class: command
    command: ./simulation
    pre: template  # Preprocessor
    post: numpytxt  # Postprocessor
    stdout: stdout  # path to log of the simulation's stdout.
    stderr: ~  # path to log of the simulation's stderr. None means output␣
↪as Worker stderr
    debug: false
    log_path: log
```

profit.run.command.CommandWorker

Template preprocessor

```
pre:
    class: template  # Variables are inserted into the template files.
    clean: true  # whether to clean the run directory after completion
```

(continues on next page)

(continued from previous page)

```
    path: template  # Path to template directory
    param_files: None  # List of relevant files for variable replacement.␣
↪None: Search all.
```

profit.run.command.TemplatePreprocessor

JSON postprocessor

```
post:
    class: json  # Reads output from a json formatted file.
    path: stdout  # Path to simulation output
```

profit.run.command.JSONPostprocessor

Numpytxt postprocessor

```
post:
    class: numpytxt  # Reads output from a tabular text file (e.g. csv,␣
↪tsv) with numpy genfromtxt.
    path: stdout  # Path to simulation output
    names: ~  # Collect only these variable names from output file.
    options:  # Options for numpy genfromtxt.
        deletechars: ""
```

profit.run.command.NumpytxtPostprocessor

HDF5 postprocessor

```
post:
    class: hdf5  # Reads output from an hdf5 file.
    path: output.hdf5  # Path to simulation output
```

profit.run.command.HDF5Postprocessor

3.3.3 Fit config

```yaml
fit:
    surrogate: GPy  # Surrogate model used.
    save: ./model.hdf5  # Path where trained model is saved.
    load: False  # Path to existing model, which is loaded.
    fixed_sigma_n: False  # True constrains the data noise hyperparameter to␣
↪its initial value.
    encoder:
        - class: Exclude  # Exclude constant variables from fit.
          variables: Constant
          parameters: {}
        - class: Log10  # Transform LogUniform variables logarithmically.
          variables: LogUniform
          parameters: {}
        - class: Normalization  # Normalize all input and output variables␣
↪(zero mean, unit variance, n-dimensional 1-cube).
          variables: all
          parameters: {}
    kernel: RBF  # Kernel used for fitting. Also sum (e.g. RBF+Matern32) andd␣
↪product kernels are possible.
    hyperparameters:  # Initial hyperparameters of the surrogate model.
        length_scale: None  # None: Inferred from training data.
        sigma_f: None  # Scaling parameter of surrogate model.
        sigma_n: None  # Data noise (standard deviation).
```

profit.sur.Surrogate
profit.sur.gp.GaussianProcess
profit.sur.gp.custom_surrogate.GPSurrogate
profit.sur.gp.gpy_surrogate.GPySurrogate
profit.sur.gp.sklearn_surrogate.SklearnGPSurrogate
profit.sur.gp.custom_surrogate.MultiOutputGPSurrogate
profit.sur.gp.gpy_surrogate.CoregionalizedGPySurrogate

3.3.4 Active learning config

```yaml
active_learning:
    algorithm: simple  # Algorithm to be used. Either SimpleAL or McmcAL.
    nwarmup: 3  # Number of warmup points.
    batch_size: 1  # Number of candidates which are learned in parallel.
    convergence_criterion: 1e-5  # Not yet implemented.
    nsearch: 50  # Number of candidate points per dimension.
    make_plot: False  # Plot each learning step.
    save_intermediate:  # Save model and data after each learning step.
        model_path: ./model.hdf5
        input_path: ./input.txt
        output_path: ./output.txt
    resume_from: None  # Float of the last run from where AL is resumed with␣
↪saved model and data files.
```

profit.al.active_learning.ActiveLearning

Simple active learning

```
algorithm:
    class: simple  # Standard active learning algorithm.
    acquisition_function: simple_exploration  # Function to select next␣
↪candidates.
    save: True  # Save active learning model after training.
```

profit.al.simple_al.SimpleAL

MCMC

```
algorithm:
    class: mcmc  # MCMC model.
    reference_data: ./yref.txt  # Path to experimental data.
    warmup_cycles: 1  # Number of MCMC warmup cycles.
    target_acceptance_rate: 0.35  # Optimal acceptance rate to be reached␣
↪after warmup.
    sigma_n: 0.05  # Estimated data noise (standard deviation).
    initial_points: None  # List of initial MCMC points.
    last_percent: 0.25  # Fraction of the main learning loop used to␣
↪calculate posterior mean and standard deviation.
    save: ./mcmc_model.hdf5  # Path where MCMC model is saved.
    delayed_acceptance: False  # Use delayed acceptance with a surrogate␣
↪model of the likelihood function.
```

profit.al.mcmc_al.McmcAL

Acquisition functions

Simple exploration

```
acquisition_function:
    class: simple_exploration  # Minimize variance.
    use_marginal_variance: False  # Add variance occurring through␣
↪hyperparameter changes.
```

profit.al.acquisition_functions.SimpleExploration

Exploration with distance penalty

```
acquisition_function:
    class: exploration_with_distance_penalty  # Penalize nearby points.
    use_marginal_variance: False  # Add variance occurring through␣
↪hyperparameter changes.
    weight: 10  # Exponential weight of penalization.
```

profit.al.acquisition_functions.ExplorationWithDistancePenalty

Weighted exploration

```
acquisition_function:
    class: weighted_exploration  # Trade-off between posterior␣
↪surrogate mean maximization and variance minimization.
    use_marginal_variance: False  # Add variance occurring through␣
↪hyperparameter changes.
    weight: 0.5  # Balance between mean and variance: weight * mean_
↪part + (1 - weight) * variance_part
```

profit.al.acquisition_functions.WeightedExploration

Probability of improvement

```
acquisition_function:
    class: probability_of_improvement
```

profit.al.acquisition_functions.ProbabilityOfImprovement

Expected improvement

```
acquisition_function:
    class: expected_improvement  #
    exploration_factor: 0.01  # 0: Only maximization of improvement. 1:␣
↪Emphasize on exploration.
    find_min: False  # Find the minimum of a function instead of the␣
↪maximum.
```

profit.al.acquisition_functions.ExpectedImprovement

Expected improvement 2

```
acquisition_function:
    class: expected_improvement_2  # Same as Expected improvement, but␣
↪with different approximation for parallel AL.
    exploration_factor: 0.01  # 0: Only maximization of improvement. 1:␣
↪Emphasize on exploration.
    find_min: False  # Find the minimum of a function instead of the␣
↪maximum.
```

profit.al.acquisition_functions.ExpectedImprovement2

Alternating exploration

```
acquisition_function:
    class: alternating_exploration  # Alternating between simple␣
↪exploration and expected improvement.
    use_marginal_variance: False  # Add variance occurring through␣
↪hyperparameter changes.
    exploration_factor: 0.01  # 0: Only maximization of improvement. 1:␣
↪Emphasize on exploration.
    find_min: False  # Find the minimum of a function instead of the␣
↪maximum.
    alternating_freq: 1  # Frequency of learning loops to change␣
↪between expected improvement and exploration.
```

profit.al.acquisition_functions.AlternatingExploration

3.3.5 UI config

```
ui:
    plot: False  # Directly show figure after executing `profit fit`. Only␣
↪possible for <= 2D.
```

profit.ui.app

CHAPTER

FOUR

CLUSTER SUPPORT

proFit is desiged to schedule simulations on a cluster. As of v0.4 only the slurm scheduler is supported. If you require a different scheduler consider contributing to proFit. All configuration is done as usual in the study's configuration file (usually `profit.yaml`). Using a provided script is also supported. It is recommended to use the `zeromq` Interface as it is designed to be used with distributed Workers.

You can also start `profit ui` on the cluster and connect to it remotely using *ssh port forwarding*. The UI is usually started on port 8050.

4.1 Example Configuration

```yaml
run:
    command: ./simulation
    runner:
        class: slurm
        OpenMP: true
        cpus: all
        options:
            job-name: profit-sim
            mem-per-cpu: 2G
    interface:
        class: zeromq
        port: 9100
```

Many clusters require specific options for each job like `account`, `mem` or `time`. These can be easily added to using the `runner/options` dictionary, where the key has to be a valid option for the slurm batch script. However some options (`cpus-per-task` and `ntasks` as well as `nodes` and `exclusive` if `cpus: all` is set) are already set internally.

4.2 Troubleshooting

- Each Worker writes a log file (usually into the *study/log* directory).
- A failed run is detected but is usually just missing from the output data, which causes the output to be misaligned with the input data. The current workaround is to delete the relevant lines from the input file manually.
- `profit run` can be started from a login node (it shouldn't use many resources) but sometimes `zeromq` can't connect from a worker node to the login node. Try `srun profit run` instead, as proFit will detect the correct host unless the `connect` address is overridden.

For more information and to submit the bugs you encountered visit the Issue Tracker on GitHub.

4.3 Available Options

- *profit.run.slurm.SlurmRunner*
- *profit.run.zeromq.ZeroMQRunnerInterface*

CHAPTER

FIVE

CODE STRUCTURE

The package consists of six main modules:

- **Run system**
  Start and manage simulation runs and collect the results.
- **Surrogate models**
  Encode and fit the data.
- **Active learning algorithms**
  Combine run system and surrogate modelling to consecutively find next input points.
- **User interface**
  Interactively view the results and fit in a Dash web server.
- **Configuration**
  Read and standardize user inputs from supplied files.
- **Utility functions**
  Definition of an abstract base class, variables and other useful functions.

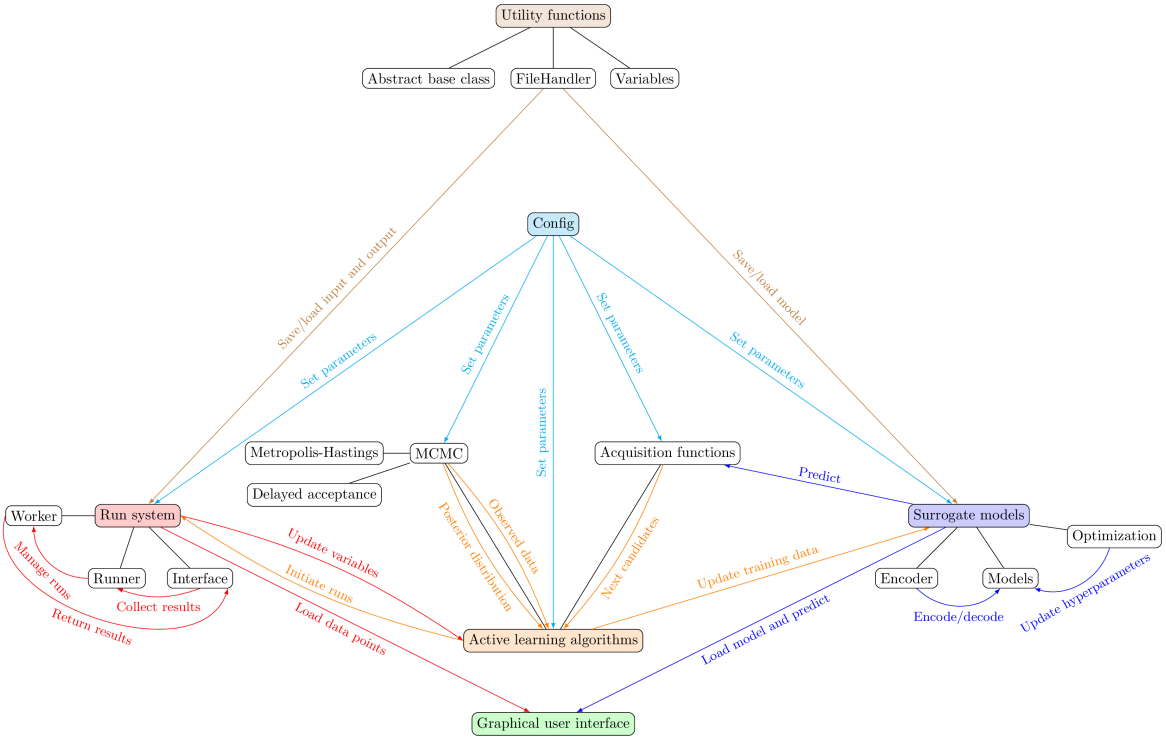


Fig. 1: proFit code structure

CHAPTER

SIX

COMPONENTS

This section lists detailed information about the different modules of proFit.

6.1 Variables

- **Input variables**
  Drawn from random distributions:
  - *Halton* sequence (quasi-random, space filling)
  - *Uniform* distribution
  - *LogUniform*: uniformly distributed in log-space
  - *Normal* distribution

  Fixed values:
  - *Linear*: linearly spaced values
  - *Constant* (excluded from the fit)

  Special:
  - *ActiveLearning*: succesively inserted according to a specified optimization strategy
- ***Independent* variables**
  The user can bind an independent variable to an output variable, if the simulation outputs a (known) vector over linear supporting points. This dimension is then not considered during fitting, in contrast to full multi-output models. This lowers necessary computing resources and can even enhance the quality of the fit, since complexity of the model is reduced.
- ***Output* variables**
  Default output is a scalar value, but with the attachment of independent variables, it becomes a vector. In the config file, also several output variables can be defined independently, which leads to multi-output surrogates during fitting. This is useful if the simulation outputs additional variables, e.g. the standard deviation or the derivative of the result.

All single variables are then stored in a VariableGroup , which is the main object for the runner, active learning and surrogates to interact with variables. This class implements efficient methods to get and set values of single variables, as only a view on the true variable objects is stored. Especially important for the active learning workflow, there is never a discrepancy between values of the runner and the surrogate model, as both access the same view.

6.1.1 Examples

Definition of variables inside the *profit.yaml* configuration file.

```yaml
ntrain: 100
variables:
    # Inputs
    a1: Uniform()  # Uniform random distribution in [0, 1].
    a2: Uniform(0, 1)  # Same as 'a1'
    b: Normal(0, 1e-2)  # Normal distribution with 0 mean and 1e-2 standard deviation.
    c1: 0.2  # Constant value.
    c2: Constant(0.2)  # Same as 'c1'.
    d: LogUniform(1e-4, 0.1)  # LogUniform distribution.
    e: Halton(0, 3)  # Quasi-random Halton sequence.
    h: Linear(-1, 1)  # Linear vector with size of 'ntrain'.
    al1: ActiveLearning(1e-4, 1e-1, Log)  # Active learning variable with␣
↪logarithmically distributed search space.
    # Independent variable
    t: Independent(0, 99, 100)  # Linear vector with 100 supporting points.
                                # Not considered as separate input, but simulation␣
↪returns vector.
    # Outputs
    f: Output(t)  # Vector output dependent on t.
    g: Output  # Scalar output.
```

Variables can be declared as strings as shown above, or with the full dict-like structure:

```yaml
variables:
    a1:
        class: Uniform
        constraints = [0, 1]
        dtype: float
    c3:
        class: Constant
        value: 3
        dtype: int
    al1:
        class: ActiveLearning
        distr: Log
        constraints: [1e-4, 1e-1]
    ...
```

Describing variable placeholders *a1* and *d* inside a simulation input file (*.txt* and *.json*).

```
# Example input file for simulation
path1='./some_path'
parameter1={a1}
parameter2={d}
...
```

```
{
    "path1": "./some_path",
    "parameter1": "{{a1}}",
```

(continues on next page)

(continued from previous page)

```
    "parameter2": "{{d}}"
}
```

6.2 The Run System

This documentation was taken directly from: R. Babin, "Generic system to manage simulation runs and result collection for distributed parameter studies", Bachelor's Thesis (Graz University of Technology, Graz, Austria, 2021). Parts of it were modified to reflect the changes to *proFit* since then.

A core feature of *proFit* is the ability to start and manage simulations. This part of *proFit* is referred to as the *run system*. A particular design challenge is the balancing of two often conflicting interests: on one side, *proFit* should do as much as possible automatically to reduce the work which the user has to do and on the other side, *proFit* should be customizable to meet specific needs a user might have.

6.2.1 Requirements

The primary use of the run system is sampling the parameter space in an effective manner. The system should therefore take arrays of input parameters, launch the simulation in parallel for each set of parameters and return the output data when the simulations have completed. Additionally, advanced fitting methods (e.g. *active learning*) iteratively determine which parameters should be scheduled next. In the future an interactive user interface should allow starting new simulations as well as monitoring currently scheduled or running ones.

It should not be necessary to adjust the simulation to be able to use it with . Therefore it has to be possible to configure how *proFit* supplies the simulation with input arguments and how the output data is collected. In addition to supporting a few common methods, the user should have the possibility to customize *proFit* to support any simulation.

Large simulations require many resources and are mostly run on clusters. The run system should be designed to schedule jobs on a given cluster and needs to be able to communicate with the simulations in a distributed manner. The other extreme use case for *proFit* would be a large number of inexpensive simulations running locally, which require the run system to have as little overhead as possible.

Finally, the program structure and API should be clean and organized to facilitate open source contributions and allow a later expansion of this system. The configuration and customization should also be cleanly structured and easy to understand even for new users.

6.2.2 Design

To fulfill the requirements above, it was decided to use a modular design with generic components. A *Runner* should manage all the simulation runs and start the simulations, while each of them is managed by a *Worker*. The *Worker* prepares the input for the simulation and collects its output after it has completed. To make the previously generated or user supplied parameters available to the simulation and to collect all the outputs together, the *Runner* needs to communicate with all simultaneously running *Workers*. This preliminary design is drawn in figure 1.

To support arbitrary simulations, the processes of preparing the input for the simulation and collecting its output need to be handled by generic (and thereby adaptable) components. Depending on the system on which the simulations should be run, the method of starting the simulations (or more precisely starting the *Workers*, which is done by the *Runner*) and the method of communication need to be adaptable. Finally, for reasons of symmetry and to reduce the overhead for simulations written in *Python*, the *Worker* should be generic and adaptable as well. To meet these requirements, the run system was divided into five generic components (see figure 2) which can come in different variations.

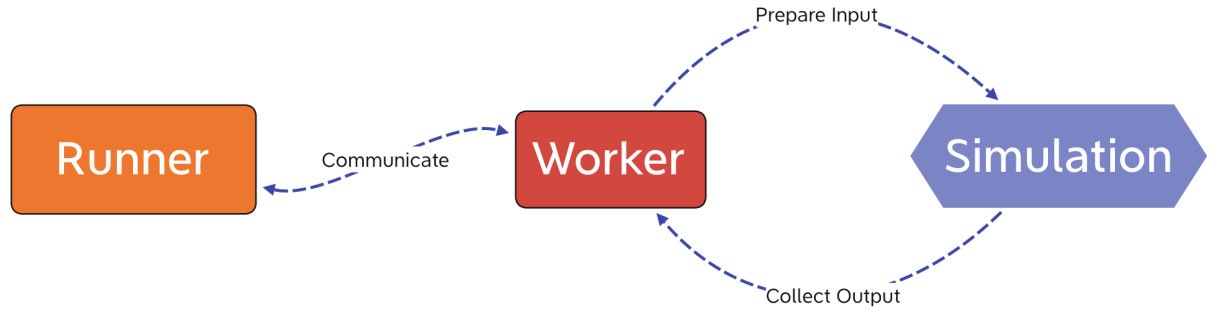


Fig. 1: Figure 1: Preliminary design of the *proFit* run system.

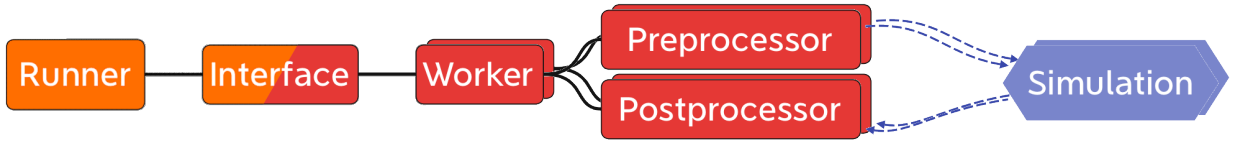


Fig. 2: Figure 2: The components of the *proFit* run system.

The central component is the *Runner* which interacts with the user and manages the individual runs. Each simulation run is represented by a *Worker* while an *Interface* provides the communication between the *Runner* and all *Workers*. The *Interface* consists of two strongly interacting parts which handle the *Worker* and *Runner* side respectively. To supply the input arguments to the simulation, the *Worker* uses a *Preprocessor* and to collect the output data a *Postprocessor* is used. Alternatively simulations which provide a python callable can be wrapped directly with a custom *Worker*. The data is then passed directly and no *Pre-* and *Postprocessor* is needed. The *Runner* and its corresponding part of the *Interface* are persistent while the other components exist for each run of the simulation separately.

Starting `profit run` initializes the *Runner* together with its corresponding *Interface*. When the user requests a new run, the *Runner* spawns a new *Worker*. This new *Worker* initializes itself with the provided configuration and environment and also initializes its part of the *Interface*. Then the *Preprocessor* is started with the input arguments which the *Runner* transmits via the *Interface*. The *Worker* is now ready and starts the simulation. After the simulation has completed, the *Postprocessor* reads the simulation's output and passes the collected data via the *Interface* back to the *Runner*. Finally, the *Worker* signals the *Runner* that the run is completed and ensures that everything is cleaned before it exits.

Each variant of a component is marked with an identifier which allows the user to select the variant in the configuration using this identifier. Each component has its independent section in the configuration. A user can add custom variants in the exact same way the default components are implemented by specifying the file with the custom components in the configuration. As this approach allows the user to execute arbitrary code, *proFit* should never be run in a privileged mode.

6.2.3 Implementation

Each component was implemented using classes in *Python* with each variant being a subclass of the respective abstract base class[1]. The two parts of the *Interface* are represented by two separate class hierarchies, which work in parallel using the same identifiers.

To facilitate adding new variants and to allow the user to provide a custom variant of a component, a class-decorator was provided for each component which registers the new variant so that it can be called using its identifier. Alternatively the label of the new variant can also be configured directly using an argument at the class definition. To select a specific variant of a component, it's identifier has to be selected in the configuration as the respective component's *class* (see for example listing 2). For the most commonly customized components *Preprocessor*, *Postprocessor* and *Worker*, which all override only a single method in most cases, an additional function-decorator was provided which wraps a given function to provide the same functionality as the subclass with minimal effort (see for example listing 3).

The default components of *proFit* are implemented using the same decorators and one of the default components, the *JSON Postprocessor* is given in listing 1. The *retrieve* method is the main method which is called to process the simualtion's output. An example of the configuration of the *JSON Postprocessor* is given in listing 2.

```
from profit.run import Postprocessor

    @Postprocessor.wrap("json", config=dict(path="stdout"))
    def JSONPostprocessor(self, data):
        """Postprocessor to read output from a JSON file

        - variables are assumed to be stored with the correct key and able to be␣
↪converted immediately
        - not extensively tested
        """
        import json

        with open(self.path) as f:
            output = json.load(f)
        for key, value in output.items():
            if key in data.dtype.names:
                data[key] = value
```

Listing 1: Registering a Postprocessor with the identifier json, to read simulation output in the JSON file format. Part of the default components of proFit [proFit].

```
run:
    runner:
        post:
            class: json
            path: simulation_output.json
include: path/to/my_custom_json_postprocessor.py
```

Listing 2: The post-section of the YAML-configuration file to select the JSON Postprocessor defined in listing 1.

To reduce the overhead for a simulation or for testing purposes it might be beneficial to let the *Worker* call a *Python* function directly instead of starting a simulation via a system call. An example for a *Worker* subclass, which uses a python function instead of an executable, is given in listing 3, using the *wrap*-decorator discussed earlier to reduce the necessary code overhead (see listing 4 for the corresponding configuration).

[1] An *abstract base class* defines the relevant methods and attributes but has no implementation and cannot be instantiated. Only subclasses which implement all abstract methods can be instantiated.

```
@Worker.wrap('new_worker')
def simulation(u) -> 'f':
    return np.cos(10 * u) + u
```

Listing 3: Registering a new Worker with the identifier python_worker, the input parameter u and output value f = cos(10 u) + u using the wrapper. Adapted from the tests of proFit [proFit].

```
run:
    worker: new_worker
include: path/to/my_custom_worker.py
```

Listing 4: The YAML configuration to select the custom Worker defined in listing 3. Adapted from the tests of proFit [proFit].

6.2.4 Components

A number of default components were added to provide basic functionality and to fulfill the different requirements.

Template Preprocessor

The *Template Preprocessor* copies a given template directory for each run and replaces special template variables within the files with the generated values for this run. Simulations which read input parameters from files can be supplied with different variables in this way easily. This functionality has been a part of *proFit* before, but has been adapted to the new system and received some small improvements. With this default component all current *Preprocessor* requirements are fulfilled and no additional variant is needed.

Postprocessors

Many simulations use a tabular format (e.g. *CSV*) for their output files. With the *Numpytxt Postprocessor* most of them can be processed easily and configuration options can be passed directly to the underlying `numpy.genfromtxt` function. Two other commonly used file formats, *JSON* and *HDF5*, are also supported with their respective *Postprocessors* to showcase the relative ease of adding new components.

Local Runner

The *Runner* is the core of the run system and by default it executes the *Workers* locally. Each new simulation run is launched in a separate process.

Fork Runner

For fast simulations (e.g. during testing) the *Local Runner* causes significant overhead as each *Worker* is started via a subprocess and has to reload all packages. To circumvent this, the *Fork Runner* now uses forking which allows the child process (a *Worker*) to inherit the loaded package with little overhead.

Slurm Runner

One of the core goals of this project was the utilization of the cluster scheduler *Slurm* instead of the local system. With the *Slurm Runner* each run of the simulation is scheduled as a job with groups of runs being scheduled as *job arrays*. The scheduler can be queried at specified intervals to detect failed or cancelled jobs. Parallelised simulations using *OpenMP* are supported, as well as passing arbitrary options through to the *Slurm* scheduler, like the job's required memory. By default the *Slurm Runner* generates a *Slurm script*, but it can be configured to take a user supplied script instead.

Memmap Interface

The default *Interface* uses a memory mapped file which allows all *Workers* and the *Runner* to access the same *numpy array*. Special care was taken to ensure that each component only accesses a small part of the mapping and each part is only written from one place to prevent race conditions.

ZeroMQ Interface

On a cluster, a file based *Interface* is problematic as the distributed file system is not fully synchronized. The *ZeroMQ Interface* uses the lightweight message queue *ZeroMQ* [2] instead. The required information is transmitted using binary messages over a configurable transport system (by default *tcp*), which allows efficient communication across the network.

6.3 Development Notes: Run System

For an overview, requirements and usage, see: *The Run System*

6.3.1 Components & Sections

- the `run` configuration contains a few global options (`debug`) and three subsections: `runner`, `interface` & `worker` * `debug` will override `runner.debug` and `worker.debug`
- the `command` Worker has another two subsections: `pre` and `post` * to retain compatiblity `pre` and `post` may also be specified a level higher. * the shorthand `command: ./simulation` will be expanded to `runner: {class: command, command: ./simulation}`
- the options in each section are mostly the same as the arguments for `__init__`, some components have additional arguments which are set during the program flow (e.g. the `run_id` for the Worker)
- if a Component takes sub-Components as arguments it should also support a config-dictionary, or a single string label

[2] The homepage of *ZeroMQ* is found at https://zeromq.org

6.3.2 Paths & Directories

The run system contains two major components: the *Runner* and many *Workers*. Additionally the *Interface* connects these two layers. *proFit* is started from some *base directory* (`base_dir`), which contains the configuration and simulation files. The generated data will be written to this directory. The *Runner* can be configured to use a different *temporary directory* (`work_dir`) which will contain temporary files (e.g. used by the *Interface*, logs and the individual run directories). Most temporary files are deleted upon successful completion, but the logs will remain. Using `profit clean --all` will delete the logs as well as the input and output data.

The *Command-Worker* will create *individual run directories* within the temporary directory for each spawned *Worker* by copying a template directory and replacing template expressions with parameter values. Unless `clean: false` is set for the *TemplatePreprocessor*, the individual run directories will be deleted after the run has completed successfully.

Most paths in the configuration, including the path to the template directory, will therefore be relative to the base directory. The paths to *Interface*-files and for log-files will be relative to the temporary directory and paths within the worker configuration (including pre and post) will be relative to the individual run directories.

6.3.3 Enivronment variables

Most of these environment variables are only set if required.

- `PROFIT_BASE_DIR` - absolute path to the base directory
- `PROFIT_INCLUDES` - JSON list of absolute paths to python files which need to be imported (e.g. contain custom components)
- `PROFIT_RUN_ID` - the `run_id` to identify the *Worker*, set for each *Worker*
- `PROFIT_ARRAY_ID` - modifier to the `run_id`, needed for batch computation on clusters
- `PROFIT_WORKER` - JSON configuration of the *Worker*
- `PROFIT_INTERFACE` - JSON configuration of the *Interface*
- `PROFIT_RUNNER_ADDRESS` - hostname/address of the *Runner*
- `SBATCH_EXPORT = ALL` - Slurm: load the full environment as passed by the *SlurmRunner*
- `OMP_NUM_THREADS` - OpenMP: number of threads (*SlurmRunner*)
- `OMP_PLACES = threads` - OpenMP: position of threads (*SlurmRunner*)

6.4 Surrogate models

6.4.1 Overview

Fitting the data with the command `profit fit` is accomplished by Gaussian process regression (GPR) surrogate models.

Currently, there are three such models implemented in proFit, which differences are explained further below:

- Custom (built from scratch in proFit)
- GPySurrogate (based on GPy)
- SklearnGPSurrogate (based on scikit-learn)

There are also two multi-output models:

- **CustomMultiOutputGP (based on the Custom surrogate)**
  Implements simple, independent GPs in every output dimension.
- **CoregionalizedGPy (based on the GPySurrogate)**
  Uses the coregionalization kernel from GPy to model dependencies between outputs.

Under development:

- Artificial neural network surrogate (using PyTorch)

6.4.2 Design

The hierarchy of implemented models is shown in the figure below. The universal `Surrogate` class is the overall base class of surrogates, which functions as abstract base class where models are registered. For experienced users, this allows implementing fully customized surrogates, starting from importing one of the available classes and modifying only one method, to writing new classes from scratch which integrate in the workflow by standard methods like `train`, `optimize`, `predict`, `save_model`, and `load_model`. These methods are used throughout every sub class to ensure compatibility between surrogates, encoders, and active learning algorithms:

- **`train`**
  Trains the model on a dataset. After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data $\mathbf{y}$ by optimizing the model's hyperparameters, which is done by minimizing the negative log-likelihood.
- **`optimize`**
  Finds the optimal hyperparameters of the model to ensure consistency with observed data.
- **`predict`**
  Predicts the output at test input points X_* by building the posterior mean and covariance using previously trained data.
- **`save_model`**
  Saves the model and used encoding as Python dictionary to a *.hdf5* file. For the coregionalization multi-output surrogate from GPySurrogate , a *.pkl* file is used instead.
- **`load_model`**
  Loads a saved model from a *.hdf5* (or *.pkl*) file, updates its attributes and restores the encoding.

To be able to use the surrogate in active learning, another two methods are necessary:

- **`add_training_data`**
  Adds X and $\mathbf{y}$ data to the model.
- **`add_ytrain`**
  Overwrites the placeholder $\mathbf{y}$ data inserted during active learning with the actual data.

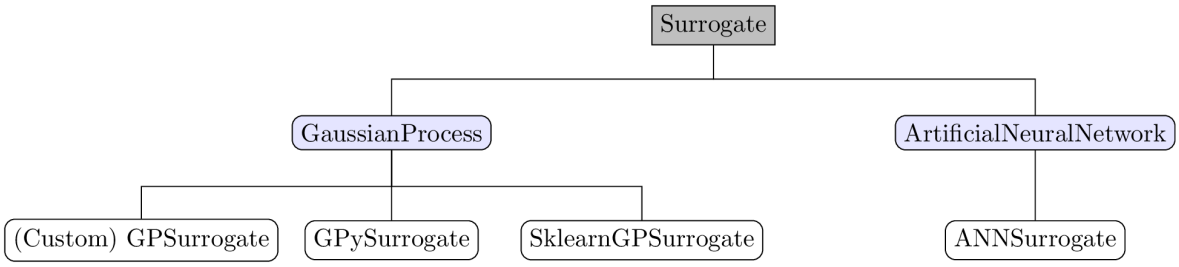


Fig. 3: Surrogate hierarchy. Grey: base class, blue: intermediate helper class, white: user selectable classes.

Each of the above mentioned GP surrogate models has its own advantages, the most important of which for the use in proFit are displayed in the following table:

Surrogate (Identifier)	Advantages
GPSurrogate (Custom)	Built from scratch. Well suited for study purpose. Customizable kernels in Python and Fortran. Customizable optimization (Laplace approximation, include derivatives, etc.).
GPySurrogate (GPy)	Large number of selectable models and kernels. Multi-output models with coregionalization available. Small package size.
SklearnGPSurrogate (Sklearn)	Low level customization possible. Good integration with other scikit-learn libraries.

6.4.3 Kernels

For the `GPySurrogate` and the `SklearnGPSurrogate` the respective kernels of their packages can be called by their corresponding class name. The kernels of the custom `GPSurrogate` are implemented by default in Python and Fortran, where the RBF, Matern32 and *Matern52* kernels are available as of now.

The automatic relevance detection (ARD) feature of the GPy kernels can be used for dimensionality reduction, as dimensions with large length-scales are excluded from the fit which makes the model less complex.

6.4.4 Encoders

Encoding of input and output data is an important topic as well, as it can make surrogate models more reliable. The most important encodings in proFit include

- **Normalization**
  The fit is always executed on the n-dimensional 1-cube with zero mean and unit variance, which usually makes the surrogate models more reliable for data with a large range. It is also planned to implement an encoder to handle heteroscedastic data.
- **Exclusion**
  Specified dimensions of the data are neglected from the beginning which makes the fitting procedure more efficient, as less dimensions have to be considered.
- **Log10**
  The transformation $log(x)$ is applied which can reduce model complex- ity, as e.g. a linear model can be

fitted instead of an exponential one.

- **Dimensionality reduction**
  Transformations can be applied, e.g. principal component analysis (PCA) and Karhunen-Loeve decomposition (KL) which contribute to minimizing computational effort of fitting high dimensions. The difference between PCA and KL here is that PCA is conducted with the collocation matrix $M = X^T \cdot X$ which is efficient if the number of samples is greater than the number of variables, while KL uses the covariance matrix $C = X \cdot X^T$ which is efficient if the number of variables is larger than the number of samples.

By default, the following encoder pipeline is used:

1. Exclude `Constant` input variables
2. Logarithmically transform `LogUniform` input variables
3. Normalize all input variables
4. Normalize all output variables

Custom encoders can be registered to the base `Encoder` class, as described generally in *Custom extensions*.

6.4.5 Examples

```
fit:
    surrogate: GPy
    save: model.hdf5  # Automatically becomes model_GPy.hdf5 for identification.
    fixed_sigma_n: False  # Fix noise in the beginning.
    kernel: RBF  # Also possible, e.g.: `Matern32`, `Matern52`, etc.
    hyperparameters:  # Initial hyperparameters. Inferred from training data if not␣
↪given.
        length_scale: 0.1
        sigma_f: 1.0
        sigma_n: 0.01
    encoder:
        - Exclude(Constant)  # Applied on `Constant` variables.
        - Log10(LogUniform)  # Applied on `LogUniform` variables.
        - Normalization(all) # Applied on all input and output variables.
```

```
fit:
    surrogate: CoregionalizedGPy  # Use coregionalization multi-output surrogate.
    save: model_CoregionalizedGPy.pkl  # Only saving to `.pkl` is implemented for this␣
↪surrogate.
    encoder:
        - Log10(input)  # Transform all input variables.
        - KarhunenLoeve(output)  # Use dimensionality reduction encoder on output.
        - Normalization(all)  # Normalize all input and output variables.
```

```
fit:
    surrogate: GPy
    load: model_1.hdf5  # Load already trained model.
```

6.5 Active Learning

Besides running simulations and building surrogate models afterwards, proFit provides algorithms to execute this simultaneously by active learning (AL). AL allows to reduce the necessary number of training points to reach a specific fit quality compared to sampling from a random or Halton distribution, as points can be chosen directly where most information is gained. proFit implements two algorithms by default, called `SimpleAL` and `McmcAL`. The former serves as the default active learning algorithm by which consecutive points are chosen according to some acquisition function, like minimization of the fit variance or finding the maximum of a function.

The second provided method is a Markov-Chain-Monte-Carlo (MCMC) algorithm, which is a somewhat different use case of proFit, compared to the other modes and the SimpleAL algorithm, as it makes use of already observed experimental data and a parameterized model to find the posterior distribution of these parameters, instead of fitting a non-parametric Gaussian process. It implements the additional feature of delayed acceptance, which further reduces computational effort. The MCMC algorithm is discussed separately in *Markov-Chain Monte-Carlo*.

In the `SimpleAL` algorithm a `warmup` method is implemented, where initial points are sampled randomly, to receive a first estimate of the surrogate model. Thereafter in the `learn` method, where the main loop is contained, the acquisition function finds the next best points which are injected into the run queue. The model and fit are updated and the successive next best points are searched. This workflow is shown graphically in the figure below:

proFit automatically activates AL if the user inserts an `ActiveLearning` variable inside the `profit.yaml` configuration file. The corresponding search interval is given as parameter of the variable, defaulting to $[0, 1]$. As optional parameter, the AL variable can be set to log-space, which means the search space is logarithmically transformed. This especially makes sense if the user wants to explore a large space. Multi-dimensional active learning is also possible by defining several AL variables. In this multi-dimensional case, Gibbs sampling is used, as of now. Furthermore, AL and non-AL variables can be combined to perform active learning only in certain dimensions. The distinctive parameters of the AL algorithm and acquisition functions are adjusted in the section `active_learning` of the config file. A special parameter to be mentioned is the `batch_size`, which allows to search for several points in parallel. This is especially useful, as today's computing clusters are much better utilized if computing in parallel.

6.5.1 Acquisition functions

The task of an acquisition function is to find points which yield the most information according to a loss function and should be explored next by the simulation. An overview of in proFit implemented acquisition functions and a short description of their purpose is shown in the table below:

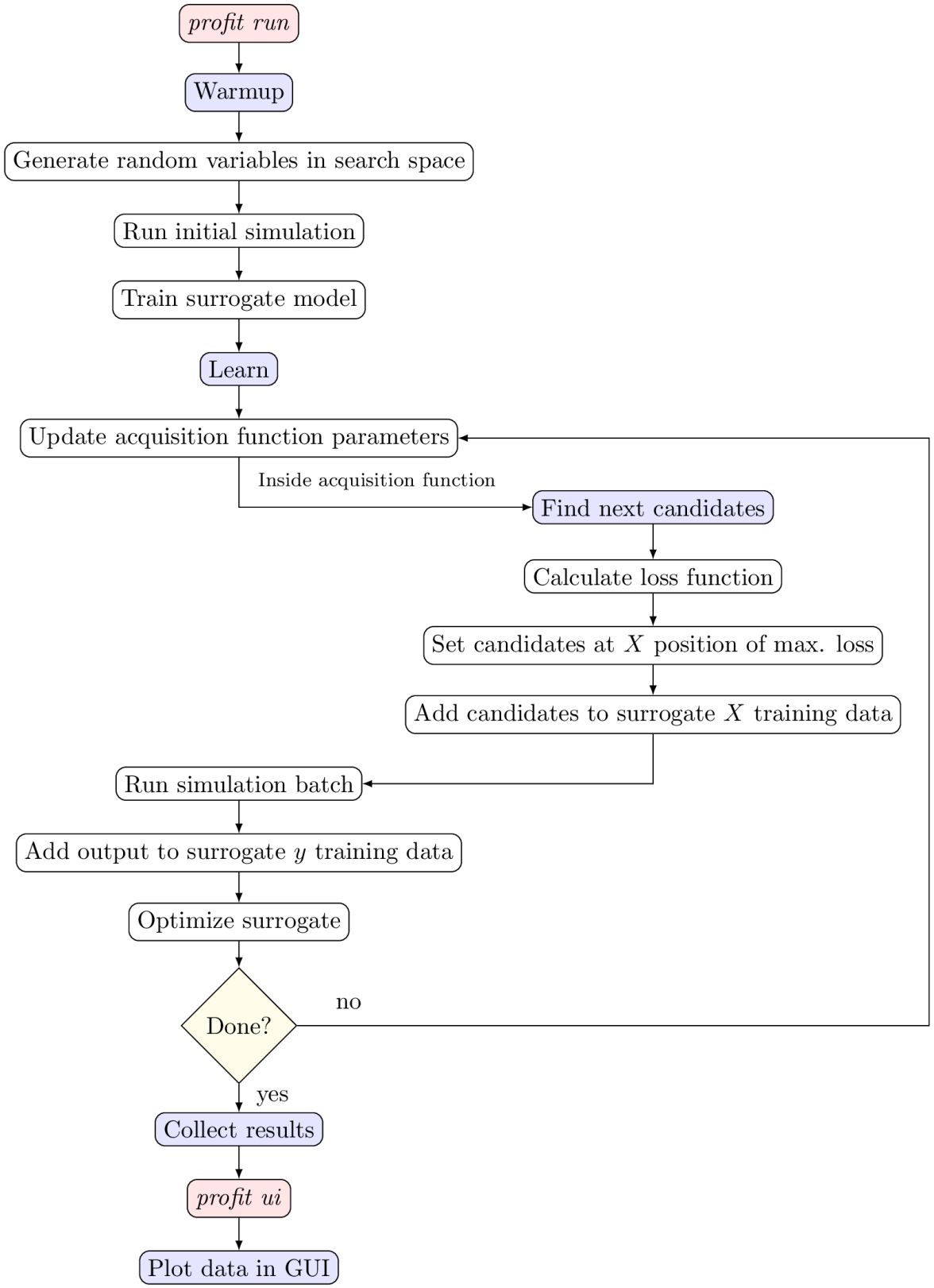


Fig. 4: Active learning workflow

Acquisition function (Identifier)	Task
Simple exploration (simple_exploration)	Minimize fit variance. Optional `bool` parameter: `use_marginal_variance`. take into account derivative of hyperparameters (only for Custom surrogate).
Exploration with distance penalty (exploration_with_distance_penalty)	Minimize fit variance. Exponential penalization of nearby points with the `weight` parameter.
Weighted exploration (weighted_exploration)	Minimize variance. Or maximize surrogate mean. Adjustable by the `weight` parameter.
Probability of improvement (probability_of_improvement)	Find global maximum or minimum. Adjustable exploration part to avoid confinement in local extremum. (Parameter `exploration_factor`)
Expected improvement (expected_improvement)	Find global maximum or minimum. Adjustable exploration part to avoid confinement in local extremum. (Parameter `exploration_factor`)
Alternating exploration (alternating_exploration)	Alternating simple exploration and expected improvement during learning cycles. Parameters of `simple_exploration` and `expected_improvement` as well as `alternating_freq` to indicate the frequency of changing between the two acquisition functions.

6.5.2 Parallel active learning

With today's supercomputers, parallel computing has become an integral part of numerical simulations. Even for expensive simulations it is thus more efficient to select several points at once instead of evaluating one point after another. For active learning this means also the search for candidates has to be parallelized to find a next best batch instead of the next best point. For the simple exploration acquisition function, this is done straightforwardly, as the predictive variance $\mathbb{V}$ does not depend on the actual simulation output $\mathbf{y}$:

$$\mathbb{V}[\mathbf{y}_*] = K_{**} - K_*^T (K + \sigma_n^2 I)^{-1} K_*$$

with $K = k(X, X)$ the training kernel matrix, $K_* = k(X_*, X)$ and $K_{**} = k(X_*, X_*)$. X represents the training input points, X_* the prediction input points, $\mathbf{y}_*$ the prediction output and I the identity matrix and σ_n the data noise.

For other acquisition functions that depend on the evaluated output $\mathbf{y}$, approximations with the predictive fit itself, i.e. with $\mathbf{y}_*$, have to be made.

As an example, in the expected improvement acquisition function, the exploration and exploitation parts are split, so the exploration can be fully parallelized and updated for each point in the batch. The exploitation term, on the other hand, is approximated by the mean function of the given surrogate model at the beginning, and is not updated throughout the batch.

6.5.3 Examples

```
ntrain: 50  # Points in total (AL warmup + AL learn).
variables:
    u: ActiveLearning()  # AL variable in interval [0, 1].
    v: ActiveLearning(1e-4, 1, Log)  # AL variable with log search space.
    mu: Normal(0, 0.2)  # Non-AL variable (Gaussian distributed).
    f: Output  # Scalar output.

run:
    ...  # Usual run configuration

fit:
    surrogate: GPy  # Use this surrogate during AL.

active_learning:
    nwarmup: 10  # Randomly sampled warmup points.
    batch_size: 1  # Sequential learning.
    algorithm:
        class: simple  # SimpleAL
        acquisition_function: simple_exploration  # Minimize fit variance.
```

```
...
active_learning:
    nwarmup: 4
    batch_size: 16  # Parallel learning
    algorithm:
        class: simple
        acquisition_function:
            class: alternating_exploration  # Alternate `simple_exploration` and␣
↪`expected_improvement`
            exploration_factor: 0.1  # Exploration factor for expected improvement.
```

(continues on next page)

(continued from previous page)

```
        find_min: True  # Find function minimum instead of maximum.
        alternating_freq: 2  # Do exploration twice, then expected improvement twice.
```

6.6 Markov-Chain Monte-Carlo

The Markov-Chain-Monte-Carlo algorithm (MCMC) is closely related to active learning but candidate points are sampled from a distribution and either accepted or rejected according to their posterior probability rather than actively learned. Furthermore, MCMC focuses on finding the highest posterior distribution of the parameters in place of optimizing a loss function as in active learning.

Given observed data and a simulation model with certain (physical) parameters, an MCMC simulation estimates the best posterior distribution of these parameters.

The ingredients to an MCMC algorithm are the following:

- **Data**
  Observed (experimental) data. Which path is given in the configuration as parameter `reference_data`.
- **Model simulation**
  Usual simulation to model the observed data with parameter vector θ.
- **Sampling algorithm**
  Method, how next input points to the simulation are chosen. In proFit the Metropolis-Hastings algorithm is used.
- **Likelihood function**
  Function to compare candidate points with already sampled points. In proFit, a Gaussian likelihood is chosen.

As the initial MCMC point starts at a random position in search space, warmup cycles are necessary to reach the desired area of high posterior probabilities. A widely used target acceptance rate of accepted versus rejected points is the well known asymptotically optimal target acceptance rate of 0.234. This value has to be treated with caution, as there may be significantly better suited acceptance rates for a particular problem. The target acceptance rate can be adjusted in the configuration with the parameter `target_acceptance_rate`. Given a specific target acceptance rate, the step size of how far in some direction in parameter space the next point is chosen, is adapted throughout the warmup cycles. After all warmup cycles are complete, the warmup points are discarded and the actual MCMC run, which yields the final posterior distribution with uncertainties for the parameters, is executed. The config parameter `last_percent` specifies which fraction of the main MCMC run is used for calculating the mean and variance of the posterior distribution. This lasts normally until a convergence criterion is satisfied or the maximum number of MCMC runs is reached. The log-likelihoods of each step are saved to `./log_likelihohod.txt` and the parameter's final mean and standard deviation are saved to `./mcmc_stats.txt`.

A special feature of the proFit MCMC algorithm is delayed acceptance (DA), which uses a surrogate model of the loss function to estimate its expected minimum and can thus reject prospected MCMC points by the surrogate model, already before an (expensive) simulation run is started, which can reduce computation time. As of now, DA can be utilized only after the first warmup cycle. Before, points are accepted or rejected only by running the simulation. It is planned to use the `SimpleAL` active learning algorithm to also sample the first points intelligently. Delayed acceptance can be activated in the configuration by setting the parameter `delayed_acceptance`.

6.6.1 Results

Setting the parameter `make_plot` in the `active_learning` section plots the MCMC results for each warmup cycle and the main MCMC loop.

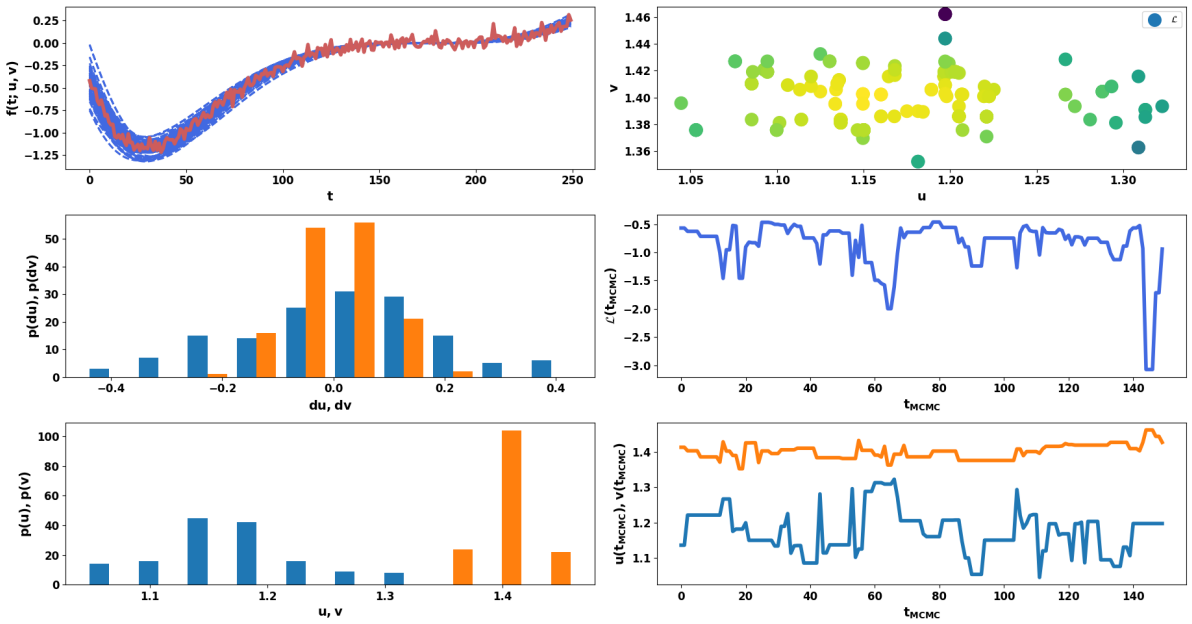


Fig. 5: Example MCMC results. Upper left: Experimental data (red) and simulation evaluations with two parameters (blue). Middle left: Distribution of step sizes for each MCMC step per parameter. Lower left: Posterior distribution of parameter values. Upper right: Likelihood for each observed sample. Middle right: Likelihood as function of MCMC steps. Lower right: Parameter values as function of MCMC steps.

6.6.2 Examples

```yaml
ntrain: 1000  # Total number of training points.
...
active_learning:
    nwarmup: 50  # Number of warmup points per cycle
    algorithm:
        class: mcmc
        reference_data: ./experimental_data.txt
        warmup_cycles: 5
        target_acceptance_rate: 0.35
        sigma_n: 0.05  # Estimated data noise (standard deviation)
        initial_points: [0.5, 1]  # Starting points for each dimension. If None,␣
↪randomly chosen in search space.
        last_percent: 0.25  # Last points used to calculate mean and variance of␣
↪posterior.
        save: ./mcmc_model.hdf5  # Save MCMC model path.
        delayed_acceptance: True  # Use delayed acceptance with surrogate specified in␣
↪`fit` configuration.
```

6.7 User Interface

The goal of the user interface (UI) is to visualize the output data and the fitted response model. A variety of options are provided for the user, ranging from the selection and restriction of the displayed data over color-scales and error-bars to specifications for the fit parameters.

6.7.1 Starting of the UI

The UI is started via the terminal with the following command:

```
profit ui
```

Then the dash app will be started and can be viewed in the web-browser of your choice at http://127.0.0.1:8050/.

6.7.2 General structure

To visualise the data and to grant a straightforward user experience, the layout is divided into the following three sections:

- axis / fit options
- graph
- filter options

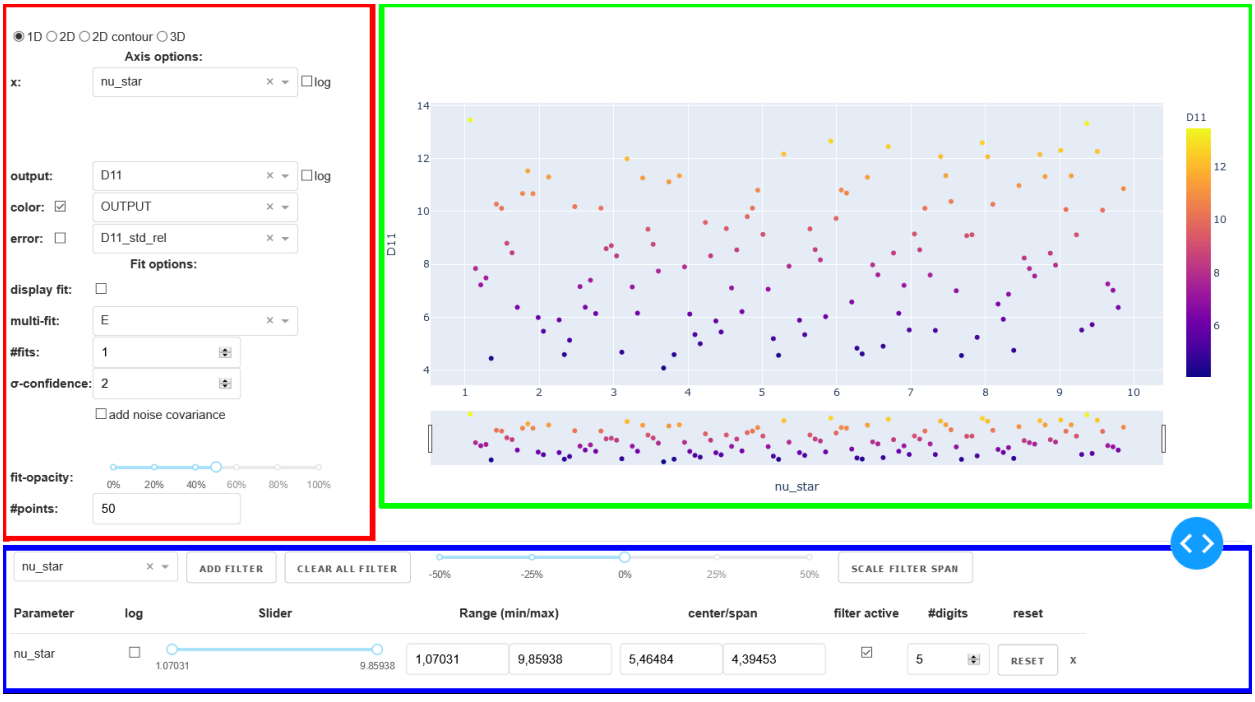


Fig. 6: Layout of the user interface with the three major sections: *axis/fit options* (red), *graph* (green) and *filter options* (blue).

Axis/Fit options

In this section of the layout the following three different types of options can be controlled:

- graph-type
- options for the axis (including color and error)
- options for the fit based on the response model

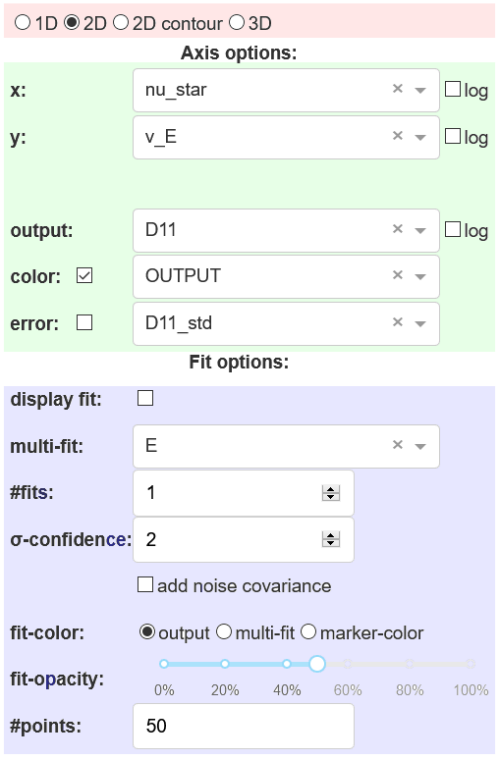


Fig. 7: Different types of options in the axis/fit options section of the UI: graph-type (red), axis options (green) and fit options (blue).

Graph-type

There are four different graph-types available:

- **1D (scatter & line)**

  input: x
  output: y

- **2D (scatter & surface)**

  input: x | y
  output: z

- **2D contour**

input: x | y
output: color

- **3D (scatter & isosurface)**

input: x | y | z
output: color

The four graph-types are shown below with sample data and a sample response model:

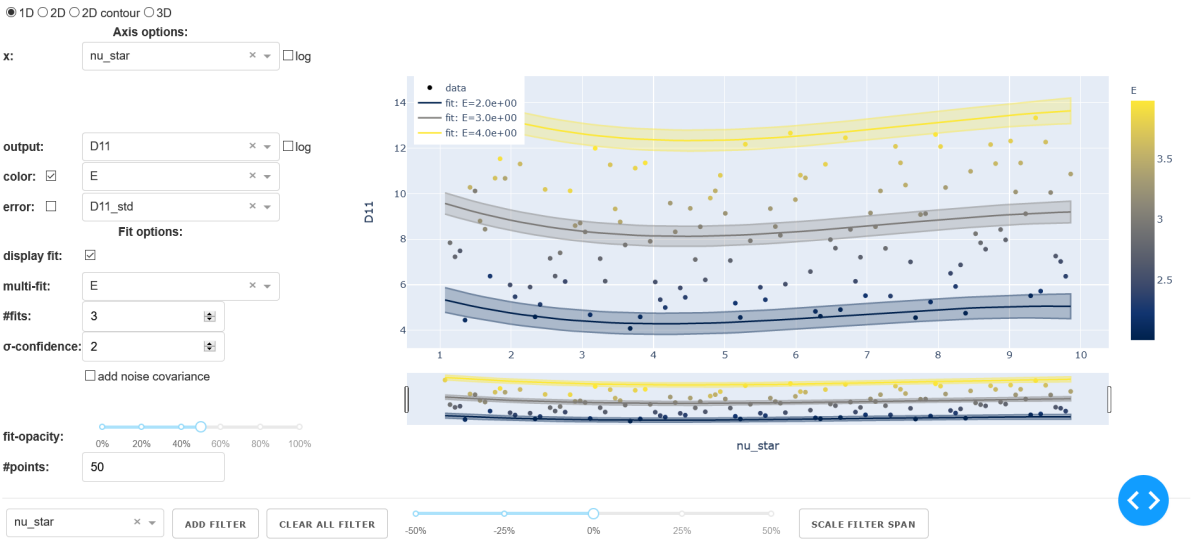


Fig. 8: Example of the UI for a 1D graph.

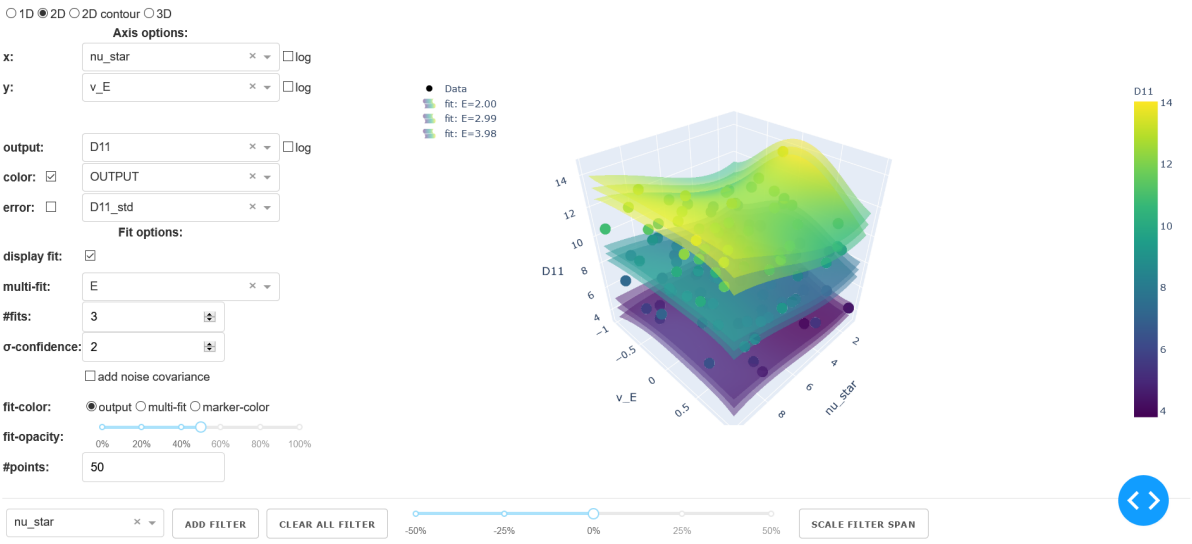


Fig. 9: Example of the UI for a 2D graph.

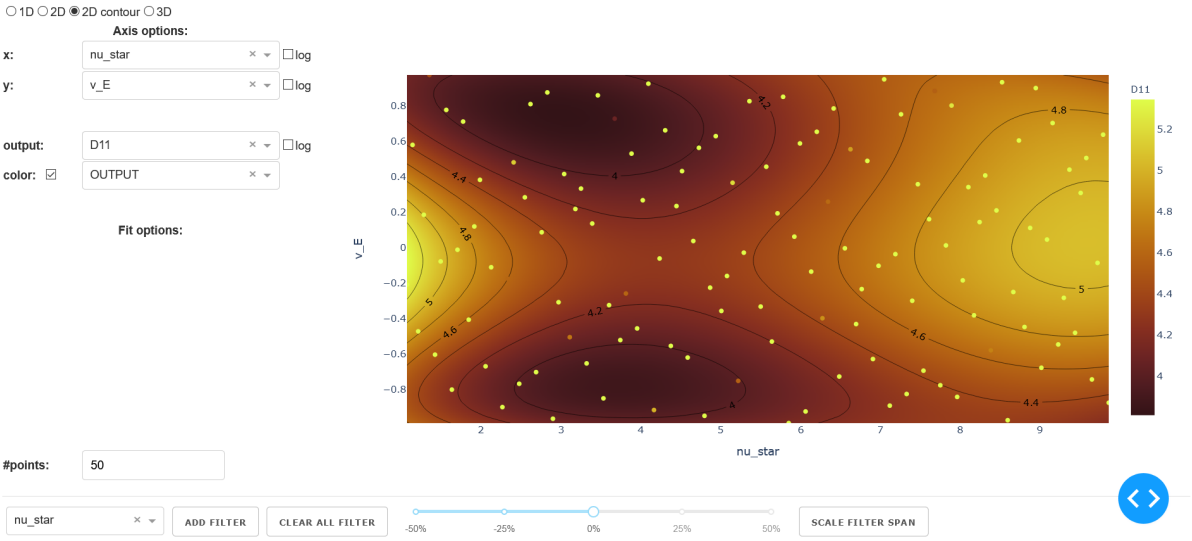


Fig. 10: Example of the UI for a 2D contour graph.

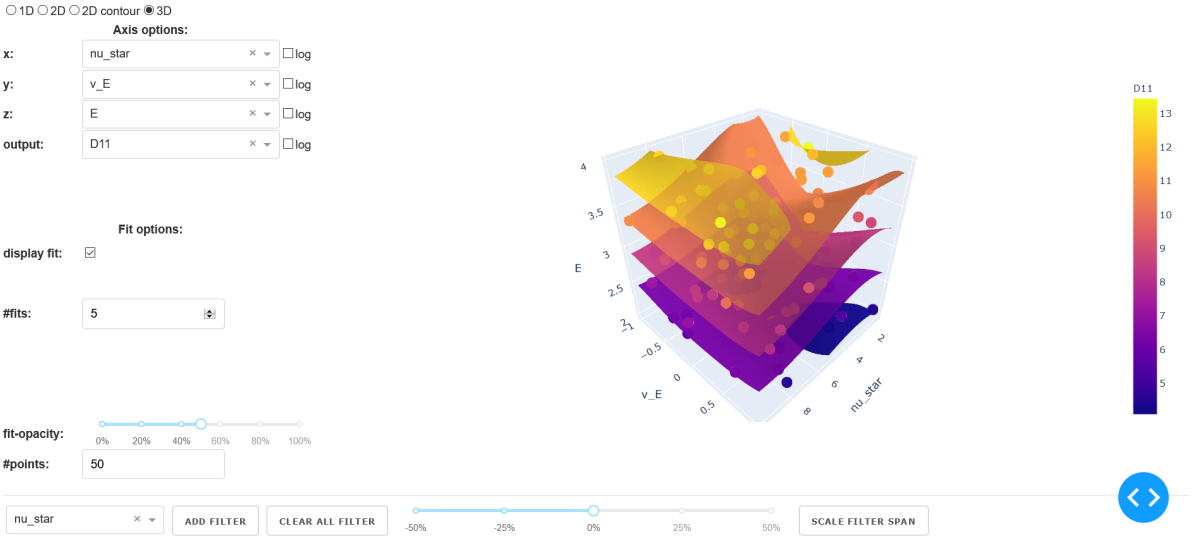


Fig. 11: Example of the UI for a 3D graph with isosurfaces.

Axis options

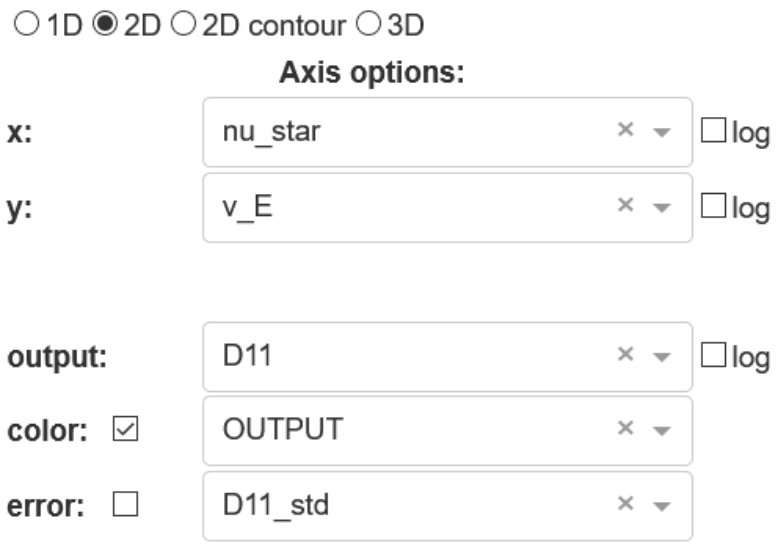


Fig. 12: Example of the axis options for a 2D graph-type.

The section **axis options** contains all the control options concerning the selection and the display of the data. Depending on the graph-type different options are available.

`x | y | z`

number of rows depending on graph-type

Type
dropdown

Options
all input-variables

`output`

Type
dropdown

Options
all output-variables

Default
first output-variable

`log`

activation of log-scale for each variable

Type
checkbox

Default
deactivated

`color`

configures the marker color of the scatter points

Type
dropdown & checkbox

Options
input-variables | output-variables | *OUTPUT*

Default
OUTPUT & activated

Available
1D | 2D | 2D contour

3D: same as `output`
The option *OUTPUT* is always synchronised with the `output`.

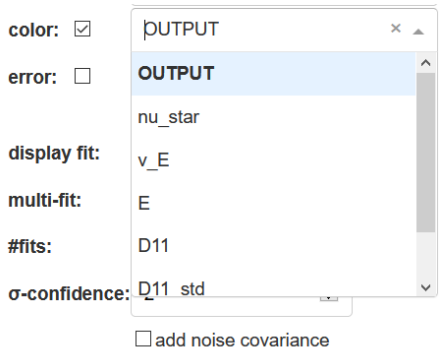


Fig. 13: Example of the dropdown options for the color consisting of *OUTPUT* and all in- and output-variables.

`error`

displays a output-variable as error

Type
dropdown & checkbox

Options
output-variables

Default
last output-variable & deactivated

Available
1D | 2D

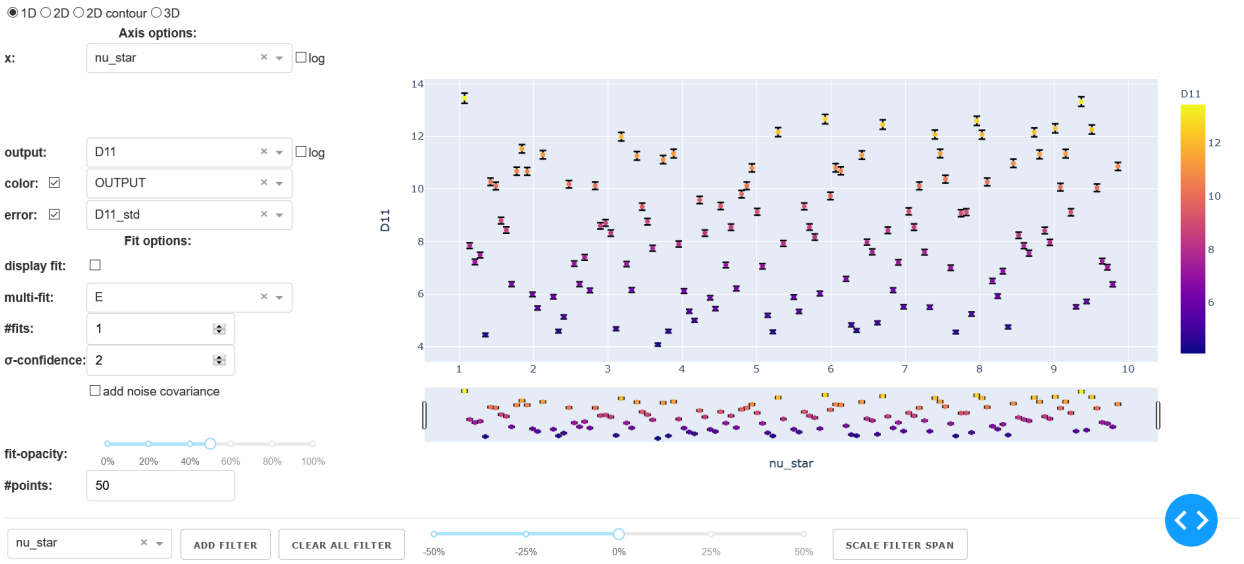


Fig. 14: Example of errorbars for a 1D graph-type.

Fit options

The section **fit options** contains the configuration for the fit based on the loaded response model. Depending on the graph-type this includes the following:

`display fit`

Type
checkbox

Default
deactivated

`multi-fit`

select the dimension (variable) along which the number of fits specified in *`#fits`* will be constructed (only relevant if *`#fits`* > 1)

Type
dropdown

Options
input-variables

Default
last input-variable

Available
1D | 2D

`#fits`

number of constructed fits along the dimension (variable) specified in *`multi-fit`*

Type
input

Default
1

Available
1D | 2D | 3D

Caution: It is possible that in 3D the top and bottom isosurface may only be partly visible. As a workaround increase *#fits* by 2.

`-confidence`

width of the confidence interval

Type
input

Default
2

Available
1D | 2D

Types of display:
1D: area around the fit line
2D: two additional surfaces under and above the fit surface

`add noise covariance`

takes uncertainty of underlying data for the response model into account

Type
checkbox

Default
deactivated

Available
1D | 2D

Caution: Not supported for all surrogate models.

`fit-color`

controls the dimension (variable) for the colorscale in 2D

Type
radiobutton & checkbox

Options
`output-variable` | *multi-fit* | `marker color`

Default
output & activated

Available
2D

1D: same as *multi-fit*
3D: same as `output-variable`

`fit-opacity`

Type
slider

Range
[0%, 100%]

Default
50%

Available
1D | 2D | 3D

1D: opacity of area between upper and lower limit
2D/3D: opacity of all surfaces

`#points`

number of predictions along the input axis for the fit based on the response model.

Type
input

Default
50

Depending on the graph-type the fit will be a line (1D), a surface (2D) or an isosurface (3D). The details for the selection of the fit parameters can be found below in the section *response model/fit*.

Graph

This section contains the actual graph. Since the graph is generated out of the plotly-library all the plotly tools are available in the upper right corner. This tools include a png-download, zoom, pan, box and lasso select, zoom in/out, autoscale, reset axis and various hover/selection tools.

Fig. 15: Graph tools provided by plotly.

The different specific properties of the graph-types are described below. In all graph-types the axis are titled according to the selected variable.

1D

The 1D graph offers a range-slider beneath the plot. With the range-slider the displayed range of data can be defined and moved around along the axis. The alternative to the range-slider is to click&drag in the graph to select a certain area. By using this method, however the viewed area can only be decreased.

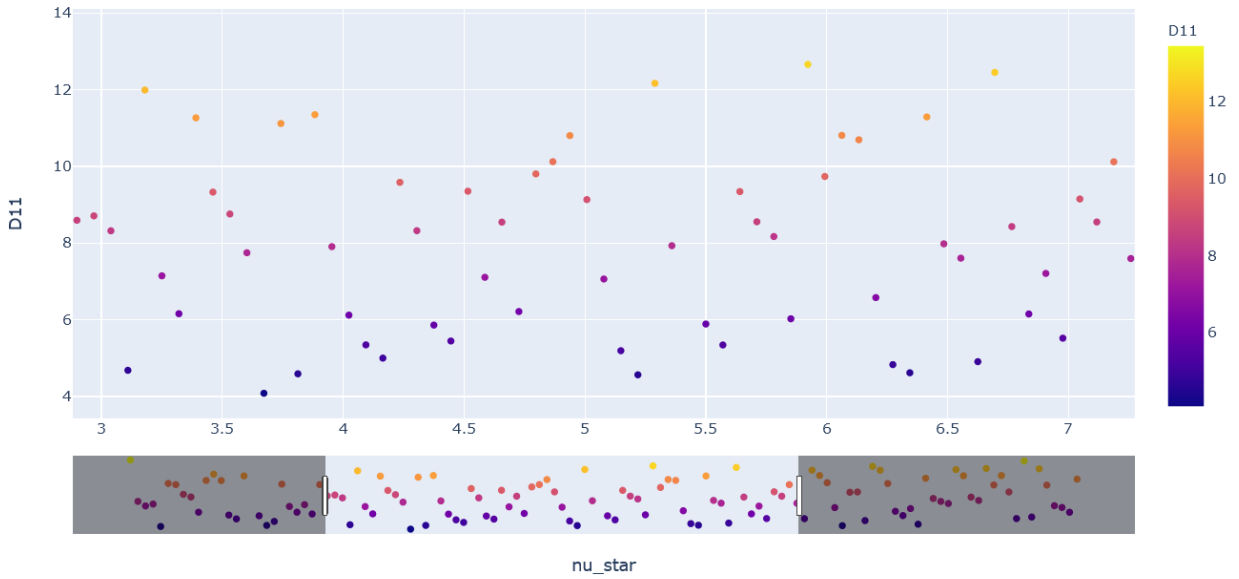


Fig. 16: Range-slider on the bottom of the 1D graph.

2D/3D

The 2D and 3D graph can be rotated and tilted by click&drag. The camera position resets as soon as an option is changed.

2D contour

In the 2D contour plot a fit surface of the 2D graph is shown from above. In addition all points in this area are displayed. Because all points (even the points with non-axis parameters far off the fit parameters) are displayed it is recommended to limit the span of the non-axis parameters via the **filter-table**.

Filter options

The main function of the **filter options** is to limit the range of the input-variables for the display in the plot and to determine the parameters for the prediction of the fit based on the response model.

The filter options are designed as a table. The controls for the entries are located at the table head and consists of the following:

1. variable-dropdown: select the input-variable to interact
2. add-filter-button: add selected dropdown-option to table
3. clear-all-button: remove all filters from table

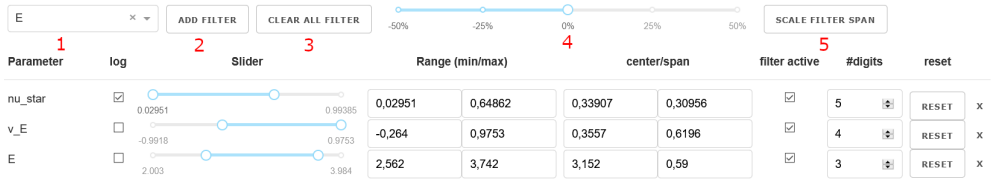


Fig. 17: The **filter-table** with the control elements (numbers according to list).

4. scale-factor-slider: select a scaling factor
5. scale-filter-span-button: apply scaling factor to al filter spans

If an variable is added to the filter table a new row appears. The table consists of the following columns:

- **Parameter:** name of the variable (dimension)
- **log:** checkbox to activate log-scale for the whole row (default: deactivated)
- **Slider:** slider to limit the range
- **Range (min/max):** input-fields for the limit of the range.
- **center/span:** input-fields for the center and the span of the range.
- **filter active:** checkbox to activate/deactivate the filter. (default: activated)
- **#digits:** input-field for the number of digits
- **reset:** button to reset the range to the default values (minimum to maximum).
- **x**-button to remove filter-row

Changes to the the values in the different columns will automatically trigger a recalculation of the other values. If the **log**-checkbox is activated the axis is mapped to a log-scale.

In addition the center values determine the value of the parameter used for the prediction of the fit as described in *response model/fit*.

6.7.3 Response model/fit

In order to predict the fit the response model needs to be evaluated at different points in the multidimensional parameter space. Therefore a multidimensional meshgrid is generated. Along the dimensions of the plot (axis-variables) the meshgrid has the same length as *#points*. The points are either linear-spaced or log-spaced based on the activation status of *log* in the **axis options**-section beside the according dimension.

In case of a single fit (*#fits* = 1) all non-axis parameters for the response model are constant. The center of the range of this dimension is used. If the range is limited via the **filter-table** the fit adjusts accordingly.

In case of a multi-fit (*#fits* > 1) along the *multi-fit* dimension the minimum and maximum of the range will be used as limits for the generation of the vector. The number of points is chosen according to *#fits*. Restrictions of the limits via the **filter-table** will be taken into account. Based on the activation-status of the **log**-checkbox in the **filter-table** a `linspace` or `logspace`-vector will be used.

For further details on the generation of the response model itself see the API documentation of the surrogate model.

6.7.4 Technical Background

The User Interface (UI) is based on `plotly/dash` for Python (see Homepage). Dash is a declarative and reactive web-based application. Dash is build on top of the following components:

- Flask
- React.js
- Plotly.js

The entire UI is running on a `Flask` web server. Flask is a WSGI (Web Server Gateway Interface) web app framework. When starting the `Dash` application a local webserver is started via `Flask`. It is possible to extend the application via Flask Plugins.

In `Dash` one is able to use the entire set of the `plotly` library. The frontend is rendered via `react.js` (react.js on github). `react.js` is a declarative, component-based JavaScript library for building user interfaces developed an maintained by Facebook.

When working with `Dash` there are a lot of standard components available for the user via the `dash_html_components` library (see dash_core_components on github) maintained by the Dash core team. In addition it is possible to write your own component library via the standard open-source React-to-Dash toolchain.

The second important library especially for structuring the UI is the `dash_html_component` library (see dash_html_component on github). It includes a set of HTML tags which are also rendered via `react.js`.

For customization it is possible to use `CSS` stylesheets for the entire interface as well as individual `CSS`-styles for each element.

The graphs itself is based on the above mentioned `plotly.js` library (see github). This graphic library maintained by Plotly.

6.8 Custom extensions

Experienced users have the possibility to add custom components which are used throughout the proFit workflow. The code should be placed in a Python file which is referenced inside the `include` section of the `profit.yaml` configuration file.

Following components are customizable:

- **Runner**

  Base class: `profit.run.Runner`
  Set in `run` config: `runner: label_of_custom_runner`

- **RunnerInterface & WorkerInterface**

  Base class: `profit.run.RunnerInterface` & `profit.run.WorkerInterface`
  Set in `run` config: `interface: label_of_custom_interface`

- **Worker**

  Base class: `profit.run.Worker`
  Set in `run` config: `worker: label_of_custom_worker`

- **Preprocessor**

Base class: `profit.run.Preprocessor`
Set in `run.runner(command)` config: `pre: label_of_custom_preprocessor`

- **Postprocessor**

  Base class: `profit.run.Postprocessor`
  Set in `run.runner(command)` config: `post: label_of_custom_postprocessor`

- **Surrogate model**

  Base class: `profit.sur.Surrogate`
  Set in `fit` config: `surrogate: label_of_custom_surrogate`

- **Active learning algorithm**

  Base class: `profit.al.ActiveLearning`
  Set in `active_learning` config: `algorithm: label_of_custom_al_algorithm`

- **Acquisition function**

  Base class: `profit.al.acquisition_functions.AcquisitionFunction`
  Set in `active_learning/algorithm` config: `acquisition_function: label_of_custom_acquisition_function`

- **FileHandler in-/output file format**

  Base class: `profit.util.file_handler.FileHandler`
  Set in `files` config: File ending of custom `FileHandler`

To create custom classes, the method `register` of the corresponding base class is used. All run components support registering using subclass arguments. For the `Worker`, `Preprocessor` and `Postprocessor` classes there exists a `wrap` method which simplifies the registering process.

6.8.1 Examples

Here, examples of registering a custom worker, custom postprocessor and a custom file format are shown.

```
# Worker

from profit.run import Worker
import numpy as np


class CustomWorker(Worker, label="custom_worker"):
"""Directly calling the wanted python function."""

    def work(self):
        self.interface.retrieve()
        u = self.interface.input["u"]
        v = self.interface.input["v"]
        self.interface.output["f"] = np.cos(10 * u) + v
        self.interface.transmit()
```

(continues on next page)

(continued from previous page)

```
@Worker.wrap("custom_worker2")
def f(u, v) -> "f":
"""Shorthand for custom_worker."""
    return np.cos(10 * u) + v
```

```
# Postprocessor

from profit.run import Postprocessor
import numpy as np


class CustomPost(Postprocessor, label="custom_post"):
"""Almost identical copy of NumpytxtPostprocessor."""

    def post(self, data):
        raw = np.loadtxt('mockup.out')
        data['f'] = raw


@Postprocessor.wrap('custom_post2')
def custom_post(data):
    """Shorthand for custom_post."""
    raw = np.loadtxt('mockup.out')
    data['f'] = raw
```

```
# FileHandler in-/output file format

from profit.util.file_handler import FileHandler


@FileHandler.register("pkl")
class PickleHandler(FileHandler):

    @classmethod
    def save(cls, filename, data, **kwargs):
        from pickle import dump
        write_method = 'wb' if not 'method' in kwargs else kwargs['method']
        dump(data, open(filename, write_method))

    @classmethod
    def load(cls, filename, as_type='raw', read_method='rb'):
        from pickle import load
        if as_type != 'raw':
            return NotImplemented
        return load(open(filename, read_method))
```

CHAPTER

SEVEN

CONTRIBUTING TO PROFIT

Contributions to proFit are always welcome.

7.1 Resources

- repository on github
  - issue tracker
  - pull requests
  - feature planning via *Projects* and *Discussions*
  - automation via *Actions*
  - managing releases
- documentation on readthedocs.io
- meeting log on HedgeDoc
- archived on zenodo.org with DOI `10.5281/zenodo.3580488`
- python package published on PyPI
- test coverage on coveralls.io

to gain access to internal documentation and regular meetings please get in touch with Christopher Albert (albert@tugraz.at)

7.2 Issues

If you encounter any bugs, problems or specific missing features, please open an issue on github. Please state the bug / problem / enhancement clearly and provide context information.

7.3 Organization and planning

Three types of project boards are used with the main repository.

Vision for strategy, use cases and user stories.

Tasks for prioritization and tracking issues and pull requests. Each version has it's own board.

Testing for gathering observations in preparation of a new release.

7.4 git

proFit uses *git* as VCS. The upstream repository is located at https://github.com/redmod-team/profit. Currently the upstream repository contains only one branch.

Contributors should fork this repository, push changes to their fork and make *pull requests* to merge them back into upstream. Before merging, the pull request should be reviewed by a maintainer and any changes can be discussed. For larger projects use *draft pull requests* to show others your current progress. They can also be tracked in the relevant *Projects*, added to release schedules and features can be discussed easily.

To try out new features which have not been merged yet, you can just add any fork to your repository with `git remote add <name> <url>` and merge it locally. Do not push this merge to your fork.

The default method to resolve conflicts is to rebase the personal fork onto `upstream` before merging.

Please also use *interactive rebase* to squash intermediate commits if you use many small commits.

7.4.1 Pre-Commit Hooks

Starting with the development for `v0.6`, proFit uses *pre-commit* to ensure consistent formatting with black and clean jupyter notebooks. Pre-commit is configured with `.pre-commit-config.yaml` and needs to be activated with `pre-commit install`. To run the hooks on all files (e.g. after adding new hooks), use `pre-commit run --all-files`. The pre-commit ci is used to enforce the hooks for all pull requests. Currently pre-commit is configured to ignore everything in `draft`.

7.5 Installing

Install proFit from your git repository using the editable install: `pip install -e .[docs,dev]`.

7.6 Documentation

The project documentation is maintained in the `doc` directory of the repository. proFit uses *Sphinx* and *rst* markup. The documentation is automatically built from the repository on every commit and hosted on readthedocs.io.

Code documentation is generated automatically using Sphinx AutoAPI. Please describe your code using docstrings, preferably following the *Google Docstring Format*. For typing please use the built in type annotations (PEP 484 / PEP 526).

To build the documentation locally, run `make html` inside the `doc` folder to create the output HTML in `_build`. This requires the additonal dependencies `docs`

7.7 Versioning

proFit follows semantic versioning (`MAJOR.MINOR.PATCH`). Each version comes with a *git tag* and a *Release* on GitHub. proFit is still in development, comes with no gurantees of backwards compatability and is therefore using versions `v0.x`. The minor version is incremented when significant features have been implemented or projects completed. Each minor version is tracked with a *Project* in GitHub and receives release notes when published. It is good practice to create a release candidate `v0.xrc` before a release. The release candidate should be tagged as *pre-release* in GitHub. It will not be shown per default on *PyPI* and *readthedocs*.

Releases are created with the GitHub interface and trigger workflows to automate the publishing and packaging. In particular:

- A python package is created an uploaded to PyPI
- The repository is archived and a new version added to zenodo
- A new version of the documentation is created on readthedocs

Before creating a version, check the metadata in `setup.cfg` (for the python package) and `.zenodo.json` (for zenodo).

proFit infers it's version from the installed metadata or the local git repository and displays it when called.

7.8 Packaging

proFit uses the new python build system, as specified by PEP 517. The build system is defined in `pyproject.toml` and uses the default `setuptools` and `wheels`. Package metadata and requirements are specified in `setup.cfg`. Building the *fortran* backend requires a `setup.py` file and `numpy` installed during the build process.

Upon publishing a new release in *GitHub*, a workflow automatically builds and uploads the package to *PyPI*. To create a release manually follow this guide.

7.9 Testing

proFit uses `pytest` for automatic testing. A pull request on *GitHub* triggers automatic testing with the supported python versions. The *GitHub* action also determines the test coverage and uploads it to coveralls.

7.10 Coding

7.10.1 Dependencies

Some calls to proFit should be completed very fast, but our many dependencies can slow down the startup time significantly. Therefore be careful where you import big packages like `GPy` or `sklearn`. Consider using import statements at the start of the function.

Investigating the tree of imported packages can be done graphically with `tuna`:

```
python -X importtime -c "import profit" 2> profit_import.log
tuna profit_import.log
```

CHAPTER

EIGHT

API REFERENCE

This page contains auto-generated API reference documentation[1].

8.1 profit

Explicit imports:
profit.fit profit.run profit.sur profit.ui

8.1.1 Subpackages

profit.al

Submodules

profit.al.active_learning

For computationally expensive simulations or experiments it is crucial to get the most information out of every training point. This is not the case in the standard procedure of randomly selecting the training points. In order to get the most out of the least number of training points, the next point is inferred by calculating an acquisition function like the minimization of local variance or expected improvement.

Module Contents

Classes

ActiveLearning	Active learning base class.

class profit.al.active_learning.**ActiveLearning**(*runner*, *variables*, *ntrain*, *nwarmup=defaults['nwarmup']*, *batch_size=defaults['batch_size']*, *convergence_criterion=defaults['convergence_criterion']*, *nsearch=defaults['nsearch']*, *make_plot=defaults['make_plot']*)

[1] Created with sphinx-autoapi

Bases: *profit.util.base_class.CustomABC*

Active learning base class.

Parameters

- **runner** (profit.run.Runner) – Runner to dynamically start runs.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **ntrain** (*int*) – Total number of training points.
- **nwarmup** (*int*) – Number of warmup (random) initialization points.
- **batch_size** (*int*) – Number of training samples learned in parallel.
- **convergence_criterion** (*float*) – AL is stopped when the loss of the acquisition function is lower than this criterion. Not implemented yet.
- **nsearch** (*int*) – Number of possible candidate points in each dimension.
- **make_plot** (*bool*) – Flat indicating if the AL progress is plotted.

krun

Current training cycle.

Type
int

labels

abstract warmup(*save_intermediate=defaults['save_intermediate']*)

Warmup cycle before the actual learning starts.

abstract learn(*resume_from=defaults['resume_from']*, *save_intermediate=defaults['save_intermediate']*)

Main loop for active learning.

update_run(*candidates*)

Run a batch of simulations with the new candidates.

Parameters
candidates (*np.array*) – Input points to run the simulation on.

update_data()

Update the variables with the runner data.

abstract save(*path*)

Save the AL model.

Parameters
path (*str*) – Path where the model is saved.

save_intermediate(*model_path=None*, *input_path=None*, *output_path=None*)

abstract plot()

Plot the progress of the AL learning.

classmethod from_config(*runner*, *variables*, *config*, *base_config*)

Instantiates an ActiveLearning object from the configuration parameters.

Parameters

- **runner** (profit.run.runner.Runner) – Runner instance.

- **variables** (profit.util.variable.VariableGroup) – Variables.
- **config** (*dict*) – Only the 'active_learning' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
AL instance.

Return type
profit.al.active_learning.ActiveLearning

profit.al.aquisition_functions

- **simple exploration**
  - variance
  - variance + penalty for near distance to previous point
  - weighted exploration/exploitation: weight * mu + (1 - weight) * sigma
- **bayesian optimization**
  - probability of improvement
  - expected improvement
- mixed exploration and bayesian optimization

Module Contents

Classes

AcquisitionFunction	Base class for acquisition functions.
SimpleExploration	Minimizes the local variance, which means the next points are generated at points of high variance.
ExplorationWithDistancePenalty	Enhanced variance minimization by adding an exponential penalty for neighboring candidates.
WeightedExploration	Combination of exploration and optimization.
ProbabilityOfImprovement	Maximizes the probability of improvement.
ExpectedImprovement	Maximising the expected improvement.
ExpectedImprovement2	Simplified batch expected improvement where the first point is calculated using normal expected improvement,
AlternatingAF	Base class for acquisition functions.

class profit.al.aquisition_functions.**AcquisitionFunction**(*Xpred*, *surrogate*, *variables*, ***parameters*)

Bases: *profit.util.base_class.CustomABC*

Base class for acquisition functions.

Parameters

- **Xpred** (*np.ndarray*) – Matrix of possible training points.
- **surrogate** (profit.sur.Surrogate) – Surrogate.

- **variables** (profit.util.variable.VariableGroup) – Variables.
- **parameters** (*dict*) – Miscellaneous parameters for the specified function. E.g. 'exploration_factor'.

labels

al_parameters

EPSILON = 1e-12

set_al_parameters(***kwargs*)

calculate_loss(**args*)

Calculate the loss of the acquisition function.

find_next_candidates(*batch_size*)

Find the next training input points which minimize the loss/maximize improvement.

_find_next_candidates(*batch_size*, **loss_args*)

normalize(*value*, *min=None*)

class profit.al.aquisition_functions.**SimpleExploration**(*Xpred*, *surrogate*, *variables*, *use_marginal_variance=se_defaults['use_marginal_variance']* ***parameters*)

Bases: *AcquisitionFunction*

Minimizes the local variance, which means the next points are generated at points of high variance.

calculate_loss()

Calculate the loss of the acquisition function.

class profit.al.aquisition_functions.**ExplorationWithDistancePenalty**(*Xpred*, *surrogate*, *variables*, *use_marginal_variance=edp_defaults['use_m* *weight=edp_defaults['weight']*)

Bases: *SimpleExploration*

Enhanced variance minimization by adding an exponential penalty for neighboring candidates.

Variables:
weight (float): Exponential penalty factor: $penalty = 1 - exp(c1 * **|X_{pred} - X_{last}|**)$.

calculate_loss()

Calculate the loss of the acquisition function.

class profit.al.aquisition_functions.**WeightedExploration**(*Xpred*, *surrogate*, *variables*, *weight=we_defaults['weight']*, *use_marginal_variance=we_defaults['use_marginal_varianc*

Bases: *AcquisitionFunction*

Combination of exploration and optimization.

Variables:
weight (float): Factor to favor maximization of the target function over exploration.

calculate_loss(*mu*)

Calculate the loss of the acquisition function.

find_next_candidates(*batch_size*)

Find the next training input points which minimize the loss/maximize improvement.

class profit.al.aquisition_functions.**ProbabilityOfImprovement**(*Xpred*, *surrogate*, *variables*, ***parameters*)

Bases: *AcquisitionFunction*

Maximizes the probability of improvement. See https://math.stackexchange.com/questions/4230985/probability-of-improvement-pi-acquisition-function-for-bayesian-optimization

calculate_loss(*mu*)

Calculate the loss of the acquisition function.

find_next_candidates(*batch_size*)

Find the next training input points which minimize the loss/maximize improvement.

class profit.al.aquisition_functions.**ExpectedImprovement**(*Xpred*, *surrogate*, *variables*, *exploration_factor=ei_defaults['exploration_factor']*, *find_min=ei_defaults['find_min']*)

Bases: *AcquisitionFunction*

Maximising the expected improvement. See https://krasserm.github.io/2018/03/21/bayesian-optimization/

To be able to execute this funciton with batches of data, some simplifications are made: The optimization part (prediction mean) is only calculated once for the first point. Thereafter, it is assumed that the data coincides with the prediction. For the next points in the batch, only the variance part is calculated as this does not need an evaluation of the function.

SIGMA_EPSILON = 1e-10

calculate_loss(*improvement*)

Calculate the loss of the acquisition function.

mu_part()

sigma_part()

find_next_candidates(*batch_size*)

Find the next training input points which minimize the loss/maximize improvement.

class profit.al.aquisition_functions.**ExpectedImprovement2**(*Xpred*, *surrogate*, *variables*, *exploration_factor=ei2_defaults['exploration_factor']*, *find_min=ei2_defaults['find_min']*)

Bases: *AcquisitionFunction*

Simplified batch expected improvement where the first point is calculated using normal expected improvement, while the others are found using the minimization of local variance acquisition function.

calculate_loss()

Calculate the loss of the acquisition function.

find_next_candidates(*batch_size*)

Find the next training input points which minimize the loss/maximize improvement.

class profit.al.aquisition_functions.**AlternatingAF**(*Xpred*, *surrogate*, *variables*, *use_marginal_variance=ae_defaults['use_marginal_variance']*, *exploration_factor=ae_defaults['exploration_factor']*, *find_min=ae_defaults['find_min']*, *alternating_freq=ae_defaults['alternating_freq']*)

Bases: *AcquisitionFunction*

Base class for acquisition functions.

Parameters

- **Xpred** (*np.ndarray*) – Matrix of possible training points.
- **surrogate** (profit.sur.Surrogate) – Surrogate.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **parameters** (*dict*) – Miscellaneous parameters for the specified function. E.g. 'exploration_factor'.

al_parameters

find_next_candidates(*batch_size*)

Find the next training input points which minimize the loss/maximize improvement.

profit.al.mcmc_al

Module Contents

Classes

McmcAL	Markov-chain Monte-Carlo active learning algorithm.

Attributes

two

profit.al.mcmc_al.**two** = **False**

class profit.al.mcmc_al.**McmcAL**(*runner*, *variables*, *reference_data*, *ntrain*, *warmup_cycles=defaults['warmup_cycles']*, *nwarmup=base_defaults['nwarmup']*, *batch_size=base_defaults['batch_size']*, *target_acceptance_rate=defaults['target_acceptance_rate']*, *convergence_criterion=base_defaults['convergence_criterion']*, *nsearch=base_defaults['nsearch']*, *sigma_n=defaults['sigma_n']*, *make_plot=base_defaults['make_plot']*, *initial_points=defaults['initial_points']*, *save=defaults['save']*, *last_percent=defaults['last_percent']*, *delayed_acceptance=defaults['delayed_acceptance']*)

Bases: *profit.al.ActiveLearning*

Markov-chain Monte-Carlo active learning algorithm.

Parameters

- **reference_data** (*np.ndarray*) – Observed experimental data points. This is not the simulated model data!
- **warmup_cycles** (*int*) – Number of warmup cycles with *nwarmup* iterations each.
- **target_acceptance_rate** (*float*) – Target rate with which probability new points are accepted.
- **sigma_n** (*float*) – Estimated standard deviation of the experimental data.
- **initial_points** (*list of float*) – Starting points for the MCMC.
- **delayed_acceptancd** (*bool*) – Whether to use delayed acceptance with a surrogate model for the likelihood.

Xpred

Matrix of the candidate points built with np.meshgrid.

Type
np.ndarray

dx

Distance between iterations in parameter space.

Type
np.ndarray

ndim

Dimension of input parameters.

Type
int

Xtrain

Array of sampled MCMC points.

Type
np.ndarray

log_likelihood

Array of the log likelihood during training.

Type
np.ndarray

accepted

Boolean array of accepted/rejected sample MCMC points.

Type
np.ndarray[bool]

labels

cost(*y*)

f(*x*)

warmup(*save_intermediate=base_defaults['save_intermediate']*)

Warmup MCMC.

learn(*resume_from=base_defaults['resume_from']*, *save_intermediate=base_defaults['save_intermediate']*)

Main loop for active learning.

do_mcmc(*rng*)

update_run(*candidates*)

Run a batch of simulations with the new candidates.

Parameters
candidates (*np.array*) – Input points to run the simulation on.

update_data()

Update the variables with the runner data.

save(*path*)

Save the AL model.

Parameters
path (*str*) – Path where the model is saved.

save_stats(*path*)

Save mean and std of X values

plot()

Plot the progress of the AL learning.

plot_mcmc(*phase*)

classmethod from_config(*runner*, *variables*, *config*, *base_config*)

Instantiates an ActiveLearning object from the configuration parameters.

Parameters

- **runner** (profit.run.runner.Runner) – Runner instance.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **config** (*dict*) – Only the 'active_learning' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
AL instance.

Return type
profit.al.active_learning.ActiveLearning

profit.al.simple_al

Module Contents

Classes

SimpleAL	Simple active learning algorithm based on a surrogate model and an acquisition function to find next candidates.

class profit.al.simple_al.**SimpleAL**(*runner*, *variables*, *surrogate*, *ntrain*, *nwarmup=base_defaults['nwarmup']*, *batch_size=base_defaults['batch_size']*, *acquisition_function=defaults['acquisition_function']*, *convergence_criterion=base_defaults['convergence_criterion']*, *nsearch=base_defaults['nsearch']*, *make_plot=base_defaults['make_plot']*, *searchtype=defaults['searchtype']*)

Bases: *profit.al.ActiveLearning*

Simple active learning algorithm based on a surrogate model and an acquisition function to find next candidates.

Parameters

- **surrogate** (profit.sur.Surrogate) – Surrogate used for fitting.
- **acquisition_function** (*str/profit.al.acquisition_functions.AcquisitionFunction*) – Acquisition function used for selecting the next candidates.

search_space

np.linspace for each AL input variable.

Type
dict[str, np.array]

Xpred

Matrix of the candidate points built with np.meshgrid.

Type
np.array

labels

warmup(*save_intermediate=base_defaults['save_intermediate']*)

To get data for active learning, sample initial points randomly.

learn(*resume_from=base_defaults['resume_from']*, *save_intermediate=base_defaults['save_intermediate']*)

Main loop for active learning.

find_next_candidates()

Find the next candidates using the acquisition function's method find_next_candidates.

Returns
Next training points.

Return type
np.array

update_run(*candidates*)

Run a batch of simulations with the new candidates.

Parameters
candidates (*np.array*) – Input points to run the simulation on.

save(*path*)

Save the AL model.

Parameters
path (*str*) – Path where the model is saved.

plot()

Plot the progress of the AL learning.

classmethod from_config(*runner*, *variables*, *config*, *base_config*)

Instantiates an ActiveLearning object from the configuration parameters.

Parameters

- **runner** (profit.run.runner.Runner) – Runner instance.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **config** (*dict*) – Only the 'active_learning' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
AL instance.

Return type
profit.al.active_learning.ActiveLearning

Package Contents

Classes

ActiveLearning	Active learning base class.
SimpleAL	Simple active learning algorithm based on a surrogate model and an acquisition function to find next candidates.
McmcAL	Markov-chain Monte-Carlo active learning algorithm.

class profit.al.**ActiveLearning**(*runner*, *variables*, *ntrain*, *nwarmup=defaults['nwarmup']*, *batch_size=defaults['batch_size']*, *convergence_criterion=defaults['convergence_criterion']*, *nsearch=defaults['nsearch']*, *make_plot=defaults['make_plot']*)

Bases: *profit.util.base_class.CustomABC*

Active learning base class.

Parameters

- **runner** (profit.run.Runner) – Runner to dynamically start runs.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **ntrain** (*int*) – Total number of training points.
- **nwarmup** (*int*) – Number of warmup (random) initialization points.
- **batch_size** (*int*) – Number of training samples learned in parallel.
- **convergence_criterion** (*float*) – AL is stopped when the loss of the acquisition function is lower than this criterion. Not implemented yet.
- **nsearch** (*int*) – Number of possible candidate points in each dimension.
- **make_plot** (*bool*) – Flat indicating if the AL progress is plotted.

krun

Current training cycle.

Type
int

labels

abstract warmup(*save_intermediate=defaults['save_intermediate']*)

Warmup cycle before the actual learning starts.

abstract learn(*resume_from=defaults['resume_from']*, *save_intermediate=defaults['save_intermediate']*)

Main loop for active learning.

update_run(*candidates*)

Run a batch of simulations with the new candidates.

Parameters
candidates (*np.array*) – Input points to run the simulation on.

update_data()

Update the variables with the runner data.

abstract save(*path*)

Save the AL model.

Parameters
path (*str*) – Path where the model is saved.

save_intermediate(*model_path=None*, *input_path=None*, *output_path=None*)

abstract plot()

Plot the progress of the AL learning.

classmethod from_config(*runner*, *variables*, *config*, *base_config*)

Instantiates an ActiveLearning object from the configuration parameters.

Parameters

- **runner** (profit.run.runner.Runner) – Runner instance.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **config** (*dict*) – Only the 'active_learning' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
AL instance.

Return type
profit.al.active_learning.ActiveLearning

class profit.al.**SimpleAL**(*runner*, *variables*, *surrogate*, *ntrain*, *nwarmup=base_defaults['nwarmup']*, *batch_size=base_defaults['batch_size']*, *acquisition_function=defaults['acquisition_function']*, *convergence_criterion=base_defaults['convergence_criterion']*, *nsearch=base_defaults['nsearch']*, *make_plot=base_defaults['make_plot']*, *searchtype=defaults['searchtype']*)

Bases: *profit.al.ActiveLearning*

Simple active learning algorithm based on a surrogate model and an acquisition function to find next candidates.

Parameters

- **surrogate** (profit.sur.Surrogate) – Surrogate used for fitting.
- **acquisition_function** (*str/profit.al.acquisition_functions.AcquisitionFunction*) – Acquisition function used for selecting the next candidates.

search_space

np.linspace for each AL input variable.

Type
dict[str, np.array]

Xpred

Matrix of the candidate points built with np.meshgrid.

Type
np.array

labels

warmup(*save_intermediate=base_defaults['save_intermediate']*)

To get data for active learning, sample initial points randomly.

learn(*resume_from=base_defaults['resume_from']*, *save_intermediate=base_defaults['save_intermediate']*)

Main loop for active learning.

find_next_candidates()

Find the next candidates using the acquisition function's method find_next_candidates.

Returns
Next training points.

Return type
np.array

update_run(*candidates*)

Run a batch of simulations with the new candidates.

Parameters
candidates (*np.array*) – Input points to run the simulation on.

save(*path*)

Save the AL model.

Parameters
path (*str*) – Path where the model is saved.

plot()

Plot the progress of the AL learning.

classmethod from_config(*runner*, *variables*, *config*, *base_config*)

Instantiates an ActiveLearning object from the configuration parameters.

Parameters

- **runner** (profit.run.runner.Runner) – Runner instance.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **config** (*dict*) – Only the 'active_learning' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
AL instance.

Return type
profit.al.active_learning.ActiveLearning

class profit.al.**McmcAL**(*runner*, *variables*, *reference_data*, *ntrain*, *warmup_cycles=defaults['warmup_cycles']*, *nwarmup=base_defaults['nwarmup']*, *batch_size=base_defaults['batch_size']*, *target_acceptance_rate=defaults['target_acceptance_rate']*, *convergence_criterion=base_defaults['convergence_criterion']*, *nsearch=base_defaults['nsearch']*, *sigma_n=defaults['sigma_n']*, *make_plot=base_defaults['make_plot']*, *initial_points=defaults['initial_points']*, *save=defaults['save']*, *last_percent=defaults['last_percent']*, *delayed_acceptance=defaults['delayed_acceptance']*)

Bases: *profit.al.ActiveLearning*

Markov-chain Monte-Carlo active learning algorithm.

Parameters

- **reference_data** (*np.ndarray*) – Observed experimental data points. This is not the simulated model data!
- **warmup_cycles** (*int*) – Number of warmup cycles with *nwarmup* iterations each.
- **target_acceptance_rate** (*float*) – Target rate with which probability new points are accepted.
- **sigma_n** (*float*) – Estimated standard deviation of the experimental data.
- **initial_points** (*list of float*) – Starting points for the MCMC.
- **delayed_acceptancd** (*bool*) – Whether to use delayed acceptance with a surrogate model for the likelihood.

Xpred

Matrix of the candidate points built with np.meshgrid.

Type
np.ndarray

dx

Distance between iterations in parameter space.

Type
np.ndarray

ndim

Dimension of input parameters.

Type
int

Xtrain

Array of sampled MCMC points.

Type
np.ndarray

log_likelihood

Array of the log likelihood during training.

Type
np.ndarray

accepted

Boolean array of accepted/rejected sample MCMC points.

Type
np.ndarray[bool]

labels

cost(*y*)

f(*x*)

warmup(*save_intermediate=base_defaults['save_intermediate']*)

Warmup MCMC.

learn(*resume_from=base_defaults['resume_from']*, *save_intermediate=base_defaults['save_intermediate']*)

Main loop for active learning.

do_mcmc(*rng*)

update_run(*candidates*)

Run a batch of simulations with the new candidates.

Parameters
candidates (*np.array*) – Input points to run the simulation on.

update_data()

Update the variables with the runner data.

save(*path*)

Save the AL model.

Parameters
path (*str*) – Path where the model is saved.

save_stats(*path*)

Save mean and std of X values

plot()

Plot the progress of the AL learning.

plot_mcmc(*phase*)

classmethod from_config(*runner*, *variables*, *config*, *base_config*)

Instantiates an ActiveLearning object from the configuration parameters.

Parameters

- **runner** (profit.run.runner.Runner) – Runner instance.
- **variables** (profit.util.variable.VariableGroup) – Variables.
- **config** (*dict*) – Only the 'active_learning' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
AL instance.

Return type
profit.al.active_learning.ActiveLearning

profit.run

Submodules

profit.run.command

Command Worker

The default Worker to run an executable simulation. The Preprocessor allows for a customized preparation of the environment for the simulation. The output of the simulation is retrieved by a Postprocessor.

- CommandWorker: run an executable simulation
- Preprocessor: new Component to prepare the environment for the simulation * TemplatePreprocessor: fill a directory according to a template directory
- Postprocessor: new Component to retrieve the simulation output * JSONPostprocessor: read a simple JSON * NumpytxtPostprocessor: read a CSV/TSV (using numpy) * HDF5Postprocessor: read a simple HDF5

Module Contents

Classes

CommandWorker	Helper class that provides a standard way to create an ABC using
Preprocessor	Helper class that provides a standard way to create an ABC using
TemplatePreprocessor	Preprocessor which substitutes the variables with a given template
Postprocessor	Helper class that provides a standard way to create an ABC using

Functions

JSONPostprocessor(self, data)	Postprocessor to read output from a JSON file
NumpytxtPostprocessor(self, data)	Postprocessor to read output from a tabular text file (e.g. csv, tsv) with numpy `genfromtxt`
HDF5Postprocessor(self, data)	Postprocessor to read output from a HDF5 file

class profit.run.command.**CommandWorker**(*run_id: int*, ***, *pre='template'*, *post='numpytxt'*, *command='./simulation'*, *stdout='stdout'*, *stderr=None*, ***kwargs*)

Bases: *profit.run.worker.Worker*

Helper class that provides a standard way to create an ABC using inheritance.

work()

class profit.run.command.**Preprocessor**(*run_dir: str*, ***, *clean=True*, *logger_parent=None*)

Bases: profit.run.worker.Component

Helper class that provides a standard way to create an ABC using inheritance.

abstract prepare(*data: Mapping*)

post()

classmethod wrap(*label*, *config={}*)

class profit.run.command.**TemplatePreprocessor**(*run_dir: str*, ***, *clean=True*, *path='template'*, *param_files=None*, *logger_parent=None*)

Bases: *Preprocessor*

Preprocessor which substitutes the variables with a given template

- copies the given template directory to the target run directory
- searches all files for variables templates of the form {name} and replaces them with their values
- for file formats which use curly braces (e.g. json) the template identifier is {{name}}
- substitution can be restricted to certain files by specifying *param_files*, *None* means no restriction
- relative symbolic links are converted to absolute symbolic links on copying
- linked files are ignored with *param_files = None*, but if specified explicitly the link target is copied to the run directory and then substituted

property template_path

prepare(*data: Mapping*)

fill_run_dir_single(*params*, *template_dir*, *run_dir_single*, *param_files=None*, *overwrite=False*, *ignore_path_exists=False*)

classmethod copy_template(*template_dir*, *out_dir*, *dont_copy=None*)

static convert_relative_symlinks(*template_dir*, *out_dir*)

When copying the template directory to the single run directories, relative paths in symbolic links are converted to absolute paths.

fill_template(*out_dir*, *params*, *param_files=None*)

Parameters
param_files (*list*) – a list of filenames which are to be substituted or None for all

classmethod fill_template_file(*template_filepath*, *output_filepath*, *params*, *copy_link=True*)

Fill template in *template_filepath* by *params* and output into *output_filepath*. If *copy_link* is set (default), do not write into symbolic links but copy them instead.

static replace_template(*content*, *params*)

Returns filled template by putting values of *params* in *content*.

Escape '{*}' for e.g. json templates by replacing it with '{{*}}'. # Variables then have to be declared as '{{*}}' which is replaced by a single '{*}'.

class profit.run.command.**Postprocessor**(*, *logger_parent: logging.Logger = None*)

Bases: profit.run.worker.Component

Helper class that provides a standard way to create an ABC using inheritance.

abstract retrieve(*data: MutableMapping*)

classmethod wrap(*label*, *config={}*)

profit.run.command.**JSONPostprocessor**(*self*, *data*)

Postprocessor to read output from a JSON file

- variables are assumed to be stored with the correct key and able to be converted immediately
- not extensively tested

profit.run.command.**NumpytxtPostprocessor**(*self*, *data*)

Postprocessor to read output from a tabular text file (e.g. csv, tsv) with numpy `genfromtxt`

- the data is assumed to be row oriented
- vector variables are spread across the row and have to be in the right order, only the name of the variable should be specified once in `names`
- `names` which are not specified as output variables are ignored
- additional options are passed directly to ``numpy.genfromtxt()`

profit.run.command.**HDF5Postprocessor**(*self*, *data*)

Postprocessor to read output from a HDF5 file

- variables are assumed to be stored with the correct key and able to be converted immediately
- not extensively tested

profit.run.interface

Runner-Worker Interface

The Interface is responsible for the data transfer between the Runner and all Workers. Each Interface consists of two components: a Runner-Interface and a Worker-Interface

Module Contents

Classes

RunnerInterface	Helper class that provides a standard way to create an ABC using
WorkerInterface	The Worker-Interface

class profit.run.interface.**RunnerInterface**(*size*, *input_config*, *output_config*, *, *logger_parent: logging.Logger = None*)

Bases: *profit.util.component.Component*

Helper class that provides a standard way to create an ABC using inheritance.

property config

property size

internal_vars = [('DONE',), ('TIME',)]

resize(*size*)

poll()

clean()

class profit.run.interface.**WorkerInterface**(*run_id: int*, *, *logger_parent: logging.Logger = None*)

Bases: *profit.util.component.Component*

The Worker-Interface

The Worker-side of the Interface performs two tasks: retrieving input data and transmitting output data.

Only the Worker interacts directly with the Interface, following the scheme: ```

self.interface.retrieve() -> self.interface.input timestamp = time.time() self.interface.output = simulate() self.interface.time = int(time.time() - timestamp) self.interface.transmit()

```

property config

abstract retrieve()

retrieve the input

1) connect to the Runner-Interface
2) retrieve the input data and store it in *.input*

abstract transmit()

transmit the output

1) transmit the output and time data (*.output* and *.time*)
2) signal the Worker has finished
3) close the connection to the Runner-Interface

clean()

## profit.run.local

Local Runner & Memory-map Interface

- LocalRunner: start Workers locally via the shell (subprocess.Popen)
- ForkRunner: start Workers locally with forking (multiprocessing.Process)
- MemmapInterface: share date using a memory-mapped, structured array (using numpy)
```

Module Contents

Classes

LocalRunner	start Workers locally via the shell
ForkRunner	start Workers locally using forking (multiprocessing.Process)
MemmapRunnerInterface	Runner-Worker Interface using a memory mapped numpy array
MemmapWorkerInterface	Runner-Worker Interface using a memory mapped numpy array

class profit.run.local.**LocalRunner**(*command='profit-worker'*, *parallel='all'*, ***kwargs*)

Bases: *profit.run.runner.Runner*

start Workers locally via the shell

property config

__repr__()

Return repr(self).

spawn(*params=None*, *wait=False*)

spawn a single run

Parameters

- **params** – a mapping which defines input parameters to be set
- **wait** – whether to wait for the run to complete

poll(*run_id*)

check the status of the run directly

cancel(*run_id*)

class profit.run.local.**ForkRunner**(*parallel='all'*, ***kwargs*)

Bases: *profit.run.runner.Runner*

start Workers locally using forking (multiprocessing.Process)

spawn(*params=None*, *wait=False*)

spawn a single run

Parameters

- **params** – a mapping which defines input parameters to be set
- **wait** – whether to wait for the run to complete

poll(*run_id*)

check the status of the run directly

cancel(*run_id*)

class profit.run.local.**MemmapRunnerInterface**(*size*, *input_config*, *output_config*, *, *path: str = 'interface.npy'*, *logger_parent: logging.Logger = None*)

Bases: *profit.run.interface.RunnerInterface*

Runner-Worker Interface using a memory mapped numpy array

- expected to be very fast with the *local* Runner as each Worker can access the array directly (unverified)
- expected to be inefficient if used on a cluster with a shared filesystem (unverified)
- reliable
- known issue: resizing the array (to add more runs) is dangerous, needs a workaround (e.g. several arrays in the same file)

property config

resize(*size*)

Resizing the Interface

Attention: this is dangerous and may lead to unexpected errors! The problem is that the memory mapped file is overwritten. Any Workers which have this file mapped will run into severe problems. Possible future workarounds: multiple files or multiple headers in one file.

clean()

class profit.run.local.**MemmapWorkerInterface**(*run_id: int*, *, *path='interface.npy'*, *logger_parent: logging.Logger = None*)

Bases: *profit.run.interface.WorkerInterface*

Runner-Worker Interface using a memory mapped numpy array

counterpart to *MemmapRunnerInterface*

property config

property time

retrieve()

retrieve the input

1) connect to the Runner-Interface
2) retrieve the input data and store it in *.input*

transmit()

transmit the output

1) transmit the output and time data (*.output* and *.time*)
2) signal the Worker has finished
3) close the connection to the Runner-Interface

clean()

profit.run.runner

Runner & Runner Interface

Module Contents

Classes

Runner	Helper class that provides a standard way to create an ABC using

class profit.run.runner.**Runner**(*, *interface:* profit.run.interface.RunnerInterface = *'memmap'*, *worker: Mapping = 'command'*, *work_dir='.'*, *debug=False*, *parallel=0*, *sleep=0.1*, *logfile='runner.log'*, *logger=None*)

Bases: *profit.util.component.Component*

Helper class that provides a standard way to create an ABC using inheritance.

property config

property input_data

property output_data

property flat_output_data

classmethod from_config(*run_config*, *base_config*)

Constructor from run config & base config dict

__repr__()

Return repr(self).

change_work_dir()

fill(*params_array*, *offset=0*)

fill Interface input with parameters

fill_output(*named_output*, *offset=0*)

fill Interface output with values

abstract spawn(*params=None*, *wait=False*)

spawn a single run

Parameters

- **params** – a mapping which defines input parameters to be set
- **wait** – whether to wait for the run to complete

spawn_array(*params_array*, *wait=False*, *progress=False*)

spawn an array of runs

maximum 'parallel' at the same time blocking until all are submitted

abstract poll(*run_id*)

check the status of the run directly

poll_all()

check_runs()

check the status of runs via the interface

abstract cancel(*run_id*)

cancel_all()

wait(*run_id*)

wait_all(*progress=False*)

clean()

profit.run.slurm

Scheduling runs on a HPC cluster with SLURM

- targeted towards aCluster@tugraz.at
- each run is submitted as a job using a slurm batch script
- run arrays are submitted as a slurm job array
- by default completed runs are recognised by the interface, but the scheduler is polled as a fallback (less often)

Module Contents

Classes

SlurmRunner	Runner which submits each run as a job to the SLURM scheduler on a cluster

class profit.run.slurm.**SlurmRunner**(*, *interface='zeromq'*, *cpus=1*, *openmp=False*, *custom=False*, *path='slurm.bash'*, *options=None*, *command='srun profit-worker'*, ***kwargs*)

Bases: *profit.run.runner.Runner*

Runner which submits each run as a job to the SLURM scheduler on a cluster

- generates a slurm batch script with the given configuration
- can also be used with a custom script
- supports OpenMP
- tries to minimize overhead by using job arrays
- polls the scheduler only at longer intervals

property config

__repr__()

Return repr(self).

spawn(*params=None*, *wait=False*)

spawn a single run

Parameters

- **params** – a mapping which defines input parameters to be set
- **wait** – whether to wait for the run to complete

spawn_array(*params_array*, *wait=False*, *progress=False*)

spawn an array of runs

maximum 'parallel' at the same time blocking until all are submitted

poll(*run_id*)

check the status of the run directly

poll_all()

cancel(*run_id*)

cancel_all()

clean()

remove generated scripts and any slurm-stdout-files which match `slurm-*.out`

generate_script()

profit.run.worker

proFit worker class & components

Module Contents

Classes

Worker	Helper class that provides a standard way to create an ABC using

Functions

main()	entry point to run a worker

class profit.run.worker.**Worker**(*run_id: int*, ***, *interface:* profit.run.interface.WorkerInterface = *'memmap'*, *debug=False*, *log_path='log'*, *logger=None*)

Bases: *profit.util.component.Component*

Helper class that provides a standard way to create an ABC using inheritance.

abstract work()

clean()

classmethod from_config(*config*, *interface*, *run_id*)

classmethod from_env(*env=None*)

classmethod wrap(*label*, *outputs=None*, *inputs=None*)

``` @Worker.wrap('label', ['f', 'g'], ['x', 'y']) def func(x, y):

...

@Worker.wrap('label', ['f', 'g']) def func(x, y):

...

@Worker.wrap('label') def func(x, y) -> ['f', 'g']:

...

@Worker.wrap('name', 'f', 'x') def func(x):

...

@Worker.wrap('name') def func(x) -> 'f':

...

@Worker.wrap('name') def f(x):

...

```

profit.run.worker.**main**()

entry point to run a worker

the run id and the path to the proFit configuration is provided via environment variables

profit.run.zeromq

zeromq Interface

Ideas & Help from the 0MQ Guide (zguide.zeromq.org, examples are licensed with MIT)

Module Contents

Classes

ZeroMQRunnerInterface	Runner-Worker Interface using the lightweight message queue ZeroMQ
ZeroMQWorkerInterface	Runner-Worker Interface using the lightweight message queue ZeroMQ

class profit.run.zeromq.**ZeroMQRunnerInterface**(*size*, *input_config*, *output_config*, *, *transport='tcp'*, *address=None*, *port=9000*, *connection=None*, *bind=None*, *timeout=4*, *retries=3*, *retry_sleep=1*, *logger_parent: logging.Logger = None*)

Bases: *profit.run.interface.RunnerInterface*

Runner-Worker Interface using the lightweight message queue ZeroMQ

- can use different transport systems, most commonly tcp
- can be used efficiently on a cluster (tested)
- expected to be inefficient for a large number of small, locally run simulations where communication overhead is a concern (unverified, could be mitigated by using a different transport system)
- known issue: some workers were unable to establish a connection with three tries, reason unknown

Parameters

- **transport** – ZeroMQ transport protocol
- **address** – override ip address or hostname of the Runner Interface (default: localhost, automatic with Slurm)
- **port** – port of the Runner Interface
- **connection** – override for the ZeroMQ connection spec (Worker side)
- **bind** – override for the ZeroMQ bind spec (Runner side)
- **timeout** – connection timeout when waiting for an answer in seconds (Worker)
- **retries** – number of tries to establish a connection (Worker)
- **retry_sleep** – sleep time in seconds between each retry (Worker)

socket

ZeroMQ backend

Type
zmq.Socket

logger

Logger

Type
logging.Logger

property bind

property config

poll()

handle_msg(*address: bytes*, *msg: list*)

clean()

__del__()

class profit.run.zeromq.**ZeroMQWorkerInterface**(*run_id: int*, *, *transport='tcp'*, *address=None*, *port=9000*, *connection=None*, *bind=None*, *timeout=4*, *retries=3*, *retry_sleep=1*, *logger_parent: logging.Logger = None*)

Bases: *profit.run.interface.WorkerInterface*

Runner-Worker Interface using the lightweight message queue ZeroMQ

counterpart to *ZeroMQRunnerInterface*

property connection

property config

retrieve()

retrieve the input

1) connect to the Runner-Interface
2) retrieve the input data and store it in *.input*

transmit()

transmit the output

1) transmit the output and time data (*.output* and *.time*)
2) signal the Worker has finished
3) close the connection to the Runner-Interface

clean()

connect()

disconnect()

__del__()

request(*request*)

0MQ - Lazy Pirate Pattern

Package Contents

Classes

RunnerInterface	Helper class that provides a standard way to create an ABC using
WorkerInterface	The Worker-Interface
Runner	Helper class that provides a standard way to create an ABC using
Worker	Helper class that provides a standard way to create an ABC using
Preprocessor	Helper class that provides a standard way to create an ABC using
Postprocessor	Helper class that provides a standard way to create an ABC using

class profit.run.**RunnerInterface**(*size*, *input_config*, *output_config*, *, *logger_parent: logging.Logger = None*)

Bases: *profit.util.component.Component*

Helper class that provides a standard way to create an ABC using inheritance.

property config

property size

internal_vars = [('DONE',), ('TIME',)]

resize(*size*)

poll()

clean()

class profit.run.**WorkerInterface**(*run_id: int*, *, *logger_parent: logging.Logger = None*)

Bases: *profit.util.component.Component*

The Worker-Interface

The Worker-side of the Interface performs two tasks: retrieving input data and transmitting output data.

Only the Worker interacts directly with the Interface, following the scheme: ```

self.interface.retrieve() -> self.interface.input timestamp = time.time() self.interface.output = simulate() self.interface.time = int(time.time() - timestamp) self.interface.transmit()

```

**property config**

**abstract retrieve**()

retrieve the input

1) connect to the Runner-Interface
2) retrieve the input data and store it in *.input*

**abstract transmit**()

transmit the output

1) transmit the output and time data (*.output* and *.time*)
2) signal the Worker has finished
3) close the connection to the Runner-Interface

**clean**()

class profit.run.**Runner**(*, *interface:* profit.run.interface.RunnerInterface *= 'memmap'*, *worker: Mapping = 'command'*, *work_dir='.'*, *debug=False*, *parallel=0*, *sleep=0.1*, *logfile='runner.log'*, *logger=None*)

Bases: *profit.util.component.Component*

Helper class that provides a standard way to create an ABC using inheritance.

**property config**

**property input_data**
```

property output_data

property flat_output_data

classmethod from_config(*run_config*, *base_config*)

Constructor from run config & base config dict

__repr__()

Return repr(self).

change_work_dir()

fill(*params_array*, *offset=0*)

fill Interface input with parameters

fill_output(*named_output*, *offset=0*)

fill Interface output with values

abstract spawn(*params=None*, *wait=False*)

spawn a single run

Parameters

- **params** – a mapping which defines input parameters to be set
- **wait** – whether to wait for the run to complete

spawn_array(*params_array*, *wait=False*, *progress=False*)

spawn an array of runs

maximum 'parallel' at the same time blocking until all are submitted

abstract poll(*run_id*)

check the status of the run directly

poll_all()

check_runs()

check the status of runs via the interface

abstract cancel(*run_id*)

cancel_all()

wait(*run_id*)

wait_all(*progress=False*)

clean()

class profit.run.Worker(*run_id: int*, ***, *interface:* profit.run.interface.WorkerInterface = *'memmap'*, *debug=False*, *log_path='log'*, *logger=None*)

Bases: *profit.util.component.Component*

Helper class that provides a standard way to create an ABC using inheritance.

abstract work()

clean()

classmethod from_config(*config*, *interface*, *run_id*)

classmethod from_env(*env=None*)

classmethod wrap(*label*, *outputs=None*, *inputs=None*)

```
@Worker.wrap('label', ['f', 'g'], ['x', 'y']) def func(x, y):
    ...
@Worker.wrap('label', ['f', 'g']) def func(x, y):
    ...
@Worker.wrap('label') def func(x, y) -> ['f', 'g']:
    ...
@Worker.wrap('name', 'f', 'x') def func(x):
    ...
@Worker.wrap('name') def func(x) -> 'f':
    ...
@Worker.wrap('name') def f(x):
    ...
```

class profit.run.**Preprocessor**(*run_dir: str*, ***, *clean=True*, *logger_parent=None*)

Bases: profit.run.worker.Component

Helper class that provides a standard way to create an ABC using inheritance.

abstract prepare(*data: Mapping*)

post()

classmethod wrap(*label*, *config={}*)

class profit.run.**Postprocessor**(***, *logger_parent: logging.Logger = None*)

Bases: profit.run.worker.Component

Helper class that provides a standard way to create an ABC using inheritance.

abstract retrieve(*data: MutableMapping*)

classmethod wrap(*label*, *config={}*)

profit.sur

Subpackages

profit.sur.ann

Submodules

profit.sur.ann.artificial_neural_network

Module Contents

Classes

ANN	Base class for all surrogate models.
ANNSurrogate	Base class for all surrogate models.
Autoencoder	Nonlinear autoencoder with activation functions

class profit.sur.ann.artificial_neural_network.**ANN**(*config*)

Bases: *profit.sur.sur.Surrogate*, abc.ABC

Base class for all surrogate models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not.

Type
bool

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

input_encoders

Encoding used on input data.

Type
list of *profit.sur.encoders.Encoder*

output_encoders

Encoding used on output data.

Type
list of *profit.sur.encoders.Encoder*

Default parameters:

surrogate: GPy save: ./model_{surrogate label}.hdf5 load: False fixed_sigma_n: False input_encoders: [{'class': 'exclude', 'columns': {constant columns}

{'class': 'log10', 'columns': {log input columns}, 'parameters': {}}, {'class': 'normalization', 'columns': {input columns}, 'parameters': {}}]

output_encoders: [{'class': 'normalization', 'columns': {output columns}, 'parameters': {}}]

`_defaults`

`train`(*X*, *y*)

Trains the surrogate on input points X and model outputs y.

Depending on the surrogate, the signature can vary.

Parameters

- **X** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **fixed_sigma_n** (*bool*) – Whether the noise σ_n is fixed during optimization.

`predict`(*Xpred*)

Predicts model output y for input Xpred based on surrogate.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Generally the uncertainty of the fit. For Gaussian Processes this is

the diagonal of the posterior covariance matrix.

Return type
tuple

`save_model`(*path*)

Saves the surrogate to a file. The file format can vary between surrogates. As default, the surrogate is saved to 'base_dir/model_{surrogate_label}.hdf5'.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

`classmethod load_model`(*path*)

Loads a saved surrogate from a file. The file format can vary between surrogates.

Identifies the surrogate by its class label in the file name.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.Surrogate

classmethod from_config(*config*, *base_config*)

Instantiates a surrogate based on the parameters given in the configuration file and delegates to child.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

classmethod handle_subconfig(*config*, *base_config*)

class profit.sur.ann.artificial_neural_network.**ANNSurrogate**

Bases: *profit.sur.sur.Surrogate*

Base class for all surrogate models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not.

Type
bool

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

input_encoders

Encoding used on input data.

Type
list of *profit.sur.encoders.Encoder*

output_encoders

Encoding used on output data.

Type
list of *profit.sur.encoders.Encoder*

Default parameters:
surrogate: GPy save: ./model_{surrogate label}.hdf5 load: False fixed_sigma_n: False input_encoders: [{'class': 'exclude', 'columns': {constant columns}

{'class': 'log10', 'columns': {log input columns}, 'parameters': {}}, {'class': 'normalization', 'columns': {input columns}, 'parameters': {}}]

output_encoders: [{'class': 'normalization', 'columns': {output columns}, 'parameters': {}}]

train(*x*, *y*)

Fits a artificial neural network to input points x and model outputs y with scale sigma_f and noise sigma_n

abstract add_training_data(*x*, *y*, *sigma=None*)

Adds input points x and model outputs y with std. deviation sigma and updates the inverted covariance matrix for the GP via the Sherman-Morrison-Woodbury formula

predict(*x*)

Predicts model output y for input Xpred based on surrogate.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Generally the uncertainty of the fit. For Gaussian Processes this is

the diagonal of the posterior covariance matrix.

Return type
tuple

class profit.sur.ann.artificial_neural_network.**Autoencoder**(*D*, *d*)

Bases: *profit.sur.sur.Surrogate*, torch.nn.Module

Nonlinear autoencoder with activation functions

forward(*x*)

train(*x*, *learning_rate=0.01*, *nstep=1000*)

Trains the surrogate on input points X and model outputs y.

Depending on the surrogate, the signature can vary.

Parameters

- **X** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **fixed_sigma_n** (*bool*) – Whether the noise σ_n is fixed during optimization.

profit.sur.gp

Subpackages

profit.sur.gp.backend

Submodules

profit.sur.gp.backend.gp_functions

Collection of functions for the Custom GPSurrogate.

Module Contents

Functions

optimize(xtrain, ytrain, a0, kernel[, fixed_sigma_n, ...])	Finds optimal hyperparameters from initial array a0, sorted as [length_scale, scale, noise].
solve_cholesky(L, b)	Solves a linear equation with a lower triangular matrix L from the Cholesky decomposition.
negative_log_likelihood_cholesky(hyp, X, y, kernel[, ...])	Computes the negative log likelihood using the Cholesky decomposition of the covariance matrix.
negative_log_likelihood(hyp, X, y, kernel[, ...])	Computes the negative log likelihood either by a Cholesky- or an Eigendecomposition.
invert_cholesky(L)	Inverts a positive-definite matrix based on a Cholesky decomposition.
invert(K[, neig, tol, eps, max_iter])	Inverts a positive-definite matrix using either a Cholesky- or an Eigendecomposition.
marginal_variance_BBQ(Xtrain, ytrain, Xpred, kernel, ...)	Calculates the marginal variance to infer the next point in active learning.
predict_f(hyp, x, y, xtest, kernel[, return_full_cov, ...])	Predicts values given only a set of hyperparameters and a kernel.

profit.sur.gp.backend.gp_functions.**optimize**(*xtrain*, *ytrain*, *a0*, *kernel*, *fixed_sigma_n=False*, *eval_gradient=False*, *return_hess_inv=False*)

Finds optimal hyperparameters from initial array a0, sorted as [length_scale, scale, noise].

The objective function, which is minimized, is the negative log likelihood. The hyperparameters are transformed logarithmically before the optimization to enhance its flexibility. They are transformed back afterwards.

Parameters

- **xtrain** (*ndarray*) –
- **ytrain** (*ndarray*) –
- **a0** (*ndarray*) – Flat array of initial hyperparameters.
- **kernel** (*function*) – Function of the kernel, not the actual matrix.
- **fixed_sigma_n** (*bool*) – If the noise σ_n should be fixed during optimization.

- **eval_gradient** (*bool*) – Whether the analytic derivatives of the kernel and likelihood are used in the optimization.
- **return_hess_inv** (*bool*) – If True, returns the inverse Hessian matrix from the optimization result. This is important for advanced active learning.

Returns

A tuple containing:

- opt_hyperparameters (ndarray): Flat array of optimized hyperparameters.
- hess_inv (scipy.optimize.LbfgsInvHessProduct): Inverse Hessian matrix in the form of a scipy linear

operator. If return_hess_inv is False, only the optimized hyperparameters are returned.

Return type
tuple

profit.sur.gp.backend.gp_functions.**solve_cholesky**(*L*, *b*)

Solves a linear equation with a lower triangular matrix L from the Cholesky decomposition.

In the context of GP's the L is the Cholesky decomposition of the training kernel matrix and b is the training output data.

$$\alpha = L^T Lb \tag{8.1}$$

Parameters

- **L** (*ndarray*) – Lower triangular matrix.
- **b** (*ndarray*) – A vector.

Returns
ndarray

profit.sur.gp.backend.gp_functions.**negative_log_likelihood_cholesky**(*hyp*, *X*, *y*, *kernel*, *eval_gradient=False*, *log_scale_hyp=False*, *fixed_sigma_n=False*)

Computes the negative log likelihood using the Cholesky decomposition of the covariance matrix.

The calculation follows Rasmussen&Williams 2006, p. 19, 113-114.

$$NL = \frac{1}{2} y^T \alpha + tr(\log(L)) + \frac{n}{2} \log(2\pi) \tag{8.2}$$

$$\frac{dNL}{d\theta} = \frac{1}{2} tr\left((K_y^{-1} - \alpha\alpha^T) \frac{\partial K}{\partial \theta} \right) \tag{8.3}$$

$$\alpha = K_y^{-1} y \tag{8.4}$$

Parameters

- **hyp** (*ndarray*) – Flat hyperparameter array [length_scale, sigma_f, sigma_n]. They can also be already log transformed.

- **X** (*ndarray*) – Training input points.
- **y** (*ndarray*) – Observed training output.
- **kernel** (*function*) – Function to build the covariance matrix.
- **eval_gradient** (*bool*) – If the analytic gradient of the negative log likelihood w.r.t. the hyperparameters should be returned.
- **log_scale_hyp** (*bool*) – Whether the hyperparameters are log transformed. This is important for the gradient calculation.
- **fixed_sigma_n** (*bool*) – If the noise σ_n is kept fixed. In this case, there is no gradient with respect to sigma_n.

Returns

A tuple containing:

- nll (float): The negative log likelihood.
- dnll (ndarray): The derivative of the negative log likelihood w.r.t. to the hyperparameters.

If eval_gradient is False, only nll is returned.

Return type
tuple

profit.sur.gp.backend.gp_functions.**negative_log_likelihood**(*hyp*, *X*, *y*, *kernel*, *eval_gradient=False*, *log_scale_hyp=False*, *fixed_sigma_n_value=None*, *neig=0*, *max_iter=1000*)

Computes the negative log likelihood either by a Cholesky- or an Eigendecomposition.

First, the Cholesky decomposition is tried. If this results in a LinAlgError, the biggest eigenvalues are calculated until convergence or until the maximum iterations are reached. The eigenvalues are cut off at $1e-10$ due ensure numerical stability.

$$NL = \frac{1}{2}\left(y^T\alpha + tr(\log(\lambda)) + n_{eig}\log(2\pi)\right) \tag{8.5}$$

$$\frac{dNL}{d\theta} = \frac{1}{2}tr\left((K_y^{-1} - \alpha\alpha^T)\frac{\partial K}{\partial\theta}\right) \tag{8.6}$$

$$\alpha = v(\lambda^{-1}(v^T y)) \tag{8.7}$$

Parameters

- **hyp** (*ndarray*) – Flat hyperparameter array [length_scale, sigma_f, sigma_n].
- **X** (*ndarray*) – Training input points.
- **y** (*ndarray*) – Observed training output.
- **kernel** (*function*) – Function to build the covariance matrix.
- **eval_gradient** (*bool*) – If the analytic gradient of the negative log likelihood w.r.t. the hyperparameters should be returned.
- **log_scale_hyp** (*bool*) – Whether the hyperparameters are log transformed. This is important for the gradient calculation.
- **fixed_sigma_n_value** (*float*) – The value of the fixed noise σ_n. If it should be optimizied as well, this should be None.

- **neig** (*int*) – Initial number of eigenvalues to calculate if the Cholesky decomposition is not successful. This is doubled during every iteration.
- **max_iter** (*int*) – Maximum number of iterations of the eigenvalue solver until convergence must be reached.

Returns

A tuple containing:

- nll (float): The negative log likelihood.
- dnll (ndarray): The derivative of the negative log likelihood w.r.t. to the hyperparameters.

If eval_gradient is False, only nll is returned.

Return type
tuple

profit.sur.gp.backend.gp_functions.**invert_cholesky**(*L*)

Inverts a positive-definite matrix based on a Cholesky decomposition.

This is used to invert the covariance matrix.

Parameters
L (*ndarray*) – Lower triangular matrix from a Cholesky decomposition.

Returns
Inverse of the matrix $L^T L$

Return type
ndarray

profit.sur.gp.backend.gp_functions.**invert**(*K*, *neig=0*, *tol=1e-10*, *eps=1e-06*, *max_iter=1000*)

Inverts a positive-definite matrix using either a Cholesky- or an Eigendecomposition.

The solution method depends on the rapidness of decay of the eigenvalues.

Parameters

- **K** (*np.ndarray*) – Kernel matrix.
- **neig** (*int*) – Initial number of eigenvalues to calculate if the Cholesky decomposition is not successful. This is doubled during every iteration.
- **tol** (*float*) – Convergence criterion for the eigenvalues.
- **eps** (*float*) – Small number to be added to diagonal of kernel matrix to ensure positive definiteness.
- **max_iter** (*int*) – Maximum number of iterations of the eigenvalue solver until convergence must be reached.

Returns
Inverse covariance matrix.

Return type
np.ndarray

profit.sur.gp.backend.gp_functions.**marginal_variance_BBQ**(*Xtrain*, *ytrain*, *Xpred*, *kernel*, *hyperparameters*, *hess_inv*, *fixed_sigma_n=False*, *alpha=None*, *predictive_variance=0*)

Calculates the marginal variance to infer the next point in active learning.

The calculation follows Osborne (2012).

$\tilde{V}$ ... Marginal covariance matrix $\hat{V}$ ... Predictive variance $\frac{dm}{d\theta}$ ... Derivative of the predictive mean w.r.t. the hyperparameters H ... Hessian matrix

$$\tilde{V} = \left(\frac{dm}{d\theta}\right) H^{-1} \left(\frac{dm}{d\theta}\right)^T \quad (8.8)$$

Parameters

- **Xtrain** (*ndarray*) – Input training points.
- **ytrain** (*ndarray*) – Observed output daat.
- **Xpred** (*ndarray*) – Possible prediction input points.
- **kernel** (*function*) – Function to build the covariance matrix.
- **hyperparameters** (*dict*) – Dictionary of the hyperparameters.
- **hess_inv** (*ndarray*) – Inverse Hessian matrix.
- **fixed_sigma_n** (*bool*) – If the noise σ_n was fixed during optimization. Then, the Hessian has to be padded with zeros.
- **alpha** (*ndarray*) – If available, the result of $K_y^{-1}y$, else None.
- **predictive_variance** (*ndarray*) – Predictive variance only. This is added to the marginal variance.

Returns
Sum of the actual marginal variance and the predictive variance.

Return type
ndarray

profit.sur.gp.backend.gp_functions.**predict_f**(*hyp*, *x*, *y*, *xtest*, *kernel*, *return_full_cov=False*, *neig=0*)

Predicts values given only a set of hyperparameters and a kernel.

This function is independent of the surrogates and used to quickly predict a function output with specific hyperparameters.

The calculation follows Rasmussen&Williams, p. 16, eq. 23-24.

Parameters

- **hyp** (*ndarray*) – Flat array of hyperparameters, sorted as [length_scale, sigma_f, sigma_n].
- **x** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **xtest** (*ndarray*) – Prediction points.
- **kernel** (*function*) – Function to build the covariance matrix.

- **return_full_cov** (*bool*) – If True, returns the full covariance matrix, otherwise only its diagonal.
- **neig** (*int*) – Initial number of eigenvalues to be computed during the inversion of the covariance matrix.

Returns

A tuple containing:

- fstar (ndarray): Posterior mean.
- vstar (ndarray): Posterior covariance matrix or its diagonal.

Return type
tuple

profit.sur.gp.backend.init_kernels

Script to initialize kernels via SymPy.

Running this script generates Fortran code for kernels that is written to *kernels_base.f90*. This code has to be compiled via *make* subsequently.

Module Contents

profit.sur.gp.backend.init_kernels.**l**

profit.sur.gp.backend.init_kernels.**kern**

profit.sur.gp.backend.init_kernels.**dkern**

profit.sur.gp.backend.init_kernels.**d2kern**

profit.sur.gp.backend.init_kernels.**funlist = []**

profit.sur.gp.backend.init_kernels.**funlist**

profit.sur.gp.backend.python_kernels

Module which includes kernels for the Custom surrogate.

Module Contents

Functions

RBF(X, Y[, length_scale, sigma_f, sigma_n, eval_gradient])	Squared exponential kernel, also call Radial-Basis-Function kernel (RBF).
LinearEmbedding(X, Y, R[, sigma_f, sigma_n, eval_gradient])	The linear embedding kernel according to Garnett (2014) is a generalisation of the RBF kernel.

profit.sur.gp.backend.python_kernels.**RBF**(*X*, *Y*, *length_scale=1*, *sigma_f=1*, *sigma_n=0*, *eval_gradient=False*)

Squared exponential kernel, also call Radial-Basis-Function kernel (RBF).

The RBF kernel is one of the most common kernels used and is especially suited for smooth functions.

$$K(X,Y) = \sigma_f^2 \exp(-\frac{1}{2}\frac{|X-Y|}{l^2}) + \sigma_n^2 \mathbf{1} \quad (8.9)$$

Parameters

- **X** (*ndarray*) – Input points.
- **Y** (*ndarray*) – Other input points.
- **length_scale** (*float/ndarray*) – Length scale l of the kernel function.
- **sigma_f** (*float*) – Scale σ_f.
- **sigma_n** (*float*) – Additive noise σ_n.
- **eval_gradient** (*bool*) – Indicates if the gradient with respect to the hyperparameters l, σ_f and σ_n should be returned.

Returns

A tuple containing:

- K (ndarray): Kernel matrix of size (X.shape[0], Y.shape[0]).
- dK (ndarray): Derivative of the kernel w.r.t. the hyperparameters l, σ_f and σ_n.

If eval_gradient is False, only K is returned.

Return type
tuple/ndarray

profit.sur.gp.backend.python_kernels.**LinearEmbedding**(*X*, *Y*, *R*, *sigma_f=1e-06*, *sigma_n=0*, *eval_gradient=False*)

The linear embedding kernel according to Garnett (2014) is a generalisation of the RBF kernel.

The RBF kernel with different length scales in each dimension (ARD kernel) is a special case where only the diagonal elements of the matrix R are non-zero and represent inverse length scales. The ARD kernel is suited to detect relevant dimensions of the input data. In contrast, the linear embedding kernel can also detect relevant linear combinations of dimensions, e.g. if a function only varies in the direction $x1 + x2$. Thus, the linear embedding kernel can be used to find a lower dimensional representation of the data in aribitrary directions.

$$K(X,Y) = \sigma_f^2 \exp(-\frac{1}{2}(X-Y)R^T R(X-Y)^T) + \sigma_n^2 \mathbf{1} \quad (8.10)$$

Here, we use the convention that the data X and Y are of shape (n, D) and therefore R has to be of shape (d, D) where D > d to find a lower dimensional representation.

Parameters:

X (ndarray): Input points of shape (n, D) Y (ndarray): Other input ponits. R (ndarray): Matrix or flattened array of hyperparameters. It is automatically reshaped to fit the input data.

Every matrix element represents a hyperparameter.

sigma_f (float): Scale σ_f. sigma_n (float): Additive noise σ_n. eval_gradient (bool): Indicates if the gradient with respect to the hyperparameters R_{ij}, σ_f and

σ_n should be returned.

Returns

A tuple containing:

- K (ndarray): Kernel matrix of size (X.shape[0], Y.shape[0]).
- dK (ndarray): Derivative of the kernel w.r.t. the hyperparameters R_{ij}, σ_f and σ_n.

If eval_gradient is False, only K is returned.

Return type

tuple/ndarray

Submodules

profit.sur.gp.custom_surrogate

Module Contents

Classes

GPSurrogate	Custom GP model made from scratch.
MultiOutputGPSurrogate	This is the base class for all Gaussian Process models.

class profit.sur.gp.custom_surrogate.**GPSurrogate**

Bases: *profit.sur.gp.GaussianProcess*

Custom GP model made from scratch.

Supports custom Python and Fortran kernels with analytic derivatives and advanced Active Learning.

hess_inv

Inverse Hessian matrix which is required for active learning. It is calculated during the hyperparameter optimization.

Type

ndarray

property Ky

Full training covariance matrix as defined in the kernel including data noise as specified in the hyperparameters.

property alpha

Convenient matrix-vector product of the inverse training matrix and the training output data. The equation is solved either exactly or with a least squares approximation.

$$\alpha = K_y^{-1} y_{train} \tag{8.11}$$

train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*, *eval_gradient=True*, *return_hess_inv=False*)

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, D) array of input training data.
- **y** (*ndarray*) – (n, 1) array of training output. Currently, only scalar output is supported.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the specific surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length_scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length_scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **eval_gradient** (*bool*) – Whether the gradients of the kernel and negative log likelihood are explicitly used in the scipy optimization or numerically calculated inside scipy.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

post_train()

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

add_training_data(*X*, *y*)

Add training points to existing data. This is important for active learning.

Only the training dataset is updated, but the hyperparameters are not optimized yet.

Parameters

- **X** (*ndarray*) – Input points to add.
- **y** (*ndarray*) – Observed output to add.

set_ytrain(*ydata*)

Set the observed training outputs. This is important for active learning.

Parameters
ydata (*np.array*) – Full training output data.

get_marginal_variance(*Xpred*)

save_model(*path*)

Saves the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Loads a saved model from a .hdf5 file and updates its attributes.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.gaussian_process.GPSurrogate

select_kernel(*kernel*)

Convert the name of the kernel as string to the kernel class object of the surrogate. First search the kernels implemented in python, then the Fortran kernels.

Parameters
kernel (*str*) – Kernel string such as 'RBF'. Only single kernels are supported currently.

Returns
Kernel object of the class. This is the function which builds the kernel and not the calculated covariance matrix.

Return type
object

optimize(*fixed_sigma_n=base_defaults['fixed_sigma_n']*, *eval_gradient=False*, *return_hess_inv=False*)

Optimize the hyperparameters length_scale l, scale σ_f and noise σ_n.

As a backend, the scipy minimize optimizer is used.

Parameters

- **fixed_sigma_n** (*bool*) – Indication if the data noise should also be optimized or not.
- **eval_gradient** (*bool*) – Flag if the gradients of the kernel and negative log likelihood should be used explicitly or numerically calculated inside the optimizer.

- **return_hess_inv** (*bool*) – Whether to set the inverse Hessian attribute hess_inv which is used to calculate the marginal variance in active learning.

_set_hyperparameters_from_model(*model_hyperparameters*)

class profit.sur.gp.custom_surrogate.**MultiOutputGPSurrogate**(*child=GPSurrogate*)

Bases: *profit.sur.gp.GaussianProcess*

This is the base class for all Gaussian Process models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not. If an ndarray is given, its length must match the training data.

Type
bool/float/ndarray

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

kernel

Kernel identifier such as 'RBF' or directly the (surrogate specific) kernel object. Defaults to 'RBF'.

Type
str/object

hyperparameters

Parameters like length-scale, variance and noise which can be optimized during training. As default, they are inferred from the training data.

Type
dict

Default parameters:
surrogate: GPy kernel: RBF

Default hyperparameters:
l … length scale σ_f … scaling σ_n … data noise

$$l = \frac{1}{2}\overline{|x - x'|} \tag{8.12}$$
$$\sigma_f = \overline{std(y)} \tag{8.13}$$
$$\sigma_n = 0.01 \cdot \overline{max(y) - min(y)} \tag{8.14}$$

train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*, *return_hess_inv=False*)

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

_set_hyperparameters_from_model()

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

add_training_data(*X*, *y*)

set_ytrain(*y*)

save_model(*path*)

Saves the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Loads a saved surrogate from a file. The file format can vary between surrogates.

Identifies the surrogate by its class label in the file name.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.Surrogate

optimize(**opt_kwargs*)

Find optimized hyperparameters of the model. Optional kwargs for tweaking optimization.

Parameters
opt_kwargs – Keyword arguments for optimization.

special_hyperparameter_decoding(*key*, *value*)

get_marginal_variance(*Xpred*)

profit.sur.gp.gaussian_process

This module contains the backends for various Gaussian Process surrogate models.

Gaussian Processes (GPs) are a generalization of Gaussian Distributions which are described by mean- and covariance functions. They can be used as as a non-parametric, supervised machine-learning technique for regression and classification. The advantages of GPs over other machine-learning techniques such as Artificial Neural Networks is the consistent, analytic mathematical derivation within probability-theory and therefore their intrinsic uncertainty-quantification of the predictions by means of the covariance function. This makes the results of GP fits intuitive to interpret.

GPs belong (like e.g. Support Vector Machines) to the kernel methods of machine learning. The mean function is often neglected and set to zero, because most functions can be modelled with a suitable covariance function or a combination of several covariance functions. The most important kernels are the Gaussian (RBF) and its generalization, the Matern kernels.

Isotropic RBF Kernel:

$$k(x, x') = \sigma_f^2 \exp(-\frac{1}{2}\frac{|x - x'|}{l^2}) \tag{8.15}$$

Literature:
Rasmussen & Williams 2006: General Introduction to Gaussian Processes Garnett 2014: Active Learning and Linear Embeddings Osborne 2012: Active Learning of Hyperparameters

Module Contents

Classes

GaussianProcess	This is the base class for all Gaussian Process models.

class profit.sur.gp.gaussian_process.**GaussianProcess**

Bases: *profit.sur.sur.Surrogate*

This is the base class for all Gaussian Process models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not. If an ndarray is given, its length must match the training data.

Type
bool/float/ndarray

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

kernel

Kernel identifier such as 'RBF' or directly the (surrogate specific) kernel object. Defaults to 'RBF'.

Type
str/object

hyperparameters

Parameters like length-scale, variance and noise which can be optimized during training. As default, they are inferred from the training data.

Type
dict

Default parameters:
surrogate: GPy kernel: RBF

Default hyperparameters:
l … length scale σ_f … scaling σ_n … data noise

$$l = \frac{1}{2}\overline{|x - x'|} \tag{8.16}$$
$$\sigma_f = \overline{std(y)} \tag{8.17}$$
$$\sigma_n = 0.01 \cdot \overline{max(y) - min(y)} \tag{8.18}$$

pre_train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*)

Check the training data, initialize the hyperparameters and set the kernel either from the given parameter, from config or from the default values.

Parameters

- **X** – (n, d) or (n,) array of input training data.
- **y** – (n, D) or (n,) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the specific surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool/float/ndarray*) – Indicates if the data noise should be optimized or not. If an ndarray is given, its length must match the training data.

set_attributes(***kwargs*)

infer_hyperparameters()

abstract train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*, *return_hess_inv=False*)

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.

- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

abstract predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

abstract optimize(***opt_kwargs*)

Find optimized hyperparameters of the model. Optional kwargs for tweaking optimization.

Parameters
opt_kwargs – Keyword arguments for optimization.

classmethod from_config(*config*, *base_config*)

Instantiate a GP model from the configuration file with kernel and hyperparameters.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
Instantiated surrogate.

Return type
profit.sur.gaussian_process.GaussianProcess

select_kernel(*kernel*)

Convert the name of the kernel as string to the kernel class object of the surrogate.

Parameters
kernel (*str*) – Kernel string such as 'RBF' or depending on the surrogate also product and sum kernels such as 'RBF+Matern52'.

Returns
Custom or imported kernel object. This is the function which builds the kernel and not the calculated covariance matrix.

Return type
object

`decode_hyperparameters`()

Decodes the hyperparameters, as encoded ones are used in the surrogate model.

`special_hyperparameter_decoding`(*key*, *value*)

`print_hyperparameters`(*prefix*)

Helper function to print the hyperparameter dict.

Parameters
prefix (*str*) – Usually 'Initialized', 'Loaded' or 'Optimized' to identify the state of the hyperparameters.

profit.sur.gp.gpy_surrogate

Module Contents

Classes

GPySurrogate	Surrogate for https://github.com/SheffieldML/GPy.
CoregionalizedGPySurrogate	Surrogate for https://github.com/SheffieldML/GPy.

`class` profit.sur.gp.gpy_surrogate.**GPySurrogate**

Bases: *profit.sur.gp.GaussianProcess*

Surrogate for https://github.com/SheffieldML/GPy.

`model`

Model object of GPy.

Type
GPy.models

`train`(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*)

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.

- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

add_training_data(*X*, *y*)

Adds training points to the existing dataset.

This is important for Active Learning. The data is added but the hyperparameters are not optimized yet.

Parameters

- **X** (*ndarray*) – Input points to add.
- **y** (*ndarray*) – Observed output to add.

set_ytrain(*y*)

Set the observed training outputs. This is important for active learning.

Parameters: y (np.array): Full training output data.

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

save_model(*path*)

Save the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Loads a saved model from a .hdf5 file and updates its attributes. In case of a multi-output model, the .pkl file is loaded, since .hdf5 is not supported yet.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
GPy.models

select_kernel(*kernel*)

Get the GPy.kern kernel by matching the given string kernel identifier.

Parameters
kernel (*str*) – Kernel string such as 'RBF' or depending on the surrogate also product and sum kernels such as 'RBF+Matern52'.

Returns
GPy kernel object. Currently, for sum and product kernels, the initial hyperparameters are the same for all kernels.

Return type
GPy.kern

optimize(*return_hess_inv=False*, ***opt_kwargs*)

For hyperparameter optimization the GPy base optimization is used.

Currently, the inverse Hessian can not be retrieved, which limits the active learning effectivity.

Parameters

- **return_hess_inv** (*bool*) – Is not considered currently.
- **opt_kwargs** – Keyword arguments used directly in the GPy base optimization.

_set_hyperparameters_from_model()

Helper function to set the hyperparameter dict from the model.

It depends on whether it is a single kernel or a combined one.

special_hyperparameter_decoding(*key*, *value*)

class profit.sur.gp.gpy_surrogate.**CoregionalizedGPySurrogate**

Bases: *GPySurrogate*

Surrogate for https://github.com/SheffieldML/GPy.

model

Model object of GPy.

Type
GPy.models

pre_train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*)

Check the training data, initialize the hyperparameters and set the kernel either from the given parameter, from config or from the default values.

Parameters

- **X** – (n, d) or (n,) array of input training data.
- **y** – (n, D) or (n,) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the specific surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool/float/ndarray*) – Indicates if the data noise should be optimized or not. If an ndarray is given, its length must match the training data.

train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*)

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

add_training_data(*X*, *y*)

Adds training points to the existing dataset.

This is important for Active Learning. The data is added but the hyperparameters are not optimized yet.

Parameters

- **X** (*ndarray*) – Input points to add.
- **y** (*ndarray*) – Observed output to add.

set_ytrain(*y*)

Set the observed training outputs. This is important for active learning.

Parameters: y (np.array): Full training output data.

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

save_model(*path*)

Save the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Loads a saved model from a .hdf5 file and updates its attributes. In case of a multi-output model, the .pkl file is loaded, since .hdf5 is not supported yet.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
GPy.models

special_hyperparameter_decoding(*key*, *value*)

profit.sur.gp.sklearn_surrogate

Module Contents

Classes

SklearnGPSurrogate	Surrogate for https://github.com/scikit-learn/scikit-learn Gaussian process.
LinearEmbedding	Mixin for kernels which are stationary: k(X, Y)= f(X-Y).

class profit.sur.gp.sklearn_surrogate.**SklearnGPSurrogate**

Bases: *profit.sur.gp.GaussianProcess*

Surrogate for https://github.com/scikit-learn/scikit-learn Gaussian process.

model

Model object of Sklearn.

Type
sklearn.gaussian_process.GaussianProcessRegressor

train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*, ***kwargs*)

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.

- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

post_train()

add_training_data(*X*, *y*)

Add training points to existing data.

Parameters

- **X** (*ndarray*) – Input points to add.
- **y** (*ndarray*) – Observed output to add.

set_ytrain(*ydata*)

Set the observed training outputs. This is important for active learning.

Parameters
ydata (*np.array*) – Full training output data.

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

save_model(*path*)

Save the SklGPSurrogate model to a pickle file. All attributes of the surrogate are loaded directly from the model.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Load a saved SklGPSurrogate model from a pickle file and update its attributes.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.gaussian_process.SklearnGPSurrogate

`optimize`(***opt_kwargs*)

For hyperparameter optimization the Sklearn base optimization is used.

Currently, the inverse Hessian can not be retrieved, which limits the active learning effectivity.

Parameters
opt_kwargs – Keyword arguments used directly in the Sklearn base optimization.

`select_kernel`(*kernel*)

Get the sklearn.gaussian_process.kernels kernel by matching the given kernel identifier.

Parameters
kernel (*str*) – Kernel string such as 'RBF' or depending on the surrogate also product and sum kernels such as 'RBF+Matern52'.

Returns
Scikit-learn kernel object. Currently, for sum and product kernels, the initial hyperparameters are the same for all kernels.

Return type
sklearn.gaussian_process.kernels

`_set_hyperparameters_from_model`()

Helper function to set the hyperparameter dict from the model.

It depends on whether σ_n is fixed. Currently this is only stable for single kernels and not for Sum and Prod kernels.

class profit.sur.gp.sklearn_surrogate.**LinearEmbedding**(*dims*, *length_scale=np.array([1.0])*, *length_scale_bounds=(1e-05, 100000.0)*)

Bases: sklearn.gaussian_process.kernels.StationaryKernelMixin, sklearn.gaussian_process.kernels.NormalizedKernelMixin, sklearn.gaussian_process.kernels.Kernel

Mixin for kernels which are stationary: k(X, Y)= f(X-Y).

New in version 0.18.

property hyperparameter_length_scale

`__call__`(*X*, *Y=None*, *eval_gradient=False*)

Evaluate the kernel.

`__repr__`()

Return repr(self).

Package Contents

Classes

GaussianProcess	This is the base class for all Gaussian Process models.
GPySurrogate	Surrogate for https://github.com/SheffieldML/GPy.
GPSurrogate	Custom GP model made from scratch.

class profit.sur.gp.**GaussianProcess**

Bases: *profit.sur.sur.Surrogate*

This is the base class for all Gaussian Process models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not. If an ndarray is given, its length must match the training data.

Type
bool/float/ndarray

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

kernel

Kernel identifier such as 'RBF' or directly the (surrogate specific) kernel object. Defaults to 'RBF'.

Type
str/object

hyperparameters

Parameters like length-scale, variance and noise which can be optimized during training. As default, they are inferred from the training data.

Type
dict

Default parameters:
surrogate: GPy kernel: RBF

Default hyperparameters:
l ... length scale σ_f ... scaling σ_n ... data noise

$$l = \frac{1}{2}\overline{|x - x'|} \tag{8.19}$$
$$\sigma_f = \overline{std(y)} \tag{8.20}$$
$$\sigma_n = 0.01 \cdot \overline{max(y) - min(y)} \tag{8.21}$$

pre_train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*)

Check the training data, initialize the hyperparameters and set the kernel either from the given parameter, from config or from the default values.

Parameters

- **X** – (n, d) or (n,) array of input training data.
- **y** – (n, D) or (n,) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the specific surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool/float/ndarray*) – Indicates if the data noise should be optimized or not. If an ndarray is given, its length must match the training data.

set_attributes(***kwargs*)

infer_hyperparameters()

abstract train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*, *return_hess_inv=False*)

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.

- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

abstract predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

abstract optimize(***opt_kwargs*)

Find optimized hyperparameters of the model. Optional kwargs for tweaking optimization.

Parameters
opt_kwargs – Keyword arguments for optimization.

classmethod from_config(*config*, *base_config*)

Instantiate a GP model from the configuration file with kernel and hyperparameters.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Returns
Instantiated surrogate.

Return type
profit.sur.gaussian_process.GaussianProcess

select_kernel(*kernel*)

Convert the name of the kernel as string to the kernel class object of the surrogate.

Parameters
kernel (*str*) – Kernel string such as 'RBF' or depending on the surrogate also product and sum kernels such as 'RBF+Matern52'.

Returns
Custom or imported kernel object. This is the function which builds the kernel and not the calculated covariance matrix.

Return type
object

`decode_hyperparameters()`

Decodes the hyperparameters, as encoded ones are used in the surrogate model.

`special_hyperparameter_decoding(key, value)`

`print_hyperparameters(prefix)`

Helper function to print the hyperparameter dict.

Parameters
prefix (*str*) – Usually 'Initialized', 'Loaded' or 'Optimized' to identify the state of the hyperparameters.

`class profit.sur.gp.GPySurrogate`

Bases: *profit.sur.gp.GaussianProcess*

Surrogate for https://github.com/SheffieldML/GPy.

`model`

Model object of GPy.

Type
GPy.models

`train(X, y, kernel=defaults['kernel'], hyperparameters=defaults['hyperparameters'], fixed_sigma_n=base_defaults['fixed_sigma_n'])`

Trains the model on the dataset.

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, d) array of input training data.
- **y** (*ndarray*) – (n, D) array of training output.
- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

`add_training_data(X, y)`

Adds training points to the existing dataset.

This is important for Active Learning. The data is added but the hyperparameters are not optimized yet.

Parameters

- **X** (*ndarray*) – Input points to add.
- **y** (*ndarray*) – Observed output to add.

set_ytrain(*y*)

Set the observed training outputs. This is important for active learning.

Parameters: y (np.array): Full training output data.

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

save_model(*path*)

Save the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Loads a saved model from a .hdf5 file and updates its attributes. In case of a multi-output model, the .pkl file is loaded, since .hdf5 is not supported yet.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
GPy.models

select_kernel(*kernel*)

Get the GPy.kern kernel by matching the given string kernel identifier.

Parameters
kernel (*str*) – Kernel string such as 'RBF' or depending on the surrogate also product and sum kernels such as 'RBF+Matern52'.

Returns
GPy kernel object. Currently, for sum and product kernels, the initial hyperparameters are the same for all kernels.

Return type
GPy.kern

optimize(*return_hess_inv=False*, ***opt_kwargs*)

For hyperparameter optimization the GPy base optimization is used.

Currently, the inverse Hessian can not be retrieved, which limits the active learning effectivity.

Parameters

- **return_hess_inv** (*bool*) – Is not considered currently.
- **opt_kwargs** – Keyword arguments used directly in the GPy base optimization.

_set_hyperparameters_from_model()

Helper function to set the hyperparameter dict from the model.

It depends on whether it is a single kernel or a combined one.

special_hyperparameter_decoding(*key*, *value*)

class profit.sur.gp.**GPSurrogate**

Bases: *profit.sur.gp.GaussianProcess*

Custom GP model made from scratch.

Supports custom Python and Fortran kernels with analytic derivatives and advanced Active Learning.

hess_inv

Inverse Hessian matrix which is required for active learning. It is calculated during the hyperparameter optimization.

Type
ndarray

property Ky

Full training covariance matrix as defined in the kernel including data noise as specified in the hyperparameters.

property alpha

Convenient matrix-vector product of the inverse training matrix and the training output data. The equation is solved either exactly or with a least squares approximation.

$$\alpha = K_y^{-1} y_{train} \tag{8.22}$$

train(*X*, *y*, *kernel=defaults['kernel']*, *hyperparameters=defaults['hyperparameters']*, *fixed_sigma_n=base_defaults['fixed_sigma_n']*, *eval_gradient=True*, *return_hess_inv=False*)

After initializing the model with a kernel function and initial hyperparameters, it can be trained on input data X and observed output data y by optimizing the model's hyperparameters. This is done by minimizing the negative log likelihood.

Parameters

- **X** (*ndarray*) – (n, D) array of input training data.
- **y** (*ndarray*) – (n, 1) array of training output. Currently, only scalar output is supported.

- **kernel** (*str/object*) – Identifier of kernel like 'RBF' or directly the kernel object of the specific surrogate.
- **hyperparameters** (*dict*) – Hyperparameters such as length_scale, variance and noise. Taken either from given parameter, config file or inferred from the training data. The hyperparameters can be different depending on the kernel. E.g. The length_scale can be a scalar, a vector of the size of the training data, or for the custom LinearEmbedding kernel a matrix.
- **fixed_sigma_n** (*bool*) – Indicates if the data noise should be optimized or not.
- **eval_gradient** (*bool*) – Whether the gradients of the kernel and negative log likelihood are explicitly used in the scipy optimization or numerically calculated inside scipy.
- **return_hess_inv** (*bool*) – Whether to the attribute hess_inv after optimization. This is important for active learning.

post_train()

predict(*Xpred*, *add_data_variance=True*)

Predicts the output at test points Xpred.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Diagonal of the predicted covariance matrix.

Return type
tuple

add_training_data(*X*, *y*)

Add training points to existing data. This is important for active learning.

Only the training dataset is updated, but the hyperparameters are not optimized yet.

Parameters

- **X** (*ndarray*) – Input points to add.
- **y** (*ndarray*) – Observed output to add.

set_ytrain(*ydata*)

Set the observed training outputs. This is important for active learning.

Parameters
ydata (*np.array*) – Full training output data.

get_marginal_variance(*Xpred*)

save_model(*path*)

Saves the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

classmethod load_model(*path*)

Loads a saved model from a .hdf5 file and updates its attributes.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.gaussian_process.GPSurrogate

select_kernel(*kernel*)

Convert the name of the kernel as string to the kernel class object of the surrogate. First search the kernels implemented in python, then the Fortran kernels.

Parameters
kernel (*str*) – Kernel string such as 'RBF'. Only single kernels are supported currently.

Returns
Kernel object of the class. This is the function which builds the kernel and not the calculated covariance matrix.

Return type
object

optimize(*fixed_sigma_n=base_defaults['fixed_sigma_n']*, *eval_gradient=False*, *return_hess_inv=False*)

Optimize the hyperparameters length_scale l, scale σ_f and noise σ_n.

As a backend, the scipy minimize optimizer is used.

Parameters

- **fixed_sigma_n** (*bool*) – Indication if the data noise should also be optimized or not.
- **eval_gradient** (*bool*) – Flag if the gradients of the kernel and negative log likelihood should be used explicitly or numerically calculated inside the optimizer.
- **return_hess_inv** (*bool*) – Whether to set the inverse Hessian attribute hess_inv which is used to calculate the marginal variance in active learning.

_set_hyperparameters_from_model(*model_hyperparameters*)

profit.sur.linreg

Submodules

profit.sur.linreg.chaospy_linreg

Module Contents

Classes

ChaospyLinReg	Linear regression surrogate using polynomial expansions as basis

class profit.sur.linreg.chaospy_linreg.**ChaospyLinReg**(*model=defaults['model']*, *order=defaults['order']*, *model_kwargs=defaults['model_kwargs']*, *sigma_n=defaults['sigma_n']*, *sigma_p=defaults['sigma_p']*)

Bases: *profit.sur.linreg.LinearRegression*

Linear regression surrogate using polynomial expansions as basis functions from chaospy https://chaospy.readthedocs.io/en/master/reference/polynomial/index.html

 # ToDo

property model

set_model(*model*, *order*, *model_kwargs=None*)

Sets model parameters for surrogate

Parameters

- **model** (*str*) – Name of chaospy model to use
- **order** (*int*) – Highest order for polynomial basis functions
- **model_kwargs** (*dict*) – Keyword arguments for the model

transform(*X*)

Transforms input data on selected basis functions

Parameters
X (*ndarray*) – Input points

Returns
Basis functions evaluated at X

Return type
Phi (ndarray)

train(*X*, *y*)

Trains the surrogate on input points X and model outputs y.

Depending on the surrogate, the signature can vary.

Parameters

- **X** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **fixed_sigma_n** (*bool*) – Whether the noise σ_n is fixed during optimization.

predict(*Xpred*, *add_data_variance=False*)

Predicts model output y for input Xpred based on surrogate.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.

- yvar (ndarray): Generally the uncertainty of the fit. For Gaussian Processes this is the diagonal of the posterior covariance matrix.

Return type
tuple

`as_dict()`

Converts class to dictionary

ToDo

`save_model(path)`

Save the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

`classmethod load_model(path)`

Loads a saved model from a .hdf5 file and updates its attributes

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
ToDo Instantiated surrogate model.

`classmethod from_config(config, base_config)`

Instantiates a surrogate based on the parameters given in the configuration file and delegates to child.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

profit.sur.linreg.linear_regression

This module contains the backend for Linear Regression models

Work in progress

Module Contents

Classes

LinearRegression	Base class for all Linear Regression models.

`class profit.sur.linreg.linear_regression.LinearRegression`

Bases: *profit.sur.Surrogate*

Base class for all Linear Regression models.

`trained`

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

 # ToDo

Parameters:

Package Contents

Classes

LinearRegression	Base class for all Linear Regression models.
ChaospyLinReg	Linear regression surrogate using polynomial expansions as basis

class profit.sur.linreg.**LinearRegression**

Bases: *profit.sur.Surrogate*

Base class for all Linear Regression models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

 # ToDo

Parameters:

class profit.sur.linreg.**ChaospyLinReg**(*model=defaults['model']*, *order=defaults['order']*, *model_kwargs=defaults['model_kwargs']*, *sigma_n=defaults['sigma_n']*, *sigma_p=defaults['sigma_p']*)

Bases: *profit.sur.linreg.LinearRegression*

Linear regression surrogate using polynomial expansions as basis functions from chaospy https://chaospy.readthedocs.io/en/master/reference/polynomial/index.html

 # ToDo

property model

set_model(*model*, *order*, *model_kwargs=None*)

Sets model parameters for surrogate

Parameters

- **model** (*str*) – Name of chaospy model to use
- **order** (*int*) – Highest order for polynomial basis functions
- **model_kwargs** (*dict*) – Keyword arguments for the model

transform(*X*)

Transforms input data on selected basis functions

Parameters
X (*ndarray*) – Input points

Returns
Basis functions evaluated at X

Return type
Phi (ndarray)

`train`(*X*, *y*)

Trains the surrogate on input points X and model outputs y.

Depending on the surrogate, the signature can vary.

Parameters

- **X** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **fixed_sigma_n** (*bool*) – Whether the noise σ_n is fixed during optimization.

`predict`(*Xpred*, *add_data_variance=False*)

Predicts model output y for input Xpred based on surrogate.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Generally the uncertainty of the fit. For Gaussian Processes this is

the diagonal of the posterior covariance matrix.

Return type
tuple

`as_dict`()

Converts class to dictionary

ToDo

`save_model`(*path*)

Save the model as dict to a .hdf5 file.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

`classmethod load_model`(*path*)

Loads a saved model from a .hdf5 file and updates its attributes

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
ToDo Instantiated surrogate model.

classmethod from_config(*config*, *base_config*)

Instantiates a surrogate based on the parameters given in the configuration file and delegates to child.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

Submodules

profit.sur.encoders

Module Contents

Classes

Encoder	Base class to handle encoding and decoding of the input and output data before creating the surrogate model.
ExcludeEncoder	Excludes specific columns from the fit. Afterwards they are inserted at the same position.
Log10Encoder	Transforms the specified columns with log_{10}. This is done for LogUniform variables by default.
Normalization	Normalization of the specified columns. Usually this is done for all input and output,
PCA	Base class to handle encoding and decoding of the input and output data before creating the surrogate model.
KarhunenLoeve	Base class to handle encoding and decoding of the input and output data before creating the surrogate model.

class profit.sur.encoders.**Encoder**(*columns*, *parameters=None*)

Bases: *profit.util.base_class.CustomABC*

Base class to handle encoding and decoding of the input and output data before creating the surrogate model.

The base class itself does nothing. It delegates the encoding process to the childs which are called by their registered labels.

Parameters

- **columns** (*list[int]*) – Columns of the data the encoder acts on.
- **parameters** (*dict*) – Miscellaneous parameters stored during encoding, which are needed for decoding. E.g. the scaling factor during normalization.

label

Label of the encoder class.

Type
str

property repr

Easy to handle representation of the encoder for saving and loading. :returns:

List of all relevant information to reconstruct the encoder.
(label, columns, parameters dict)

Return type
list

`labels`

`encode`(*x*)

Applies the encoding function on given columns.

Parameters
x (*ndarray*) – Array to which the encoding is applied.

Returns
An encoded copy of the array x.

Return type
ndarray

`decode`(*x*)

Applies the decoding function on given columns.

Parameters
x (*ndarray*) – Array to which the decoding is applied.

Returns
A decoded copy of the array x.

Return type
ndarray

`decode_hyperparameters`(*value*)

Decoder for the surrogate hyperparameters, as the direct model uses encoded values. As a default, the unchanged value is returned.

Parameters
value (*np.array*) – The (encoded) value of the hyperparameter.

Returns
Decoded value.

Return type
np.array

`decode_variance`(*variance*)

`encode_func`(*x*)

Returns
Function used for decoding the data. E.g. $\log_{10}(x)$.

Return type
ndarray

`decode_func`(*x*)

Returns

Inverse transform of the encoding function. For an encoding of $\log_{10}(x)$ this would be 10^x.

Return type
ndarray

class profit.sur.encoders.**ExcludeEncoder**(*columns*, *parameters=None*)

Bases: *Encoder*

Excludes specific columns from the fit. Afterwards they are inserted at the same position.

Variables:
excluded_values (np.array): Slice of the input data which is excluded.

encode(x)

Applies the encoding function on given columns.

Parameters
x (*ndarray*) – Array to which the encoding is applied.

Returns
An encoded copy of the array x.

Return type
ndarray

decode(x)

Applies the decoding function on given columns.

Parameters
x (*ndarray*) – Array to which the decoding is applied.

Returns
A decoded copy of the array x.

Return type
ndarray

class profit.sur.encoders.**Log10Encoder**(*columns*, *parameters=None*)

Bases: *Encoder*

Transforms the specified columns with log_{10}. This is done for LogUniform variables by default.

encode_func(x)

Returns
Function used for decoding the data. E.g. $\log_{10}(x)$.

Return type
ndarray

decode_func(x)

Returns

Inverse transform of the encoding function. For an encoding of $\log_{10}(x)$ this would be 10^x.

Return type
ndarray

class profit.sur.encoders.**Normalization**(*columns*, *parameters=None*)

Bases: *Encoder*

Normalization of the specified columns. Usually this is done for all input and output,
so the surrogate can fit on a (0, 1)^n cube with zero mean and unit variance.

$$x' = (x - x_{min})/(x_{max} - x_{min}) \tag{8.23}$$
$$x = (x_{max} - x_{min}) * x' + x_{min} \tag{8.24}$$

Parameters

- **xmax** (*np.array*) – Max. value of the data for each column.
- **xmin** (*np.array*) – Min. value of the data for each column.
- **xmean** (*np.array*) – Mean value of the data for each column.
- **xstd** (*np.array*) – Standard deviation of the data for each column.
- **xmax_centered** (*np.array*) – Max. value of the data after mean and variance standardization.
- **xmin_centered** (*np.array*) – Min. value of the data after mean and variance standardization.

encode(*x*)

Applies the encoding function on given columns.

Parameters
x (*ndarray*) – Array to which the encoding is applied.

Returns
An encoded copy of the array x.

Return type
ndarray

encode_func(*x*)

Returns
Function used for decoding the data. E.g. $\log_{10}(x)$.

Return type
ndarray

decode_func(*x*)

Returns

Inverse transform of the encoding function. For an encoding of $\log_{10}(x)$ this would be 10^x.

Return type
ndarray

decode_hyperparameters(*value*)

Decode surrogate's hyperparameters. Distinguish between length_scale (only input_encoders) and sigma_f, sigma_n (only output_encoders) done in profit.sur.gp.gaussian_process.GaussianProcess.

Parameters
value (*np.array*) – The (encoded) value of the hyperparameter.

Returns
Decoded value.

Return type
np.array

decode_variance(*variance*)

class profit.sur.encoders.**PCA**(*columns=()*, *parameters=None*)

Bases: *Encoder*

Base class to handle encoding and decoding of the input and output data before creating the surrogate model.

The base class itself does nothing. It delegates the encoding process to the childs which are called by their registered labels.

Parameters

- **columns** (*list[int]*) – Columns of the data the encoder acts on.
- **parameters** (*dict*) – Miscellaneous parameters stored during encoding, which are needed for decoding. E.g. the scaling factor during normalization.

label

Label of the encoder class.

Type
str

property features

Returns: neig feature vectors of length N.

init_eigvalues(*y*)

encode(*y*)

Parameters
y – ntest sample vectors of length N.

Returns
Expansion coefficients of y in eigenbasis.

decode(*z*)

Parameters
z – Expansion coefficients of y in eigenbasis.

Returns
Reconstructed ntest sample vectors of length N.

decode_variance(*variance*)

class profit.sur.encoders.**KarhunenLoeve**(*columns=()*, *parameters=None*)

Bases: *PCA*

Base class to handle encoding and decoding of the input and output data before creating the surrogate model.

The base class itself does nothing. It delegates the encoding process to the childs which are called by their registered labels.

Parameters

- **columns** (*list[int]*) – Columns of the data the encoder acts on.
- **parameters** (*dict*) – Miscellaneous parameters stored during encoding, which are needed for decoding. E.g. the scaling factor during normalization.

`label`

Label of the encoder class.

Type
str

`property features`

Returns: neig feature vectors of length N.

`encode`(*y*)

Parameters
y – ntest sample vectors of length N.

Returns
Expansion coefficients of y in eigenbasis.

`init_eigvalues`(*y*)

`decode`(*z*)

Parameters
z – Expansion coefficients of y in eigenbasis.

Returns
Reconstructed ntest sample vectors of length N.

`decode_variance`(*variance*)

profit.sur.sur

Abstract base class for all surrogate models.

Class structure: - Surrogate

- **GaussianProcess**
  - GPSurrogate (Custom)
  - GPySurrogate (GPy)
  - SklearnGPSurrogate (Sklearn)
- **ANN**
  - ANNSurrogate (Pytorch)
  - Autoencoder (Pytorch)
- **LinearRegression**
  - Linear regression surrogates (Work in progress)

Module Contents

Classes

Surrogate	Base class for all surrogate models.

class profit.sur.sur.**Surrogate**

Bases: *profit.util.base_class.CustomABC*

Base class for all surrogate models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not.

Type
bool

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

input_encoders

Encoding used on input data.

Type
list of *profit.sur.encoders.Encoder*

output_encoders

Encoding used on output data.

Type
list of *profit.sur.encoders.Encoder*

Default parameters:

surrogate: GPy save: ./model_{surrogate label}.hdf5 load: False fixed_sigma_n: False input_encoders: [{'class': 'exclude', 'columns': {constant columns}

{'class': 'log10', 'columns': {log input columns}, 'parameters': {}}, {'class': 'normalization', 'columns': {input columns}, 'parameters': {}}]

output_encoders: [{'class': 'normalization', 'columns': {output columns}, 'parameters': {}}]

`labels`

`encode_training_data()`

Encodes the input and output training data.

`decode_training_data()`

Applies the decoding function of the encoder in reverse order on the input and output training data.

`encode_predict_data(x)`

Transforms the input prediction points according to the encoder used for training.

Parameters
x (*ndarray*) – Prediction input points.

Returns
Encoded and normalized prediction points.

Return type
ndarray

`decode_predict_data(ym, yv)`

Rescales and then back-transforms the predicted output.

Parameters

- **ym** (*ndarray*) – Predictive output.
- **yv** (*ndarray*) – Variance of predicted output.

Returns

a tuple containing:

- ym (ndarray) Rescaled and decoded output values at the test input points.
- yv (ndarray): Rescaled predictive variance.

Return type
tuple

`add_input_encoder(encoder)`

Add encoder on input data.

Parameters
encoder (*profit.sur.encoder.Encoder*) –

`add_output_encoder(encoder)`

Add encoder on output data.

Parameters
encoder (*profit.sur.encoder.Encoder*) –

abstract train(*X*, *y*, *fixed_sigma_n=defaults['fixed_sigma_n']*)

Trains the surrogate on input points X and model outputs y.

Depending on the surrogate, the signature can vary.

Parameters

- **X** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **fixed_sigma_n** (*bool*) – Whether the noise σ_n is fixed during optimization.

pre_train(*X*, *y*)

Check the training data

Parameters

- **X** – (n, d) or (n,) array of input training data.
- **y** – (n, D) or (n,) array of training output.

post_train()

abstract predict(*Xpred*, *add_data_variance=True*)

Predicts model output y for input Xpred based on surrogate.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Generally the uncertainty of the fit. For Gaussian Processes this is

the diagonal of the posterior covariance matrix.

Return type
tuple

pre_predict(*Xpred*)

Prepares the surrogate for prediction by checking if it is trained and validating the data.

Parameters
Xpred (*ndarray*) – (n, d) or (n,) array of input points for prediction

Returns
Checked input data or default values inferred from training data.

Return type
ndarray

abstract save_model(*path*)

Saves the surrogate to a file. The file format can vary between surrogates. As default, the surrogate is saved to 'base_dir/model_{surrogate_label}.hdf5'.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

abstract classmethod load_model(*path*)

Loads a saved surrogate from a file. The file format can vary between surrogates.

Identifies the surrogate by its class label in the file name.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.Surrogate

abstract classmethod from_config(*config*, *base_config*)

Instantiates a surrogate based on the parameters given in the configuration file and delegates to child.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

plot(*Xpred=None*, *independent=None*, *show=False*, *ref=None*, *add_data_variance=True*, *axes=None*)

Simple plotting for dimensions <= 2.

Fore more sophisticated plots use the command 'profit ui'.

Parameters

- **Xpred** (*ndarray*) – Prediction points where the fit is plotted. If None, it is inferred from the training points.
- **independent** (*dict*) – Dictionary of independent variables from config.
- **show** (*bool*) – If the figure should be shown directly.
- **ref** (*ndarray*) – Reference function which is fitted.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance.
- **axes** (*matplotlib.pyplot.axes*) – Axes object to insert the plot into. If None, a new figure is created.

default_Xpred()

Infer prediction values from training points in each dimension.

Currently a dense grid is created. This becomes inefficient for > 3 dimensions.

Returns
Prediction points.

Return type
ndarray

Package Contents

Classes

Surrogate	Base class for all surrogate models.

class profit.sur.**Surrogate**

Bases: *profit.util.base_class.CustomABC*

Base class for all surrogate models.

trained

Flag that indicates if the model is already trained and ready to make predictions.

Type
bool

fixed_sigma_n

Indicates if the data noise should be optimized or not.

Type
bool

Xtrain

Input training points.

Type
ndarray

ytrain

Observed output data. Vector output is supported for independent variables only.

Type
ndarray

ndim

Dimension of input data.

Type
int

output_ndim

Dimension of output data.

Type
int

input_encoders

Encoding used on input data.

Type
list of *profit.sur.encoders.Encoder*

output_encoders

Encoding used on output data.

Type
list of *profit.sur.encoders.Encoder*

Default parameters:

surrogate: GPy save: ./model_{surrogate label}.hdf5 load: False fixed_sigma_n: False input_encoders: [{'class': 'exclude', 'columns': {constant columns}

{'class': 'log10', 'columns': {log input columns}, 'parameters': {}}, {'class': 'normalization', 'columns': {input columns}, 'parameters': {}}]

output_encoders: [{'class': 'normalization', 'columns': {output columns}, 'parameters': {}}]

`labels`

`encode_training_data()`

Encodes the input and output training data.

`decode_training_data()`

Applies the decoding function of the encoder in reverse order on the input and output training data.

`encode_predict_data(x)`

Transforms the input prediction points according to the encoder used for training.

Parameters
x (*ndarray*) – Prediction input points.

Returns
Encoded and normalized prediction points.

Return type
ndarray

`decode_predict_data(ym, yv)`

Rescales and then back-transforms the predicted output.

Parameters

- **ym** (*ndarray*) – Predictive output.
- **yv** (*ndarray*) – Variance of predicted output.

Returns

a tuple containing:

- ym (ndarray) Rescaled and decoded output values at the test input points.
- yv (ndarray): Rescaled predictive variance.

Return type
tuple

`add_input_encoder(encoder)`

Add encoder on input data.

Parameters
encoder (*profit.sur.encoder.Encoder*) –

`add_output_encoder(encoder)`

Add encoder on output data.

Parameters
encoder (*profit.sur.encoder.Encoder*) –

abstract train(*X*, *y*, *fixed_sigma_n=defaults['fixed_sigma_n']*)

Trains the surrogate on input points X and model outputs y.

Depending on the surrogate, the signature can vary.

Parameters

- **X** (*ndarray*) – Input training points.
- **y** (*ndarray*) – Observed output data.
- **fixed_sigma_n** (*bool*) – Whether the noise σ_n is fixed during optimization.

pre_train(*X*, *y*)

Check the training data

Parameters

- **X** – (n, d) or (n,) array of input training data.
- **y** – (n, D) or (n,) array of training output.

post_train()

abstract predict(*Xpred*, *add_data_variance=True*)

Predicts model output y for input Xpred based on surrogate.

Parameters

- **Xpred** (*ndarray/list*) – Input points for prediction.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance. This is especially useful for plotting.

Returns

a tuple containing:

- ymean (ndarray) Predicted output values at the test input points.
- yvar (ndarray): Generally the uncertainty of the fit. For Gaussian Processes this is

the diagonal of the posterior covariance matrix.

Return type
tuple

pre_predict(*Xpred*)

Prepares the surrogate for prediction by checking if it is trained and validating the data.

Parameters
Xpred (*ndarray*) – (n, d) or (n,) array of input points for prediction

Returns
Checked input data or default values inferred from training data.

Return type
ndarray

abstract save_model(*path*)

Saves the surrogate to a file. The file format can vary between surrogates. As default, the surrogate is saved to 'base_dir/model_{surrogate_label}.hdf5'.

Parameters
path (*str*) – Path including the file name, where the model should be saved.

abstract classmethod load_model(*path*)

Loads a saved surrogate from a file. The file format can vary between surrogates.

Identifies the surrogate by its class label in the file name.

Parameters
path (*str*) – Path including the file name, from where the model should be loaded.

Returns
Instantiated surrogate model.

Return type
profit.sur.Surrogate

abstract classmethod from_config(*config*, *base_config*)

Instantiates a surrogate based on the parameters given in the configuration file and delegates to child.

Parameters

- **config** (*dict*) – Only the 'fit' part of the base_config.
- **base_config** (*dict*) – The whole configuration parameters.

plot(*Xpred=None*, *independent=None*, *show=False*, *ref=None*, *add_data_variance=True*, *axes=None*)

Simple plotting for dimensions <= 2.

Fore more sophisticated plots use the command 'profit ui'.

Parameters

- **Xpred** (*ndarray*) – Prediction points where the fit is plotted. If None, it is inferred from the training points.
- **independent** (*dict*) – Dictionary of independent variables from config.
- **show** (*bool*) – If the figure should be shown directly.
- **ref** (*ndarray*) – Reference function which is fitted.
- **add_data_variance** (*bool*) – Adds the data noise σ_n^2 to the prediction variance.
- **axes** (*matplotlib.pyplot.axes*) – Axes object to insert the plot into. If None, a new figure is created.

default_Xpred()

Infer prediction values from training points in each dimension.

Currently a dense grid is created. This becomes inefficient for > 3 dimensions.

Returns
Prediction points.

Return type
ndarray

profit.ui

Submodules

profit.ui.app

Module Contents

Functions

init_app(config)	

profit.ui.app.**init_app**(*config*)

profit.ui.condhist

Module Contents

Functions

generate_table()	
gen_callback(k)	

Attributes

df	
data	
param	
external_stylesheets	
app	

profit.ui.condhist.**df**

profit.ui.condhist.**data**

profit.ui.condhist.**param**

profit.ui.condhist.**generate_table**()

profit.ui.condhist.**external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']**

profit.ui.condhist.**app**

profit.ui.condhist.**gen_callback**(*k*)

profit.ui.hist_utils

Module Contents

Functions

dens_hist(a, edges, outside)	
format_hist(fig, bins, edges, density, title, outside, ...)	
draw_hist(edges, density)	
fig_hist(da, bins, title[, condi, outside, colors])	
add_border(fig)	

profit.ui.hist_utils.**dens_hist**(*a*, *edges*, *outside*)

profit.ui.hist_utils.**format_hist**(*fig*, *bins*, *edges*, *density*, *title*, *outside*, *colors*)

profit.ui.hist_utils.**draw_hist**(*edges*, *density*)

profit.ui.hist_utils.**fig_hist**(*da*, *bins*, *title*, *condi=None*, *outside=(False, True)*, *colors=('#8ecbad', '#b3ffd9')*)

profit.ui.hist_utils.**add_border**(*fig*)

Package Contents

Functions

init_app(config)	

profit.ui.**init_app**(*config*)

profit.util

Submodules

profit.util.base_class

Module Contents

Classes

CustomABC	Helper class that provides a standard way to create an ABC using

class profit.util.base_class.**CustomABC**

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

labels

classmethod get_label()

Returns the string label of a class object.

classmethod register(*label*)

Decorator to register new classes.

classmethod __class_getitem__(*item*)

Returns the child.

profit.util.component

Component Base Class

abstract base class to register subclasses

Module Contents

Classes

Component	Helper class that provides a standard way to create an ABC using

class profit.util.component.**Component**

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

_components

component = 'Component'

classmethod __init_subclass__(*label=None*)

This method is called when a class is subclassed.

Register a new (sub-)component

classmethod register(*label*)

Decorator to register new subcomponents.

This will allow access to the subcomponent via `Component[label]`. Internally the subcomponents are stored in `_components`. Warning: this is totally unrelated to ABC.register Not necessary when a Component is subclassed, __init_subclass__ is used instead

classmethod __class_getitem__(*label*)

Returns the subcomponent.

profit.util.file_handler

Module Contents

Classes

FileHandler	Helper class that provides a standard way to create an ABC using
TxtHandler	Helper class that provides a standard way to create an ABC using
HDF5Handler	Helper class that provides a standard way to create an ABC using
PickleHandler	Helper class that provides a standard way to create an ABC using

class profit.util.file_handler.**FileHandler**

Bases: *profit.util.base_class.CustomABC*

Helper class that provides a standard way to create an ABC using inheritance.

labels

associated_types

classmethod save(*filename*, *data*, ***kwargs*)

Parameters

- **filename** (*str*) –
- **data** (*ndarray, dict*) –
- **kwargs** – Options like header and format for specific child classes.

classmethod load(*filename*, *as_type='dtype'*)

Parameters

- **filename** (*str*) –

- **as_type** (*str*) – Identifier in which format the data should be returned. Options: dtype (structured array), dict

class profit.util.file_handler.**TxtHandler**

Bases: *FileHandler*

Helper class that provides a standard way to create an ABC using inheritance.

classmethod save(*filename*, *data*, *header=None*, *fmt=None*)

Parameters

- **filename** (*str*) –
- **data** (*ndarray, dict*) –
- **kwargs** – Options like header and format for specific child classes.

classmethod load(*filename*, *as_type='dtype'*)

Parameters

- **filename** (*str*) –
- **as_type** (*str*) – Identifier in which format the data should be returned. Options: dtype (structured array), dict

class profit.util.file_handler.**HDF5Handler**

Bases: *FileHandler*

Helper class that provides a standard way to create an ABC using inheritance.

classmethod save(*filename*, *data*, ***kwargs*)

Parameters

- **filename** (*str*) –
- **data** (*ndarray, dict*) –
- **kwargs** – Options like header and format for specific child classes.

classmethod load(*filename*, *as_type='dtype'*)

Parameters

- **filename** (*str*) –
- **as_type** (*str*) – Identifier in which format the data should be returned. Options: dtype (structured array), dict

classmethod _recursive_dict2hdf(*file*, *path*, *_dict*)

static hdf2numpy(*dataset*)

static hdf2dict(*dataset*)

class profit.util.file_handler.**PickleHandler**

Bases: *FileHandler*

Helper class that provides a standard way to create an ABC using inheritance.

classmethod save(*filename*, *data*, ***kwargs*)

Parameters

- **filename** (*str*) –
- **data** (*ndarray, dict*) –
- **kwargs** – Options like header and format for specific child classes.

classmethod load(*filename*, *as_type='raw'*, *read_method='rb'*)

Parameters

- **filename** (*str*) –
- **as_type** (*str*) – Identifier in which format the data should be returned. Options: dtype (structured array), dict

profit.util.halton

Halton low discrepancy sequence.

This snippet implements the Halton sequence following the generalization of a sequence of *Van der Corput* in n-dimensions.

Module Contents

Functions

primes_from_2_to(n)	Prime number from 2 to n.
van_der_corput(n_sample[, base])	Van der Corput sequence.
halton(n_sample, dim)	Halton sequence.

profit.util.halton.**primes_from_2_to**(*n*)

Prime number from 2 to n.

From StackOverflow.

Parameters
n (*int*) – sup bound with `n >= 6`.

Returns
primes in 2 <= p < n.

Return type
list

profit.util.halton.**van_der_corput**(*n_sample*, *base=2*)

Van der Corput sequence.

Parameters

- **n_sample** (*int*) – number of element of the sequence.
- **base** (*int*) – base of the sequence.

Returns
sequence of Van der Corput.

Return type
list (n_samples,)

profit.util.halton.**halton**(*n_sample*, *dim*)

Halton sequence.

Parameters

- **n_sample** (*int*) – number of samples.
- **dim** (*int*) – dimension

Returns
sequence of Halton.

Return type
array_like (n_samples, n_features)

profit.util.util

Utility functions.

This file contains miscellaneous useful functions.

Module Contents

Classes

SafeDict	dict() -> new empty dictionary

Functions

safe_path(arg, default[, valid_extensions])	
quasirand([ndim, npoint])	
check_ndim(arr)	
params2map(params)	
load_includes(paths)	load python modules from the specified paths
flatten_struct(struct_array)	

profit.util.util.**safe_path**(*arg*, *default*, *valid_extensions=('.yaml', '.py')*)

profit.util.util.**quasirand**(*ndim=1*, *npoint=1*)

profit.util.util.**check_ndim**(*arr*)

class profit.util.util.**SafeDict**(*obj*, *pre='{'*, *post='}'*)

Bases: dict

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:

d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**

in the keyword argument list. For example: dict(one=1, two=2)

classmethod from_params(*params*, ***kwargs*)

__missing__(*key*)

profit.util.util.**params2map**(*params: None | collections.abc.MutableMapping | numpy.ndarray | numpy.void*)

profit.util.util.**load_includes**(*paths*)

load python modules from the specified paths

profit.util.util.**flatten_struct**(*struct_array: numpy.ndarray*)

profit.util.variable

Module Contents

Classes

VariableGroup	Table of input, output and independent variables.
Variable	Base class for a single variable.
InputVariable	Sub class for input variables.
IndependentVariable	Sub class for independent variables.
ActiveLearningVariable	Sub class for active learning variables.
OutputVariable	Sub class for output variables.

Functions

halton([size])	
uniform([start, end, size])	
loguniform([start, end, size])	
normal([mu, std, size])	
linear([start, end, size])	
independent([start, end, size])	
constant([value, size])	

Attributes

EXCLUDE_FROM_HALTON	

profit.util.variable.**EXCLUDE_FROM_HALTON = ('output', 'constant', 'uniform', 'loguniform', 'normal', 'linear', 'independent')**

profit.util.variable.**halton**(*size=(1, 1)*)

profit.util.variable.**uniform**(*start=0*, *end=1*, *size=None*)

profit.util.variable.**loguniform**(*start=1e-06*, *end=1*, *size=None*)

profit.util.variable.**normal**(*mu=0*, *std=1*, *size=None*)

profit.util.variable.**linear**(*start=0*, *end=1*, *size=1*)

profit.util.variable.**independent**(*start=0*, *end=1*, *size=1*)

profit.util.variable.**constant**(*value=0*, *size=None*)

class profit.util.variable.**VariableGroup**(*samples*)

Table of input, output and independent variables.

Parameters
samples (*int*) – Samples of the training data.

samples

Samples of the training data.

Type
int

list

List of all variables in the order of the user entry.

Type
list

property all

Returns
View on all variables.

Not working yet for vector output.

property as_dict

Returns: View of all variables as a dictionary.

property input

Returns: View of the input variables only. Also excluded are independent variables.

property named_input

Returns: Ndarray with dtype of the input variables.

property input_dict

Returns: Dictionary of the input variables.

property input_list

Returns: List of input variables without independent variables.

property kind_dict

property output

Returns: View on the output variables only.

property named_output

Returns: Ndarray with dtype of the output variables.

property formatted_output

property output_dict

Returns: Dictionary of the output variables.

property output_list

Returns: List of output variables.

__getitem__(*item*)

Implements dict like behavior to get a variable by its identifier or index.

Parameters
item (*int/str*) – Index or label of variable.

Returns
Variable.

add(*variables*)

Adds a single or a list of variables to the table. If a list is added, a common n-D halton sequence is generated and the variables are transformed according to their distribution.

Parameters
variables (*Variable/list*) – Variable(s) to add.

delete_variable(*columns*)

Deletes one or more variables from the table.

Parameters
columns (*int/list*) – Columns of the table to remove.

delete_sample(*rows*)

Deletes one or more rows of the table.

Parameters
rows (*int/list*) – Rows to delete.

generate_from_halton()

Generates a common halton sequence for all variables where this is possible and transforms them according to their distribution.

class profit.util.variable.**Variable**(*name*, *kind*, *size*, *value=None*, *dtype=np.float64*)

Bases: *profit.util.base_class.CustomABC*

Base class for a single variable. To create input, independent and output variables, use the cls.create() or cls.create_from_str() methods.

name

Name of the variable.

Type
str

kind

Distribution for input variables, 'Output' or 'Independent'.

Type
str

size

Size as (nsamples, ndim).

Type
tuple

value

Data.

Type
ndarray

`dtype`

Datatype.

Type
dtype

`property named_value`

Returns: Ndarray with dtype.

`labels`

`classmethod create_from_str(name, size, v_str)`

Creates a Variable instance from a string. E.g. 'Uniform(3.4, 7.8)'

Parameters

- **name** (*str*) – Name of the variable.
- **size** (*tuple*) – Size as (nsamples, ndim).
- **v_str** (*str*) – String from which the variable is constructed.

Returns
Variable.

`classmethod create(name, kind, size, value=None, dtype=np.float64, **kwargs)`

Directly creates a variable from keyword entries.

Parameters

- **name** (*str*) – Name of the variable.
- **kind** (*str*) – Distribution of input variables, 'Output' or 'Independent'.
- **size** (*tuple*) – Size as (nsamples, ndim).
- **kwargs** (*tuple/str*) – Keyword arguments depending on the sub variables. E.g. constraints for input variables, a search distribution for active learning variables or dependent variables of outputs.
- **value** (*ndarray*) – Data.
- **dtype** (*dtype*) – Datatype.

Returns
Variable.

`as_dict()`

Dictionary of the variable attributes.

`__getitem__(item)`

Implement dict like behavior to get an attribute by using square brackets.

Paramters:
item (str)

Returns
Attribute

class profit.util.variable.**InputVariable**(*name*, *kind*, *size*, *constraints=(0, 1)*, *value=None*, *dtype=np.float64*)

Bases: *Variable*

Sub class for input variables.

generate_values(*halton_seq=None*)

classmethod parse_entries(*entries*)

create_Xpred(*size*, *spacing=None*)

creates an array of suitably spaced X-values for prediction

spacing (shape: size) can be used to override the default linear spacing

class profit.util.variable.**IndependentVariable**(*name*, *kind*, *size*, *constraints=(0, 1)*, *value=None*, *dtype=np.float64*)

Bases: *InputVariable*

Sub class for independent variables.

class profit.util.variable.**ActiveLearningVariable**(*name*, *kind*, *size*, *distr='uniform'*, *constraints=(0, 1)*, *value=None*, *dtype=np.float64*)

Bases: *InputVariable*

Sub class for active learning variables.

classmethod parse_entries(*entries*)

generate_values(*halton_seq=None*)

create_Xpred(*size*, *spacing=None*)

creates an array of suitably spaced X-values for prediction

spacing (shape: size) can be used to override the default linear spacing

class profit.util.variable.**OutputVariable**(*name*, *kind*, *size*, *dependent=()*, *value=None*, *dtype=np.float64*)

Bases: *Variable*

Sub class for output variables.

classmethod parse_entries(*entries*)

resolve_dependent(*ind*)

Create a *Variable* instance for the independent variables of vector outputs and set `dependent`.

Parameters

ind (*profit.util.variable.IndependentVariable or list[profit.util.variable.IndependentVariable]*) – Independent variables.

as_dict()

Dictionary of the variable attributes.

Package Contents

Classes

SafeDict	dict() -> new empty dictionary

Functions

safe_path(arg, default[, valid_extensions])	
quasirand([ndim, npoint])	
check_ndim(arr)	
params2map(params)	
load_includes(paths)	load python modules from the specified paths
flatten_struct(struct_array)	

profit.util.**safe_path**(*arg*, *default*, *valid_extensions=('.yaml', '.py')*)

profit.util.**quasirand**(*ndim=1*, *npoint=1*)

profit.util.**check_ndim**(*arr*)

class profit.util.**SafeDict**(*obj*, *pre='{'*, *post='}'*)

Bases: dict

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:

d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**

in the keyword argument list. For example: dict(one=1, two=2)

classmethod from_params(*params*, ***kwargs*)

__missing__(*key*)

profit.util.**params2map**(*params: None | collections.abc.MutableMapping | numpy.ndarray | numpy.void*)

profit.util.**load_includes**(*paths*)

load python modules from the specified paths

profit.util.**flatten_struct**(*struct_array: numpy.ndarray*)

8.1.2 Submodules

profit.config

Module Contents

Classes

AbstractConfig	General class with methods which are useful for all Config classes.
BaseConfig	This class and its modular subclasses provide all possible configuration parameters.
RunConfig	Run configuration with the following sub classes:
FitConfig	Configuration for the surrogate and encoder. Currently, the only sub config is for the GaussianProcess classes.
ALConfig	Active learning configuration.
AlgorithmALConfig	General class with methods which are useful for all Config classes.
SimpleALConfig	General class with methods which are useful for all Config classes.
McmcConfig	General class with methods which are useful for all Config classes.
AcquisitionFunctionConfig	Acquisition function configuration.
SimpleExplorationConfig	Acquisition function configuration.
ExplorationWithDistancePenaltyConfig	Acquisition function configuration.
WeightedExplorationConfig	Acquisition function configuration.
ProbabilityOfImprovementConfig	Acquisition function configuration.
ExpectedImprovementConfig	Acquisition function configuration.
ExpectedImprovement2Config	Acquisition function configuration.
AlternatingExplorationConfig	Acquisition function configuration.
UIConfig	Configuration for the Graphical User Interface.
DefaultConfig	Default config for all run sub configs which just updates the attributes with user entries.

Functions

represent_ordereddict(dumper, data)	
dict_constructor(loader, node)	
load_config_from_py(filename)	Load the configuration parameters from a python file into dict.

Attributes

VALID_FORMATS	yaml has to be configured to represent OrderedDict
_mapping_tag	

profit.config.**VALID_FORMATS = ('.yaml', '.py')**

yaml has to be configured to represent OrderedDict see https://stackoverflow.com/questions/16782112/can-pyyaml-dump-dict-items-in-non-alphabetical-order and https://stackoverflow.com/questions/5121931/in-python-how-can-you-load-yaml-mappings-as-ordereddicts

profit.config.**represent_ordereddict**(*dumper*, *data*)

profit.config.**dict_constructor**(*loader*, *node*)

profit.config.**_mapping_tag**

profit.config.**load_config_from_py**(*filename*)

Load the configuration parameters from a python file into dict.

class profit.config.**AbstractConfig**(***entries*)

Bases: *profit.util.base_class.CustomABC*

General class with methods which are useful for all Config classes.

labels

defaults

update(***entries*)

Updates the attributes with user inputs. A warning is issued if the attribute set by the user is unknown.

Parameters
entries (*dict*) – User input of the config parameters.

process_entries(*base_config*)

After the attributes are set, they are formatted and edited to standardize the user inputs.

Parameters
base_config (BaseConfig) – In sub configs, the data from the base config is needed.

set_defaults(*default_dict*)

Default values are set from a default dictionary, which is usually located in the global profit.defaults file.

create_subconfig(*sub_config_label*, ***entries*)

Instances of sub configs are created from a string or a dictionary.

Parameters

- **sub_config_label** (*str*) – Dict key of registered sub config.
- **entries** (*dict*) – User input parameters.

__getitem__(*item*)

Implements the dictionary like get method with brackets.

Parameters
item (*str*) – Label of the attribute to return.

Returns
Attribute or if the attribute is a sub config, a dictionary of the sub config items.

`items()`

Implements the dictionary like self.items() method.

Returns
List of (key, value) tuples of the class attributes.

Return type
list

`get(item, default=None)`

Implements the dictionary like get method with a default value.

Parameters

- **item** (*str*) – Label of the attribute to return.
- **default** – Default value, if the attribute is not found.

Returns
Attribute or the default value.

`class profit.config.BaseConfig(base_dir=defaults.base_dir, **entries)`

Bases: *AbstractConfig*

This class and its modular subclasses provide all possible configuration parameters.

Parts of the Config:

- base_dir
- run_dir
- config_file
- include
- ntrain
- variables
- **files**
  - input
  - output
- **run**
  - runner
  - interface
  - pre
  - post
- **fit**
  - surrogate
  - save / load
  - fixed_sigma_n
- active_learning

- ui

Base configuration for fundamental parameters.

Parameters

- **base_dir** (*str*) – Base directory.
- **run_dir** (*str*) – Run directory.
- **config_path** (*str*) – Path to configuration file.
- **include** (*list*) – Paths to custom files which are loaded in the beginning.
- **files** (*dict*) – Paths for input and output files.
- **ntrain** (*int*) – Number of training samples.
- **variables** (*dict*) – All variables.
- **input** (*dict*) – Input variables.
- **output** (*dict*) – Output variables.
- **independent** (*dict*) – Independent variables, if the result of the simulation is a vector.

labels

process_entries()

Sets absolute paths, creates variables and delegates to the sub configs.

classmethod from_file(*filename=defaults.config_file*)

Creates a configuration class from a .yaml or .py file.

load_includes()

class profit.config.**RunConfig**(***entries*)

Bases: *AbstractConfig*

Run configuration with the following sub classes:

- **runner**
  - local
  - slurm
- **interface**
  - memmap
  - zeromq
- **pre**
  - template
- **post**
  - json
  - numpytxt
  - hdf5

A default sub class which just updates the entries from a user input is also implemented and used if the class from the user input is not found.

Custom config classes can also be registered, e.g. as a custom runner:

```
@RunnerConfig.register("custom")
class CustomRunner(LocalRunnerConfig):
    def process_entries(self, base_config):
        # do something else than the usual LocalRunnerConfig
        pass
```

Default values from the global profit.defaults.py file are loaded.

labels

defaults = 'run'

update(***entries*)

Updates the attributes with user inputs. No warning is issued if the attribute set by the user is unknown.

Parameters
entries (*dict*) – User input of the config parameters.

class profit.config.**FitConfig**(***entries*)

Bases: *AbstractConfig*

Configuration for the surrogate and encoder. Currently, the only sub config is for the GaussianProcess classes.

labels

defaults = 'fit'

update(***entries*)

Updates the attributes with user inputs. A warning is issued if the attribute set by the user is unknown.

Parameters
entries (*dict*) – User input of the config parameters.

process_entries(*base_config*)

Set 'load' and 'save' as well as the encoder.

class profit.config.**ALConfig**(***entries*)

Bases: *AbstractConfig*

Active learning configuration.

labels

defaults = 'active_learning'

process_entries(*base_config*)

After the attributes are set, they are formatted and edited to standardize the user inputs.

Parameters
base_config (BaseConfig) – In sub configs, the data from the base config is needed.

class profit.config.**AlgorithmALConfig**(***entries*)

Bases: *AbstractConfig*

General class with methods which are useful for all Config classes.

labels

defaults

class profit.config.**SimpleALConfig**(***entries*)

Bases: *AlgorithmALConfig*

General class with methods which are useful for all Config classes.

labels

defaults = 'al_algorithm_simple'

process_entries(*base_config*)

After the attributes are set, they are formatted and edited to standardize the user inputs.

Parameters
base_config (BaseConfig) – In sub configs, the data from the base config is needed.

class profit.config.**McmcConfig**(***entries*)

Bases: *AlgorithmALConfig*

General class with methods which are useful for all Config classes.

labels

defaults = 'al_algorithm_mcmc'

process_entries(*base_config*)

After the attributes are set, they are formatted and edited to standardize the user inputs.

Parameters
base_config (BaseConfig) – In sub configs, the data from the base config is needed.

class profit.config.**AcquisitionFunctionConfig**(***entries*)

Bases: *AbstractConfig*

Acquisition function configuration.

labels

defaults

process_entries(*base_config*)

After the attributes are set, they are formatted and edited to standardize the user inputs.

Parameters
base_config (BaseConfig) – In sub configs, the data from the base config is needed.

class profit.config.**SimpleExplorationConfig**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_simple_exploration'

class profit.config.**ExplorationWithDistancePenaltyConfig**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_exploration_with_distance_penalty'

class profit.config.**WeightedExplorationConfig**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_weighted_exploration'

class profit.config.**ProbabilityOfImprovementConfig**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_probability_of_improvement'

class profit.config.**ExpectedImprovementConfig**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_expected_improvement'

class profit.config.**ExpectedImprovement2Config**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_expected_improvement_2'

class profit.config.**AlternatingExplorationConfig**(***entries*)

Bases: *AcquisitionFunctionConfig*

Acquisition function configuration.

labels

defaults = 'al_acquisition_function_alternating_exploration'

class profit.config.**UIConfig**(***entries*)

Bases: *AbstractConfig*

Configuration for the Graphical User Interface.

labels

defaults = 'ui'

class profit.config.**DefaultConfig**(***entries*)

Bases: *AbstractConfig*

Default config for all run sub configs which just updates the attributes with user entries.

labels

defaults

update(***entries*)

Updates the attributes with user inputs. A warning is issued if the attribute set by the user is unknown.

Parameters
entries (*dict*) – User input of the config parameters.

profit.defaults

Global default configuration values.

Module Contents

profit.defaults.**base_dir**

profit.defaults.**run_dir**

profit.defaults.**config_file = 'profit.yaml'**

profit.defaults.**include = []**

profit.defaults.**files**

profit.defaults.**ntrain = 10**

profit.defaults.**variables**

profit.defaults.**run**

profit.defaults.**fit**

profit.defaults.**fit_gaussian_process**

profit.defaults.**fit_linear_regression**

profit.defaults.**active_learning**

profit.defaults.**al_algorithm_simple**

profit.defaults.**al_algorithm_mcmc**

profit.defaults.**al_acquisition_function_simple_exploration**

profit.defaults.**al_acquisition_function_exploration_with_distance_penalty**

profit.defaults.**al_acquisition_function_weighted_exploration**

profit.defaults.**al_acquisition_function_probability_of_improvement**

profit.defaults.**al_acquisition_function_expected_improvement**

profit.defaults.**al_acquisition_function_expected_improvement_2**

profit.defaults.**al_acquisition_function_alternating_exploration**

profit.defaults.**ui**

profit.main

proFit main script.

This script is called when running the *profit* command inside a shell.

Module Contents

Functions

main()	Main command line interface

profit.main.**main**()

Main command line interface

8.1.3 Package Contents

profit.**__version__**

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[proFit] C. Albert, M. Kendler, R. Babin, M. Hadwiger, R. Hofmeister, M. Khallaayoune, F. Kramp, K. Rath, & B. Rubino-Moyner, "proFit: Probabilistic Response Model Fitting with Interactive Tools", 10.5281/zenodo.6624446 (2022)

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z