
pyepal

Release 0.7.0-dev

Kevin Maik Jablonka, Brian Yoo, Berend Smit

May 12, 2022

CONTENTS

1	Contents	3
1.1	Getting Started	3
1.1.1	Installation	3
1.1.2	Which class do I use?	3
1.1.3	Running an active learning experiment	4
1.1.4	Caveats and tricks with Gaussian processes	7
1.2	Background	9
1.2.1	The intuition behind the algorithm	10
1.2.2	How do the hyperparameters influence the algorithm?	10
1.3	Tutorials	12
1.3.1	1. One active learning step using GPR models built with GPy	12
1.3.2	2. Active learning with “measure” function and sklearn models	13
1.3.3	3. Quantile regression	13
1.3.4	4. Plotting a learning curve	14
1.4	The PyePAL API reference	14
1.4.1	The PAL package	14
1.4.2	The models package	28
1.5	Developer notes	29
1.5.1	Contribution Guidelines	29
1.5.2	Implementing a new PAL class	30
2	Indices and tables	33
	Python Module Index	35
	Index	37

This package implements an active learning approach to efficiently and confidently identify the Pareto front with any regression model that can output a mean and a standard deviation.

It works with any number of objectives, missing data, and is highly customizable.

If you find this code useful for your work, please cite:

- Jablonka, K. M.; Giriprasad, M. J.; Wang, S.; Smit, B.; Yoo, B. Bias Free Multiobjective Active Learning for Materials Design and Discovery, ChemRxiv 2020 (10.26434/chemrxiv.13200197.v1).
- Zuluaga, M.; Krause, A.; Püschel, M. E-PAL: An Active Learning Approach to the Multi-Objective Optimization Problem. Journal of Machine Learning Research 2016, 17 (104), 1–32.

CONTENTS

1.1 Getting Started

1.1.1 Installation

We recommend installing PyePAL in a dedicated [virtual environment](#) or [conda environment](#). Note that we currently support Python 3.7 and 3.8.

To install the latest stable release use

```
pip install pyepal
```

or the conda channel (recommended)

```
conda install pyepal -c conda-forge
```

The latest version of PyePAL can be installed from GitHub using

```
pip install git+https://github.com/kjappelbaum/pyepal.git
```

On macOS you might need to install *libomp* (e.g., *brew install libomp*) for multithreading in some models.

If you want to [limit how many CPUs openblas uses](#), you can `export OPENBLAS_NUM_THREADS=1`

1.1.2 Which class do I use?

- For Gaussian processes built with `sklearn` use [PALSklearn](#)
- For Gaussian processes built with `GPy` use [PALGPy](#)
- For coregionalized Gaussian processes (built with `GPy`) use [PALCoregionalized](#)
- For quantile regression using `LightGBM` gradient boosted decision trees use [PALGBDT](#)
- For infinite wide neural networks with the neural tangent kernel or exact Bayesian inference (Novak et al., 2019) use [PALNT](#)
- For an ensemble of finite width neural networks (Lakshminarayanan et al., 2017) (built with `JAX`) use [PALNTensemble](#)

If your favorite model is not listed, you can easily implement it yourself (see [Implementing a new PAL class](#))!

1.1.3 Running an active learning experiment

The *examples* directory contains a [Jupyter notebook](#) with an [example](#) that can also be run on MyBinder.

If using a Gaussian process model built with `sklearn` or `GPY` we recommend using a pre-built class such as [PALSklearn](#), [PALCoregionalized](#), [PALGPY](#) and following the subsequent steps (for more details on which class to use see [Which class do I use?](#)):

1. For each objective create a model (if using a coregionalized Gaussian process model, only one model needs to be created)
2. Sample a few initial points from the design space. We provide the [get_maxmin_samples\(\)](#) or [get_kmeans_samples\(\)](#) utilities that can help with the sampling. Our code assumes that `X` is a `np.array`.

```
from pyepal import get_kmeans_samples, get_maxmin_samples

# This selects the 10 points closest to the centroids of a k=10 means_
↳ clustering
indices = get_kmeans_samples(X, 10)

# This selects the 10 farthest points in feature space
indices = get_maxmin_samples(X, 10)
```

3. Now we can initialize the instance of one PAL class. If using a `sklearn` Gaussian process model, we would use

```
from pyepal import PALSklearn

# Each of these models is an instance of sklearn.gaussian_process.
↳ GaussianProcessRegressor
models = [gpr0, gpr1, gpr2]

# We always need to provide the feature matrix (X), a list of models, and_
↳ the number of objectives
palinstance = PALSklearn(X, models, 3)

# Now, we can also feed in the first measurements
# this here assumes that we have all measurements for y and we now
# provide those which are present in the indices array
palinstance.update_train_set(indices, y[indices])

# Now we can run one step
next_idx = palinstance.run_one_step()
```

At this level, we have a range of different optional arguments we can set.

- `epsilon`: one ϵ per dimension in a `np.ndarray`. This can be used to set different tolerances for each objective. Note that $\epsilon_i \in [0, 1]$.
- `delta`: the δ hyperparameter ($\delta \in [0, 1]$). Increasing this value will speed up the convergence.
- `beta_scale`: an empirical scaling parameter for β . The theoretical guarantees in the PAL paper are derived for this parameter set to 1. But in practice, a much faster convergence can be achieved by setting it to a number $0 < \beta_{\text{scale}} \ll 1$.
- `goal`: By default, PyePAL assumes that the goal is to maximize every objective. If this is not the case, this argument can be set using a list of “min” and “max” strings, with “min” specifying whether to minimize the *i*th objective and “max” indicating whether to maximize this objective.

- `coef_var_threshold`: By default, PyePAL will not consider points with a coefficient of variation ≥ 3 for the classification step of the algorithm. This is meant to avoid classifying design points for which the model is entirely unsure. This tends to happen when a model is severely overfit on the training data (i.e., the training data uncertainties are very low, whereas the prediction uncertainties are very high). To change this setting, reduce this value to make the check tighter or increase it to avoid this check (as in the original implementation).

In the case of missing observations, i.e., only two of three outputs are measured, report the missing observations as `np.nan`. The call could look like

```
import numpy as np

palinstance.update_train_set(np.array([1,2]), np.array([[1, 2, 3], [np.nan, 1, 2, 0]]))
```

for a case in which we performed measurements for samples with index 1 and 2 of our design space, but did not measure the first target for sample 2.

Hyperparameter optimization

Usually, the hyperparameters of a machine learning model, in particular the kernel hyperparameters of a Gaussian process regression model, should be optimized as new training data is added. However, since this is usually a computationally expensive process, it may not be desirable to perform this at every iteration of the active learning process. The iteration frequency of the hyperparameter optimization is internally set by the `_should_optimize_hyperparameters` function, which by default uses a schedule that optimizes the hyperparameter every 10th iteration. This behavior can be changed by overriding this function.

Reclassification schedule

A problem of the e-PAL algorithm can be the case where the initial GPR predictions are wrong. This might lead to points that are actually Pareto-efficient being confidently discarded (in case the GPR predicts with low variance a performance that is dominated by some other point). Under the assumption that the GPR predictions improve over the course of the use of the algorithm, this behavior can be mitigated by “resetting” all the classification, i.e. reconsidering discarded points. In PyePAL, we automatically do this in case there is only one point left. In general, you might want to customize this behavior, which you can do, for example, by [monkey patching](#)

```
from pyepal.pal.schedules import linear

def my_schedule(self):
    return linear(self.iteration, 1)

PALGPyReclassify._should_reclassify = my_schedule
```

or by subclassing

```
from pyepal.pal.schedules import linear

class PALGPyReclassify(PALGPy):
    def _should_reclassify(self):
        return linear(self.iteration, 1)
```

In the examples above the full design space would be re-classified each iteration.

Logging

Basic information such as the current iteration and the classification status are logged and can be viewed by printing the PAL object

```
print(palinstance)

# returns: pyepal at iteration 1. 10 Pareto optimal points, 1304 discarded points, 200
↪ unclassified points.
```

We also provide calculation of the hypervolume enclosed by the Pareto front with the function `get_hypervolume()`

```
hv = get_hypervolume(palinstance.means[palinstance.pareto_optimal])
```

Properties of the PAL object

For debugging there are some properties and attributes of the *PAL* class that can be used to inspect the progress of the active learning loop.

- **get the points in the design space, \mathbf{x} :**
 - `design_space` returns the full design space matrix
 - `pareto_optimal_points`: returns the points that are classified as Pareto-efficient
 - `sampled_points`: returns the points that have been sampled
 - `discarded_points`: returns the points that have been discarded
- get the indices of Pareto efficient, sampled, discarded, and unclassified points with `pareto_optimal_indices`, `sampled_indices`, `discarded_indices`, and `unclassified_indices`
- similarly, the number of points in the different classes can be obtained using `number_pareto_optimal_points`, `number_discarded_points`, `number_unclassified_points`, and `number_sampled_points`. The total number of design points can be obtained with `number_design_points`.
- `hyperrectangle_size` returns the sizes of the hyperrectangles, i.e., the weights that are used in the sampling step
- `means` and `std` contain the predictions of the model
- `sampled` is a mask array. In case one objective has not been measured its cell is `False`

Exploring a space where all objectives are known

In some cases, we may already possess all measurements, but would like to run PAL with different settings to test how the algorithm performs. In this case, we provide the `exhaust_loop()` wrapper.

```
from pyepal import PALSklearn, exhaust_loop
models = [gpr0, gpr1, gpr2]
palinstance = PALSklearn(X, models, 3)

exhaust_loop(palinstance, y)
```

This will continue calling `run_one_step()` until there is no unclassified sample left.

Batch sampling

By default, the `run_one_step` function of the PAL classes will return a `np.ndarray` with only one index for the point in the design space for which the next experiment should be performed. In some situations, it may be more practical to run multiple experiments as batches before running the next active learning iteration. In such cases, we provide the `batch_size` argument which can be set to an integer greater than one.

```
next_idx = palinstance.run_one_step(batch_size=10)
# next_idx will be a np.array of length 10
```

Note that the `exhaust_loop` also supports the `batch_size` keyword argument

```
palinstance = PALSklearn(X, models, 3)

# sample always 10 points and do this until there is no unclassified
# point left
exhaust_loop(palinstance, y, batch_size=10)
```

Adding new points to the design matrix

In some applications, you might want to augment the design matrix after a few iterations of PyePAL. This could be useful, for example, if you start with a coarse discretization of your design space then want to refine this grid in subsequent iterations in the relevant regions of the design space.

Adding new points to the design matrix can be easily achieved using the `augment_design_matrix()` function that takes the new design matrix as input. By default, it will run the current model for the new, augmented, design matrix, and re-classify all points. You can turn this behavior off using the `clean_classify` parameter.

Alternatively, you can use the `classify` flag that keeps all previous classifications. This means that if there is a point that was previously Pareto-efficient in the non-augmented design space but is now dominated by a new design point, it will no longer certainly be classified as Pareto-efficient.

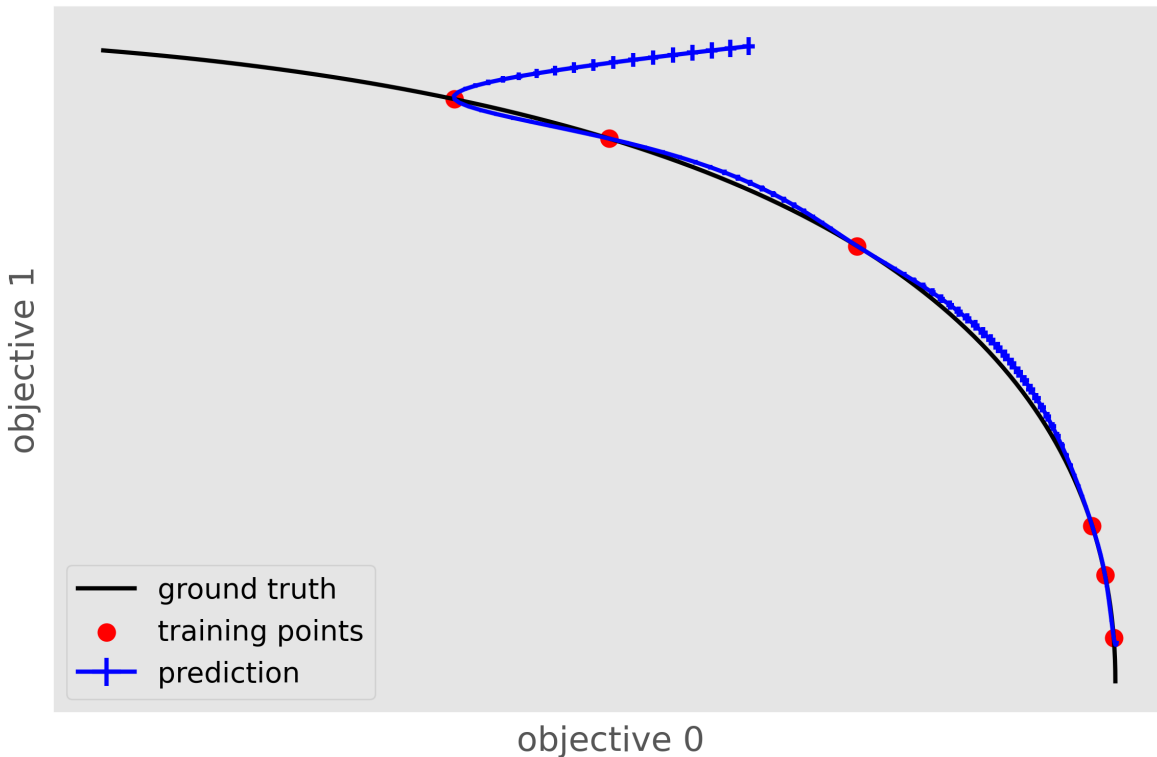
Note that is important that the new points are sampled from the same distribution as the previous points in the design space. Otherwise, the model will have to deal with unexpected data shift.

Plotting a learning curve

To plot a learning curve one can use the `plot_learning_curve()` function. This can be useful to estimate if the number of initial points is reasonable and to see if the model is predictive.

1.1.4 Caveats and tricks with Gaussian processes

One caveat to keep in mind is that ϵ -PAL will not work if the predictive variance does not make sense, for example, when the model is overconfident and the uncertainties for the training set is significantly lower than those for the predicted set. In this case, PyePAL will untimely, and often incorrectly, label the design points. An example situation where the predictions for an overconfident model due to a training set that excludes a part of design space is shown in the figure below



This problem is exacerbated in conjunction with $\beta_{\text{scale}} < 1$. To make the model more robust we suggest trying:

- to set reasonable bounds on the length scale parameters
- to increase the regularization parameter/noise kernel (`alpha` in `sklearn`)
- to increase the number of data points, especially the coverage of the design space
- to use a kernel that suits the problem
- to turn off ARD. Automatic relevance determination (ARD) might increase the predictive performance, but also makes the model more prone to overfitting

We also recommend cross-validating the Gaussian process models and checking that the predicted variances make sense. When performing cross-validation, make sure that the index provided to PyePAL is the same size as the cross-validation folds. By default, the code will run a simple cross-validation only on the first iteration and provide a warning if the mean absolute error is above the mean standard deviation. The warning will look something like

The mean absolute error **in** cross-validation **is** 64.29, the mean variance **is** 0.36.
 Your model might **not** be predictive **and/or** overconfident.
 In the docs, you find hints on how to make GPRs more robust.

This behavior can be changed with the cross-validation test being performed more frequently by overriding the `should_run_crossvalidation` function.

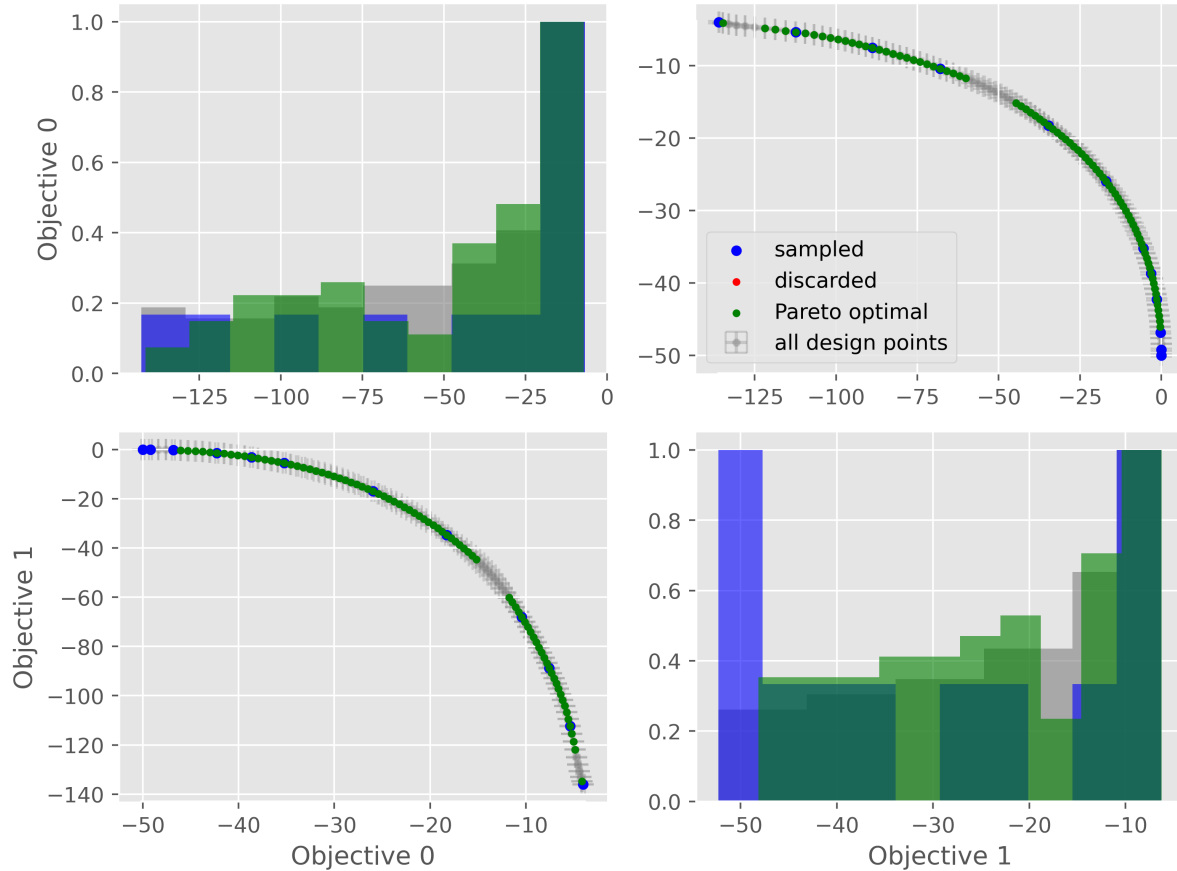
Another way to detect overfitting is to use `plot_jointplot()` function from the plotting subpackage. This function will plot all objectives against each other (with errorbars and different classes indicated with colors) and histograms of the objectives on the diagonal. If the majority of predicted points tend to overlap one another and get discarded by PyePAL, this may suggest that the surrogate model is overfitted.

```
from pyepal.plotting import plot_jointplot
```

(continues on next page)

(continued from previous page)

```
# palinstance is a instance of a PAL class after
# calling run_one_step
fig = plot_jointplot(palinstance.means, palinstance)
```



1.2 Background

This package implements a modified version of the [-PAL algorithm from Zuluaga et al.](#) in an object-oriented design. The algorithm efficiently searches for the Pareto efficient points in an unbiased manner for any number of dimensions. This package can be used with any regression model that can output means and standard deviations.

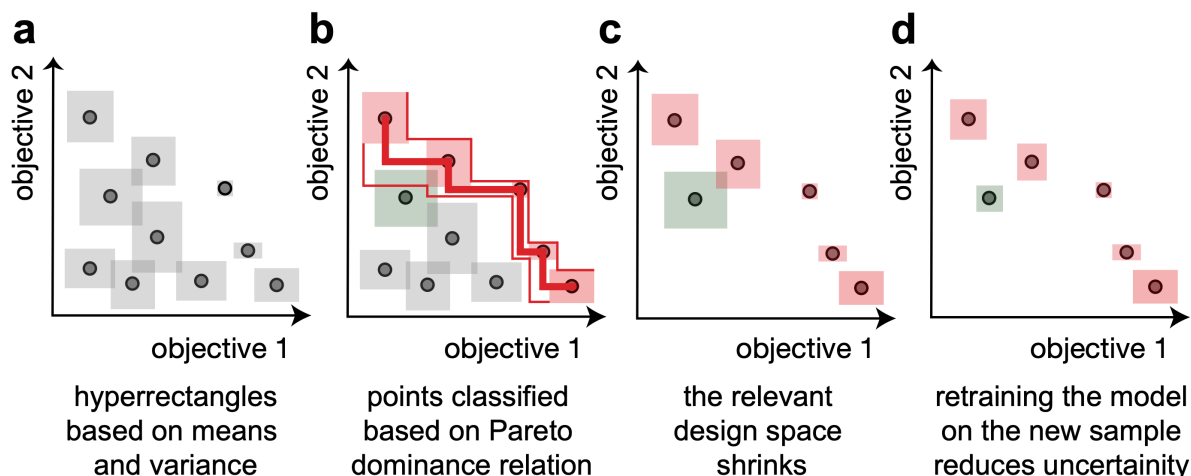
This implementation has the following features:

- We ensure that the sampling is scale-invariant and that the algorithm can deal with positive and negative objective values.
- In contrast to the [original implementation by Zuluaga et al.](#) we do not assume that the range of the objectives is known a priori. In their implementation it is used to calculate fixed tolerance values $\epsilon_i \cdot r_i$ (where r_i is the range of objective i). We instead use by default $\epsilon_i \cdot |\mu_i|$.
- Instead of using the predicted $\hat{\mu}$ and $\hat{\sigma}$ also for the sampled points we use the measured μ and σ .
- This implementation is directly scalable to n -dimensional problems.
- It can be easily used with any kind of regression model with uncertainty measures. For example, one can replace the Gaussian process model with a neural network with [Dropout Monte Carlo](#) for the uncertainty estimate.

- The support for missing data is implemented. For example, if you measure for labeled data for only some of the objectives, you will simply need to provide `np.nan` for the missing measurements. The code will automatically estimate these measurements. If using coregionalized GPR models, the models will try to utilize the correlations between the objectives to improve these predictions.

In our own work, we used this algorithm for materials discovery applications.

1.2.1 The intuition behind the algorithm



The PAL algorithm iterates through the following steps:

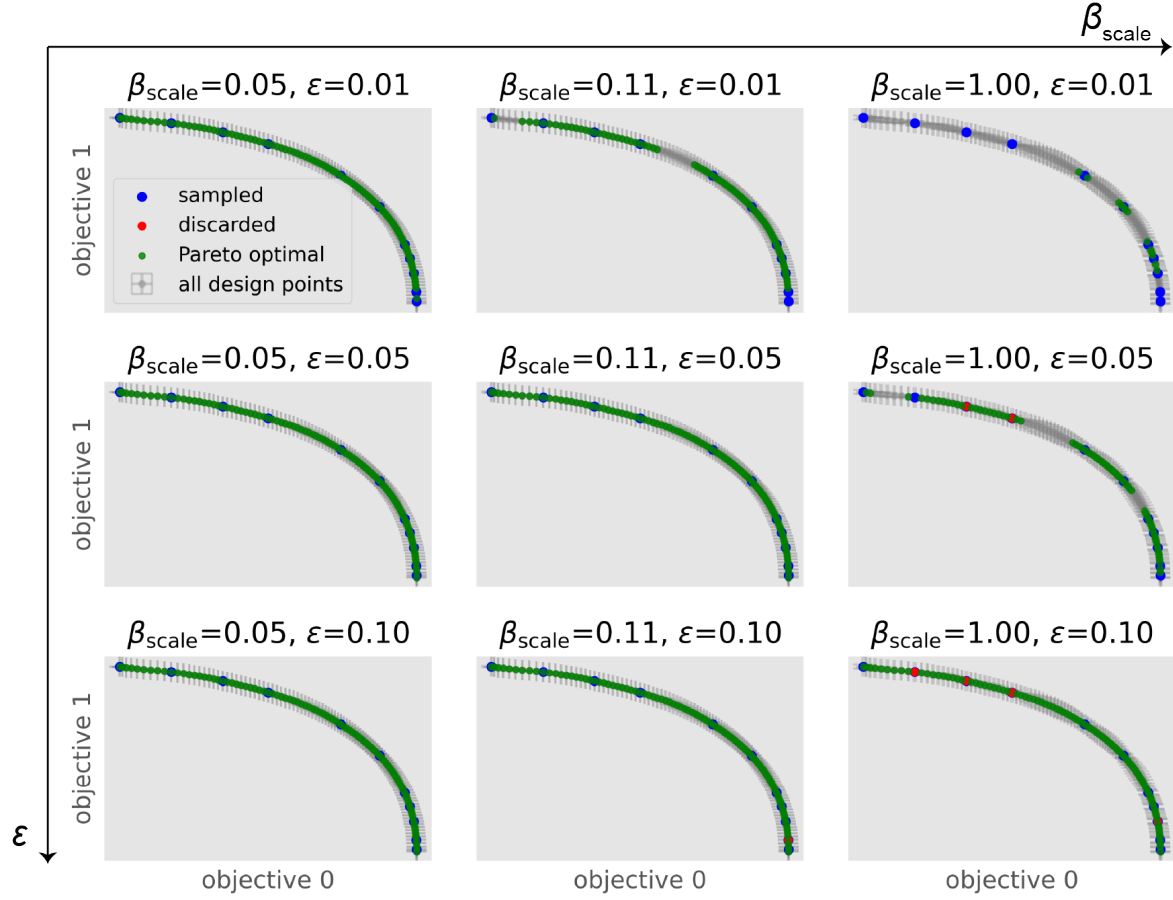
- Training a machine learning model to predict means and standard deviations for all points of the design space. This can be used to construct uncertainty hyperrectangles.
- Using these points, we can use the Pareto dominance relation to classify points as Pareto optimal or to discard them. In some cases, e.g., when uncertainty hyperrectangles overlap, we will not be able to perform a classification with confidence. This is different from many Bayesian optimization approaches that couples an acquisition function such as expected improvement, which introduces a total order in the design space and hence biases the search.
- Since in step (b) we discard many points (with confidence) the effective design space shrinks. This enables us to sample the next experiment from the regions near the Pareto front with points labeled as Pareto optimal and “unclassified”. Since the ultimate goal is to perform a classification of the full design space—with confidence—we sample the point with the largest hyperrectangle.
- Retraining a model with this new measurement will reduce the width of the hyperrectangles.

1.2.2 How do the hyperparameters influence the algorithm?

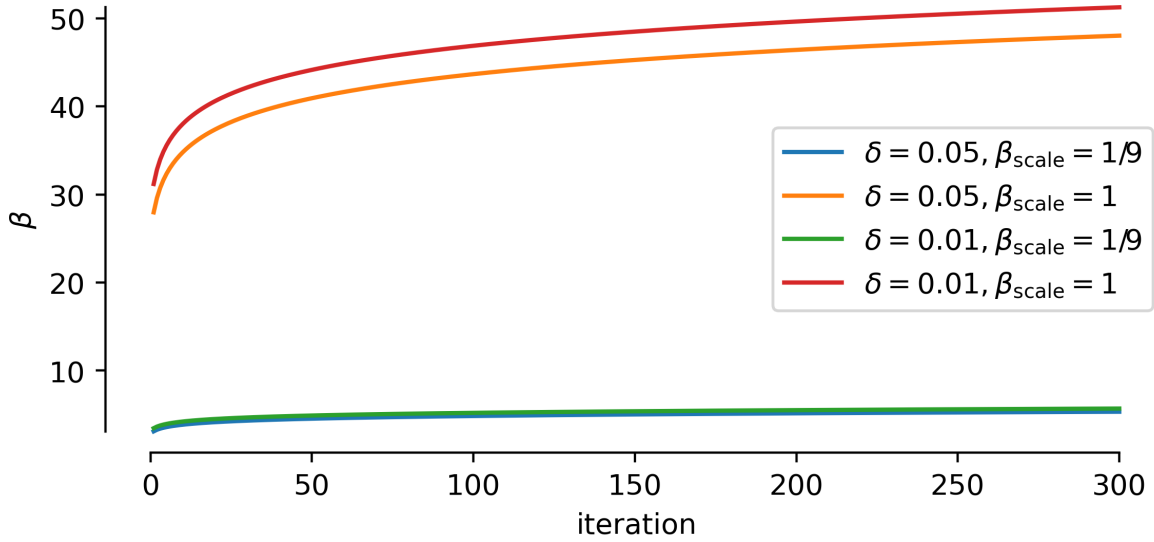
To test the influence of hyperparameter settings we provide an example where we ran one step of the algorithm on the [Binh-Korn test function](#).

We model every objective separately with a Matérn-3/2 kernel, leave $\delta = 0.05$ fixed and vary ϵ and β_{scale} .

We find that increasing ϵ speeds up the algorithm, but gives us a sparser Pareto frontier. Similarly, β_{scale} speeds up the algorithm by scaling the size of the hyperrectangles.



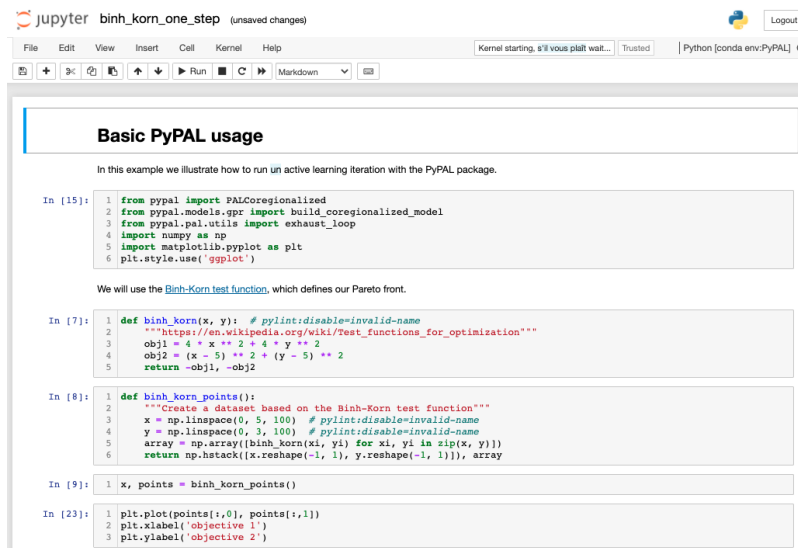
As shown in the figure below, β depends on δ and scaling beta down will drastically reduce the size of the uncertainty rectangles and in this way influence the convergence behavior.



1.3 Tutorials

To explore different use cases of PyePAL, we recommend checking out the [example notebooks](#). All notebooks can be run without installation on MyBinder. In the folder you find the notebooks with pre-executed output cells. Rerunning them should take no more than a few minutes.

1.3.1 1. One active learning step using GPR models built with GPy



Basic PyPAL usage

In this example we illustrate how to run an active learning iteration with the PyPAL package.

```
In [15]: 1 from pypal import PALCoregionalized
2 from pypal.models.gpr import build_coregionalized_model
3 from pypal.pal.utils import exhaust_loop
4 import numpy as np
5 import matplotlib.pyplot as plt
6 plt.style.use('ggplot')
```

We will use the [Binh-Korn test function](#), which defines our Pareto front.

```
In [7]: 1 def binh_korn(x, y): # pylint:disable=invalid-name
2     """https://en.wikipedia.org/wiki/Test_functions_for_optimisation"""
3     obj1 = 4 * x ** 2 + 4 * y ** 2
4     obj2 = (x - 5) ** 2 + (y - 5) ** 2
5     return -obj1, -obj2
```

```
In [8]: 1 def binh_korn_points():
2     """Create a dataset based on the Binh-Korn test function"""
3     x = np.linspace(0, 5, 100) # pylint:disable=invalid-name
4     y = np.linspace(0, 3, 100) # pylint:disable=invalid-name
5     array = np.array([binh_korn(xi, yi) for xi, yi in zip(x, y)])
6     return np.hstack([x.reshape(-1, 1), y.reshape(-1, 1)]), array
```

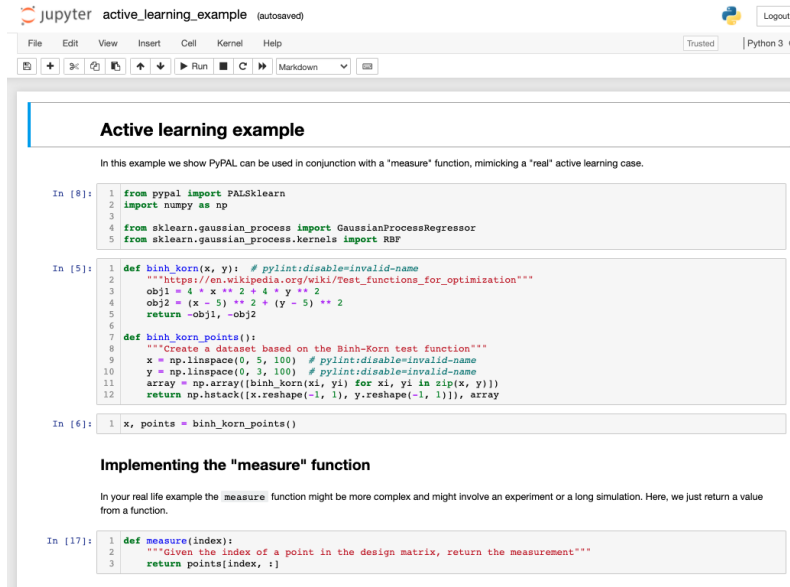
```
In [9]: 1 x, points = binh_korn_points()
```

```
In [23]: 1 plt.plot(points[:,0], points[:,1])
2 plt.xlabel('objective 1')
3 plt.ylabel('objective 2')
```

Topic covered

- building a pal_coregionalized GPR model using `build_coregionalized_model()`
- using coregionalized models with *PALCoregionalized*
- attributes of the PAL instance
- `exhaust_loop()`

1.3.2 2. Active learning with “measure” function and sklearn models



Active learning example

In this example we show PyPAL can be used in conjunction with a “measure” function, mimicking a “real” active learning case.

```
In [8]: 1 from pyPAL import PALSklearn
2 import numpy as np
3
4 from sklearn.gaussian_process import GaussianProcessRegressor
5 from sklearn.gaussian_process.kernels import RBF
```

```
In [5]: 1 def binh_korn(x, y): # pylint:disable=invalid-name
2     """https://en.wikipedia.org/wiki/Test_functions_for_optimization"""
3     obj1 = 4 * x ** 2 + 4 * y ** 2
4     obj2 = (x - 5) ** 2 + (y - 5) ** 2
5     return -obj1, -obj2
6
7 def binh_korn_points():
8     """Create a dataset based on the Binh-Korn test function"""
9     x = np.linspace(0, 5, 100) # pylint:disable=invalid-name
10    y = np.linspace(0, 3, 100) # pylint:disable=invalid-name
11    array = np.array([binh_korn(xi, yi) for xi, yi in zip(x, y)])
12    return np.hstack([x.reshape(-1, 1), y.reshape(-1, 1)], array)
```

```
In [6]: 1 x, points = binh_korn_points()
```

Implementing the “measure” function

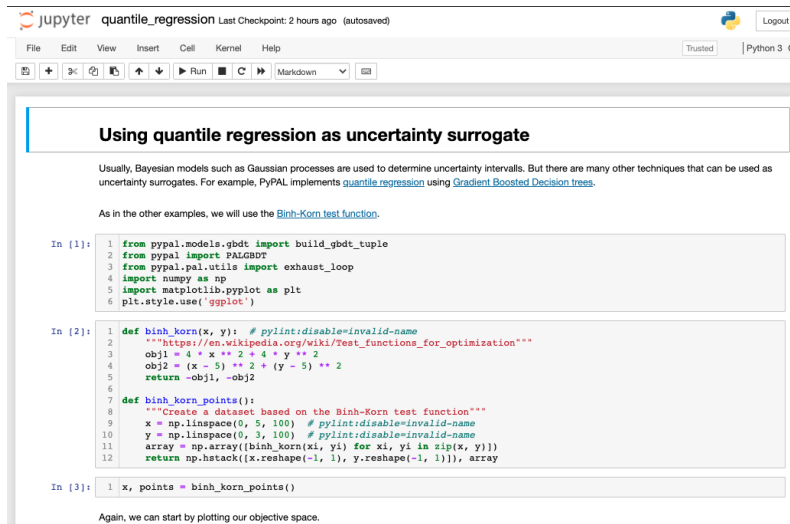
In your real life example the “measure” function might be more complex and might involve an experiment or a long simulation. Here, we just return a value from a function.

```
In [17]: 1 def measure(index):
2     """Given the index of a point in the design matrix, return the measurement"""
3     return points[index, 1]
```

Topic covered

- using sklearn models with *PALSklearn*
- selecting an initial set with *get_maxmin_samples()*
- plotting with *plot_jointplot()*

1.3.3 3. Quantile regression



Using quantile regression as uncertainty surrogate

Usually, Bayesian models such as Gaussian processes are used to determine uncertainty intervals. But there are many other techniques that can be used as uncertainty surrogates. For example, PyPAL implements *quantile regression* using *Gradient Boosted Decision trees*.

As in the other examples, we will use the *Binh-Korn test function*.

```
In [1]: 1 from pyPAL.models.gbdtd import build_gbdtd_tuple
2 from pyPAL import PALGBDT
3 from pyPAL.palutils import exhaust_loop
4 import numpy as np
5 import matplotlib.pyplot as plt
6 plt.style.use('ggplot')
```

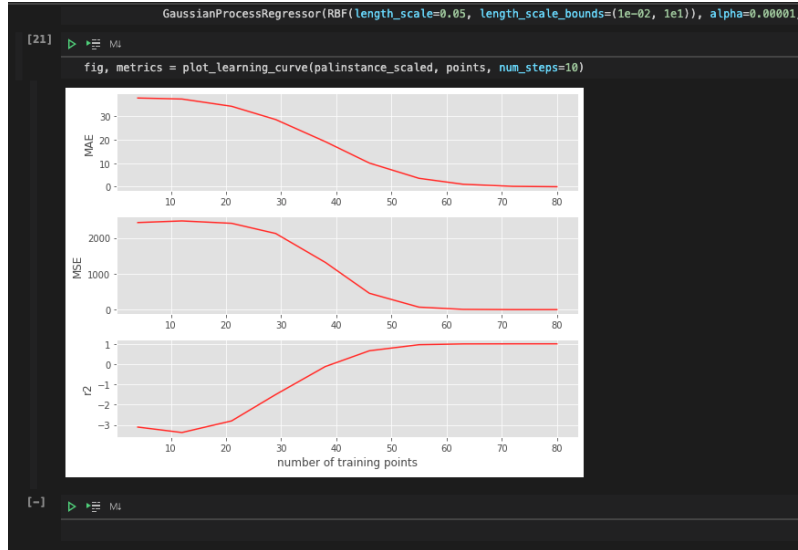
```
In [2]: 1 def binh_korn(x, y): # pylint:disable=invalid-name
2     """https://en.wikipedia.org/wiki/Test_functions_for_optimization"""
3     obj1 = 4 * x ** 2 + 4 * y ** 2
4     obj2 = (x - 5) ** 2 + (y - 5) ** 2
5     return -obj1, -obj2
6
7 def binh_korn_points():
8     """Create a dataset based on the Binh-Korn test function"""
9     x = np.linspace(0, 5, 100) # pylint:disable=invalid-name
10    y = np.linspace(0, 3, 100) # pylint:disable=invalid-name
11    array = np.array([binh_korn(xi, yi) for xi, yi in zip(x, y)])
12    return np.hstack([x.reshape(-1, 1), y.reshape(-1, 1)], array)
```

```
In [3]: 1 x, points = binh_korn_points()
```

Again, we can start by plotting our objective space.

- Using LightGBM models with quantile loss with *PALGBDT*
- selecting an initial set with *get_kmeans_samples()*

1.3.4 4. Plotting a learning curve



- Using *sklearn* Gaussian process models with RBF kernels
- Plotting learning curves

1.4 The PyePAL API reference

1.4.1 The PAL package

Core functions

Core functions for PAL

Base class

Base class for PAL

```
class pyepal.pal.pal_base.PALBase(X_design, models, ndim, epsilon=0.01, delta=0.05,
                                   beta_scale=0.1111111111111111, goals=None, coef_var_threshold=3,
                                   ranges=None)
```

Bases: object

PAL base class

```
__init__(X_design, models, ndim, epsilon=0.01, delta=0.05, beta_scale=0.1111111111111111,
          goals=None, coef_var_threshold=3, ranges=None)
```

Initialize the PAL instance

Parameters

- ***X_design*** (*np.array*) – Design space (feature matrix)
- ***models*** (*list*) – Machine learning models
- ***ndim*** (*int*) – Number of objectives

- **epsilon** (*Union[list, float], optional*) – Epsilon hyperparameter. Defaults to 0.01.
- **delta** (*float, optional*) – Delta hyperparameter. Defaults to 0.05.
- **beta_scale** (*float, optional*) – Scaling parameter for beta. If not equal to 1, the theoretical guarantees do not necessarily hold. Also note that the parametrization depends on the kernel type. Defaults to 1/9.
- **goals** (*List[str], optional*) – If a list, provide “min” for every objective that shall be minimized and “max” for every objective that shall be maximized. Defaults to None, which `_means` that the code maximizes all objectives.
- **coef_var_threshold** (*float, optional*) – Use only points with a coefficient of variation below this threshold in the classification step. Defaults to 3.
- **ranges** (*np.ndarray, optional*) – Numpy array of length `ndmin`, where each element contains the value range of given objective. If this is provided, we will use $\epsilon \cdot \text{ranges}$ to compute the uncertainties of the hyperrectangles instead of the default behavior $\epsilon \cdot |\mu|$

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

augment_design_space(*X_design, classify=False, clean_classify=True*)

Add new design points to PAL instance

Parameters

- **X_design** (*np.ndarray*) – Design matrix. Two-dimensional array containing measurements in the rows and the features as the columns.
- **classify** (*bool*) – Reclassifies the new design space, using the old model. This is, it runs inference, calculates the hyperrectangles, and runs the classification. Does not increase the iteration count. Note though that points that already have been classified as Pareto-optimal will not be re-classified, e.g., discarded—even if the new design points dominate the existing “Pareto optimal” points. Defaults to False.
- **clean_classify** (*bool*) – Reclassifies the new design space, using the old model. This is, it runs inference, calculates the hyperrectangles, and runs the classification. Does not increase the iteration count. But, in contrast to `classify` it erases all previous classifications, before running the new classification. Hence, if some new design point dominates a previously Pareto efficient point, the previous Pareto optimal point will no longer be classified as Pareto efficient. This flag is incompatible with `classify`. If you choose `clean_classify`, PyePAL will erase all previous classifications, independent of what you choose for `classify`. Defaults to True.

Return type None

property discarded_indices

Return the indices of the discarded points

property discarded_points

Return the discarded points

property hyperrectangle_sizes

Return the sizes of the hyperrectangles

property means

Return the means predicted by the model

property number_design_points

Return the number of points in the design space

property number_discarded_points

Return the number of discarded points

property number_pareto_optimal_points

Return the number of Pareto optimal points

property number_sampled_points

Return the number of sampled points

property number_unclassified_points

Return the number of unclassified points

property pareto_optimal_indices

Return the indices of the Pareto optimal points

property pareto_optimal_points

Return the pareto optimal points

run_one_step(*batch_size=1, pooling_method='fro', sample_discarded=False, use_coef_var=True, replace_mean=True, replace_std=True*)

[summary]

Parameters

- **batch_size** (*int, optional*) – Number of indices that will be returned. Defaults to 1.
- **pooling_method** (*str*) – Method that is used to aggregate the uncertainty in different objectives into one scalar. Available options are: “fro” (Frobenius/Euclidean norm), “mean”, “median”. Defaults to “fro”.
- **sample_discarded** (*bool*) – if true, it will sample from all points and not only from the unclassified and Pareto optimal ones
- **use_coef_var** (*bool*) – If True, uses the coefficient of variation instead of the unscaled rectangle sizes
- **replace_mean** (*bool*) – If true uses the measured `_means` for the sampled points
- **replace_std** (*bool*) – If true uses the measured standard deviation for the sampled points

Raises ValueError – In case the PAL instance was not initialized with measurements.

Returns

Returns array of indices if there are unclassified points that can be sample left.

Return type Union[np.array, None]

sample(*exclude_idx=None, pooling_method='fro', sample_discarded=False, use_coef_var=True*)

Runs the sampling step based on the size of the hyperrectangle. I.e., favoring exploration.

Parameters

- **exclude_idx** (*Union[np.array, None], optional*) – Points in design space to exclude from sampling. Defaults to None.

- **pooling_method** (*str*) – Method that is used to aggregate the uncertainty in different objectives into one scalar. Available options are: “fro” (Frobenius/Euclidean norm), “mean”, “median”. Defaults to “fro”.
- **sample_discarded** (*bool*) – if true, it will sample from all points and not only from the unclassified and Pareto optimal ones
- **use_coef_var** (*bool*) – If True, uses the coefficient of variation instead of the unscaled rectangle sizes

Raises **ValueError** – In case there are no uncertainty rectangles, i.e., when the `_predict` has not been successfully called.

Returns Index of next point to evaluate in design space

Return type `int`

property **sampler_indices**

Return the indices of the sampled points

property **sampler_mask**

Create a mask for the sampled points We count a point as sampled if at least one objective has been measured, i.e., `self.sampler` is a `N * number objectives` array in which some columns can be false if a measurement has not been performed

property **sampler_points**

Return the sampled points

should_cross_validate()

Override for more complex cross validation schedules

property **unclassified_indices**

Return the indices of the unclassified points

property **unclassified_points**

Return the discarded points

update_train_set (*indices, measurements, measurement_uncertainty=None*)

Update training set following a measurement

Parameters

- **indices** (*np.ndarray*) – Indices of design space at which the measurements were taken
- **measurements** (*np.ndarray*) – Measured values, 2D array. the length must equal the length of the indices array. the second direction must equal the number of objectives. If an objective is missing, provide `np.nan`. For example, `np.array([1, 1, np.nan])`
- **measurement_uncertainty** (*np.ndarray*) – uncertainty in the measurements, if not provided (`None`) will be zero. If it is not `None`, it must be an array with the same shape as the measurements. If an objective is missing, provide `np.nan`. For example, `np.array([1, 1, np.nan])`

property **uses_fixed_epsilon**

True if it uses the fixed epsilon $\epsilon \cdot ranges$

For GPy models

PAL using GPy GPR models

class pyepal.pal.pal_gpy.PALGPy(*args, **kwargs)

Bases: [pyepal.pal.pal_base.PALBase](#)

PAL class for a list of GPy GPR models, with one model per objective

__init__(*args, **kwargs)

Construct the PALGPy instance

Parameters

- **X_design** (*np.array*) – Design space (feature matrix)
- **models** (*list*) – Machine learning models
- **ndim** (*int*) – Number of objectives
- **epsilon** (*Union[list, float], optional*) – Epsilon hyperparameter. Defaults to 0.01.
- **delta** (*float, optional*) – Delta hyperparameter. Defaults to 0.05.
- **beta_scale** (*float, optional*) – Scaling parameter for beta. If not equal to 1, the theoretical guarantees do not necessarily hold. Also note that the parametrization depends on the kernel type. Defaults to 1/9.
- **goals** (*List[str], optional*) – If a list, provide “min” for every objective that shall be minimized and “max” for every objective that shall be maximized. Defaults to None, which means that the code maximizes all objectives.
- **coef_var_threshold** (*float, optional*) – Use only points with a coefficient of variation below this threshold in the classification step. Defaults to 3.
- **restarts** (*int*) – Number of random restarts that are used for hyperparameter optimization. Defaults to 20.
- **n_jobs** (*int*) – Number of parallel processes that are used to fit the GPR models. Defaults to 1.

For coregionalized GPy models

PAL for coregionalized GPR models

class pyepal.pal.pal_coregionalized.PALCoregionalized(*args, **kwargs)

Bases: [pyepal.pal.pal_base.PALBase](#)

PAL class for a coregionalized GPR model

__init__(*args, **kwargs)

Construct the PALCoregionalized instance

Parameters

- **X_design** (*np.array*) – Design space (feature matrix)
- **models** (*list*) – Machine learning models
- **ndim** (*int*) – Number of objectives

- **epsilon** (*Union[list, float], optional*) – Epsilon hyperparameter. Defaults to 0.01.
- **delta** (*float, optional*) – Delta hyperparameter. Defaults to 0.05.
- **beta_scale** (*float, optional*) – Scaling parameter for beta. If not equal to 1, the theoretical guarantees do not necessarily hold. Also note that the parametrization depends on the kernel type. Defaults to 1/9.
- **goals** (*List[str], optional*) – If a list, provide “min” for every objective that shall be minimized and “max” for every objective that shall be maximized. Defaults to None, which means that the code maximizes all objectives.
- **coef_var_threshold** (*float, optional*) – Use only points with a coefficient of variation below this threshold in the classification step. Defaults to 3.
- **restarts** (*int*) – Number of random restarts that are used for hyperparameter optimization. Defaults to 20.
- **parallel** (*bool*) – If true, model hyperparameters are optimized in parallel, using the GPy implementation. Defaults to False.

For sklearn GPR models

PAL using Sklearn GPR models

class `pyepal.pal.pal_sklearn.PALSklearn(*args, **kwargs)`

Bases: `pyepal.pal.pal_base.PALBase`

PAL class for a list of Sklearn (GPR) models, with one model per objective

__init__ (**args, **kwargs*)

Construct the PALSklearn instance

Parameters

- **X_design** (*np.array*) – Design space (feature matrix)
- **models** (*list*) – Machine learning models. You can provide a list of GaussianProcessRegressor instances or a list of *fitted* RandomizedSearchCV/GridSearchCV instances with GaussianProcessRegressor models
- **ndim** (*int*) – Number of objectives
- **epsilon** (*Union[list, float], optional*) – Epsilon hyperparameter. Defaults to 0.01.
- **delta** (*float, optional*) – Delta hyperparameter. Defaults to 0.05.
- **beta_scale** (*float, optional*) – Scaling parameter for beta. If not equal to 1, the theoretical guarantees do not necessarily hold. Also note that the parametrization depends on the kernel type. Defaults to 1/9.
- **goals** (*List[str], optional*) – If a list, provide “min” for every objective that shall be minimized and “max” for every objective that shall be maximized. Defaults to None, which means that the code maximizes all objectives.
- **coef_var_threshold** (*float, optional*) – Use only points with a coefficient of variation below this threshold in the classification step. Defaults to 3.
- **n_jobs** (*int*) – Number of parallel processes that are used to fit the GPR models. Defaults to 1.

For quantile regression with LightGBM

Implements a PAL class for GBDT models which can predict uncertainty intervals when used with quantile loss. For an example of GBDT with quantile loss see Jablonka, Kevin Maik; Moosavi, Seyed Mohamad; Asgari, Mehrdad; Ireland, Christopher; Patiny, Luc; Smit, Berend (2020): A Data-Driven Perspective on the Colours of Metal-Organic Frameworks. ChemRxiv. Preprint. <https://doi.org/10.26434/chemrxiv.13033217.v1>

For general information about quantile regression see https://en.wikipedia.org/wiki/Quantile_regression

Note that the scaling of the hyperrectangles has been derived for GPR models (with RBF kernels).

```
class pyepal.pal.pal_gbdtd.PALGBDT(*args, **kwargs)
```

Bases: `pyepal.pal.pal_base.PALBase`

PAL class for a list of LightGBM GBDT models

```
__init__(*args, **kwargs)
```

Construct the PALGBDT instance

Parameters

- **X_design** (*np.array*) – Design space (feature matrix)
- **(List[Iterable[LGBMRegressor] (models)** – Machine learning models. You need to provide a list of iterables. One iterable per objective and every iterable contains three LGBMRegressors. The first one for the lower uncertainty limits, the middle one for the median and the last one for the upper limit. To create appropriate models, you need to use the quantile loss. If you want to parallelize training, we recommend that you use the LightGBM parallelization and fit the models for the different objectives in serial fashion.s
- **LGBMRegressor** – Machine learning models. You need to provide a list of iterables. One iterable per objective and every iterable contains three LGBMRegressors. The first one for the lower uncertainty limits, the middle one for the median and the last one for the upper limit. To create appropriate models, you need to use the quantile loss. If you want to parallelize training, we recommend that you use the LightGBM parallelization and fit the models for the different objectives in serial fashion.s
- **LGBMRegressor]]** – Machine learning models. You need to provide a list of iterables. One iterable per objective and every iterable contains three LGBMRegressors. The first one for the lower uncertainty limits, the middle one for the median and the last one for the upper limit. To create appropriate models, you need to use the quantile loss. If you want to parallelize training, we recommend that you use the LightGBM parallelization and fit the models for the different objectives in serial fashion.s
- **ndim** (*int*) – Number of objectives
- **epsilon** (*Union[list, float], optional*) – Epsilon hyperparameter. Defaults to 0.01.
- **delta** (*float, optional*) – Delta hyperparameter. Defaults to 0.05.
- **beta_scale** (*float, optional*) – Scaling parameter for beta. If not equal to 1, the theoretical guarantees do not necessarily hold. Also note that the parametrization depends on the kernel type. Defaults to 1/9.
- **goals** (*List[str], optional*) – If a list, provide “min” for every objective that shall be minimized and “max” for every objective that shall be maximized. Defaults to None, which means that the code maximizes all objectives.
- **coef_var_threshold** (*float, optional*) – Use only points with a coefficient of variation below this threshold in the classification step. Defaults to 3.

- **interquartile_scaler** (*float, optional*) – Used to convert the difference between the upper and lower quantile into a standard deviation. This, is $\text{std} = (\text{up-low})/\text{interquartile_scaler}$. Defaults to 1.35, following Wan, X., Wang, W., Liu, J. et al. Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range. BMC Med Res Methodol 14, 135 (2014). <https://doi.org/10.1186/1471-2288-14-135>

For GPR with GPFlow

PAL using GPy GPR models

class pyepal.pal.pal_gpflowgpr.PALGPflowGPR(*args, **kwargs)

Bases: [pyepal.pal.pal_base.PALBase](#)

PAL class for a list of GPFlow GPR models, with one model per objective. Please consider that there are specific multioutput models (<https://gpflow.readthedocs.io/en/master/notebooks/advanced/multioutput.html>) for which the train and prediction function would need to be adjusted. You might also consider using streaming GPRs (https://github.com/thangbui/streaming_sparse_gp). In future releases we might support this case automatically (i.e., handle the case in which only one model is provided).

__init__(*args, **kwargs)

Construct the PALGPflowGPR instance

Parameters

- **X_design** (*np.array*) – Design space (feature matrix)
- **models** (*list*) – Machine learning models
- **ndim** (*int*) – Number of objectives
- **epsilon** (*Union[list, float], optional*) – Epsilon hyperparameter. Defaults to 0.01.
- **delta** (*float, optional*) – Delta hyperparameter. Defaults to 0.05.
- **beta_scale** (*float, optional*) – Scaling parameter for beta. If not equal to 1, the theoretical guarantees do not necessarily hold. Also note that the parametrization depends on the kernel type. Defaults to 1/9.
- **goals** (*List[str], optional*) – If a list, provide “min” for every objective that shall be minimized and “max” for every objective that shall be maximized. Defaults to None, which means that the code maximizes all objectives.
- **coef_var_threshold** (*float, optional*) – Use only points with a coefficient of variation below this threshold in the classification step. Defaults to 3.
- **opt** (*function, optional*) – Optimizer function for the GPR parameters. If None (default), then we will use `gpflow.optimizers.Scipy()`
- **opt_kwargs** (*dict, optional*) – Keyword arguments passed to the optimizer. If None, PyePAL will pass `{“maxiter”: 100}`
- **n_jobs** (*int*) – Number of parallel threads that are used to fit the GPR models. Defaults to 1.

Schedules for hyperparameter optimization

Provides some scheduling functions that can be used to implement the `_should_optimize_hyperparameters` function

`pyepal.pal.schedules.exp_decay(iteration, base=10)`

Optimize hyperparameters at logarithmically spaced intervals

Parameters

- **iteration** (*int*) – current iteration
- **base** (*int*, *optional*) – Base of the logarithm. Defaults to 10.

Returns True if iteration is on the log scaled grid

Return type bool

`pyepal.pal.schedules.linear(iteration, frequency=10)`

Optimize hyperparameters at equally spaced intervals

Parameters

- **iteration** (*int*) – current iteration
- **frequency** (*int*, *optional*) – Spacing between the True outputs. Defaults to 10.

Returns True if iteration can be divided by frequency without remainder

Return type bool

Utilities for multiobjective optimization

Utilities for dealing with Pareto fronts in general

`pyepal.pal.utils.dominance_check(point1, point2)`

One point dominates another if it is not worse in all objectives and strictly better in at least one. This here assumes we want to maximize

Return type bool

`pyepal.pal.utils.dominance_check_jitted(point, array)`

Check if point dominates any point in array

Return type bool

`pyepal.pal.utils.dominance_check_jitted_2(array, point)`

Check if any point in array dominates point

Return type bool

`pyepal.pal.utils.dominance_check_jitted_3(array, point, ignore_me)`

Check if any point in array dominates point. `ignore_me` since numba does not understand masked arrays

Return type bool

`pyepal.pal.utils.exhaust_loop(palinstance, y, batch_size=1)`

Helper function that takes an initialized PAL instance and loops the sampling until there is no unclassified point left. This is useful if all measurements are already taken and one wants to test the algorithm with different hyperparameters.

Parameters

- **palinstance** (*PALBase*) – A initialized instance of a class that inherited from *PALBase* and implemented the `._train()` and `._predict()` functions
- **y** (*np.array*) – Measurements. The number of measurements must equal the number of points in the design space.
- **batch_size** (*int, optional*) – Number of indices that will be returned. Defaults to 10.

Returns None. The PAL instance is updated in place

`pyepal.pal.utils.get_hypervolume(pareto_front, reference_vector, prefactor=-1)`

Compute the hypervolume indicator of a Pareto front *I* multiply it with minus one as we assume that we want to maximize all objective and then we calculate the area

$f1 \mid | \text{---} | \mid - \mid - \mid \text{---} \mid f2$

But the code we use for the hv indicator assumes that the reference vector is larger than all the points in the Pareto front. For this reason, we then flip all the signs using *prefactor*

This indicator is not needed for the epsilon-PAL algorithm itself but only to allow tracking a metric that might help the user to see if the algorithm converges.

Return type float

`pyepal.pal.utils.get_kmeans_samples(X, n_samples, **kwargs)`

Get the samples that are closest to the *k=n_samples* centroids

Parameters

- **X** (*np.array*) – Feature array, on which the KMeans clustering is run
- **n_samples** (*int*) – number of samples are should be selected
- **KMeans** (***kwargs passed to the*) –

Returns selected_indices

Return type np.array

`pyepal.pal.utils.get_maxmin_samples(X, n_samples, metric='euclidean', init='mean', seed=None, **kwargs)`

Greedy maxmin sampling, also known as Kennard-Stone sampling (1). Note that a greedy sampling is not guaranteed to give the ideal solution and the output will depend on the random initialization (if this is chosen).

If you need a good solution, you can restart this algorithm multiple times with random initialization and different random seeds and use a coverage metric to quantify how well the space is covered. Some metrics are described in (2). In contrast to the code provided with (2) and (3) we do not consider the feature importance for the selection as this is typically not known beforehand.

You might want to standardize your data before applying this sampling function.

Some more sampling options are provided in our *structure_comp* (4) Python package. Also, this implementation here is quite memory hungry.

References: (1) Kennard, R. W.; Stone, L. A. Computer Aided Design of Experiments. *Technometrics* 1969, 11 (1), 137–148. <https://doi.org/10.1080/00401706.1969.10490666>. (2) Moosavi, S. M.; Nandy, A.; Jablonka, K. M.; Ongari, D.; Janet, J. P.; Boyd, P. G.; Lee, Y.; Smit, B.; Kulik, H. J. Understanding the Diversity of the Metal-Organic Framework Ecosystem. *Nature Communications* 2020, 11 (1), 4068. <https://doi.org/10.1038/s41467-020-17755-8>. (3) Moosavi, S. M.; Chidambaram, A.; Talirz, L.; Haranczyk, M.; Stylianou, K. C.; Smit, B. Capturing Chemical Intuition in Synthesis of Metal-Organic Frameworks. *Nat Commun* 2019, 10 (1), 539. <https://doi.org/10.1038/s41467-019-08483-9>. (4) https://github.com/kjappelbaum/structure_comp

Parameters

- **X** (*np.array*) – Feature array, this is the array that is used to perform the sampling
- **n_samples** (*int*) – number of points that will be selected, needs to be lower than the length of X
- **metric** (*str, optional*) – Distance metric to use for the maxmin calculation. Must be a valid option of `scipy.spatial.distance.cdist` ('braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'jensenshannon', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'). Defaults to 'euclidean'
- **init** (*str, optional*) – either 'mean', 'median', or 'random'. Determines how the initial point is chosen. Defaults to 'center'
- **seed** (*int, optional*) – seed for the random number generator. Defaults to None.
- **cdist** (***kwargs passed to the*) –

Returns `selected_indices`

Return type `np.array`

`pyepal.pal.utils.is_pareto_efficient(costs, return_mask=True)`

Find the Pareto efficient points Based on <https://stackoverflow.com/questions/32791911/fast-calculation-of-pareto-front-in-python>

Parameters

- **costs** (*np.array*) – An (n_points, n_costs) array
- **return_mask** (*bool, optional*) – True to return a mask, Otherwise it will be a (n_efficient_points,) integer array of indices. Defaults to True.

Returns [description]

Return type `np.array`

Utilities for plotting

Plotting utilities

`pyepal.plotting.plot_bar_iterations(pareto_optimal, non_pareto_points, unclassified_points, ax=None)`

Plot stacked barplots for every step of the iteration.

Parameters

- **pareto_optimal** (*np.ndarray*) – Number of pareto optimal points for every iteration.
- **non_pareto_points** (*np.ndarray*) – Number of discarded points for every iteration
- **unclassified_points** (*np.ndarray*) – Number of unclassified points for every iteration

Returns

matplotlib axis (the same that was provided as input or one from a new figure if no axis was provided)

Return type `axis`

`pyepal.plotting.plot_histogram(y, palinstance, ax=None)`

Plot histograms, with maxima scaled to one and different categories indicated in color for one objective

Parameters

- **y** (*np.ndarray*) – objective (measurement)

- **palinstance** ([PALBase](#)) – instance of a PAL class
- **ax** (*ax*) – Matplotlib figure axis

Returns

matplotlib axis (the same that was provided as input or one from a new figure if no axis was provided)f

Return type *ax*

`pyepal.plotting.plot_jointplot(y, palinstance, labels=None, figsize=(8.0, 6.0))`

Plot a jointplot of the objective space with histograms on the diagonal and 2D-Pareto plots on the off-diagonal.

Parameters

- **y** (*np.array*) – Two-dimensional array with the objectives (measurements)
- **palinstance** ([PALBase](#)) – “trained” PAL instance
- **labels** (*Union[List[str], None]*, *optional*) – Labels for each objective. Defaults to “objective [index]”.
- **figsize** (*tuple*, *optional*) – Figure size for joint plot. Defaults to (8.0, 6.0).

Returns matplotlib Figure object.

Return type *fig*

`pyepal.plotting.plot_pareto_front_2d(y_0, y_1, std_0, std_1, palinstance, ax=None)`

Plot a 2D pareto front, with the different categories indicated in color.

Parameters

- **y_0** (*np.ndarray*) – objective 0
- **y_1** (*np.ndarray*) – objective 1
- **std_0** (*np.ndarray*) – standard deviation objective 0
- **std_1** (*np.ndarray*) – standard deviation objective 0
- **palinstance** ([PALBase](#)) – PAL instance
- **ax** (*axis*, *optional*) – Matplotlib figure axis. Defaults to None.

Returns

matplotlib axis (the same that was provided as input or one from a new figure if no axis was provided)

Return type *ax*

`pyepal.plotting.plot_residuals(y, palinstance, labels=None, figsize=(6.0, 4.0))`

Plot signed residual (on y axis) vs fitted (on x axis) plot of sampled points. Will create subplots for *y.ndim* > 1.

Parameters

- **y** (*np.array*) – Two-dimensional array with the objectives (measurements)
- **palinstance** ([PALBase](#)) – “trained” PAL instance
- **labels** (*Union[List[str], None]*, *optional*) – Labels for each objective. Defaults to “objective [index]”.
- **figsize** (*tuple*, *optional*) – Figure size for each individual residual vs fitted objective plot. Defaults to (6.0, 4.0).

Returns matplotlib Figure object

Return type fig

Input validation

Methods to validate inputs for the PAL classes

`pyepal.pal.validate_inputs.base_validate_models(models)`

Currently no validation as the predict and train function are implemented independet of the base class

Return type list

`pyepal.pal.validate_inputs.validate_beta_scale(beta_scale)`

Parameters `beta_scale` (*Any*) – scaling factor for beta

Raises **ValueError** – If beta is smaller than 0

Returns scaling factor for beta

Return type float

`pyepal.pal.validate_inputs.validate_coef_var(coef_var)`

Make sure that the coef_var makes sense

`pyepal.pal.validate_inputs.validate_coregionalized_gpy(models)`

Make sure that model is a coregionalized GPR model

`pyepal.pal.validate_inputs.validate_delta(delta)`

Make sure that delta is in a reasonable range

Parameters `delta` (*Any*) – Delta hyperparameter

Raises **ValueError** – Delta must be in [0,1].

Returns delta

Return type float

`pyepal.pal.validate_inputs.validate_epsilon(epsilon, ndim)`

Validate epsilon and return a np.array

Parameters

- **epsilon** (*Any*) – Epsilon hyperparameter
- **ndim** (*int*) – Number of dimensions/objectives

Raises

- **ValueError** – If epsilon is a list there must be one float per dimension
- **ValueError** – Epsilon must be in [0,1]
- **ValueError** – If epsilon is an array there must be one float per dimension

Returns Array of one epsilon per objective

Return type np.ndarray

`pyepal.pal.validate_inputs.validate_gbdm_models(models, ndim)`

Make sure that the number of iterables is equal to the number of objectives and that every iterable contains three LGBMRegressors. Also, we check that at least the first and last models use quantile loss

Return type List[Iterable]

`pyepal.pal.validate_inputs.validate_goals(goals, ndim)`

Create a valid array of goals. 1 for maximization, -1 for objectives that are to be minimized.

Parameters

- **goals** (*Any*) – List of goals, typically provided as strings ‘max’ for maximization and ‘min’ for minimization
- **ndim** (*int*) – number of dimensions

Raises

- **ValueError** – If goals is a list and the length is not equal to ndim
- **ValueError** – If goals is a list and the elements are not strings ‘min’, ‘max’ or -1 and 1

Returns Array of -1 and 1

Return type np.ndarray

`pyepal.pal.validate_inputs.validate_gpy_model(models)`

Make sure that all elements of the list a GPRegression models

`pyepal.pal.validate_inputs.validate_interquartile_scaler(interquartile_scaler)`

Make sure that the interquartile_scaler makes sense

Return type float

`pyepal.pal.validate_inputs.validate_ndim(ndim)`

Make sure that the number of dimensions makes sense

Parameters **ndim** (*Any*) – number of dimensions

Raises

- **ValueError** – If the number of dimensions is not an integer
- **ValueError** – If the number of dimensions is not greater than 0

Returns the number of dimensions

Return type int

`pyepal.pal.validate_inputs.validate_njobs(njobs)`

Make sure that njobs is an int > 1

Return type int

`pyepal.pal.validate_inputs.validate_nt_models(models, ndim)`

Make sure that we can work with a sequence of `pyepal.pal.models.nt.NTModel()`

Return type Sequence

`pyepal.pal.validate_inputs.validate_number_models(models, ndim)`

Make sure that there are as many models as objectives

Parameters

- **models** (*Any*) – List of models
- **ndim** (*int*) – Number of objectives

Raises **ValueError** – If the number of models does not equal the number of objectives

`pyepal.pal.validate_inputs.validate_optimizers(optimizers, ndim)`

Make sure that we can work with a Sequence if JaxOptimizer

Return type Sequence

`pyepal.pal.validate_inputs.validate_positive_integer_list(seq, ndim,
parameter_name='Parameter')`

Can be used, e.g., to validate and standardize the ensemble size and epochs input

Return type Sequence[int]

`pyepal.pal.validate_inputs.validate_sklearn_gpr_models(models, ndim)`

Make sure that there is a list of GPR models, one model per objective

Return type List[GaussianProcessRegressor]

1.4.2 The models package

Helper functions for GPR with GPy

Wrappers for Gaussian Process Regression models.

We typically use the GPy package as it offers most flexibility for Gaussian processes in Python. Typically, we use automatic relevance determination (ARD), where one lengthscale parameter per input dimension is used.

If your task requires training on larger training sets, you might consider replacing the models with their sparse version but for the epsilon-PAL algorithm this typically shouldn't be needed.

For kernel selection, you can have a look at <https://www.cs.toronto.edu/~duvenaud/cookbook/> Matérn, RBF and RationalQuadrat are good quick and dirty solutions but have their caveats

`pyepal.models.gpr.build_coregionalized_model(X_train, y_train, kernel=None, w_rank=1, **kwargs)`

Wrapper for building a coregionalized GPR, it will have as many outputs as `y_train.shape[1]`. Each output will have its own noise term

Return type GPCoregionalizedRegression

`pyepal.models.gpr.build_model(X_train, y_train, index=0, kernel=None, **kwargs)`

Build a single-output GPR model

Return type GPRegression

`pyepal.models.gpr.get_matern_32_kernel(NFEAT, ARD=True, **kwargs)`

Matern-3/2 kernel without ARD

Return type Matern32

`pyepal.models.gpr.get_matern_52_kernel(NFEAT, ARD=True, **kwargs)`

Matern-5/2 kernel without ARD

Return type Matern52

`pyepal.models.gpr.get_ratquad_kernel(NFEAT, ARD=True, **kwargs)`

Rational quadratic kernel without ARD

Return type RatQuad

`pyepal.models.gpr.predict(model, X)`

Wrapper function for the prediction method of a GPy regression model. It return the standard deviation instead of the variance

Return type Tuple[array, array]

`pyepal.models.gpr.predict_coregionalized(model, X, index=0)`

Wrapper function for the prediction method of a coregionalized GPy regression model. It return the standard deviation instead of the variance

Return type Tuple[array, array]

`pyepal.models.gpr.set_xy_coregionalized(model, X, y, mask=None)`

Wrapper to update a coregionalized model with new data

1.5 Developer notes

1.5.1 Contribution Guidelines

Commit messages

- To automatically generate the changelog and releases we use [conventional commits](#) use the prefix `feat` for new features, `chore` for updating grunt tasks etc; `no production code change`, `fix` for bug fixes and `docs` for changes to the documentation. Use `feat!:`, or `fix!:`, `refactor!:`, etc., to represent a breaking change (indicated by the `!`). This will result in bump of the SemVer major version number.

Python code

Please install the pre-commit hooks using
to automatically

- format the code with [black](#)
- sort the imports with [isort](#)
- lint the code with [pylint](#)

We use type hints, which we feel is a good way of documentation and helps us find bugs using [mypy](#).

Some of the pre-commit hooks modify the files, e.g., they trim whitespaces or format the code. If they modify your file, you will have to run `git add` and `git commit` again. To skip the pre-commit checks (not recommended) you can use `git commit --no-verify`.

New features

Please make a new branch for the development of new features. [Rebase on the upstream master](#) and include a test for your new feature. (The CI checks for a drop in code coverage.)

Releases

Releases are automated using a GitHub actions based on the commit message. Maintainers manually upload the release to PyPi.

1.5.2 Implementing a new PAL class

If you want to use PyePAL with a model that we do not support yet, i.e., not GPy or sklearn Gaussian process regression, it is easy to write your own class. For this, you will need to inherit from [PALBase](#) and implement your `_train` and `_predict()` functions (and maybe also the `pyepal.pal.pal_base.PALBase._set_hyperparameters` and `pyepal.pal.pal_base.PALBase._should_optimize_hyperparameters` functions) using the `design_space` and `y` attributes of the class.

For instance, if we develop some multioutput model that has a `train()` and a `predict()` method, we could simply use the following design pattern

```
from pyepal import PALBase

class PALMyModel(PALBase):
    def _train(self):
        self.models[0].train(self.design_space[self.sampled], self.y[self.sampled])

    def _predict(self):
        self.mu, self.std = self.models[0].predict(self.design_space)
```

Note that we typically provide the models, even if it is only one, in a list to keep the API consistent.

In some instances, you may want to perform an operation in parallel, e.g., train the models for different objectives in parallel. One convenient way to do this in Python is by using [concurrent.futures](#). The only caveat to this is that this approach requires that the function is pickleable. To ensure this, you may want to implement the function that you want to parallelize, outside the class. For example, you could use the following design pattern

```
from pyepal import PALBase
import concurrent.futures
from functools import partial

def _train_model_pickleable(i, models, design_space, objectives, sampled):
    model = models[i]
    model.fit(
        design_space[sampled[:, i]],
        objectives[sampled[:, i], i].reshape(-1, 1),
    )
    return model

class MyPal(PALBase):
    def __init__(self, *args, **kwargs):
        n_jobs = kwargs.pop("n_jobs", 1)
        validate_njobs(n_jobs)
```

(continues on next page)

(continued from previous page)

```
self.n_jobs = n_jobs
super().__init__(*args, **kwargs)

validate_number_models(self.models, self.ndim)

def _train(self):
    train_single_partial = partial(
        _train_model_picklable,
        models=self.models,
        design_space=self.design_space,
        objectives=self.y,
        sampled=self.sampled,
    )
    models = []
    with concurrent.futures.ProcessPoolExecutor(
        max_workers=self.n_jobs
    ) as executor:
        for model in executor.map(train_single_partial, range(self.ndim)):
            models.append(model)
    self.models = models
```


INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

p

- `pyepal.models.gpr`, 28
- `pyepal.pal.core`, 14
- `pyepal.pal.pal_base`, 14
- `pyepal.pal.pal_coregionalized`, 18
- `pyepal.pal.pal_gbd`, 20
- `pyepal.pal.pal_gpflowgpr`, 21
- `pyepal.pal.pal_gpy`, 18
- `pyepal.pal.pal_sklearn`, 19
- `pyepal.pal.schedules`, 22
- `pyepal.pal.utils`, 22
- `pyepal.pal.validate_inputs`, 26
- `pyepal.plotting`, 24

Symbols

`__init__()` (*pyepal.pal.pal_base.PALBase* method), 14
`__init__()` (*pyepal.pal.pal_coregionalized.PALCoregionalized* method), 18
`__init__()` (*pyepal.pal.pal_gbdtd.PALGBDT* method), 20
`__init__()` (*pyepal.pal.pal_gpflowgpr.PALGPflowGPR* method), 21
`__init__()` (*pyepal.pal.pal_gpy.PALGPpy* method), 18
`__init__()` (*pyepal.pal.pal_sklearn.PALSklearn* method), 19
`__repr__()` (*pyepal.pal.pal_base.PALBase* method), 15
`__weakref__` (*pyepal.pal.pal_base.PALBase* attribute), 15

A

`augment_design_space()`
(pyepal.pal.pal_base.PALBase method), 15

B

`base_validate_models()` (in module *pyepal.pal.validate_inputs*), 26
`build_coregionalized_model()` (in module *pyepal.models.gpr*), 28
`build_model()` (in module *pyepal.models.gpr*), 28

D

`discarded_indices` (*pyepal.pal.pal_base.PALBase* property), 15
`discarded_points` (*pyepal.pal.pal_base.PALBase* property), 15
`dominance_check()` (in module *pyepal.pal.utils*), 22
`dominance_check_jitted()` (in module *pyepal.pal.utils*), 22
`dominance_check_jitted_2()` (in module *pyepal.pal.utils*), 22
`dominance_check_jitted_3()` (in module *pyepal.pal.utils*), 22

E

`exhaust_loop()` (in module *pyepal.pal.utils*), 22

`exp_decay()` (in module *pyepal.pal.schedules*), 22

G

`get_hypervolume()` (in module *pyepal.pal.utils*), 23
`get_kmeans_samples()` (in module *pyepal.pal.utils*), 23
`get_matern_32_kernel()` (in module *pyepal.models.gpr*), 28
`get_matern_52_kernel()` (in module *pyepal.models.gpr*), 28
`get_maxmin_samples()` (in module *pyepal.pal.utils*), 23
`get_ratquad_kernel()` (in module *pyepal.models.gpr*), 28

H

`hyperrectangle_sizes` (*pyepal.pal.pal_base.PALBase* property), 15

I

`is_pareto_efficient()` (in module *pyepal.pal.utils*), 24

L

`linear()` (in module *pyepal.pal.schedules*), 22

M

`means` (*pyepal.pal.pal_base.PALBase* property), 15
module
pyepal.models.gpr, 28
pyepal.pal.core, 14
pyepal.pal.pal_base, 14
pyepal.pal.pal_coregionalized, 18
pyepal.pal.pal_gbdtd, 20
pyepal.pal.pal_gpflowgpr, 21
pyepal.pal.pal_gpy, 18
pyepal.pal.pal_sklearn, 19
pyepal.pal.schedules, 22
pyepal.pal.utils, 22
pyepal.pal.validate_inputs, 26
pyepal.plotting, 24

N

`number_design_points` (*pyepal.pal.pal_base.PALBase* property), 16

number_discarded_points
 (*pyepal.pal.pal_base.PALBase* *property*),
 16
 number_pareto_optimal_points
 (*pyepal.pal.pal_base.PALBase* *property*),
 16
 number_sampled_points
 (*pyepal.pal.pal_base.PALBase* *property*),
 16
 number_unclassified_points
 (*pyepal.pal.pal_base.PALBase* *property*),
 16

P

PALBase (*class in pyepal.pal.pal_base*), 14
 PALCoregionalized (*class in pyepal.pal.pal_coregionalized*), 18
 PALGBDT (*class in pyepal.pal.pal_gbd*), 20
 PALGPflowGPR (*class in pyepal.pal.pal_gpflowgpr*), 21
 PALGPy (*class in pyepal.pal.pal_gpy*), 18
 PALSklearn (*class in pyepal.pal.pal_sklearn*), 19
 pareto_optimal_indices
 (*pyepal.pal.pal_base.PALBase* *property*),
 16
 pareto_optimal_points
 (*pyepal.pal.pal_base.PALBase* *property*),
 16
 plot_bar_iterations() (*in module pyepal.plotting*),
 24
 plot_histogram() (*in module pyepal.plotting*), 24
 plot_jointplot() (*in module pyepal.plotting*), 25
 plot_pareto_front_2d() (*in module pyepal.plotting*),
 25
 plot_residuals() (*in module pyepal.plotting*), 25
 predict() (*in module pyepal.models.gpr*), 29
 predict_coregionalized() (*in module pyepal.models.gpr*), 29
 pyepal.models.gpr
 module, 28
 pyepal.pal.core
 module, 14
 pyepal.pal.pal_base
 module, 14
 pyepal.pal.pal_coregionalized
 module, 18
 pyepal.pal.pal_gbd
 module, 20
 pyepal.pal.pal_gpflowgpr
 module, 21
 pyepal.pal.pal_gpy
 module, 18
 pyepal.pal.pal_sklearn
 module, 19
 pyepal.pal.schedules
 module, 22
 pyepal.pal.utils
 module, 22
 pyepal.pal.validate_inputs
 module, 26
 pyepal.plotting
 module, 24

R

run_one_step() (*pyepal.pal.pal_base.PALBase*
 method), 16

S

sample() (*pyepal.pal.pal_base.PALBase* *method*), 16
 sampled_indices (*pyepal.pal.pal_base.PALBase* *prop-*
 erty), 17
 sampled_mask (*pyepal.pal.pal_base.PALBase* *property*),
 17
 sampled_points (*pyepal.pal.pal_base.PALBase* *prop-*
 erty), 17
 set_xy_coregionalized() (*in module pyepal.models.gpr*), 29
 should_cross_validate()
 (*pyepal.pal.pal_base.PALBase* *method*),
 17

U

unclassified_indices (*pyepal.pal.pal_base.PALBase*
 property), 17
 unclassified_points (*pyepal.pal.pal_base.PALBase*
 property), 17
 update_train_set() (*pyepal.pal.pal_base.PALBase*
 method), 17
 uses_fixed_epsilon (*pyepal.pal.pal_base.PALBase*
 property), 17

V

validate_beta_scale() (*in module pyepal.pal.validate_inputs*), 26
 validate_coef_var() (*in module pyepal.pal.validate_inputs*), 26
 validate_coregionalized_gpy() (*in module pyepal.pal.validate_inputs*), 26
 validate_delta() (*in module pyepal.pal.validate_inputs*), 26
 validate_epsilon() (*in module pyepal.pal.validate_inputs*), 26
 validate_gbd_models() (*in module pyepal.pal.validate_inputs*), 26
 validate_goals() (*in module pyepal.pal.validate_inputs*), 27
 validate_gpy_model() (*in module pyepal.pal.validate_inputs*), 27

`validate_interquartile_scaler()` (in *module*
 pyepal.pal.validate_inputs), 27

`validate_ndim()` (in *module*
 pyepal.pal.validate_inputs), 27

`validate_njobs()` (in *module*
 pyepal.pal.validate_inputs), 27

`validate_nt_models()` (in *module*
 pyepal.pal.validate_inputs), 27

`validate_number_models()` (in *module*
 pyepal.pal.validate_inputs), 27

`validate_optimizers()` (in *module*
 pyepal.pal.validate_inputs), 28

`validate_positive_integer_list()` (in *module*
 pyepal.pal.validate_inputs), 28

`validate_sklearn_gpr_models()` (in *module*
 pyepal.pal.validate_inputs), 28