MuyGPyS

Release beta

Benjamin W. Priest

May 03, 2024

PACKAGE DOCUMENTATION:

MuyGPyS is toolkit for training approximate Gaussian Process (GP) models using the MuyGPs (Muyskens, Goumiri, Priest, Schneider) algorithm.

CHAPTER

ONE

CITATION

If you use MuyGPyS in a research paper, please reference our article:

```
@article{muygps2021,
  title={MuyGPs: Scalable Gaussian Process Hyperparameter Estimation Using Local Cross-
↪Validation},
  author={Muyskens, Amanda and Priest, Benjamin W. and Goumiri, Im{\`e}ne and Schneider,␣
↪Michael},
  journal={arXiv preprint arXiv:2104.14581},
  year={2021}
}
```

1.1 neighbors

KNN lookup management

`MuyGPyS.neighbors.NN_Wrapper` is an api for tasking several KNN libraries with the construction of lookup indexes that empower fast training and inference. The wrapper constructor expects the training features, the number of nearest neighbors, and a method string specifying which algorithm to use, as well as any additional kwargs used by the methods. Currently supported implementations include exact KNN using sklearn ("exact") and approximate KNN using hnsw ("hnsw").

class MuyGPyS.neighbors.**NN_Wrapper**(*train*, *nn_count*, *nn_method='exact'*, ***kwargs*)

Nearest Neighbors lookup datastructure wrapper.

Wraps the logic driving nearest neighbor data structure training and querying. Currently supports `sklearn.neighbors.NearestNeighbors` for exact computation and `hnswlib.Index` for approximate nearest neighbors.

An example constructing exact and approximate KNN data lookups with k = 10.

Example

```
>>> from MuyGPyS.neighors import NN_Wrapper
>>> train_features = load_train_features()
>>> nn_count = 10
>>> exact_nbrs_lookup = NN_Wrapper(
...         train_features, nn_count, nn_method="exact", algorithm="ball_tree"
... )
>>> approx_nbrs_lookup = NN_Wrapper(
```

(continues on next page)

(continued from previous page)

```
...         train_features, nn_count, nn_method="hnsw", space="l2", M=16
... )
```

Parameters

- **train** (ndarray) – The full training data of shape `(train_count, feature_count)` that will construct the nearest neighbor query datastructure.
- **nn_count** (int) – The number of nearest neighbors to return in queries.
- **nn_method** (str) – Indicates which nearest neighbor algorithm should be used. Currently "exact" indicates `sklearn.neighbors.NearestNeighbors`, while "hnsw" indicates `hnswlib.Index` (requires installing MuyGPyS with the "hnswlib" extras flag).
- **kwargs** – Additional kwargs used for lookup data structure construction. `nn_method="exact"` supports "radius", "algorithm", "leaf_size", "metric", "p", "metric_params", and "n_jobs" kwargs. `nn_method="hnsw"` supports "space", "ef_construction", "M", and "random_seed" kwargs.

get_batch_nns(*batch_indices*)

Get the non-self nearest neighbors for indices into the training data.

Find the nearest neighbors and associated distances for each specified index into the training data.

Example

```
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> from numpy.random import choice
>>> train_features = load_train_features()
>>> nn_count = 10
>>> nbrs_lookup = NN_Wrapper(
...         train_features, nn_count, nn_method="exact", algorithm="ball_tree"
... )
>>> train_count, _ = train_features.shape
>>> batch_count = 50
>>> batch_indices = choice(train_count, batch_count, replace=False)
>>> nn_indices, nn_dists = nbrs_lookup.get_nns(batch_indices)
```

Parameters

batch_indices (ndarray) – Indices into the training data of shape `(batch_count,)`.

Return type

`Tuple[ndarray, ndarray]`

Returns

- *batch_nn_indices* – Matrix of nearest neighbor indices of shape `(batch_count, nn_count)`. Each row lists the nearest neighbor indices (self excluded) of the corresponding batch element.
- **batch_nn_dists** (*numpy.ndarray(int), shape=(batch_count, nn_count)*) – Matrix of distances of shape `(batch_count, nn_count)`. Each row lists the distance to the batch element of the corresponding element in `batch_nn_indices`.

get_nns(*test*)

Get the nearest neighbors for each row of `test` dataset.

Find the nearest neighbors and associated distances for each element of the given test dataset. Here we assume that the test dataset is distinct from the train dataset used in the construction of the nearest neighbor lookup data structure.

Example

```
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> train_features = load_train_features()
>>> test_features = load_test_features()
>>> nn_count = 10
>>> nbrs_lookup = NN_Wrapper(
...         train_features, nn_count, nn_method="exact", algorithm="ball_tree"
... )
>>> nn_indices, nn_dists = nbrs_lookup.get_nns(test_features)
```

Parameters

test (ndarray) – Testing data matrix of shape `(test_count, feature_count)`.

Return type

`Tuple[ndarray, ndarray]`

Returns

- *nn_indices* – Matrix of nearest neighbor indices of shape `(test_count, nn_count)`. Each row lists the nearest neighbor indices of the corresponding test element.
- *nn_dists* – Matrix of distances of shape `(test_count, nn_count)`. Each row lists the distance to the test element of the corresponding element in `nn_indices`.

1.2 gp

MuyGPyS.gp module reference.

1.2.1 deformation

MuyGPyS.gp.deformation module reference.

class MuyGPyS.gp.deformation.**Isotropy**(*metric*, *length_scale*)

An isotropic deformation model.

Isotropy defines a scaled elementwise distance function $d_e ll(\cdot, \cdot)$, and is paramterized by a scalar $\ell > 0$ length scale hyperparameter.

$$d_\ell(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{d} \frac{d(\mathbf{x}_i, \mathbf{y}_i)}{\ell}$$

Parameters

- **metric** (*MetricFn*) – A MetricFn object defining the behavior of the feature metric space.
- **length_scale** (*Parameter*) – Some scalar nonnegative hyperparameter object.

__call__(*dists*, *length_scale=None*, ***kwargs*)

Apply isotropic deformation to an elementwise difference tensor.

This function is not intended to be invoked directly by a user. It is instead functionally incorporated into some MuyGPyS.gp.kernels.KernelFn in its constructor.

Parameters

- **dists** (ndarray) – A tensor of distances between sets of observables.
- **length_scale** (Union[float, ndarray, None]) – A floating point length scale.

Return type
ndarray

Returns
A scaled distance matrix of the same shape as shape (data_count, nn_count) or a pairwise distance tensor of shape (data_count, nn_count, nn_count) whose last two dimensions are pairwise distance matrices.

crosswise_tensor(*data*, *nn_data*, *data_indices*, *nn_indices*, ***kwargs*)

Compute a crosswise distance tensor between data and their nearest neighbors.

Takes full datasets of records of interest data and neighbor candidates nn_data and produces a scalar distance between each element of data indicated by data_indices and each of the nearest neighbors in nn_data as indicated by the corresponding rows of nn_indices. data and nn_data can refer to the same dataset.

Parameters

- **data** (ndarray) – The data matrix of shape (data_count, feature_count) containing batch elements.
- **nn_data** (ndarray) – The data matrix of shape (candidate_count, feature_count) containing the universe of candidate neighbors for the batch elements. Might be the same as data.
- **indices** – An integral vector of shape (batch_count,) containing the indices of the batch.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape (batch_count, nn_count) whose second dimension indicates distance vectors between each batch element and its nearest neighbors.

pairwise_tensor(*data*, *nn_indices*, ***kwargs*)

Compute a pairwise distance tensor among sets of nearest neighbors.

Takes a full dataset of records of interest data and produces the pairwise distances between the elements indicated by each row of nn_indices.

Parameters

- **data** (ndarray) – The data matrix of shape (batch_count, feature_count) containing batch elements.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped pairwise nearest neighbor distance tensors corresponding to each of the batch elements.

class MuyGPyS.gp.deformation.**Anisotropy**(*metric*, *length_scale*)

An anisotropic deformation model.

Anisotropy parameterizes a scaled elementwise distance function $d_\ell(\cdot, \cdot)$, and is paramterized by a vector-valued $\ell > 0$ length scale hyperparameter.

$$d_\ell(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{d} \frac{d(\mathbf{x}_i, \mathbf{y}_i)}{\ell_i}$$

Parameters

- **metric** (*MetricFn*) – A MetricFn object defining the behavior of the feature metric space.
- **length_scales** – Keyword arguments `length_scale#`, mapping to scalar hyperparameters.

__call__(*dists*, ***length_scales*)

Apply anisotropic deformation to an elementwise difference tensor.

This function is not intended to be invoked directly by a user. It is instead functionally incorporated into some `MuyGPyS.gp.kernels.KernelFn` in its constructor.

Parameters

- **dists** (ndarray) – A tensor of pairwise differences of shape `(..., feature_count)` representing the difference in feature dimensions between sets of observables.
- **batch_features** – A `(batch_count, feature_count)` matrix of features to be used with a hierarchical hyperparameter. `None` otherwise.
- **length_scale** – A floating point length scale, or a vector of `(knot_count,)` knot length scales.

Return type
ndarray

Returns
A crosswise distance matrix of shape `(data_count, nn_count)` or a pairwise distance tensor of shape `(data_count, nn_count, nn_count)` whose last two dimensions are pairwise distance matrices.

crosswise_tensor(*data*, *nn_data*, *data_indices*, *nn_indices*, ***kwargs*)

Compute a crosswise difference tensor between data and their nearest neighbors.

Takes full datasets of records of interest `data` and neighbor candidates `nn_data` and produces a difference vector between each element of `data` indicated by `data_indices` and each of the nearest neighbors in `nn_data` as indicated by the corresponding rows of `nn_indices`. `data` and `nn_data` can refer to the same dataset.

Parameters

- **data** (ndarray) – The data matrix of shape `(data_count, feature_count)` containing batch elements.

- **nn_data** (ndarray) – The data matrix of shape (candidate_count, feature_count) containing the universe of candidate neighbors for the batch elements. Might be the same as data.
- **indices** – An integral vector of shape (batch_count,) containing the indices of the batch.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape (batch_count, nn_count, feature_count) whose last two dimensions indicate difference vectors between the feature dimensions of each batch element and those of its nearest neighbors.

pairwise_tensor(*data*, *nn_indices*, ***kwargs*)

Compute a pairwise difference tensor among sets of nearest neighbors.

Takes a full dataset of records of interest data and produces the pairwise differences for each feature dimension between the elements indicated by each row of nn_indices.

Parameters

- **data** (ndarray) – The data matrix of shape (batch_count, feature_count) containing batch elements.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape (batch_count, nn_count, nn_count, feature_count) containing the (nn_count, nn_count, feature_count)-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.

1.2.2 tensors

Tensor functions

Compute special tensors for the purposes of kernel construction.

MuyGPyS.gp.tensors.**batch_features_tensor**(*features*, *batch_indices*)

Compute a tensor of feature vectors for each batch element.

Parameters

- **features** (ndarray) – The full floating point training or testing data matrix of shape (train_count, feature_count) or (test_count, feature_count).
- **batch_indices** (ndarray) – A vector of integers of shape (batch_count,) identifying the training batch of observations to be approximated.

Return type
ndarray

Returns
A tensor of shape `(batch_count, feature_count)` containing the feature vectors for each batch element.

`MuyGPyS.gp.tensors.`**`fast_nn_update`**`(`*`train_nn_indices`*`)`

Modify the nearest neighbor indices of the training data to include self.

This function is only intended for use in concert with :func:~MuyGPyS.gp.tensors.make_fast_predict_tensors` and *MuyGPyS.gp.muygps.MuyGPS.fast_coefficients()*.

Example

```
>>> train_nn_indices, _ = nbrs_lookup.get_nns(train_features)
>>> train_nn_indices = fast_nn_update(train_nn_indices)
>>> pairwise_diffs, nn_targets = make_fast_predict_tensors(
...     train_nn_indices,
...     train_features,
...     train_responses,
... )
>>> Kin = muygps_fast.kernel(pairwise_diffs)
>>> precomputed_coefficients_matrix = muygps_fast.fast_coefficients(
...     Kin, nn_targets
... )
>>> # Late on, once test data is encountered
>>> test_indices = np.arange(test_count)
>>> test_nn_indices, _ = nbrs_lookup.get_nns(test_features)
>>> closest_neighbor = test_nn_indices[:, 0]
>>> closest_set = train_nn_indices[closest_neighbor, :]
```

Parameters
`train_nn_indices` (ndarray) – A matrix of integers of shape `(train_count, nn_count)` listing the nearest neighbor indices for all observations in the batch.

Return type
ndarray

Returns
An integral matrix of shape `(train_count, nn_count)` that is similar to the input, but the most distant neighbor index is removed and the index reference to self has been inserted.

`MuyGPyS.gp.tensors.`**`make_fast_predict_tensors`**`(`*`batch_nn_indices`*`, `*`train_features`*`, `*`train_targets`*`)`

Create the difference and target tensors for fast posterior mean inference.

Creates `pairwise_diffs` and `batch_nn_targets` tensors required by *MuyGPyS.gp.muygps.MuyGPS.fast_posterior_mean()*.

Parameters

- **`batch_nn_indices`** (ndarray) – A matrix of integers of shape `(batch_count, nn_count)` listing the nearest neighbor indices for all observations in the batch.
- **`train_features`** (ndarray) – The full floating point training data matrix of shape `(train_count, feature_count)`.
- **`train_targets`** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows are vector-valued responses for each training element.

Return type
Tuple[ndarray, ndarray]

Returns

- *pairwise_diffs* – A tensor of shape (batch_count, nn_count, nn_count, feature_count) containing the (nn_count, nn_count, feature_count)-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- *batch_nn_targets* – Tensor of floats of shape (batch_count, nn_count, response_count) containing the expected response for each nearest neighbor of each batch element.

MuyGPyS.gp.tensors.**make_heteroscedastic_tensor**(*measurement_noise*, *batch_nn_indices*)

Create the heteroscedastic noise tensor for nonuniform noise values.

Used to produce the noise tensor needed during batched training and prediction. Creates the noise_tensor tensor required by heteroscedastic MuyGPs models.

Parameters

- **measurement_noise** (ndarray) – A matrix of floats of shape (batch_count,) providing the noise corresponding to the response variable at each input value in the data.
- **batch_nn_indices** (ndarray) – A matrix of integers of shape (batch_count, nn_count, nn_count) listing the measurement noise for the nearest neighbors for all observations in the batch.

Return type
ndarray

Returns
A matrix of floats of shape (batch_count, nn_count) providing the noise corresponding to the nearest neighbor responses for all observations in the batch.

1.2.3 hyperparameter

MuyGPyS.gp.hyperparameters module reference.

class MuyGPyS.gp.hyperparameter.scalar.**Parameter**(*val*, *bounds='fixed'*)

A MuyGPs kernel or model Hyperparameter. Also called ScalarParam.

Hyperparameters are defined by a value and optimization bounds. Values must be scalar numeric types, and bounds are either a len == 2 iterable container whose elements are numeric scalars in increasing order, or the string fixed. If bounds == "fixed" (the default behavior), the hyperparameter value will remain fixed during optimization. val must remain within the range of the upper and lower bounds, if not fixed.

Parameters

- **val** (Union[str, float]) – A scalar within the range of the upper and lower bounds (if given). val can also be the strings "sample" or "log_sample", which will result in randomly sampling a value within the range given by the bounds.
- **bounds** (Union[str, Tuple[float, float]]) – Iterable container of len 2 containing lower and upper bounds (in that order), or the string "fixed".

Raises

- **ValueError** – Any bounds string other than "fixed" will produce an error.
- **ValueError** – A non-iterable non-string type for bounds will produce an error.

- **ValueError** – A bounds iterable of len other than 2 will produce an error.
- **ValueError** – Iterable bounds values of non-numeric types will produce an error.
- **ValueError** – A lower bound that is not less than an upper bound will produce an error.
- **ValueError** – val == "sample" or val == "log_sample" will produce an error if self._bounds == "fixed".
- **ValueError** – Any string other than "sample" or "log_sample" will produce an error.
- **ValueError** – A val outside of the range specified by self._bounds will produce an error.

__call__(***kwargs*)

Value accessor.

Return type
float

Returns
The current value of the hyperparameter.

fixed()

Report whether the parameter is fixed, and is to be ignored during optimization.

Return type
bool

Returns
True if fixed, False otherwise.

get_bounds()

Bounds accessor.

Return type
Tuple[float, float]

Returns
The lower and upper bound tuple.

class MuyGPyS.gp.hyperparameter.tensor.**TensorParam**(*val*)

A MuyGPs kernel or model Tensor Hyperparameter.

TensorParam are defined solely by a value, which must be numeric arrays. Currently only used for heteroscedastic noise.

Parameters
val (ndarray) – A mm.ndarray containing the value of the tensor hyperparameter

__call__()

Value accessor.

Return type
ndarray

Returns
The current value of the tensor hyperparameter.

fixed()

Report whether the parameter is fixed, and is to be ignored during optimization. Always returns True for tensor hyperparameters.

Return type
bool

Returns
True.

class MuyGPyS.gp.hyperparameter.scale.**ScaleFn**(*val=1.0*, ***kwargs*)

A σ^2 covariance scale parameter base functor.

σ^2 is a scaling parameter that one multiplies with the found diagonal variances of a *MuyGPyS.gp.muygps.MuyGPS* regression in order to obtain the predicted posterior variance. Trained values assume a number of dimensions equal to the number of response dimensions, and correspond to scalar scaling parameters along the corresponding dimensions.

Parameters
val (float) – A floating point value, if intended to be set manually. Defaults to 1.0.

__call__()

Value accessor.

Return type
ndarray

Returns
The current value of the hyperparameter.

scale_fn(*fn*)

Modify a function to outer product its output with scale.

Parameters
fn (Callable) – A function.

Return type
Callable

Returns
A function that returns the outer product of the output of fn

property trained: bool

Report whether the value has been set.

Returns
True if trained, False otherwise.

class MuyGPyS.gp.hyperparameter.scale.**FixedScale**(*val=1.0*, ***kwargs*)

A σ^2 covariance scale parameter.

A Scale parameter with a null optimization method. This parameter is therefore insensitive to optimization.

Parameters
response_count – The integer number of response dimensions.

get_opt_fn(*muygps*)

Return a function that optimizes the value of the variance scale.

Parameters
muygps – A model to be ignored.

Return type
Callable

Returns
A function that always returns the value of this scale parameter.

class MuyGPyS.gp.hyperparameter.scale.**AnalyticScale**(*iteration_count=1*, *_backend_fn=<function _analytic_scale_optim>*, ***kwargs*)

An optimizable σ^2 covariance scale parameter.

Identical to `FixedScale`, save that its `get_opt_fn` method performs an analytic optimization.

Parameters

- **response_count** – The integer number of response dimensions.
- **iteration_count** (`int`) – The number of iterations to run during optimization.

get_opt_fn(*muygps*)

Get a function to optimize the value of the σ^2 scale parameter for each response dimension.

We approximate a scalar σ^2 by way of averaging over the analytic solution from each local kernel. Given observations X with responses Y, noise model ε, and kernel function $Kin_\theta(\cdot,\cdot)$, computes:

$$\sigma^2 = \frac{1}{bk} * \sum_{i \in B} Y(X_{N_i})^T \left(Kin_\theta(X_{N_i}, X_{N_i}) + \varepsilon_{N_i}\right)^{-1} Y(X_{N_i}).$$

Here N_i is the set of nearest neighbor indices of the $i`thbatchelement, : math : `k$ is the number of nearest neighbors and $b = |B|$ is the number of batch elements considered.

Parameters
muygps – The model to used to create and perturb the kernel.

Return type
`Callable`

Returns
A function with signature `(Kin, nn_targets, *args, **kwargs) -> mm.ndarray` that perturbs the `(batch_count, nn_count, nn_count)` tensor `Kin` with `muygps`'s noise model before solving it against the `(batch_count, nn_count, response_count)` tensor `nn_targets`.

1.2.4 kernels

Hyperparameters and kernel functors

Defines kernel functors (inheriting *KernelFn*) that transform crosswise difference tensors into cross-covariance matrices and pairwise difference matrices into covariance or kernel tensors.

See the following example to initialize an `MuyGPyS.gp.kernels.Matern` object. Other kernel functors are similar, but require different hyperparameters.

Example

```
>>> from MuyGPyS.gp.kernels import Matern
>>> kern = Matern(
...     smoothness=Parameter("log_sample", (0.1, 2.5)),
...     deformation=Isotropy(
...         metric=l2,
...         length_scale=Parameter(1.0),
...     ),
... )
```

One uses a previously computed *pairwise_diffs* tensor (see `MuyGPyS.gp.tensor.pairwise_tensor()`) to compute a kernel tensor whose second two dimensions contain square kernel matrices. Similarly, one uses a previously computed *crosswise_diffs* matrix (see `MuyGPyS.gp.tensor.crosswise_diffs()`) to compute a cross-covariance matrix. See the following example, which assumes that you have already constructed the difference *numpy.ndarrays* and the kernel *kern* as shown above.

Example

```
>>> Kin = kern(pairwise_diffs)
>>> Kcross = kern(crosswise_diffs)
```

class MuyGPyS.gp.kernels.kernel_fn.**KernelFn**(*deformation*)

Bases: `object`

A kernel functor.

Base class for kernel functors that include a hyperparameter Dict and a call mechanism.

Parameters
kwargs – Ignored (by this base class) keyword arguments.

__call__(*diffs*, ***kwargs*)

Call self as a function.

Return type
`ndarray`

get_opt_params()

Report lists of unfixed hyperparameter names, values, and bounds.

Return type
`Tuple[List[str], List[float], List[Tuple[float, float]]]`

Returns

names:
A list of unfixed hyperparameter names.

params:
A list of unfixed hyperparameter values.

bounds:
A list of unfixed hyperparameter bound tuples.

set_params(***kwargs*)

Reset hyperparameters using hyperparameter dict(s).

Parameters
kwargs – Hyperparameter kwargs.

Return type
`None`

class MuyGPyS.gp.kernels.rbf.**RBF**(*deformation=<MuyGPyS.gp.deformation.isotropy.Isotropy object>*, *_backend_fn=<function _rbf_fn>*, *_backend_ones=<function fix_type.<locals>.typed_fn.<locals>.fn_wrapper>*, *_backend_zeros=<function fix_type.<locals>.typed_fn.<locals>.fn_wrapper>*, *_backend_squeeze=<function squeeze>*)

The radial basis function (RBF) or squared-exponential kernel.

The RBF kernel includes a parameterized scaled distance function $d_\ell(\cdot, \cdot)$.

The kernel is defined by

$$Kin(x_i, x_j) = \exp\left(-d_\ell(x_i, x_j)\right).$$

Typically, $d(\cdot, \cdot)$ is the squared Euclidean distance or second frequency moment of the difference of the operands.

Parameters
deformation (DeformationFn) – The deformation functor to be used. Includes length_scale hyperparameter information via the MuyGPyS.gp.deformation module

__call__(*diffs*, ***kwargs*)

Compute RBF kernel(s) from a difference tensor.

Parameters
diffs (ndarray) – A tensor of pairwise diffs of shape (data_count, nn_count, nn_count, feature_count) or (data_count, nn_count, feature_count). In the four dimensional case, it is assumed that the diagonals dists diffs[i, j, j, :] == 0.

Return type
ndarray

Returns
A cross-covariance matrix of shape (data_count, nn_count) or a tensor of shape (data_count, nn_count, nn_count) whose last two dimensions are kernel matrices.

get_opt_fn()

Return a kernel function with fixed parameters set.

Assumes that optimization parameter literals will be passed as keyword arguments.

Return type
Callable

Returns
A function implementing the kernel where all fixed parameters are set. The function expects keyword arguments corresponding to current hyperparameter values for unfixed parameters.

class MuyGPyS.gp.kernels.matern.**Matern**(*smoothness=<MuyGPyS.gp.hyperparameter.scalar.Parameter object>*, *deformation=<MuyGPyS.gp.deformation.isotropy.Isotropy object>*, *_backend_ones=<function fix_type.<locals>.typed_fn.<locals>.fn_wrapper>*, *_backend_zeros=<function fix_type.<locals>.typed_fn.<locals>.fn_wrapper>*, *_backend_squeeze=<function squeeze>*, ***_backend_fns*)

The Matérn kernel.

The Màtern kernel includes a parameterized deformation model $d_\ell(\cdot, \cdot)$ and an additional smoothness parameter $\nu > 0$. ν is proportional to the smoothness of the resulting function. As $\nu \to \infty$, the kernel becomes equivalent to the RBF kernel. When $\nu = 1/2$, the Matérn kernel is identical to the absolute exponential kernel. Important intermediate values are $\nu = 1.5$ (once differentiable functions) and $\nu = 2.5$ (twice differentiable functions).

The kernel is defined by

$$k(x_i, x_j) = \frac{1}{\Gamma(\nu)2^{\nu-1}}\left(\frac{\sqrt{2\nu}}{l} d_\ell(x_i, x_j)\right)^\nu Kin_\nu\left(\frac{\sqrt{2\nu}}{l} d(x_i, x_j)\right),$$

where $Kin_\nu(\cdot)$ is a modified Bessel function and $\Gamma(\cdot)$ is the gamma function. Typically, $d(\cdot, \cdot)$ is the Euclidean distance or ℓ_2 norm of the difference of the operands.

Parameters

- **smoothness** (*Parameter*) – A parameter determining the differentiability of the function distribution.
- **deformation** (DeformationFn) – The deformation functor to be used. Includes length_scale hyperparameter information via the MuyGPyS.gp.deformation module.

__call__(*diffs*, ***kwargs*)

Compute Matern kernels from distance tensor.

Takes inspiration from scikit-learn's implementation.

Parameters
diffs – A tensor of pairwise differences of shape (data_count, nn_count, nn_count, feature_count). It is assumed that the vectors along the diagonals diffs[i, j, j, :] == 0.

Returns
A cross-covariance matrix of shape (data_count, nn_count) or a tensor of shape (data_count, nn_count, nn_count) whose last two dimensions are kernel matrices.

get_opt_fn()

Return a kernel function with fixed parameters set.

Assumes that optimization parameter literals will be passed as keyword arguments.

Return type
Callable

Returns
A function implementing the kernel where all fixed parameters are set. The function expects keyword arguments corresponding to current hyperparameter values for unfixed parameters.

get_opt_params()

Report lists of unfixed hyperparameter names, values, and bounds.

Return type
Tuple[List[str], List[float], List[Tuple[float, float]]]

Returns

names:
A list of unfixed hyperparameter names.

params:
A list of unfixed hyperparameter values.

bounds:
A list of unfixed hyperparameter bound tuples.

1.2.5 metric

Metric Function Handling

MuyGPyS includes predefined metric functions with convenience functions for interacting with the rest of the library.

MuyGPyS.gp.deformation.metric.**F2 = <MuyGPyS.gp.deformation.metric.MetricFn object>**

F2 or squared Euclidean metric function.

Computes the Euclidean distance between points:

$$d_{F_2}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} (x_i - y_i)^2$$

Parameters
dists – A difference tensor of shape `(..., feature_count)`.

Returns
A distance tensor of shape `(...)`.

class MuyGPyS.gp.deformation.metric.**MetricFn**(*differences_metric_fn*, *crosswise_differences_fn*, *pairwise_diffferences_fn*, *apply_length_scale_fn*)

Metric functor class.

MuyGPyS-compatible metric functions are objects of this class. Creating a new metric function is as simple as instantiating a new `MetricFn` object with the desired behavior.

Parameters

- **differences_metric_fn** (`Callable`) – A Callable taking an ndarray of feature-wise dimensional comparisons with shape `(..., feature_count)` that collapses the last dimension into scalar distances.
- **crosswise_distances_fn** – A Callable of signature `(data, nn_data, data_indices, nn_indices) -> distances` that produces a crosswise distance tensor between data and their nearest neighbors.
- **crosswise_differences_fn** (`Callable`) – A Callable of signature `(data, nn_data, data_indices, nn_indices) -> differences` that produces a feature dimension-wise crosswise differences tensor between data and their nearest neighbors.
- **pairwise_distances_fn** – A Callable of signature `(data, nn_indices) -> distances` that produces a pairwise distance tensor among sets of nearest neighbors.
- **pairwise_differences_fn** – A Callable of signature `(data, nn_data) -> differences` that produces a feature dimension-wise pairwise differences tensor among sets of nearest neighbors.
- **apply_length_scale_fn** (`Callable`) – A Callable of signature `(dists) -> dists` that applies a length scale parameter appropriately to a distances tensor.

apply_length_scale(*dists*, *length_scale*)

Compute a pairwise distance tensor among sets of nearest neighbors.

Takes a full dataset of records of interest `data` and produces the pairwise distances between the elements indicated by each row of `nn_indices`.

Parameters
dists (`ndarray`) – A distance tensor of any shape.

Return type
`ndarray`

Returns
A tensor of the same shape that has been element-wise scaled by the provided length scale as befits the metric.

`crosswise_differences`(*data*, *nn_data*, *data_indices*, *nn_indices*, ***kwargs*)

Compute a crosswise difference tensor between data and their nearest neighbors.

Takes full datasets of records of interest `data` and neighbor candidates `nn_data` and produces a difference vector between each element of `data` indicated by `data_indices` and each of the nearest neighbors in `nn_data` as indicated by the corresponding rows of `nn_indices`. `data` and `nn_data` can refer to the same dataset.

Parameters

- **data** (ndarray) – The data matrix of shape `(data_count, feature_count)` containing batch elements.
- **nn_data** (ndarray) – The data matrix of shape `(candidate_count, feature_count)` containing the universe of candidate neighbors for the batch elements. Might be the same as `data`.
- **indices** – An integral vector of shape `(batch_count,)` containing the indices of the batch.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape `(batch_count, nn_count, feature_count)` whose last two dimensions indicate difference vectors between the feature dimensions of each batch element and those of its nearest neighbors.

`crosswise_distances`(*data*, *nn_data*, *data_indices*, *nn_indices*, ***kwargs*)

Compute a crosswise distance tensor between data and their nearest neighbors.

Takes full datasets of records of interest `data` and neighbor candidates `nn_data` and produces a scalar distance between each element of `data` indicated by `data_indices` and each of the nearest neighbors in `nn_data` as indicated by the corresponding rows of `nn_indices`. `data` and `nn_data` can refer to the same dataset.

Parameters

- **data** (ndarray) – The data matrix of shape `(data_count, feature_count)` containing batch elements.
- **nn_data** (ndarray) – The data matrix of shape `(candidate_count, feature_count)` containing the universe of candidate neighbors for the batch elements. Might be the same as `data`.
- **indices** – An integral vector of shape `(batch_count,)` containing the indices of the batch.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape `(batch_count, nn_count)` whose second dimension indicates distance vectors between each batch element and its nearest neighbors.

pairwise_differences(*data*, *nn_indices*, ***kwargs*)

Compute a pairwise difference tensor among sets of nearest neighbors.

Takes a full dataset of records of interest `data` and produces the pairwise differences for each feature dimension between the elements indicated by each row of `nn_indices`.

Parameters

- **data** (ndarray) – The data matrix of shape `(batch_count, feature_count)` containing batch elements.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.

pairwise_distances(*data*, *nn_indices*, ***kwargs*)

Compute a pairwise distance tensor among sets of nearest neighbors.

Takes a full dataset of records of interest `data` and produces the pairwise distances between the elements indicated by each row of `nn_indices`.

Parameters

- **data** (ndarray) – The data matrix of shape `(batch_count, feature_count)` containing batch elements.
- **nn_indices** (ndarray) – An integral matrix of shape (batch_count, nn_count) listing the nearest neighbor indices for the batch of data points.

Return type
ndarray

Returns
A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped pairwise nearest neighbor distance tensors corresponding to each of the batch elements.

MuyGPyS.gp.deformation.metric.**l2 = <MuyGPyS.gp.deformation.metric.MetricFn object>**

l2 or Euclidean metric function.

Computes the Euclidean distance between points:

$$d_{\ell_2}(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^2 \right)^{1/2}$$

Parameters
dists – A difference tensor of shape `(..., feature_count)`.

Returns
A distance tensor of shape `(...)`.

1.2.6 noise

class MuyGPyS.gp.noise.homoscedastic.**HomoscedasticNoise**(*val*, *bounds='fixed'*, *_backend_fn=<function _homoscedastic_perturb>*)

A scalar prior noise parameter.

A homoscedastic noise parameter used to build the "nugget" with the prior assumption that all observations are subject to i.i.d. unbiased Gaussian noise. Can be set at initialization time or left subject to optimization, in which case (positive) bounds are specified.

Parameters

- **val** (Union[str, float]) – A positive scalar, or the strings "sample" or "log_sample".
- **bounds** (Union[str, Tuple[float, float]]) – Iterable container of len 2 containing positive lower and upper bounds (in that order), or the string "fixed".

Raises

ValueError – Any nonpositive bounds string will produce an error.

perturb(*Kin*, *noise=None*, ***kwargs*)

Perturb a kernel tensor with homoscedastic noise.

Applies a homoscedastic noise model to a kernel tensor, whose last two dimensions are assumed to be the same length. For each such square submatrix Kin, computes the form $Kin + \tau^2 * I$, where τ^2 is the shared noise prior variance and I is the conforming identity matrix.

Parameters

- **Kin** (ndarray) – A tensor of shape (batch_count, nn_count, nn_count) containing the (nn_count, nn_count)-shaped kernel matrices corresponding to each of the batch elements.
- **noise** (Optional[float]) – A floating-point value for the noise variance prior, or None. None prompts the use of the stored value, whereas supplying alternative values is employed during optimization.

Return type

ndarray

Returns

A tensor of shape (batch_count, nn_count, nn_count) where the final two dimensions consist of the perturbed matrices of the input Kin.

perturb_fn(*fn*)

Perturb a function of kernel tensors with homoscedastic noise.

Applies a homoscedastic noise model to the first argument of the given function, which is assumed to be a kernel tensor whose last two dimensions are the same length. The returned function is the same as the input, save that it perturbs any passed kernel tensors.

Parameters

fn (Callable) – A callable whose first argument is assumed to be a tensor of shape (batch_count, nn_count, nn_count) containing the (nn_count, nn_count)-shaped kernel matrices corresponding to each of the batch elements.

Return type

Callable

Returns
A Callable with the same signature that applies a homoscedastic perturbation to its first argument. Also adds a `noise` keyword argument that is only used for optimization.

`class MuyGPyS.gp.noise.null.NullNoise(*args, **kwargs)`

A zero noise assumption model.

`perturb(Kin, **kwargs)`

Null noise perturbation.

Simply returns the input tensor unchanged.

Parameters
Kin (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped kernel matrices corresponding to each of the batch elements.

Return type
ndarray

Returns
The same tensor.

`class MuyGPyS.gp.noise.heteroscedastic.HeteroscedasticNoise(val, _backend_fn=<function _heteroscedastic_perturb>)`

A tensor noise parameter.

A heteroscedastic noise tensor used to build the "nugget" with the prior assumption that all observations are have a corresponding measurement noise prior variance.

Parameters
val (ndarray) – An ndarray of shape `(batch_count, nn_count)` containing the heteroscedastic nugget matrix.

Raises
ValueError – Any strictly negative entry in the array will produce an error.

`fixed()`

Overloading fixed function to return True for heteroscedastic noise.

Return type
bool

Returns
`True` - we do not allow optimizing Heteroscedastic Noise.

`perturb(Kin, **kwargs)`

Perturb a kernel tensor with heteroscedastic noise.

Applies a heteroscedastic noise model to a kernel tensor, whose last two dimensions are assumed to be the same length. For each such square submatrix Kin, computes the form $Kin + D$, where D is the diagonal matrix containing the observation-wise noise priors.

Parameters
Kin (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped kernel matrices corresponding to each of the batch elements.

Return type
ndarray

Returns
A tensor of shape `(batch_count, nn_count, nn_count)` where the final two dimensions consist of the perturbed matrices of the input Kin.

`perturb_fn`(*fn*)

Perturb a function of kernel tensors with heteroscedastic noise.

Applies a heteroscedastic noise model to the first argument of the given function, which is assumed to be a kernel tensor whose last two dimensions are the same length. The returned function is the same as the input, save that it perturbs any passed kernel tensors.

Parameters
fn (`Callable`) – A callable whose first argument is assumed to be a tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped kernel matrices corresponding to each of the batch elements.

Return type
`Callable`

Returns
A Callable with the same signature that applies a homoscedastic perturbation to its first argument.

1.2.7 MuyGPS

`class` MuyGPyS.gp.muygps.**MuyGPS**(*kernel*, *noise=<MuyGPyS.gp.noise.homoscedastic.HomoscedasticNoise object>*, *scale=<MuyGPyS.gp.hyperparameter.scale.FixedScale object>*, *_backend_mean_fn=<function _muygps_posterior_mean>*, *_backend_var_fn=<function _muygps_diagonal_variance>*, *_backend_fast_mean_fn=<function _muygps_fast_posterior_mean>*, *_backend_fast_precompute_fn=<function _muygps_fast_posterior_mean_precompute>*)

Local Kriging Gaussian Process.

Performs approximate GP inference by locally approximating an observation's response using its nearest neighbors. Implements the MuyGPs algorithm as articulated in [muyskens2021muygps].

Kernels accept different hyperparameter dictionaries specifying hyperparameter settings. Keys can include `val` and `bounds`. `bounds` must be either a len == 2 iterable container whose elements are scalars in increasing order, or the string `fixed`. If `bounds == fixed` (the default behavior), the hyperparameter value will remain fixed during optimization. `val` must be either a scalar (within the range of the upper and lower bounds if given) or the strings `"sample"` or `log_sample"`, which will randomly sample a value within the range given by the bounds.

In addition to individual kernel hyperparamters, each MuyGPS object also possesses a noise model, possibly with parameters, and a vector of σ^2 indicating the scale parameter associated with the posterior variance of each dimension of the response.

Example

```
>>> from MuyGPyS.gp import MuyGPS
>>> muygps = MuyGPS(
...     kernel=Matern(
...         smoothness=Parameter(0.38, (0.1, 2.5)),
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(0.2),
...         ),
...     ),
...     noise=HomoscedasticNoise(1e-5),
...     scale=AnalyticScale(),
... )
```

MuyGPyS depends upon linear operations on specially-constructed tensors in order to efficiently estimate GP realizations. One can use (see their documentation for details) `MuyGPyS.gp.tensors.pairwise_tensor()` to construct pairwise difference tensors and `MuyGPyS.gp.tensors.crosswise_tensor()` to produce crosswise diff tensors that `MuyGPS` can then use to construct kernel tensors and cross-covariance matrices, respectively.

We can easily realize kernel tensors using a `MuyGPS` object's `kernel` functor once we have computed a `pairwise_diffs` tensor and a `crosswise_diffs` matrix.

Example

```
>>> Kin = muygps.kernel(pairwise_diffs)
>>> Kcross = muygps.kernel(crosswise_diffs)
```

Parameters

- **kernel** (*KernelFn*) – The kernel to be used. Defines $Kin_\theta(\cdot,\cdot)$ as referenced in `MuyGPS` functions.
- **noise** (`NoiseFn`) – A noise model. Defines ε as referenced in `MuyGPS` functions.
- **scale** (*ScaleFn*) – A variance scale parameter. Defines σ^2 as referenced in `MuyGPS` functions.

fast_coefficients(*Kin*, *train_nn_targets_fast*)

Produces coefficient matrix for the fast posterior mean given in Equation (8) of [dunton2022fast].

Given observation set X with responses Y, noise prior set ε, and kernel function $Kin_\theta(\cdot,\cdot)$, computes the following for each observation element $\mathbf{x}_i$ with nearest neighbors index set N_i^*, containing `i` and the indices of the `nn_count - 1` nearest neighbors of $\mathbf{x}_i$:

$$C_i = (Kin_\theta(X_{N_i}, X_{N_i}) + \varepsilon_{N_i})^{-1} Y(X_{N_i}).$$

Parameters

- **Kin** (`ndarray`) – A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped kernel matrices corresponding to each of the batch elements.
- **Kcross** – A matrix of shape `(batch_count, nn_count)` whose rows consist of `(1, nn_count)`-shaped cross-covariance vector corresponding to each of the batch elements and its nearest neighbors.

Return type
ndarray

Returns
A matrix C of shape `(train_count, nn_count)` whose rows are the precomputed coefficients for fast posterior mean inference.

fast_posterior_mean(*Kcross*, *coeffs_tensor*)

Performs fast posterior mean inference using provided cross-covariance and precomputed coefficient matrix.

Assumes that cross-covariance matrix `Kcross` is already computed and given as an argument.

Returns the predicted response in the form of a posterior mean for each element of the batch of observations, as computed in Equation (9) of [dunton2022fast]. Given the coefficients C created by *fast_coefficients()* and Equation (8) of [dunton2022fast], observation set X, noise prior set ε, and kernel function $Kin_\theta(\cdot,\cdot)$, computes the following for each test point $\mathbf{z}$ and index set N_i^* containing the union of the index i of the nearest neighbor $\mathbf{x}_i$ of $\mathbf{z}$ and the `nn_count - 1` nearest neighbors of $\mathbf{x}_i$:

$$\widehat{Y}(\mathbf{z} \mid X) = \sigma^2 Kin_\theta(\mathbf{z}, X_{N_i^*}) C_i.$$

Parameters

- **Kcross** (ndarray) – A matrix of shape `(batch_count, nn_count)` whose rows consist of `(1, nn_count)`-shaped cross-covariance vector corresponding to each of the batch elements and its nearest neighbors.
- **coeffs_tensor** (ndarray) – A matrix of shape `(batch_count, nn_count, response_count)` whose rows are given by precomputed coefficients.

Return type
ndarray

Returns
A matrix of shape `(batch_count, response_count)` whose rows are the predicted response for each of the given indices.

fixed()

Checks whether all kernel and model parameters are fixed.

This is a convenience utility to determine whether optimization is required.

Return type
bool

Returns
Returns `True` if all parameters are fixed, and `False` otherwise.

get_opt_mean_fn()

Return a posterior mean function for use in optimization.

Assumes that optimization parameter literals will be passed as keyword arguments.

Return type
Callable

Returns
A function implementing the posterior mean, where `noise` is either fixed or takes updating values during optimization. The function expects keyword arguments corresponding to current hyperparameter values for unfixed parameters.

`get_opt_params()`

Return lists of unfixed hyperparameter names, values, and bounds.

Return type
`Tuple[List[str], ndarray, ndarray]`

Returns

names:
A list of unfixed hyperparameter names.

params:
A list of unfixed hyperparameter values.

bounds:
A list of unfixed hyperparameter bound tuples.

`get_opt_var_fn()`

Return a posterior variance function for use in optimization.

Assumes that optimization parameter literals will be passed as keyword arguments.

Return type
`Callable`

Returns
A function implementing posterior variance, where `noise` is either fixed or takes updating values during optimization. The function expects keyword arguments corresponding to current hyperparameter values for unfixed parameters.

`make_predict_tensors(`*batch_indices*, *batch_nn_indices*, *test_features*, *train_features*, *train_targets*, ***kwargs*`)`

Create the metric and target tensors for prediction using the model's deformation.

Creates the `crosswise_tensor`, `pairwise_tensor` and `batch_nn_targets` tensors required by `posterior_mean()` and `posterior_variance()`.

Parameters

- **batch_indices** (`ndarray`) – A vector of integers of shape `(batch_count,)` identifying the training batch of observations to be approximated.
- **batch_nn_indices** (`ndarray`) – A matrix of integers of shape `(batch_count, nn_count)` listing the nearest neighbor indices for all observations in the batch.
- **test_features** (`Optional[ndarray]`) – The full floating point testing data matrix of shape `(test_count, feature_count)`.
- **train_features** (`ndarray`) – The full floating point training data matrix of shape `(train_count, ...)`.
- **train_targets** (`ndarray`) – A matrix of shape `(train_count, ...)` whose rows are vector-valued responses for each training element.

Return type
`Tuple[ndarray, ndarray, ndarray]`

Returns

- *crosswise_tensor* – A tensor of shape `(batch_count, nn_count, ...)` whose second and subsequent dimensions list the metric comparison between each batch element element and its nearest neighbors.

- *pairwise_diffs* – A tensor of shape `(batch_count, nn_count, nn_count, ...)` containing the `(nn_count, nn_count, ...)`-shaped pairwise nearest neighbor metrics tensors corresponding to each of the batch elements.
- *batch_nn_targets* – Tensor of floats of shape `(batch_count, nn_count, ...)` containing the expected response for each nearest neighbor of each batch element.

make_train_tensors(*batch_indices*, *batch_nn_indices*, *train_features*, *train_targets*, ***kwargs*)

Create the metric and target tensors needed for training.

Similar to *make_predict_tensors()* but returns the additional `batch_targets` matrix, which is only defined for a batch of training data.

Parameters

- **batch_indices** (ndarray) – A vector of integers of shape `(batch_count,)` identifying the training batch of observations to be approximated.
- **batch_nn_indices** (ndarray) – A matrix of integers of shape `(batch_count, nn_count)` listing the nearest neighbor indices for all observations in the batch.
- **train_features** (ndarray) – The full floating point training data matrix of shape `(train_count, ...)`.
- **train_targets** (ndarray) – A matrix of shape `(train_count, ...)` whose rows are vector-valued responses for each training element.

Return type
`Tuple[ndarray, ndarray, ndarray, ndarray]`

Returns

- *crosswise_tensor* – A tensor of shape `(batch_count, nn_count, ...)` whose second and subsequent dimensions list the metric comparison between each batch element element and its nearest neighbors.
- *pairwise_diffs* – A tensor of shape `(batch_count, nn_count, nn_count, ...)` containing the `(nn_count, nn_count, ...)`-shaped pairwise nearest neighbor metrics tensors corresponding to each of the batch elements.
- *batch_targets* – Matrix of floats of shape `(batch_count, ...)` whose rows give the expected response for each batch element.
- *batch_nn_targets* – Tensor of floats of shape `(batch_count, nn_count, ...)` containing the expected response for each nearest neighbor of each batch element.

optimize_scale(*pairwise_diffs*, *nn_targets*)

Optimize the value of the $sigma^2$ scale parameter.

Uses the optimization method specified by the type of the `scale` parameter to optimize its value.

Parameters

- **pairwise_diffs** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **nn_targets** (ndarray) – Tensor of floats of shape `(batch_count, nn_count, response_count)` containing the expected response for each nearest neighbor of each batch element.

Returns
A reference to this model with a freshly-optimized `scale` parameter.

posterior_mean(*Kin*, *Kcross*, *batch_nn_targets*)

Returns the posterior mean from the provided covariance, cross-covariance, and target tensors.

Computes parallelized local solves of systems of linear equations using the last two dimensions of `Kin` along with `Kcross` and `batch_nn_targets` to predict responses in terms of the posterior mean. Assumes that kernel tensor `Kin` and cross-covariance matrix `Kcross` are already computed and given as arguments.

Returns the predicted response in the form of a posterior mean for each element of the batch of observations, as computed in Equation (3.4) of [muyskens2021muygps]. Given observation set X with responses Y, noise prior set ε, and kernel function $Kin_\theta(\cdot,\cdot)$, computes the following for each prediction element $\mathbf{z}_i$ with nearest neighbors index set N_i:

$$\widehat{Y}(\mathbf{z}_i \mid X_{N_i}) = \sigma^2 Kin_\theta(\mathbf{z}_i, X_{N_i})(Kin_\theta(X_{N_i}, X_{N_i}) + \varepsilon_{N_i})^{-1} Y(X_{N_i}).$$

Parameters

- **Kin** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped kernel matrices corresponding to each of the batch elements.
- **Kcross** (ndarray) – A matrix of shape `(batch_count, nn_count)` whose rows consist of `(1, nn_count)`-shaped cross-covariance vector corresponding to each of the batch elements and its nearest neighbors.
- **batch_nn_targets** (ndarray) – A tensor of shape `(batch_count, nn_count, response_count)` whose last dimension lists the vector-valued responses for the nearest neighbors of each batch element.

Return type
ndarray

Returns
A matrix of shape `(batch_count, response_count)` whose rows are the predicted response for each of the given indices.

posterior_variance(*Kin*, *Kcross*)

Returns the posterior variance from the provided covariance and cross-covariance tensors.

Return the local posterior variances of each prediction, corresponding to the diagonal elements of a covariance matrix. Given observation set X with responses Y, noise prior set ε, and kernel function $Kin_\theta(\cdot,\cdot)$, computes the following for each prediction element $\mathbf{z}_i$ with nearest neighbors index set N_i:

$$Var\left(\widehat{Y}(\mathbf{z}_i \mid X_{N_i})\right) = \sigma^2 \left(Kin_\theta(\mathbf{z}_i, \mathbf{z}_i) - Kin_\theta(\mathbf{z}_i, X_{N_i}) \left(Kin_\theta(X_{N_i}, X_{N_i}) + \varepsilon_{N_i}\right)^{-1} Kin_\theta(X_{N_i}, \mathbf{z}_i)\right).$$

Parameters

- **Kin** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count)` containing the `(nn_count, nn_count)`-shaped kernel matrices corresponding to each of the batch elements.
- **Kcross** (ndarray) – A matrix of shape `(batch_count, nn_count)` whose rows consist of `(1, nn_count)`-shaped cross-covariance vector corresponding to each of the batch elements and its nearest neighbors.

Return type
ndarray

Returns

A vector of shape `(batch_count, response_count)` consisting of the diagonal elements of the posterior variance.

1.2.8 MultivariateMuyGPS

class MuyGPyS.gp.multivariate_muygps.**MultivariateMuyGPS**(**model_args*)

Multivariate Local Kriging Gaussian Process.

Performs approximate GP inference by locally approximating an observation's response using its nearest neighbors with a separate kernel allocated for each response dimension, implemented as individual *MuyGPyS.gp.muygps.MuyGPS* objects.

This class is similar in interface to *MuyGPyS.gp.muygps.MuyGPS*, but requires a list of hyperparameter dicts at initialization.

Example

```
>>> from MuyGPyS.gp import MultivariateMuyGPS as MMuyGPS
>>> k_kwargs1 = {
...     "noise": Parameter(1e-5),
...     "kernel": Matern(
...         smoothness=Parameter(0.67, (0.1, 2.5)),
...         deformation=Isotropy(
...             metric=l2,
...             length_scale=Parameter(0.2),
...         scale=AnalyticScale(),
...     ),
... }
>>> k_kwargs2 = {
...     "noise": Parameter(1e-5),
...     "kernel": Matern(
...         smoothness=Parameter(0.67, (0.1, 2.5)),
...         deformation=Isotropy(
...             metric=l2,
...             length_scale=Parameter(0.2),
...         scale=AnalyticScale(),
...     ),
... }
>>> k_args = [k_kwargs1, k_kwargs2]
>>> mmuygps = MMuyGPS(*k_args)
```

We can realize kernel tensors for each of the models contained within a `MultivariateMuyGPS` object by iterating over its `models` member. Once we have computed `pairwise_diffs` and `crosswise_diffs` tensors, it is straightforward to perform each of these realizations.

Example

```
>>> for model in MuyGPyS.models:
>>>     Kin = model.kernel(pairwise_diffs)
>>>     Kcross = model.kernel(crosswise_diffs)
>>>     # do something with Kin and Kcross...
```

Args

model_args:
Dictionaries defining each internal *MuyGPyS.gp.muygps.MuyGPS* instance.

fast_coefficients(*pairwise_diffs_fast*, *train_nn_targets_fast*)

Produces coefficient matrix for the fast posterior mean given in Equation (8) of [dunton2022fast] for each response dimenion.

Fro each response dimension j, given observation set X with responses Y, noise prior set $\varepsilon^{(j)}$, and kernel function $Kin_{\theta^{(j)}}(\cdot,\cdot)$, computes the following for each observation element $\mathbf{x}_i$ with nearest neighbors index set N_i^*, containing `i` and the indices of the `nn_count - 1` nearest neighbors of $\mathbf{x}_i$:

$$C_i^{(j)} = \left(Kin_{\theta^{(j)}}(X_{N_i}, X_{N_i}) + \varepsilon_{N_i}^{(j)}\right)^{-1} Y(X_{N_i})_{:,j}.$$

Parameters

- **pairwise_diffs** – A tensor of shape `(train_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **batch_nn_targets** – A tensor of shape `(train_count, nn_count, response_count)` listing the vector-valued responses for the nearest neighbors of each batch element.

Return type
`ndarray`

Returns
A tensor of shape `(batch_count, nn_count, response_count)` whose entries comprise the precomputed coefficients for fast posterior mean inference.

fast_posterior_mean(*crosswise_diffs*, *coeffs_tensor*)

Performs fast posterior mean inference using provided cross-covariance and precomputed coefficient matrix for each response dimension.

Returns the predicted response across each response dimension in the form of a posterior mean for each element of the batch of observations, as computed in Equation (9) of [dunton2022fast]. For each response dimension j, given the coefficients $C^{(j)}$ created by `fast_coefficients()` and Equation (8) of [dunton2022fast], observation set X, noise prior set $\varepsilon^{(j)}$, and kernel function $Kin_{\theta^{(j)}}(\cdot,\cdot)$, computes the following for each test point $\mathbf{z}$ and index set N_i^* containing the union of the index i of the nearest neighbor $\mathbf{x}_i$ of $\mathbf{z}$ and the `nn_count - 1` nearest neighbors of $\mathbf{x}_i$:

$$\widehat{Y}(\mathbf{z} \mid X)_j = \sigma^2 Kin_{\theta^{(j)}}(\mathbf{z}, X_{N_i^*}) C_i^{(j)}.$$

Parameters

- **crosswise_diffs** (ndarray) – A matrix of shape `(batch_count, nn_count, feature_count)` whose rows list the difference between each feature of each batch element element and its nearest neighbors.

- **coeffs_tensor** (ndarray) – A tensor of shape (batch_count, nn_count, response_count) providing the precomputed coefficients.

Return type
ndarray

Returns
A matrix of shape (batch_count, response_count) whose rows are the predicted response for each of the given indices.

fixed()

Checks whether all kernel and model parameters are fixed for each model, excluding σ^2.

Return type
bool

Returns
Returns True if all parameters in all models are fixed, and False otherwise.

make_predict_tensors(*batch_indices*, *batch_nn_indices*, *test_features*, *train_features*, *train_targets*, ***kwargs*)

Create the metric and target tensors for prediction using the model's deformation.

@NOTE[mwp] uses the first model's deformation, and expects all model deformations to agree in tensor shapes.

Creates the crosswise_tensor, pairwise_tensor and batch_nn_targets tensors required by posterior_mean() and posterior_variance().

Parameters

- **batch_indices** (ndarray) – A vector of integers of shape (batch_count,) identifying the training batch of observations to be approximated.
- **batch_nn_indices** (ndarray) – A matrix of integers of shape (batch_count, nn_count) listing the nearest neighbor indices for all observations in the batch.
- **test_features** (Optional[ndarray]) – The full floating point testing data matrix of shape (test_count, feature_count).
- **train_features** (ndarray) – The full floating point training data matrix of shape (train_count, ...).
- **train_targets** (ndarray) – A matrix of shape (train_count, ...) whose rows are vector-valued responses for each training element.

Return type
Tuple[ndarray, ndarray, ndarray]

Returns

- *crosswise_tensor* – A tensor of shape (batch_count, nn_count, ...) whose second and subsequent dimensions list the metric comparison between each batch element element and its nearest neighbors.
- *pairwise_diffs* – A tensor of shape (batch_count, nn_count, nn_count, ...) containing the (nn_count, nn_count, ...)-shaped pairwise nearest neighbor metrics tensors corresponding to each of the batch elements.
- *batch_nn_targets* – Tensor of floats of shape (batch_count, nn_count, ...) containing the expected response for each nearest neighbor of each batch element.

make_train_tensors(*batch_indices*, *batch_nn_indices*, *train_features*, *train_targets*, ***kwargs*)

Create the metric and target tensors needed for training.

@NOTE[mwp] uses the first model's deformation, and expects all model deformations to agree in tensor shapes.

Similar to *make_predict_tensors()* but returns the additional `batch_targets` matrix, which is only defined for a batch of training data.

Parameters

- **batch_indices** (ndarray) – A vector of integers of shape `(batch_count,)` identifying the training batch of observations to be approximated.
- **batch_nn_indices** (ndarray) – A matrix of integers of shape `(batch_count, nn_count)` listing the nearest neighbor indices for all observations in the batch.
- **train_features** (ndarray) – The full floating point training data matrix of shape `(train_count, ...)`.
- **train_targets** (ndarray) – A matrix of shape `(train_count, ...)` whose rows are vector-valued responses for each training element.

Return type

`Tuple`[`ndarray`, `ndarray`, `ndarray`, `ndarray`]

Returns

- *crosswise_tensor* – A tensor of shape `(batch_count, nn_count, ...)` whose second and subsequent dimensions list the metric comparison between each batch element element and its nearest neighbors.
- *pairwise_diffs* – A tensor of shape `(batch_count, nn_count, nn_count, ...)` containing the `(nn_count, nn_count, ...)`-shaped pairwise nearest neighbor metrics tensors corresponding to each of the batch elements.
- *batch_targets* – Matrix of floats of shape `(batch_count, ...)` whose rows give the expected response for each batch element.
- *batch_nn_targets* – Tensor of floats of shape `(batch_count, nn_count, ...)` containing the expected response for each nearest neighbor of each batch element.

optimize_scale(*pairwise_diffs*, *nn_targets*)

Optimize the value of the $sigma^2$ scale parameter for each response dimension.

Uses the optimization method specified by the types of the `scale` parameters to optimize their value.

Parameters

- **pairwise_diffs** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **nn_targets** (ndarray) – Tensor of floats of shape `(batch_count, nn_count, response_count)` containing the expected response for each nearest neighbor of each batch element.

Returns

A reference to this model whose global scale parameter (and those of its submodels) has been optimized.

posterior_mean(*pairwise_diffs*, *crosswise_diffs*, *batch_nn_targets*)

Performs simultaneous posterior mean inference on provided difference tensors and the target matrix.

Computes parallelized local solves of systems of linear equations using the kernel realizations, one for each internal model, of the last two dimensions of `pairwise_diffs` along with `crosswise_diffs` and `batch_nn_targets` to predict responses in terms of the posterior mean. Assumes that difference tensors `pairwise_diffs` and `crosswise_diffs` are already computed and given as arguments.

Returns the predicted response in the form of a posterior mean for each element of the batch of observations by solving a system of linear equations induced by each kernel functor, one per response dimension, in a generalization of Equation (3.4) of [muyskens2021muygps]. For each response dimension j, given observation set X with responses Y, noise prior set $\varepsilon^{(j)}$, and kernel function $Kin_{\theta^{(j)}}(\cdot,\cdot)$, computes the following for each prediction element $\mathbf{z}_i$ with nearest neighbors index set N_i:

$$\widehat{Y}(\mathbf{z}_i \mid X_{N_i})_j = \sigma_j^2 Kin_{\theta^{(j)}}(\mathbf{z}_i, X_{N_i}) \left(Kin_{\theta^{(j)}}(X_{N_i}, X_{N_i}) + \varepsilon_{N_i}^{(j)}\right)^{-1} Y(X_{N_i})_{:,j}.$$

Parameters

- **pairwise_diffs** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **crosswise_diffs** (ndarray) – A matrix of shape `(batch_count, nn_count, feature_count)` whose rows list the difference between each feature of each batch element element and its nearest neighbors.
- **batch_nn_targets** (ndarray) – A tensor of shape `(batch_count, nn_count, response_count)` listing the vector-valued responses for the nearest neighbors of each batch element.

Return type
ndarray

Returns
A matrix of shape `(batch_count, response_count)` whose rows are the predicted response for each of the given indices.

posterior_variance(*pairwise_diffs*, *crosswise_diffs*)

Returns the posterior variance from the provided difference tensors.

Return the local posterior variances of each prediction, corresponding to the diagonal elements of a covariance matrix. For each response dimension, given observation set X with responses Y, noise prior set $\varepsilon^{(j)}$, and kernel function $Kin_{\theta^{(j)}}(\cdot,\cdot)$, computes the following for each prediction element $\mathbf{z}_i$ with nearest neighbors index set N_i:

$$Var\left(\widehat{Y}(\mathbf{z}_i \mid X_{N_i})\right)_j = \sigma_j^2 \left(Kin_{\theta^{(j)}}(\mathbf{z}_i, \mathbf{z}_i) - Kin_{\theta^{(j)}}(\mathbf{z}_i, X_{N_i}) \left(Kin_{\theta^{(j)}}(X_{N_i}, X_{N_i}) + \varepsilon_{N_i}^{(j)}\right)^{-1} Kin_{\theta^{(j)}}(X_{N_i}, \mathbf{z}_i)\right).$$

Parameters

- **pairwise_diffs** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **crosswise_diffs** (ndarray) – A matrix of shape `(batch_count, nn_count, feature_count)` whose rows list the difference between each feature of each batch element element and its nearest neighbors.

Return type
ndarray

Returns
A vector of shape `(batch_count, response_count)` consisting of the diagonal elements of the posterior variance for each model.

1.3 optimize

MuyGPyS.optimize module reference.

1.3.1 batch

Sampling elements with their nearest neighbors from data

MuyGPyS includes convenience functions for sampling batches of data from existing datasets. These batches are returned in the form of row indices, both of the sampled data as well as their nearest neighbors. Also included is the ability to sample "balanced" batches, where the data is partitioned by class and we attempt to sample as close to an equal number of items from each class as is possible.

MuyGPyS.optimize.batch.**full_filtered_batch**(*nbrs_lookup*, *labels*)

Return a batch composed of the entire training set, filtering out elements with constant nearest neighbor sets.

Parameters

- **nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.
- **labels** (ndarray) – List of class labels of shape `(train_count,)` for all train data.

Return type
Tuple[ndarray, ndarray]

Returns

- *indices* – The indices of the sampled training points of shape `(batch_count,)`.
- *nn_indices* – The indices of the nearest neighbors of the sampled training points of shape `(batch_count, nn_count)`.

MuyGPyS.optimize.batch.**get_balanced_batch**(*nbrs_lookup*, *labels*, *batch_count*)

Decide whether to sample a balanced batch or return the full filtered batch.

This method is the go-to method for sampling from classification datasets when one desires a sample with equal representation of every class. The function simply calls *MuyGPyS.optimize.batch.full_filtered_batch()* if the supplied list of training data class labels is smaller than the batch count, otherwise calling MuyGPyS.optimize.batch_sample_balanced_batch().

Example

```
>>> import numpy as np
>>> From MuyGPyS.optimize.batch import get_balanced_batch
>>> train_features, train_responses = get_train()
>>> nn_count = 10
>>> nbrs_lookup = NN_Wrapper(train_features, nn_count)
>>> batch_count = 200
>>> train_labels = np.argmax(train_responses, axis=1)
>>> balanced_indices, balanced_nn_indices = get_balanced_batch(
...         nbrs_lookup, train_labels, batch_count
>>> )
```

Parameters

- **nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.
- **labels** (ndarray) – List of class labels of shape `(train_count,)` for all training data.
- **batch_count** (int) – int The number of batch elements to sample.

Return type
Tuple[ndarray, ndarray]

Returns

- *indices* – The indices of the sampled training points of shape `(batch_count,)`.
- *nn_indices* – The indices of the nearest neighbors of the sampled training points of shape `(batch_count, nn_count)`.

MuyGPyS.optimize.batch.**sample_balanced_batch**(*nbrs_lookup*, *labels*, *batch_count*)

Collect a class-balanced batch of training indices.

The returned batch is filtered to remove samples whose nearest neighbors share the same class label, and is balanced so that each class is equally represented (where possible.)

Parameters

- **nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.
- **labels** (ndarray) – List of class labels of shape `(train_count,)` for all train data.
- **batch_count** (int) – The number of batch elements to sample.

Return type
Tuple[ndarray, ndarray]

Returns

- *nonconstant_balanced_indices* – The indices of the sampled training points of shape `(batch_count,)`. These indices are guaranteed to have nearest neighbors with differing class labels.
- *batch_nn_indices* – The indices of the nearest neighbors of the sampled training points of shape `(batch_count, nn_count)`.

MuyGPyS.optimize.batch.**sample_batch**(*nbrs_lookup*, *batch_count*, *train_count*)

Collect a batch of training indices.

This is a simple sampling method where training examples are selected uniformly at random, without replacement.

Example

```
>>> From MuyGPyS.optimize.batch import sample_batch
>>> train_features, train_responses = get_train()
>>> train_count, _ = train_features.shape
>>> nn_count = 10
>>> nbrs_lookup = NN_Wrapper(train_features, nn_count)
>>> batch_count = 200
>>> batch_indices, batch_nn_indices = sample_batch(
...         nbrs_lookup, batch_count, train_count
>>> )
```

Parameters

- **nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.
- **batch_count** (`int`) – The number of batch elements to sample.
- **train_count** (`int`) – int The total number of training examples.

Return type
`Tuple`[`ndarray`, `ndarray`]

Returns

- *batch_indices* – The indices of the sampled training points of shape `(batch_count,)`.
- *batch_nn_indices* – The indices of the nearest neighbors of the sampled training points of shape `(batch_count, nn_count)`.

1.3.2 chassis

class MuyGPyS.optimize.chassis.**OptimizeFn**(*optimize_fn*, *make_obj_fn*)

Outer-loop optimization functor class.

MuyGPyS-compatible optimization functions are objects of this class. Creating a new outer-loop optimization function is as simple as instantiating a new `OptimizeFn` object.

Parameters

- **optimize_fn** – A Callable with the signature `(muygps, obj_fn, verbose=verbose, **kwargs) -> MuyGPS`.
- **make_obj_fn** – A Callable taking the following objects, in order: `loss_fn`, `kernel_fn`, `mean_fn`, `var_fn`, `scale_fn`, `pairwise_diffs`, `crosswise_diffs` `batch_nn_targets`, `batch_targets` `batch_features`, `loss_kwargs`.

make_obj_fn(*muygps*, *batch_targets*, *batch_nn_targets*, *crosswise_diffs*, *pairwise_diffs*, *batch_features=None*, *target_mask=None*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *loss_kwargs={}*, ***kwargs*)

Returns the objective function specified by the training batch and model choices.

Parameters

- **muygps** (*MuyGPS*) – The model to be optimized.
- **batch_targets** (`ndarray`) – Matrix of floats of shape `(batch_count, response_count)` whose rows give the expected response for each batch element.

- **batch_nn_targets** (ndarray) – Tensor of floats of shape `(batch_count, nn_count, response_count)` containing the expected response for each nearest neighbor of each batch element.
- **crosswise_diffs** (ndarray) – A tensor of shape `(batch_count, nn_count, feature_count)` whose last two dimensions list the difference between each feature of each batch element element and its nearest neighbors.
- **pairwise_diffs** (ndarray) – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **loss_fn** (*LossFn*) – The loss functor used to evaluate model performance.
- **target_mask** (`Optional[ndarray]`) – An array of indices, listing the output dimensions of the prediction to be used for optimization.
- **loss_kwargs** (`Dict`) – A dictionary of additional keyword arguments to apply to the *LossFn*. Loss function specific.
- **kwargs** – Additional keyword arguments to be passed to the wrapper optimizer.

Return type
`Callable`

Returns
A Callable function that evaluates the objective function for a given value of the free parameters.

`MuyGPyS.optimize.chassis.`**Bayes_optimize = <MuyGPyS.optimize.chassis.OptimizeFn object>**

Optimize a *MuyGPS* model using Bayesian optimization.

See the following example, where we have already created a `batch_indices` vector and a `batch_nn_indices` matrix using *MuyGPyS.neighbors.NN_Wrapper*, a `crosswise_diffs` matrix using `MuyGPyS.gp.tensors.crosswise_tensor()` and `pairwise_diffs` using `MuyGPyS.gp.tensors.pairwise_tensor()`, and initialized a *MuyGPS* model `muygps`.

Example

```
>>> from MuyGPyS.optimize import Bayes_optimize
>>> muygps = Bayes_optimize(
...         muygps,
...         batch_targets,
...         batch_nn_targets,
...         crosswise_diffs,
...         pairwise_diffs,
...         train_responses,
...         loss_fn=lool_fn,
...         verbose=True,
... )
parameters to be optimized: ['nu']
bounds: [[0.1 5. ]]
initial x0: [0.92898658]
|   iter    |  target   |    nu     |
-------------------------------------
| 1         | 1.826e+03 | 0.929     |
```

(continues on next page)

(continued from previous page)

```
| 2          | 2.359e+03 | 2.143     |
| 3          | 1.953e+03 | 3.63      |
| 4          | 614.4     | 0.1006    |
| 5          | 2.309e+03 | 1.581     |
| 6          | 1.707e+03 | 0.8191    |
| 7          | 1.48e+03  | 5.0       |
| 8          | 2.202e+03 | 2.83      |
| 9          | 2.373e+03 | 1.883     |
| 10         | 2.373e+03 | 1.996     |
| 11         | 2.375e+03 | 1.938     |
| 12         | 2.375e+03 | 1.938     |
| 13         | 2.375e+03 | 1.938     |
| 14         | 2.375e+03 | 1.938     |
| 15         | 2.375e+03 | 1.938     |
| 16         | 2.375e+03 | 1.938     |
| 17         | 2.375e+03 | 1.938     |
| 18         | 2.375e+03 | 1.945     |
| 19         | 2.375e+03 | 1.927     |
| 20         | 2.375e+03 | 1.95      |
| 21         | 2.375e+03 | 1.926     |
=====================================
```

Parameters

- **muygps** – The model to be optimized.
- **batch_targets** – Matrix of floats of shape `(batch_count, response_count)` whose rows give the expected response for each batch element.
- **batch_nn_targets** – Tensor of floats of shape `(batch_count, nn_count, response_count)` containing the expected response for each nearest neighbor of each batch element.
- **crosswise_diffs** – A tensor of shape `(batch_count, nn_count, feature_count)` whose last two dimensions list the difference between each feature of each batch element element and its nearest neighbors.
- **pairwise_diffs** – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **loss_fn** – The loss functor used to evaluate model performance.
- **loss_kwargs** – A dictionary of additional keyword arguments to apply to the *LossFn*. Loss function specific.
- **verbose** – If True, print debug messages.
- **kwargs** – Additional keyword arguments to be passed to the wrapper optimizer.

Returns

A new MuyGPs model whose specified hyperparameters have been optimized.

MuyGPyS.optimize.chassis.**L_BFGS_B_optimize = <MuyGPyS.optimize.chassis.OptimizeFn object>**

Optimize a *MuyGPS* model using the L-BFGS-B algorithm.

See the following example, where we have already created a `batch_indices` vector and a `batch_nn_indices` matrix using *MuyGPyS.neighbors.NN_Wrapper*, a `crosswise_diffs` matrix using `MuyGPyS.gp.tensors.`

crosswise_tensor() and pairwise_diffs using MuyGPyS.gp.tensors.pairwise_tensor(), and initialized a *MuyGPS* model muygps.

Example

```
>>> from MuyGPyS.optimize import L_BFGS_B_optimize
>>> muygps = L_BFGS_B_optimize(
...         muygps,
...         batch_targets,
...         batch_nn_targets,
...         crosswise_diffs,
...         pairwise_diffs,
...         train_responses,
...         loss_fn=lool_fn,
...         verbose=True,
... )
parameters to be optimized: ['nu']
bounds: [[0.1 1. ]]
sampled x0: [0.8858425]
optimizer results:
    fun: 0.4797763813693626
hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
    jac: array([-3.06976666e-06])
message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
    nfev: 16
    nit: 5
    njev: 8
status: 0
success: True
        x: array([0.39963594])
```

Parameters

- **muygps** – The model to be optimized.
- **batch_targets** – Matrix of floats of shape (batch_count, response_count) whose rows give the expected response for each batch element.
- **batch_nn_targets** – Tensor of floats of shape (batch_count, nn_count, response_count) containing the expected response for each nearest neighbor of each batch element.
- **crosswise_diffs** – A tensor of shape (batch_count, nn_count, feature_count) whose last two dimensions list the difference between each feature of each batch element element and its nearest neighbors.
- **pairwise_diffs** – A tensor of shape (batch_count, nn_count, nn_count, feature_count) containing the (nn_count, nn_count, feature_count)-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **loss_fn** – The loss functor used to evaluate model performance.
- **loss_kwargs** – A dictionary of additional keyword arguments to apply to the *LossFn*. Loss function specific.
- **verbose** – If True, print debug messages.

- **kwargs** – Additional keyword arguments to be passed to the wrapper optimizer.

Returns

A new MuyGPs model whose specified hyperparameters have been optimized.

1.3.3 loss

Loss Function Handling

MuyGPyS includes predefined loss functions and convenience functions for indicating them to optimization.

class MuyGPyS.optimize.loss.**LossFn**(*loss_fn*, *make_predict_and_loss_fn*)

Loss functor class.

MuyGPyS-compatible loss functions are objects of this class. Creating a new loss function is as simple as instantiation a new `LossFn` object.

Parameters

- **loss_fn** (`Callable`) – A Callable with signature `(predictions, targets, **kwargs)` or `(predictions, targets, variances, scale, **kwargs)` tha computes a floating-point loss score of a set of predictions given posterior means and possibly posterior variances. Individual loss functions can implement different `kwargs` as needed.
- **make_precit_and_loss_fn** – A Callable with signature `(loss_fn, mean_fn, var_fn, scale_fn, batch_nn_targets, batch_targets, **loss_kwargs)` that produces a function that computes posterior predictions and scores them using the loss function. :func:~MuyGPyS.optimize.loss._make_raw_predict_and_loss_fn` and :func:~MuyGPyS.optimize.loss._make_var_predict_and_loss_fn` are two candidates.

MuyGPyS.optimize.loss.**cross_entropy_fn = <MuyGPyS.optimize.loss.LossFn object>**

Cross entropy function.

Computes the cross entropy loss the predicted versus known response. Transforms `predictions` to be row-stochastic, and ensures that `targets` contains no negative elements. Only defined for two or more labels. For a sample with true labels $y_i \in \{0, 1\}$ and estimates $\mu(x_i) = \Pr(y = 1)$, the function computes

$$\ell_{\text{cross-entropy}}(\bar{\mu}, y) = \sum_{i=1}^{b} y_i \log(\bar{\mu}_i) - (1 - y_i) \log(1 - \bar{\mu}_i).$$

The numpy backend uses sklearn's implementation.

Parameters

- **predictions** – The predicted response of shape `(batch_count, response_count)`.
- **targets** – The expected response of shape `(batch_count, response_count)`.
- **eps** – Probabilities are clipped to the range `[eps, 1 - eps]`.

Returns

The cross-entropy loss of the prediction.

MuyGPyS.optimize.loss.**lool_fn = <MuyGPyS.optimize.loss.LossFn object>**

Leave-one-out likelihood function.

Computes leave-one-out likelihood (LOOL) loss of the predicted versus known response. Treats multivariate outputs as interchangeable in terms of loss penalty. The function computes

$$\ell_{\text{lool}}(\bar{\mu}, y, \bar{\Sigma}) = \sum_{i=1}^{b}\sum_{j=1}^{s}\left(\frac{(\bar{\mu}_i - y)^2}{\bar{\Sigma}_{ii}}\right)_j + \left(\log \bar{\Sigma}_{ii}\right)_j$$

Parameters

- **predictions** – The predicted response of shape `(batch_count, response_count)`.
- **targets** – The expected response of shape `(batch_count, response_count)`.
- **variances** – The unscaled variance of the predicted responses of shape `(batch_count, response_count)`.
- **scale** – The scale variance scaling parameter of shape `(response_count,)`.

Returns

The LOOL loss of the prediction.

MuyGPyS.optimize.loss.**lool_fn_unscaled = <MuyGPyS.optimize.loss.LossFn object>**

Leave-one-out likelihood function.

Computes leave-one-out likelihood (LOOL) loss of the predicted versus known response. Treats multivariate outputs as interchangeable in terms of loss penalty. Unlike lool_fn, does not require scale as an argument. The function computes

$$\ell_{\text{lool}}(\bar{\mu}, y, \bar{\Sigma}) = \sum_{i=1}^{b}\frac{(\bar{\mu}_i - y)^2}{\bar{\Sigma}_{ii}} + \log \bar{\Sigma}_{ii}.$$

Parameters

- **predictions** – The predicted response of shape `(batch_count, response_count)`.
- **targets** – The expected response of shape `(batch_count, response_count)`.
- **variances** – The unscaled variance of the predicted responses of shape `(batch_count, response_count)`.

Returns

The LOOL loss of the prediction.

MuyGPyS.optimize.loss.**looph_fn = <MuyGPyS.optimize.loss.LossFn object>**

Variance-regularized pseudo-Huber loss function.

Computes a smooth approximation to the Huber loss function, similar to *pseudo_huber_fn()*, with the addition of both a variance scaling and a additive logarithmic variance regularization term to avoid exploding the variance. This is intended to be an outlier-robust replacement to the lool function. Similar to pseudo-Huber, the `boundary_scale` parameter is sensitive to the units of the responses, and must be set accordingly. The function computes

$$\ell_{\text{lool}}(\bar{\mu}, y, \bar{\Sigma} \mid \delta) = \sum_{i=1}^{b}\sum_{j=1}^{s} 2\delta^2\left(\sqrt{1 + \left(\frac{(y_i - \bar{\mu}_i)^2}{\delta^2\bar{\Sigma}_{ii}}\right)_j} - 1\right) + \left(\log \bar{\Sigma}_{ii}\right)_j.$$

Parameters

- **predictions** – The predicted response of shape `(batch_count, response_count)`.
- **targets** – The expected response of shape `(batch_count, response_count)`.

- **variances** – The unscaled variance of the predicted responses of shape `(batch_count, response_count)`.
- **scale** – The scale variance scaling parameter of shape `(response_count,)`.
- **boundary_scale** – The boundary value for the residual beyond which the loss becomes approximately linear. Corresponds to the number of standard deviations beyond which to linearize the loss. The default value is 3.0, which is sufficiently tight for most realistic problems.

Returns
The sum of leave-one-out pseudo-Huber losses of the predictions.

`MuyGPyS.optimize.loss.`**make_raw_predict_and_loss_fn**(*loss_fn*, *mean_fn*, *var_fn*, *scale_fn*, *batch_nn_targets*, *batch_targets*, *target_mask=None*, ***loss_kwargs*)

Make a predict_and_loss function that depends only on the posterior mean.

Assembles a new function with signature `(Kin, Kcross, *args, **kwargs)` that computes the posterior mean and uses the passed `loss_fn` to score it against the batch targets.

Parameters

- **loss_fn** (`Callable`) – A loss function Callable with signature `(predictions, responses, **kwargs)`, where `predictions` and `targets` are matrices of shape `(batch_count, response_count)`.
- **mean_fn** (`Callable`) – A MuyGPS posterior mean function Callable with signature `(Kin, Kcross, batch_nn_targets)`, which are tensors of shape `(batch_count, nn_count, nn_count)`, `(batch_count, nn_count)`, and `(batch_count, nn_count, response_count)`, respectively.
- **var_fn** (`Callable`) – A MuyGPS posterior variance function Callable with signature `(Kin, Kcross)`, which are tensors of shape `(batch_count, nn_count, nn_count)` and `(batch_count, nn_count)`, respectively. Unused by this function, but still required by the signature.
- **scale_fn** (`Callable`) – A MuyGPS `scale` optimization function Callable with signature `(Kin, batch_nn_targets)`, which are tensors of shape `(batch_count, nn_count, nn_count)` and `(batch_count, nn_count, response_count)`, respectively. Unused by this function, but still required by the signature.
- **batch_nn_targets** (ndarray) – A tensor of shape `(batch_count, nn_count, response_count)` containing the expected response of the nearest neighbors of each batch element.
- **batch_targets** (ndarray) – A matrix of shape `(batch_count, response_count)` containing the expected response of each batch element.
- **loss_kwargs** – Additionall keyword arguments used by the loss function.

Return type
`Callable`

Returns
A Callable with signature `(Kin, Kcross, *args, **kwargs) -> float` that computes the posterior mean and applies the loss function to it and the `batch_targets`.

`MuyGPyS.optimize.loss.`**make_var_predict_and_loss_fn**(*loss_fn*, *mean_fn*, *var_fn*, *scale_fn*, *batch_nn_targets*, *batch_targets*, *target_mask=None*, ***loss_kwargs*)

Make a predict_and_loss function that depends on the posterior mean and variance.

Assembles a new function with signature `(Kin, Kcross, *args, **kwargs)` that computes the posterior mean and variance and uses the passed `loss_fn` to score them against the batch targets.

Parameters

- **loss_fn** (`Callable`) – A loss function Callable with signature `(predictions, responses, **kwargs)`, where `predictions` and `targets` are matrices of shape `(batch_count, response_count)`.
- **mean_fn** (`Callable`) – A MuyGPS posterior mean function Callable with signature `(Kin, Kcross, batch_nn_targets)`, which are tensors of shape `(batch_count, nn_count, nn_count)`, `(batch_count, nn_count)`, and `(batch_count, nn_count, response_count)`, respectively.
- **var_fn** (`Callable`) – A MuyGPS posterior variance function Callable with signature `(Kin, Kcross)`, which are tensors of shape `(batch_count, nn_count, nn_count)` and `(batch_count, nn_count)`, respectively.
- **scale_fn** (`Callable`) – A MuyGPS `scale` optimization function Callable with signature `(Kin, batch_nn_targets)`, which are tensors of shape `(batch_count, nn_count, nn_count)` and `(batch_count, nn_count, response_count)`, respectively.
- **batch_nn_targets** (`ndarray`) – A tensor of shape `(batch_count, nn_count, response_count)` containing the expected response of the nearest neighbors of each batch element.
- **batch_targets** (`ndarray`) – A matrix of shape `(batch_count, response_count)` containing the expected response of each batch element.
- **target_mask** (`Optional[ndarray]`) – An array of indices, listing the output dimensions of the prediction to be used for optimization.
- **loss_kwargs** – Additionall keyword arguments used by the loss function.

Return type
`Callable`

Returns
A Callable with signature `(Kin, Kcross, *args, **kwargs) -> float` that computes the posterior mean and applies the loss function to it and the `batch_targets`.

`MuyGPyS.optimize.loss.`**`mse_fn = <MuyGPyS.optimize.loss.LossFn object>`**

Mean squared error function.

Computes mean squared error loss of the predicted versus known response. Treats multivariate outputs as interchangeable in terms of loss penalty. The function computes

$$\ell_{\text{MSE}}(\bar{\mu}, y) = \frac{1}{b}\sum_{i=1}^{b}(\bar{\mu}_i - y)^2.$$

Parameters

- **predictions** – The predicted response of shape `(batch_count, response_count)`.
- **targets** – The expected response of shape `(batch_count, response_count)`.

Returns
The mse loss of the prediction.

MuyGPyS.optimize.loss.**pseudo_huber_fn = <MuyGPyS.optimize.loss.LossFn object>**

Pseudo-Huber loss function.

Computes a smooth approximation to the Huber loss function, which balances sensitive squared-error loss for relatively small errors and robust-to-outliers absolute loss for larger errors, so that the loss is not overly sensitive to outliers. Uses the form from wikipedia. The function computes

$$\ell_{\text{Pseudo-Huber}}(\bar{\mu}, y \mid \delta) = \sum_{i=1}^{b} \delta^2 \left(\sqrt{1 + \left(\frac{y_i - \bar{\mu}_i}{\delta} \right)^2} - 1 \right).$$

Parameters

- **predictions** – The predicted response of shape `(batch_count, response_count)`.
- **targets** – The expected response of shape `(batch_count, response_count)`.
- **boundary_scale** – The boundary value for the residual beyond which the loss becomes approximately linear. Useful values depend on the scale of the response.

Returns

The sum of pseudo-Huber losses of the predictions.

1.3.4 objective

Objective Handling

MuyGPyS includes predefined objective functions and convenience functions for indicating them to optimization.

MuyGPyS.optimize.objective.**make_loo_crossval_fn**(*loss_fn*, *kernel_fn*, *mean_fn*, *var_fn*, *scale_fn*, *pairwise_diffs*, *crosswise_diffs*, *batch_nn_targets*, *batch_targets*, *batch_features=None*, *target_mask=None*, *loss_kwargs={}*)

Prepare a leave-one-out cross validation function as a function purely of the hyperparameters to be optimized.

This function is designed for use with *MuyGPyS.optimize.chassis.OptimizeFn*.

Parameters

- **loss_fn** (*LossFn*) – The loss functor used to evaluate model performance.
- **kernel_fn** (`Callable`) – A function that realizes kernel tensors given a list of the free parameters.
- **mean_fn** (`Callable`) – A function that realizes MuyGPs posterior mean prediction given a noise model.
- **var_fn** (`Callable`) – A function that realizes MuyGPs posterior variance prediction given a noise model.
- **scale_fn** (`Callable`) – A function that realizes variance scale parameter optimization given a noise model.
- **pairwise_diffs** (`ndarray`) – A tensor of shape `(batch_count, nn_count, nn_count, feature_count)` containing the `(nn_count, nn_count, feature_count)`-shaped pairwise nearest neighbor difference tensors corresponding to each of the batch elements.
- **crosswise_diffs** (`ndarray`) – A tensor of shape `(batch_count, nn_count, feature_count)` whose last two dimensions list the difference between each feature of each batch element element and its nearest neighbors.

- **batch_nn_targets** (ndarray) – Tensor of floats of shape (batch_count, nn_count, response_count) containing the expected response for each nearest neighbor of each batch element.
- **batch_targets** (ndarray) – Matrix of floats of shape (batch_count, response_count) whose rows give the expected response for each batch element.
- **batch_features** (Optional[ndarray]) – Matrix of floats of shape (batch_count, feature_count) whose rows give the features for each batch element.
- **target_mask** (Optional[ndarray]) – An array of indices, listing the output dimensions of the prediction to be used for optimization.
- **loss_kwargs** (Dict) – A dict listing any additional kwargs to pass to the loss function.

Return type
Callable

Returns
A Callable objective_fn.

1.4 examples

MuyGPyS.examples module reference. Includes the high-level APIs for automated model creation and training, and some automated prediction workflows.

1.4.1 regress

Resources and high-level API for a simple regression workflow.

make_regressor() is a high-level API for creating and training *MuyGPyS.gp.muygps.MuyGPS* objects for regression. *make_multivariate_regressor()* is a high-level API for creating and training MuyGPyS.gp.muygps.MultivariateMuyGPS objects for regression.

do_regress() is a high-level api for executing a simple, generic regression workflow given data. It calls the maker APIs above and *regress_any()*.

MuyGPyS.examples.regress.**do_regress**(*test_features*, *train_features*, *train_targets*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_kwargs={}*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function initializing a model and performing regression.

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Also supports workflows relying upon multivariate models. In order to create a multivariate model, pass a list of hyperparameter dicts to k_kwargs.

Example

```
>>> from MuyGPyS.examples.regress import do_regress
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.hyperparameter import AnalyticScale
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.examples.regress import do_regress
>>> from MuyGPyS.optimize import Bayes_optimize
>>> from MuyGPyS.optimize.objective import mse_fn
>>> train_features, train_responses = make_train()  # stand-in function
>>> test_features, test_responses = make_test()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(1.0, (1e-2, 1e2))
...         )
...     ),
...     "noise": HomoscedasticNoise(1e-5),
...     "scale": AnalyticScale(),
... }
>>> muygps, nbrs_lookup, predictions, variance = do_regress(
...         test_features,
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_fn=lool_fn,
...         opt_fn=Bayes_optimize,
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> mse = mse_fn(test_responses, predictions)
>>> print(f"obtained mse: {mse}")
obtained mse: 0.20842...
```

Parameters

- **test_features** (ndarray) – A matrix of shape `(test_count, feature_count)` whose rows consist of observation vectors of the test data.
- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_targets** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.
- **nn_count** (`int`) – The number of nearest neighbors to employ.
- **batch_count** (`int`) – The number of elements to sample batch for hyperparameter optimization.

- **loss_fn** (*LossFn*) – The loss functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_kwargs** (`Union[Dict, List[Dict], Tuple[Dict, ...]]`) – If given a list or tuple of length `response_count`, assume that the elements are dicts containing kernel initialization keyword arguments for the creation of a multivariate model (see *make_multivariate_regressor()*). If given a dict, assume that the elements are keyword arguments to a MuyGPs model (see *make_regressor()*).
- **nn_kwargs** (`Dict`) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (`Dict`) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (`bool`) – If `True`, print summary statistics.

Return type
`Tuple[Union[MuyGPS, MultivariateMuyGPS], NN_Wrapper, ndarray, ndarray]`

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into `train_features`.
- *predictions* – The predicted response associated with each test observation.
- *variance* – Estimated `(test_count, response_count)` posterior variance of each test prediction.

MuyGPyS.examples.regress.**make_multivariate_regressor**(*train_features*, *train_targets*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_args=[]*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function for creating a Multivariate MuyGPyS functor and neighbor lookup data structure.

Expected parameters include a list of keyword argument dicts specifying kernel parameters and a dict listing nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Example

```
>>> from MuyGPyS.examples.regress import make_multivariate_regressor
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.hyperparameter import AnalyticScale
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.optimize import Bayes_optimize
>>> train_features, train_responses = make_train()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_args = [
```

(continues on next page)

(continued from previous page)

```
...         {
...             "kernel": RBF(
...                 deformation=Isotropy(
...                     metric=F2,
...                     length_scale=Parameter(1.0, (1e-2, 1e2))
...                 )
...             ),
...             "noise": HomoscedasticNoise(1e-5),
...             "scale": AnalyticScale(),
...         },
...         {
...             "kernel": RBF(
...                 deformation=Isotropy(
...                     metric=F2,
...                     length_scale=Parameter(1.0, (1e-2, 1e2))
...                 )
...             ),
...             "noise": HomoscedasticNoise(1e-5),
...             "scale": AnalyticScale(),
...         },
... ]
>>> mmuygps, nbrs_lookup = make_multivariate_regressor(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_fn=lool_fn,
...         opt_fn=Bayes_optimize,
...         k_args=k_args,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
```

Parameters

- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_targets** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.
- **nn_count** (`int`) – The number of nearest neighbors to employ.
- **batch_count** (`int`) – The number of elements to sample batch for hyperparameter optimization.
- **loss_fn** (*LossFn*) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_args** (`Union[List[Dict], Tuple[Dict, ...]]`) – A list of `response_count` dicts containing kernel initialization keyword arguments. Each dict specifies parameters for the kernel, possibly including noise and scale hyperparameter specifications and specifications for

specific kernel hyperparameters. If all of the hyperparameters are fixed or are not given optimization bounds, no optimization will occur.

- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (Dict) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (bool) – If True, print summary statistics.

Return type
Tuple[*MultivariateMuyGPS*, *NN_Wrapper*]

Returns

- *mmuygps* – A Multivariate MuyGPs object with a separate (possibly trained) kernel function associated with each response dimension.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into train_features.

MuyGPyS.examples.regress.**make_regressor**(*train_features*, *train_targets*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_kwargs={}*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function for creating MuyGPyS functor and neighbor lookup data structure.

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Example

```
>>> from MuyGPyS.examples.regress import make_regressor
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.hyperparameter import AnalyticScale
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.optimize import Bayes_optimize
>>> from MuyGPyS.examples.regress import make_regressor
>>> train_features, train_responses = make_train()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(1.0, (1e-2, 1e2))
...         )
...     ),
...     "noise": HomoscedasticNoise(1e-5),
...     "scale": AnalyticScale(),
... }
>>> muygps, nbrs_lookup = make_regressor(
...         train_features,
...         train_responses,
...         nn_count=30,
```

(continues on next page)

(continued from previous page)

```
...         batch_count=200,
...         loss_fn=lool_fn,
...         opt_fn=Bayes_optimize,
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
```

Parameters

- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_targets** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.
- **nn_count** (int) – The number of nearest neighbors to employ.
- **batch_count** (int) – The number of elements to sample batch for hyperparameter optimization.
- **loss_fn** (*LossFn*) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_kwargs** (Dict) – Parameters for the kernel, possibly including kernel type, deformation function, noise and scale hyperparameter specifications, and specifications for kernel hyperparameters. See *kernels* for examples and requirements. If all of the hyperparameters are fixed or are not given optimization bounds, no optimization will occur.
- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (Dict) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (bool) – If `True`, print summary statistics.

Return type
Tuple[*MuyGPS*, *NN_Wrapper*]

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into `train_features`.

MuyGPyS.examples.regress.**regress_any**(*regressor*, *test_features*, *train_features*, *train_nbrs_lookup*, *train_targets*)

Simultaneously predicts the response for each test item.

Parameters

- **regressor** (Union[*MuyGPS*, *MultivariateMuyGPS*]) – Regressor object.
- **test_features** (ndarray) – Test observations of shape `(test_count, feature_count)`.

- **train_features** (ndarray) – Train observations of shape (train_count, feature_count).
- **train_nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.
- **train_targets** (ndarray) – Observed response for all training data of shape (train_count, class_count).

Return type
Tuple[ndarray, ndarray, Dict[str, float]]

Returns

- *means* – The predicted response of shape (test_count, response_count,) for each of the test examples.
- *variances* – The independent posterior variances for each of the test examples. Of shape (test_count,) if the argument regressor is an instance of *MuyGPyS.gp.muygps.MuyGPS*, and of shape (test_count, response_count) if regressor is an instance of MuyGPyS.gp.muygps.MultivariateMuyGPS.
- **timing** (*dict*) – Timing for the subroutines of this function.

1.4.2 fast_posterior_mean

Resources and high-level API for a fast posterior mean inference workflow.

make_fast_regressor() is a high-level API for creating the necessary components for fast posterior mean inference. *make_fast_multivariate_regressor()* is a high-level API for creating the necessary components for fast posterior mean inference with multiple outputs.

do_fast_posterior_mean() is a high-level api for executing a simple, generic fast posterior medan workflow given data. It calls the maker APIs above and *fast_posterior_mean_any()*.

MuyGPyS.examples.fast_posterior_mean.**do_fast_posterior_mean**(*test_features*, *train_features*, *train_targets*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_kwargs={}*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function initializing a model and performing fast posterior mean inference.

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Also supports workflows relying upon multivariate models. In order to create a multivariate model, specify the kern argument and pass a list of hyperparameter dicts to k_kwargs.

Example

```
>>> from MuyGPyS.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.examples.fast_posterior_mean import do_fast_posterior_mean
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.hyperparameter import AnalyticScale
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.optimize import Bayes_optimize
>>> from MuyGPyS.optimize.objective import mse_fn
>>> train_features, train_responses = make_train()  # stand-in function
>>> test_features, test_responses = make_test()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(1.0, (1e-2, 1e2))
...         )
...     ),
...     "noise": HomoscedasticNoise(1e-5),
...     "scale": AnalyticScale(),
... }
>>> (
...     muygps, nbrs_lookup, predictions, precomputed_coefficients_matrix
... ) = do_fast_posterior_mean(
...         test_features,
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_fn=lool_fn,
...         opt_fn=Bayes_optimize,
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
```

Parameters

- **test_features** (ndarray) – A matrix of shape `(test_count, feature_count)` whose rows consist of observation vectors of the test data.
- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_targets** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.
- **nn_count** (int) – The number of nearest neighbors to employ.
- **batch_count** (int) – The number of elements to sample batch for hyperparameter optimization.

- **loss_fn** (*LossFn*) – The loss functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_kwargs** (`Union[Dict, List[Dict], Tuple[Dict, ...]]`) – If given a list or tuple of length `response_count`, assume that the elements are dicts containing kernel initialization keyword arguments for the creation of a multivariate model (see *make_multivariate_regressor()*). If given a dict, assume that the elements are keyword arguments to a MuyGPs model (see *make_regressor()*).
- **nn_kwargs** (`Dict`) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (`Dict`) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (`bool`) – If `True`, print summary statistics.

Return type
`Tuple[ndarray, NN_Wrapper, ndarray, ndarray, Dict]`

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into `train_features`.
- *predictions* – The predicted response associated with each test observation.
- *precomputed_coefficients_matrix* – A matrix of shape `(train_count, nn_count)` whose rows list the precomputed coefficients for each nearest neighbors set in the training data.
- *timing* – A dictionary containing timings for the training, precomputation, nearest neighbor computation, and prediction.

MuyGPyS.examples.fast_posterior_mean.**fast_posterior_mean_any**(*muygps*, *test_features*, *train_features*, *nbrs_lookup*, *train_targets*)

Convenience function performing fast posterior mean inference using a pre-trained model.

Also supports workflows relying upon multivariate models.

Parameters

- **muygps** (`Union[MuyGPS, MultivariateMuyGPS]`) – A (possibly trained) MuyGPS object.
- **test_features** (`ndarray`) – A matrix of shape `(test_count, feature_count)` whose rows consist of observation vectors of the test data.
- **train_features** (`ndarray`) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **nbrs_lookup** (*NN_Wrapper*) – A data structure supporting nearest neighbor queries into `train_features`.
- **train_targets** (`ndarray`) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.

Return type
`Tuple[ndarray, ndarray, Dict]`

Returns

- *posterior_mean* – The predicted response associated with each test observation.

- *precomputed_coefficients_matrix* – A matrix of shape `(train_count, nn_count)` whose rows list the precomputed coefficients for each nearest neighbors set in the training data.
- *timing* – A dictionary containing timings for the training, precomputation, nearest neighbor computation, and prediction.

MuyGPyS.examples.fast_posterior_mean.**make_fast_multivariate_regressor**(*mmuygps*, *nbrs_lookup*, *train_features*, *train_targets*)

Convenience function for creating precomputed coefficient matrix and neighbor lookup data structure.

Parameters

- **muygps** – A trained MultivariateMuyGPS object.
- **nbrs_lookup** (*NN_Wrapper*) – A data structure supporting nearest neighbor queries into `train_features`.
- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_targets** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.

Return type
Tuple[ndarray, ndarray]

Returns

- *precomputed_coefficients_matrix* – A matrix of shape `(train_count, nn_count)` whose rows list the precomputed coefficients for each nearest neighbors set in the training data.
- *nn_indices* – An array supporting nearest neighbor queries.

MuyGPyS.examples.fast_posterior_mean.**make_fast_regressor**(*muygps*, *nbrs_lookup*, *train_features*, *train_targets*)

Convenience function for creating precomputed coefficient matrix and neighbor lookup data structure.

Parameters

- **muygps** (*MuyGPS*) – A (possibly trained) MuyGPS object.
- **nbrs_lookup** (*NN_Wrapper*) – A data structure supporting nearest neighbor queries into `train_features`.
- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_targets** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of response vectors of the train data.

Return type
Tuple[ndarray, ndarray]

Returns

- *precomputed_coefficients_matrix* – A matrix of shape `(train_count, nn_count)` whose rows list the precomputed coefficients for each nearest neighbors set in the training data.
- *nn_indices* – A numpy.ndarrray supporting nearest neighbor queries.

1.4.3 classify

Resources and high-level API for a simple classification workflow.

make_classifier() is a high-level API for creating and training *MuyGPyS.gp.muygps.MuyGPS* objects for classification. *make_multivariate_classifier()* is a high-level API for creating and training `MuyGPyS.gp.muygps.MultivariateMuyGPS` objects for classification.

do_classify() is a high-level api for executing a simple, generic classification workflow given data. It calls the maker APIs above and *classify_any()*.

`MuyGPyS.examples.classify.`**`classify_any`**(*surrogate*, *test_features*, *train_features*, *train_nbrs_lookup*, *train_labels*)

Simulatneously predicts the surrogate regression means for each test item.

Parameters

- **surrogate** (Union[*MuyGPS*, *MultivariateMuyGPS*]) – Surrogate regressor.
- **test_features** (ndarray) – Test observations of shape `(test_count, feature_count)`.
- **train_features** (ndarray) – Train observations of shape `(train_count, feature_count)`.
- **train_nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.
- **train_labels** (ndarray) – One-hot encoding of class labels for all training data of shape `(train_count, class_count)`.

Return type
Tuple[ndarray, Dict[str, float]]

Returns

- *predictions* – The surrogate predictions of shape `(test_count, class_count)` for each test observation.
- *timing* – Timing for the subroutines of this function.

`MuyGPyS.examples.classify.`**`do_classify`**(*test_features*, *train_features*, *train_labels*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_kwargs={}*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function for initializing a model and performing surrogate classification.

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Example

```
>>> import numpy as np
>>> from MuyGPyS.examples.classify import do_classify
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.optimize import Bayes_optimize
>>> train_features, train_responses = make_train()  # stand-in function
```

(continues on next page)

(continued from previous page)

```
>>> test_features, test_responses = make_test()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(0.5, (0.01, 1)),
...         ),
...     )
...     "noise": HomoscedasticNoise(1e-5),
... }
>>> muygps, nbrs_lookup, surrogate_predictions = do_classify(
...         test_features,
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_fn=cross_entropy_fn,
...         opt_fn=Bayes_optimize,
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> predicted_labels = np.argmax(surrogate_predictions, axis=1)
>>> true_labels = np.argmax(test_features, axis=1)
>>> acc = np.mean(predicted_labels == true_labels)
>>> print(f"obtained accuracy {acc}")
obtained accuracy: 0.973...
```

Parameters

- **test_features** (ndarray) – A matrix of shape `(test_count, feature_count)` whose rows consist of observation vectors of the test data.
- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_labels** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of label vectors for the training data.
- **nn_count** (`int`) – The number of nearest neighbors to employ.
- **batch_count** (`int`) – The batch size for hyperparameter optimization.
- **loss_fn** (*LossFn*) – The loss functor to use in hyperparameter optimization. Ignored if all of the parameters specified by `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_kwargs** (`Union[Dict, List[Dict], Tuple[Dict, ...]]`) – Parameters for the kernel, possibly including kernel type, deformation function, noise and scale hyperparameter specifications, and specifications for kernel hyperparameters. If all of the hyperparameters are fixed or are not given optimization bounds, no optimization will occur. If `"k_kwargs"` is a list of such dicts, will create a multivariate classifier model.

- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (Dict) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (bool) – If True, print summary statistics.

Return type
Tuple[Union[*MuyGPS*, *MultivariateMuyGPS*], *NN_Wrapper*, ndarray]

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into train_features.
- *surrogate_predictions* – A matrix of shape (test_count, response_count) whose rows indicate the surrogate predictions of the model. The predicted classes are given by the indices of the largest elements of each row.

MuyGPyS.examples.classify.**make_classifier**(*train_features*, *train_labels*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_kwargs={}*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function for creating MuyGPyS functor and neighbor lookup data structure.

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Example

```
>>> from MuyGPyS.examples.regress import make_classifier
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.optimize import Bayes_optimize
>>> from MuyGPyS.examples.classify import make_classifier
>>> train_features, train_responses = make_train()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(1.0, (1e-2, 1e2))
...         )
...     ),
...     "noise": HomoscedasticNoise(1e-5),
... }
>>> muygps, nbrs_lookup = make_classifier(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
```

(continues on next page)

(continued from previous page)

```
...         loss_fn=cross_entropy_fn,
...         opt_fn=Bayes_optimize,
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
```

Parameters

- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_labels** (ndarray) – A matrix of shape `(train_count, class_count)` whose rows consist of one-hot class label vectors of the train data.
- **nn_count** (int) – The number of nearest neighbors to employ.
- **batch_count** (int) – The number of elements to sample batch for hyperparameter optimization.
- **loss_fn** (*LossFn*) – The loss functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_kwargs** (Dict) – Parameters for the kernel, possibly including kernel type, deformation function, noise and scale hyperparameter specifications, and specifications for kernel hyperparameters. See *kernels* for examples and requirements. If all of the hyperparameters are fixed or are not given optimization bounds, no optimization will occur.
- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (Dict) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (bool) – Boolean If `True`, print summary statistics.

Return type

Tuple[*MuyGPS*, *NN_Wrapper*]

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into `train_features`.

MuyGPyS.examples.classify.**make_multivariate_classifier**(*train_features*, *train_labels*, *nn_count=30*, *batch_count=200*, *loss_fn=<MuyGPyS.optimize.loss.LossFn object>*, *opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>*, *k_args=[]*, *nn_kwargs={}*, *opt_kwargs={}*, *verbose=False*)

Convenience function for creating MuyGPyS functor and neighbor lookup data structure.

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Example

```
>>> from MuyGPyS.examples.classify import make_multivariate_classifier
>>> from MuyGPyS.gp.deformation import F2, Isotropy
>>> from MuyGPyS.gp.hyperparameter import Parameter
>>> from MuyGPyS.gp.kernels import RBF
>>> from MuyGPyS.gp.noise import HomoscedasticNoise
>>> from MuyGPyS.optimize import Bayes_optimize
>>> train_features, train_responses = make_train()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_args = [
...     {
...         "kernel": RBF(
...             deformation=Isotropy(
...                 metric=F2,
...                 length_scale=Parameter(0.5, (0.01, 1)),
...             ),
...         )
...         "noise": HomoscedasticNoise(1e-5),
...     },
...     {
...         "kernel": RBF(
...             deformation=Isotropy(
...                 metric=F2,
...                 length_scale=Parameter(0.5, (0.01, 1)),
...             ),
...         )
...         "noise": HomoscedasticNoise(1e-5),
...     },
... ]
>>> mmuygps, nbrs_lookup = make_multivariate_classifier(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_fn=cross_entropy_fn,
...         opt_fn=Bayes_optimize,
...         k_args=k_args,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
```

Parameters

- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_labels** (ndarray) – A matrix of shape `(train_count, class_count)` whose rows consist of one-hot encoded label vectors of the train data.
- **nn_count** (int) – The number of nearest neighbors to employ.
- **batch_count** (int) – The number of elements to sample batch for hyperparameter optimization.

- **loss_fn** (*LossFn*) – The loss functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **k_args** (`Union[List[Dict], Tuple[Dict, ...]]`) – A list of `response_count` dicts containing kernel initialization keyword arguments. Each dict specifies parameters for the kernel, possibly including noise and scale hyperparameter specifications and specifications for specific kernel hyperparameters. If all of the hyperparameters are fixed or are not given optimization bounds, no optimization will occur.
- **nn_kwargs** (`Dict`) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (`Dict`) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (`bool`) – If `True`, print summary statistics.

Return type
`Tuple[MultivariateMuyGPS, NN_Wrapper]`

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into `train_features`.

1.4.4 two-class classify with uq

Resources and high-level API for a two-class classification with UQ workflow.

Implements a two-class classification workflow with a bespoke uncertainty quantification tuning method. [muyskens2021star] describes this method and its application to a star-galaxy image separation problem.

`do_classify_uq()` is a high-level api for executing a two-class classification workflow with the uncertainty quantification. It calls the maker APIs *MuyGPyS.examples.classify.make_classifier()* and *MuyGPyS.examples.classify.make_multivariate_classifier()* to create and train models, and performs the inference using the functions `classify_two_class_uq()`, `make_masks()`, and `train_two_class_interval()`. `do_uq()` takes the true labels of the test data and the `surrgoate_prediction` and `masks` outputs to report the statistics of the confidence intervals associated with each supplied objective function.

`MuyGPyS.examples.two_class_classify_uq.`**classify_two_class_uq**(*surrogate*, *test_features*, *train_features*, *train_nbrs_lookup*, *train_labels*)

Simultaneously predicts the surrogate means and variances for each test item under the assumption of binary classification.

Parameters

- **surrogate** (`Union[MuyGPS, MultivariateMuyGPS]`) – Surrogate regressor.
- **test_features** (`ndarray`) – Test observations of shape `(test_count, feature_count)`.
- **train_features** (`ndarray`) – Train observations of shape `(train_count, feature_count)`.
- **train_nbrs_lookup** (*NN_Wrapper*) – Trained nearest neighbor query data structure.

- **train_labels** (ndarray) – One-hot encoding of class labels for all training data of shape `(train_count, class_count)`.

Return type
`Tuple[ndarray, ndarray, Dict[str, float]]`

Returns

- *means* – The surrogate predictions for each test observation of shape `(test_count, 2)`.
- *variances* – The posterior variances for each test observation of shape `(test_count,)`
- *timing* – Timing for the subroutines of this function.

`MuyGPyS.examples.two_class_classify_uq.`**do_classify_uq**(*test_features, train_features, train_labels, nn_count=30, opt_batch_count=200, uq_batch_count=500, loss_fn=<MuyGPyS.optimize.loss.LossFn object>, opt_fn=<MuyGPyS.optimize.chassis.OptimizeFn object>, uq_objectives=[<function <lambda>>, <function <lambda>>, <function <lambda>>, <function <lambda>>, <function <lambda>>], k_kwargs={}, nn_kwargs={}, opt_kwargs={}, verbose=False*)

Convenience function for initializing a model and performing two-class surrogate classification, while tuning uncertainty quantification.

Performs the classification workflow with uncertainty quantification tuning as described in [muyskens2021star].

Expected parameters include keyword argument dicts specifying kernel parameters and nearest neighbor parameters. See the docstrings of the appropriate functions for specifics.

Example

```
>>> import numpy as np
>>> from MuyGPyS.examples.regress import do_classify_uq, do_uq
>>> train_features, train_responses = make_train()  # stand-in function
>>> test_features, test_responses = make_test()  # stand-in function
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         deformation=Isotropy(
...             metric=F2,
...             length_scale=Parameter(0.5, (0.01, 1)),
...         ),
...     )
...     "noise": HomoscedasticNoise(1e-5),
... }
>>> muygps, nbrs_lookup, surrogate_predictions = do_classify_uq(
...         test_features,
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
```

(continues on next page)

(continued from previous page)

```
...         loss_fn=cross_entropy_fn,
...         opt_fn=Bayes_optimize,
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> accuracy, uq = do_uq(surrogate_predictions, test_responses, masks)
>>> print(f"obtained accuracy {accuracy}")
obtained accuracy: 0.973...
>>> print(f"obtained mask uq : \n{uq}")
obtained mask uq :
[[8.21000000e+02 8.53836784e-01 9.87144569e-01]
[8.59000000e+02 8.55646100e-01 9.87528717e-01]
[1.03500000e+03 8.66666667e-01 9.88845510e-01]
[1.03500000e+03 8.66666667e-01 9.88845510e-01]
[5.80000000e+01 6.72413793e-01 9.77972239e-01]]
```

Parameters

- **test_features** (ndarray) – A matrix of shape `(test_count, feature_count)` whose rows consist of observation vectors of the test data.
- **train_features** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows consist of observation vectors of the train data.
- **train_labels** (ndarray) – A matrix of shape `(train_count, response_count)` whose rows consist of label vectors for the training data.
- **nn_count** (int) – The number of nearest neighbors to employ.
- **opt_batch_count** (int) – The batch size for hyperparameter optimization.
- **uq_batch_count** (int) – The batch size for uncertainty quantification calibration.
- **loss_fn** (*LossFn*) – The loss functor to use in hyperparameter optimization. Ignored if all of the parameters specified by `k_kwargs` are fixed.
- **opt_fn** (*OptimizeFn*) – The optimization functor to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed.
- **uq_objectives** (Union[List[Callable], Tuple[Callable, ...]]) – list(Callable) List of `objective_count`functions taking four arguments: bit masks `alpha` and `beta` - the type 1 and type 2 error counts at each grid location, respectively - and the numbers of correctly and incorrectly classified training examples. Used to tune the scale parameter σ^2 for setting confidence intervals. See `MuyGPyS.examples.classify.example_lambdas` for examples.
- **k_kwargs** (Dict) – Parameters for the kernel, possibly including kernel type, deformation function, noise and scale hyperparameter specifications, and specifications for kernel hyperparameters. If all of the hyperparameters are fixed or are not given optimization bounds, no optimization will occur.
- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.
- **opt_kwargs** (Dict) – Parameters for the wrapped optimizer. See the docs of the corresponding library for supported parameters.
- **verbose** (bool) – If `True`, print summary statistics.

Return type
Tuple[*MuyGPS*, *NN_Wrapper*, ndarray, ndarray]

Returns

- *muygps* – A (possibly trained) MuyGPs object.
- *nbrs_lookup* – A data structure supporting nearest neighbor queries into `train_features`.
- *surrogate_predictions* – A matrix of shape `(test_count, response_count)` whose rows indicate the surrogate predictions of the model. The predicted classes are given by the indices of the largest elements of each row.
- *masks* – A matrix of shape `(objective_count, test_count)` whose rows consist of index masks into the training set. Each `True` index includes 0.0 within the associated prediction's confidence interval.

MuyGPyS.examples.two_class_classify_uq.**do_uq**(*surrogate_predictions*, *test_labels*, *masks*)

Convenience function performing uncertainty quantification given predicted labels and ground truth for a given set of confidence interval scales.

Parameters

- **predictions** – A matrix of shape `(test_count, class_count)` whose rows consist of the surrogate predictions.
- **test_labels** (ndarray) – A matrix of shape `(test_count, class_count)` listing the true one-hot encodings of each test observation's class.
- **masks** (ndarray) – A matrix of shape `(objective_count, test_count)` whose rows consist of index masks into the training set. Each `True` index includes `0.0` within the associated prediction's confidence interval.

Return type
Tuple[float, ndarray]

Returns

- *accuracy* – The accuracy over all of the test data.
- *uq* – A matrix of shape `(objective_count, 3)` listing the uncertainty quantification associated with each input mask (i.e. each objective function). The first column is the total number of ambiguous samples, i.e. those whose confidence interval contains the `mid_value`, usually `0.0`. The second column is the accuracy of the ambiguous samples. The third column is the accuracy of the unambiguous samples.

MuyGPyS.examples.two_class_classify_uq.**make_masks**(*predictions*, *cutoffs*, *variances*, *mid_value*)

Compute boolean masks over all of the test data indicating which test indices are considered ambiguous

Parameters

- **predictions** (ndarray) – A matrix of shape `(test_count, class_count)` whose rows consist of the surrogate predictions.
- **cutoffs** (ndarray) – A vector of shape `(objective_count,)` indicating the confidence interval scale parameter σ^2 that minimizes each of the considered objective function.
- **variances** (ndarray) – A vector of shape `(test_count, 1)` indicating the diagonal posterior variance of each test item.
- **mid_value** (float) – The discriminating value determining absolute uncertainty. Usually `0.0` or `0.5`.

Return type
ndarray

Returns
A matrix of shape `(objective_count, test_count)` whose rows consist of index masks into the training set. Each `True` index includes `mid_value` within the associated prediction's confidence interval.

MuyGPyS.examples.two_class_classify_uq.**train_two_class_interval**(*surrogate*, *batch_indices*, *batch_nn_indices*, *train_features*, *train_responses*, *train_labels*, *objective_fns*)

For 2-class classification problems, get estimate of the confidence interval scaling parameter.

Parameters

- **surrogate** (*MuyGPS*) – Surrogate regressor.
- **batch_indices** (ndarray) – Batch observation indices of shape `(batch_count)`.
- **batch_nn_indices** (ndarray) – Indices of the nearest neighbors of shape `(batch_count, nn_count)`.
- **train** – The full training data matrix of shape `(train_count, feature_count)`.
- **train_responses** (ndarray) – One-hot encoding of class labels for all training data of shape `(train_count, class_count)`.
- **train_labels** (ndarray) – List of class labels for all training data of shape `(train_count,)`.
- **objective_fns** (Union[List[Callable], Tuple[Callable, ...]]) – A collection of `objective_count` functions taking the four arguments bit masks alpha and beta - the type 1 and type 2 error counts at each grid location, respectively - and the numbers of correctly and incorrectly classified training examples. Each objective function effervesces a cutoff value to calibrate UQ for class decision-making.

Return type
ndarray

Returns
A vector of shape `(objective_count)` indicating the confidence interval scale parameter that minimizes each considered objective function.

1.4.5 muygps_torch

Resources and high-level API for a deep kernel learning with MuyGPs.

train_deep_kernel_muygps() is a high-level API for training deep kernel MuyGPs models for regression.

predict_model() is a high-level API for generating predictions at test locations given a trained model.

MuyGPyS.examples.muygps_torch.**predict_model**(*model*, *test_features*, *train_features*, *train_responses*, *nbrs_lookup*, *nn_count*)

Generate predictions using a PyTorch model containing a `MuyGPyS.torch.muygps_layer.MuyGPs_layer` layer or a `MuyGPyS.torch.muygps_layer.MultivariateMuyGPs_layer` layer in its structure. Note that the custom PyTorch layers for MuyGPs objects only support the Matern kernel. Support for more kernels will be added in future releases.

Example

```
>>> #model must be defined as a PyTorch model inheriting from
... #torch.nn.Module. Must have two components: model.embedding
... #(e.g., a neural net) and another component model.GP_layer.
>>> from MuyGPyS.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> train, test = _make_gaussian_data(10000, 1000, 100, 10)
>>> nn_count = 10
>>> nbrs_lookup = NN_Wrapper(train['input'], nn_count, nn_method="hnsw")
>>> predictions, variances = predict_model(
... model,
... torch.from_numpy(test['input']),
... torch.from_numpy(train['input']),
... torch.from_numpy(train['output']),
... nbrs_lookup,
... nn_count)
```

Parameters

- **model** – A custom PyTorch.nn.Module object containing an embedding component and one MuyGPs_layer or MultivariateMuyGPS_layer layer.
- **test_features** (Tensor) – A torch.Tensor of shape `(test_count, feature_count)` containing the test features to be regressed.
- **train_features** (Tensor) – A torch.Tensor of shape `(train_count, feature_count)` containing the training features.
- **train_responses** (Tensor) – A torch.Tensor of shape `(train_count, response_count)` containing the training responses corresponding to each feature.
- **nbrs_lookup** (*NN_Wrapper*) – A NN_Wrapper nearest neighbor lookup data structure.

Returns

- *predictions* – A torch.Tensor of shape `(test_count, response_count)` whose rows are the predicted response for each of the given test feature.
- *variances* – A torch.Tensor of shape `(batch_count,)` consisting of the diagonal elements of the posterior variance, or a matrix of shape `(batch_count, response_count)` for a multidimensional response.

MuyGPyS.examples.muygps_torch.**predict_multiple_model**(*model*, *test_features*, *train_features*, *train_responses*, *nbrs_lookup*, *nn_count*)

Generate predictions using a PyTorch model containing a `MuyGPyS.torch.muygps_layer.MultivariateMuyGPs_layer` in its structure. Meant for the case in which there is more than one GP model used to model multiple outputs. Note that the custom PyTorch MultivariateMuyGPs_layer objects only support the Matern kernel. Support for more kernels will be added in future releases.

Parameters

- **model** – A custom PyTorch.nn.Module object containing an `embedding` component and one `MuyGPyS.torch.muygps_layer.MultivariateMuyGPs_layer` layer.
- **test_features** (Tensor) – A torch.Tensor of shape `(test_count, feature_count)` containing the test features to be regressed.

- **train_features** (Tensor) – A torch.Tensor of shape `(train_count, feature_count)` containing the training features.
- **train_responses** (Tensor) – A torch.Tensor of shape `(train_count, response_count)` containing the training responses corresponding to each feature.
- **nbrs_lookup** (*NN_Wrapper*) – A NN_Wrapper nearest neighbor lookup data structure.

Returns

- *predictions* – A torch.Tensor of shape `(test_count, response_count)` whose rows are the predicted response for each of the given test feature.
- *variances* – A torch.Tensor of shape `(batch_count,)` consisting of the diagonal elements of the posterior variance, or a matrix of shape `(batch_count, response_count)` for a multidimensional response.

MuyGPyS.examples.muygps_torch.**predict_single_model**(*model*, *test_features*, *train_features*, *train_responses*, *nbrs_lookup*, *nn_count*)

Generate predictions using a PyTorch model containing at least one `MuyGPyS.torch.muygps_layer.MuyGPs_layer` in its structure. Note that the custom PyTorch MuyGPs_layer objects only support the Matern kernel. Support for more kernels will be added in future releases.

Parameters

- **model** – A custom PyTorch.nn.Module object containing an `embedding` component and one `MuyGPyS.torch.muygps_layer.MuyGPs_layer` layer.
- **test_features** (Tensor) – A torch.Tensor of shape `(test_count, feature_count)` containing the test features to be regressed.
- **train_features** (Tensor) – A torch.Tensor of shape `(train_count, feature_count)` containing the training features.
- **train_responses** (Tensor) – A torch.Tensor of shape `(train_count, response_count)` containing the training responses corresponding to each feature.
- **nbrs_lookup** (*NN_Wrapper*) – A NN_Wrapper nearest neighbor lookup data structure.

Returns

- *predictions* – A torch.Tensor of shape `(test_count, response_count)` whose rows are the predicted response for each of the given test feature.
- *variances* – A torch.Tensor of shape `(batch_count,response_count)` shape consisting of the diagonal elements of the posterior variance.

MuyGPyS.examples.muygps_torch.**train_deep_kernel_muygps**(*model*, *train_features*, *train_responses*, *batch_indices*, *nbrs_lookup*, *training_iterations=10*, *optimizer_method=<class 'torch.optim.adam.Adam'>*, *learning_rate=0.001*, *scheduler_decay=0.95*, *loss_function='lool'*, *update_frequency=1*, *verbose=False*, *nn_kwargs={}*)

Train a PyTorch model containing an embedding component and a `MuyGPyS.torch.muygps_layer.MuyGPs_layer` layer or a `MuyGPyS.torch.muygps_layer. MultivariateMuyGPs_layer` layer in its structure. Note that the custom PyTorch layers for MuyGPs models only support the Matern kernel. Support for more kernels will be added in future releases.

Example

```
>>> #model must be defined as a PyTorch model inheriting from
... #torch.nn.Module. Must have two components: model.embedding
... #(e.g., a neural net) and another component model.GP_layer.
>>> from MuyGPyS.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> from MuyGPyS.examples.muygps_torch import train_deep_kernel_muygps
>>> from MuyGPyS._src.optimize.loss import _lool_fn as lool_fn
>>> train, test = _make_gaussian_data(10000, 1000, 100, 10)
>>> nn_count = 10
>>> nbrs_lookup = NN_Wrapper(train['input'], nn_count, nn_method="hnsw")
>>> batch_count = 100
>>> train_count = 10000
>>> batch_indices, batch_nn_indices = sample_batch(nbrs_lookup,
... batch_count, train_count)
>>> nbrs_struct, model_trained = train_deep_kernel_muygps(
... model=model,
... train_features=torch.from_numpy(train['input']),
... train_responses=torch.from_numpy(train['output']),
... batch_indices=torch.from_numpy(batch_indices),
... nbrs_lookup=nbrs_lookup,
... training_iterations=10,
... optimizer_method=torch.optim.Adam,
... learning_rate=1e-3,
... scheduler_decay=0.95,
... loss_function=lool_fn,
... update_frequency=1)
```

Parameters

- **model** – A custom PyTorch.nn.Module object containing at least one embedding layer and one MuyGPs_layer or MultivariateMuyGPS_layer layer.
- **train_features** (Tensor) – A torch.Tensor of shape `(train_count, feature_count)` containing the training features.
- **train_responses** (Tensor) – A torch.Tensor of shape `(train_count, response_count)` containing the training responses corresponding to each feature.
- **batch_indices** (Tensor) – A torch.Tensor of shape `(batch_count,)` containing the indices of the training batch.
- **nbrs_lookup** (*NN_Wrapper*) – A NN_Wrapper nearest neighbor lookup data structure.
- **training_iterations** – The number of training iterations to be used in training.
- **method** (*optimizer*) – An optimization method from the torch.optim class.
- **learning_rate** – The learning rate to be applied during training.
- **schedule_decay** – The exponential decay rate to be applied to the learning rate.
- **function** (*loss*) – The loss function to be used in training. Defaults to "lool" for leave-one-out likelihood. Other options are "mse" for mean-squared error, "ce" for cross entropy loss, "bce" for binary cross entropy loss, and "l1" for L1 loss.

- **update_frequency** – Tells the training procedure how frequently the nearest neighbor structure should be updated. An update frequency of n indicates that every n epochs the nearest neighbor structure should be updated.
- **verbose** – Indicates whether or not to include print statements during training.
- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.

Returns

- *nbrs_lookup* – A NN_Wrapper object containing the nearest neighbors of the embedded training data.
- *model* – A trained deep kernel MuyGPs model.

MuyGPyS.examples.muygps_torch.**update_nearest_neighbors**(*model*, *train_features*, *train_responses*, *batch_indices*, *nn_count*, *nn_kwargs={}*)

Update the nearest neighbors after deformation via a PyTorch model containing an embedding component and a `MuyGPyS.torch.muygps_layer.MuyGPs_layer` layer or a `MuyGPyS.torch.muygps_layer.MultivariateMuyGPs_layer` layer in its structure.

Example

```
>>> #model must be defined as a PyTorch model inheriting from
... #torch.nn.Module. Must have two components: model.embedding
... #(e.g., a neural net) and another component model.GP_layer.
>>> from MuyGPyS.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> from MuyGPyS.examples.muygps_torch import update_nearest_neighbors
>>> train, test = _make_gaussian_data(10000, 1000, 100, 10)
>>> nn_count = 10
>>> batch_count = 100
>>> train_count = 10000
>>> batch_indices, batch_nn_indices = sample_batch(nbrs_lookup, batch_count, train_
↪count)
>>> nbrs_struct, model_trained = update_nearest_neighbors(
... model=model,
... train_features=torch.from_numpy(train['input']),
... train_responses=torch.from_numpy(train['output']),
... batch_indices=torch.from_numpy(batch_indices),
... nn_count=nn_count,)
```

Parameters

- **model** – A custom PyTorch.nn.Module object containing at least one embedding layer and one MuyGPs_layer or MultivariateMuyGPS_layer layer.
- **train_features** (Tensor) – A torch.Tensor of shape `(train_count, feature_count)` containing the training features.
- **train_responses** (Tensor) – A torch.Tensor of shape `(train_count, response_count)` containing the training responses corresponding to each feature.
- **batch_indices** (Tensor) – A torch.Tensor of shape `(batch_count,)` containing the indices of the training batch.

- **nn_count** (int) – A torch.int64 giving the number of nearest neighbors.
- **nn_kwargs** (Dict) – Parameters for the nearest neighbors wrapper. See *MuyGPyS.neighbors.NN_Wrapper* for the supported methods and their parameters.

Returns

- *nbrs_lookup* – A NN_Wrapper object containing the updated nearest neighbors of the embedded training data.
- *model* – A deep kernel MuyGPs model with updated nearest neighbors.

1.5 torch

MuyGPyS.torch module reference.

1.5.1 muygps_layer

MuyGPs PyTorch implementation

class MuyGPyS.torch.muygps_layer.**MuyGPs_layer**(*muygps_model*, *batch_indices*, *batch_nn_indices*, *batch_targets*, *batch_nn_targets*)

MuyGPs model written as a custom PyTorch layer using nn.Module.

Implements the MuyGPs algorithm as articulated in [muyskens2021muygps]. See documentation on MuyGPs class for more detail.

The MuyGPs_layer class only supports the Matern kernel currently. More kernels will be added to the torch module of MuyGPs in future releases.

PyTorch does not currently support the Bessel function required to compute the Matern kernel for non-special case smoothness values of ν, e.g. 1/2, 3/2, 5/2, and ∞. The MuyGPs layer allows the lengthscale parameter ρ to be trained (provided an initial value by the user) as well as the homoscedastic τ^2 noise prior variance.

The MuyGPs layer returns the posterior mean, posterior variance, and a vector of σ^2 indicating the scale parameter associated with the posterior variance of each dimension of the response.

Example

```
>>> from MuyGPyS.torch.muygps_layer import MuyGPs_layer
>>> muygps_model = MuyGPS(
...     Matern(
...         smoothness=ScalarParam(0.5),
...         deformation=Isotropy(
...             metric=l2,
...             length_scale=ScalarParam(1.0)
...         ),
...     ),
...     noise=HomoscedasticNoise(1e-5),
... )
>>> batch_indices = torch.arange(100,)
>>> batch_nn_indices = torch.arange(100,)
>>> batch_targets = torch.ones(100,)
>>> batch_nn_targets = torch.ones(100,)
```

(continues on next page)

(continued from previous page)

```
>>> muygps_layer_object = MuyGPs_layer(
... muygps_model,
... batch_indices,
... batch_nn_indices,
... batch_targets,
... batch_nn_targets)
```

Parameters

- **muygps_model** (*MuyGPS*) – A MuyGPs object providing the Gaussian Process final layer.
- **batch_indices** – A torch.Tensor of shape `(batch_count,)` containing the indices of the training data to be sampled for training.
- **batch_nn_indices** – A torch.Tensor of shape `(batch_count, nn_count)` containing the indices of the k nearest neighbors of the batched training samples.
- **batch_targets** – A torch.Tensor of shape `(batch_count, response_count)` containing the responses corresponding to each batched training sample.
- **batch_nn_targets** – A torch.Tensor of shape `(batch_count, nn_count, response_count)` containing the responses corresponding to the nearest neighbors of each batched training sample.
- **kwargs** – Addition parameters to be passed to the kernel, possibly including additional hyperparameter dicts and a metric keyword.

forward(*x*)

Produce the output of a MuyGPs custom PyTorch layer.

Returns

- *predictions* – A torch.ndarray of shape `(batch_count, response_count)` whose rows are the predicted response for each of the given batch feature.
- *variances* – A torch.ndarray of shape `(batch_count,response_count)` consisting of the diagonal elements of the posterior variance.

1.6 Univariate Regression Tutorial

This notebook walks through a simple regression workflow and explains the components of `MuyGPyS`.

[2]:
```
import numpy as np

from MuyGPyS._test.sampler import UnivariateSampler, print_results
from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.deformation import Isotropy, l2
from MuyGPyS.gp.hyperparameter import AnalyticScale, Parameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.noise import HomoscedasticNoise
```

(continues on next page)

(continued from previous page)

```
from MuyGPyS.neighbors import NN_Wrapper
from MuyGPyS.optimize import Bayes_optimize
from MuyGPyS.optimize.batch import sample_batch
from MuyGPyS.optimize.loss import lool_fn
```

We will set a random seed here for consistency when building docs. In practice we would not fix a seed.

[3]:
```
np.random.seed(0)
```

1.6.1 Sampling a Curve from a Conventional GP

This notebook will use a simple one-dimensional curve sampled from a conventional Gaussian process. We will specify the domain as a grid on a one-dimensional surface and divide the observations into train and test data.

Feel free to download the source notebook and experiment with different parameters.

First we specify the region of space, the data size, and the proportion of the train/test split.

[4]:
```
data_count = 3000
train_ratio = 0.075
```

We will assume that the true data is produced with no noise, so we specify a very small noise prior for numerical stability. This is an idealized experiment with effectively no instrument error.

[5]:
```
nugget_noise = HomoscedasticNoise(1e-14)
```

We will perturb our simulated observations (the training data) with some i.i.d Gaussian measurement noise.

[6]:
```
measurement_noise = HomoscedasticNoise(1e-7)
```

Finally, we will specify kernel hyperparameters `smoothness` and `length_scale`. The `length_scale` scales the distances that are inputs to the kernel function, while the `smoothness` parameter determines how differentiable the GP prior is. The larger `smoothness` grows, the smoother sampled functions will become.

[7]:
```
sim_length_scale = Parameter(0.05)
sim_smoothness = Parameter(2.0)
```

We use all of these parameters to define a Matérn kernel GP and a sampler for convenience. The `UnivariateSampler` class is a convenience class for this tutorial, and is not a part of the library.

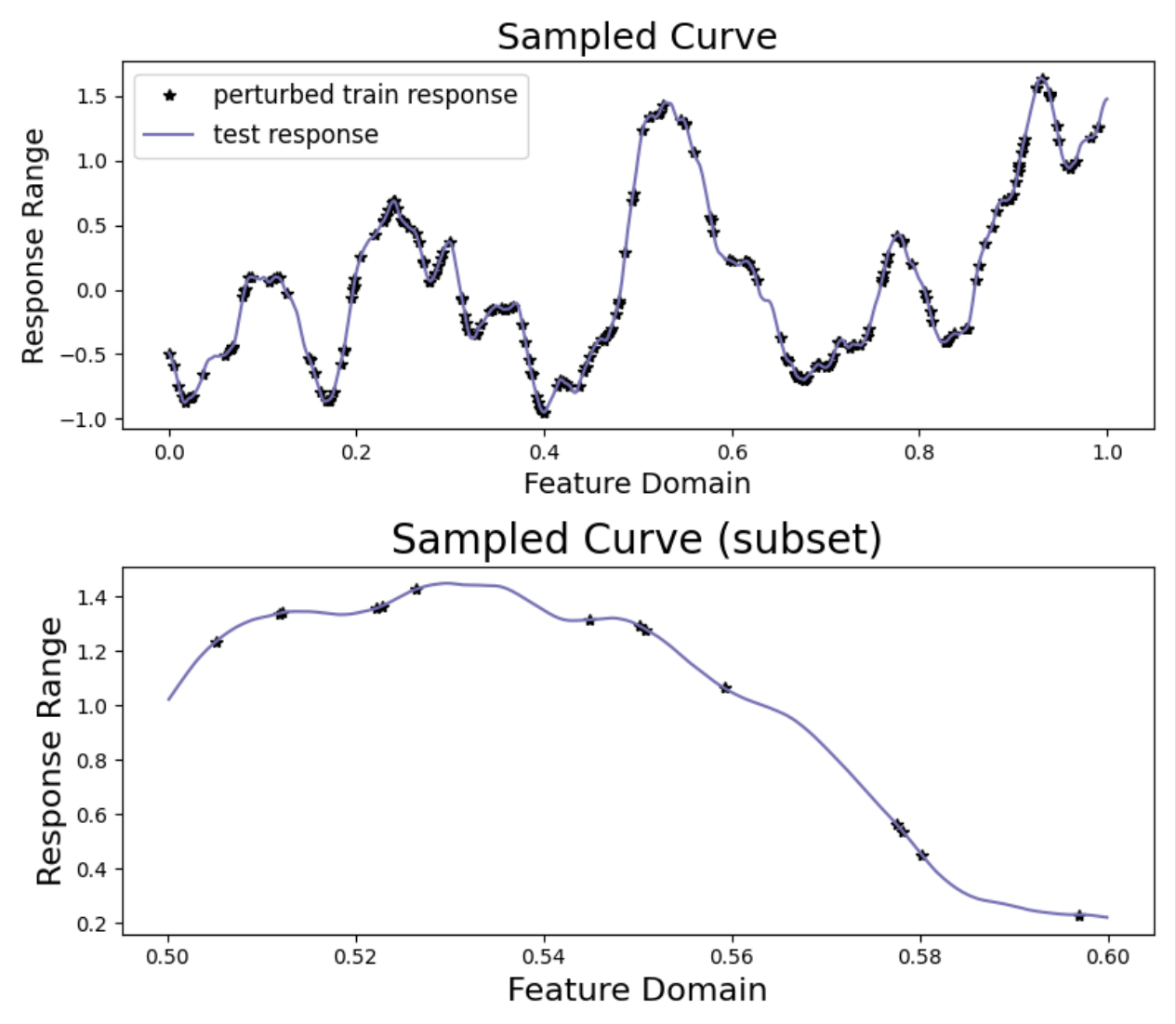
[8]:
```
sampler = UnivariateSampler(
    data_count=data_count,
    train_ratio=train_ratio,
    kernel=Matern(
        smoothness=sim_smoothness,
        deformation=Isotropy(
            l2,
            length_scale=sim_length_scale,
        ),
    ),
    noise=nugget_noise,
    measurement_noise=measurement_noise,
)
```

Finally, we will sample a curve from this GP prior and visualize it. Note that we perturb the train responses (the values that our model will actual receive) with Gaussian measurement noise. Further note that this is not especially fast, as sampling from a conventional Gaussian process requires computing the Cholesky decomposition of a `(data_count, data_count)` matrix.

[9]:
```
train_features, test_features = sampler.features()
```

[10]:
```
train_responses, test_responses = sampler.sample()
```

[11]:
```
sampler.plot_sample()
```



We will now attempt to recover the response on the held-out test data by training a univariate `MuyGPS` model on the perturbed training data.

1.6.2 Constructing Nearest Neighbor Lookups

NN_Wrapper is an api for tasking several KNN libraries with the construction of lookup indexes that empower fast training and inference. The wrapper constructor expects the training features, the number of nearest neighbors, and a method string specifying which algorithm to use, as well as any additional kwargs used by the methods. Currently supported implementations include exact KNN using sklearn ("exact") and approximate KNN using hnsw ("hnsw", requires installing `MuyGPyS` using the `hnswlib` extras flag).

Here we construct an exact KNN data example with k = 30

[12]:
```
nn_count = 30
nbrs_lookup = NN_Wrapper(train_features, nn_count, nn_method="exact", algorithm="ball_
↪tree")
```

This `nbrs_lookup` index is then usable to find the nearest neighbors of queries in the training data.

1.6.3 Sampling Batches of Data

`MuyGPyS` includes convenience functions for sampling batches of data from existing datasets. These batches are returned in the form of row indices, both of the sampled data as well as their nearest neighbors.

Here we sample a random batch of `train_count` elements. This results in using *all* of the train data for training. We only do that in this case because this example uses a relatively small amount of data. In practice, we would instead set `batch_count` to a resaonable number. In practice we find reasonable values to be in the range of 500-2000.

[13]:
```
batch_count = sampler.train_count
batch_indices, batch_nn_indices = sample_batch(
    nbrs_lookup, batch_count, sampler.train_count
)
```

These `indices` and `nn_indices` arrays are the basic operating blocks of `MuyGPyS` linear algebraic inference. The elements of `indices.shape == (batch_count,)` lists all of the row indices into `train_features` and `train_responses` corresponding to the sampled data. The rows of `nn_indices.shape == (batch_count, nn_count)` list the row indices into `train_features` and `train_responses` corresponding to the nearest neighbors of the sampled data.

While the user need not use the *MuyGPyS.optimize.batch* sampling tools to construct these data, they will need to construct similar indices into their data in order to use `MuyGPyS`.

1.6.4 Setting and Optimizing Hyperparameters

One initializes a *MuyGPS* object by indicating the kernel, as well as optionally specifying hyperparameters.

Consider the following example, which constructs a MuyGPs object with a Matérn kernel. The `MuyGPS` object expects a kernel function object, a `noise` noise model parameter, and a variance scale parameter. We will use an `AnalyticScale` instance, which has an analytic optimization method. The `Matern` object expects a deformation function object and a smoothness parameter. We use an isotropic deformation, so `Isotropy` expects a Callable indicating the metric to use (`l2` distance in this case) and a length scale parameter.

Hyperparameters can be optionally given a lower and upper optimization bound tuple on creation. If `"bounds"` is set, one can also set the hyperparameter value with the arguments `"sample"` and `"log_sample"` to generate a uniform or log uniform sample, respectively. Hyperparameters without optimization bounds will remain fixed during optimization.

In this experiment, we make the simplifying assumptions that we know the true `length_scale` and `measurement_noise`, and reuse the parameters used to create the sampler. We will try to learn the smoothness parameter.

[14]:
```
muygps = MuyGPS(
    kernel=Matern(
        smoothness=Parameter("log_sample", (0.1, 5.0)),
        deformation=Isotropy(
            l2,
            length_scale=sim_length_scale,
        ),
    ),
    noise=measurement_noise,
    scale=AnalyticScale(),
)
```

There is one additionally common hyperparameter, the `scale` variance scale parameter, that is treated differently than the others. `scale` cannot be directly set by the user, and always initializes to the value `"unlearned"`. We will show how to train `scale` below. All hyperparameters other than `scale` are assumed to be fixed unless otherwise specified.

MuyGPyS depends upon linear operations on specially-constructed tensors in order to efficiently estimate GP realizations. Constructing these tensors depends upon the nearest neighbor index matrices that we described above. We can construct a distance tensor coalescing all of the square pairwise distance matrices of the nearest neighbors of a batch of points.

This snippet constructs a matrix of shape `(batch_count, nn_count)` coalescing all of the crosswise distances between our set of points and their nearest neighbors. This method is verbose; we will see a more concise version below.

[15]:
```
batch_crosswise_dists = muygps.kernel.deformation.crosswise_tensor(
    train_features,
    train_features,
    batch_indices,
    batch_nn_indices,
)
```

We can similarly construct a difference tensor of shape `(batch_count, nn_count, nn_count)` containing the pairwise distances of the nearest neighbor sets of each sampled batch element.

[16]:
```
pairwise_dists = muygps.kernel.deformation.pairwise_tensor(
    train_features, batch_nn_indices
)
```

The `MuyGPS` object we created earlier allows us to easily realize corresponding kernel tensors by way of its kernel function. We do not need to construct these directly for training - out optimization function will do so internally.

[17]:
```
Kcross = muygps.kernel(batch_crosswise_dists)
Kin = muygps.kernel(pairwise_dists)
```

In order to perform Gaussian process regression, we must utilize these kernel tensors in conjunction with their associated known responses. We can construct these matrices using the index matrices we derived earlier.

[18]:
```
batch_targets = train_responses[batch_indices]
batch_nn_targets = train_responses[batch_nn_indices]
```

Since we often must realize `batch_targets` and `batch_nn_targets` in close proximity to `batch_crosswise_dists` and `batch_pairwise_dists`, the `MuyGPS` class includes a convenience function bundles these operations.

[19]:
```
(
    batch_crosswise_dists,
    batch_pairwise_dists,
    batch_targets,
    batch_nn_targets,
) = muygps.make_train_tensors(
    batch_indices,
    batch_nn_indices,
    train_features,
    train_responses,
)
```

[20]:
```
Kcross = muygps.kernel(batch_crosswise_dists)
Kin = muygps.kernel(pairwise_dists)
```

We supply a convenient leave-one-out cross-validation utility functor class (`OptimizeFn <../MuyGPyS/gp/optimize.rst>`__) that utilizes these tensors to repeatedly realize kernel tensors during optimization. Optimization implementations are objects of this class. The library currently natively supports two optimization workflows: This optimization loop wraps a few different batch optimization methods (importable from `MuyGPyS.optimize`): * `Bayes_optimize <../MuyGPyS/gp/optimize.rst>`__, which wraps `bayes_opt.BayesianOptimization <https://github.com/fmfn/BayesianOptimization>`__ in batch mode only. * `L_BFGS_B_optimize <../MuyGPyS/gp/optimize.rst>`__, which wraps the "L-BFGS-B" implementation in `scipy.optimize.minimize <https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.minimize.html>`__. It is possible to create a new instance of `OptimizeFn` to support custom outer-loop optimizations.

This example uses `Bayes_optimize`. There are several additional parameters inherited from the implementation that a user might want to set. In particular, `init_points` (the number of "exploration" objective function evaluations to perform) and `n_iter` (the number of "exploitation" objective function evaluations to perform) are of use to most users. This example also sets `random_state` for consistency. See the documentation of BayesianOptimization for more examples.

[21]:
```
muygps_optimized = Bayes_optimize(
    muygps,
    batch_targets,
    batch_nn_targets,
    batch_crosswise_dists,
    batch_pairwise_dists,
    loss_fn=lool_fn,
    verbose=True,
    random_state=1,
    init_points=5,
    n_iter=15,
)
```

```
parameters to be optimized: ['smoothness']
bounds: [[0.1 5. ]]
initial x0: [0.92898658]
|   iter    |  target   | smooth... |
-------------------------------------
| 1         | 1.826e+03 | 0.929     |
| 2         | 2.359e+03 | 2.143     |
| 3         | 1.953e+03 | 3.63      |
| 4         | 614.4     | 0.1006    |
| 5         | 2.309e+03 | 1.581     |
```

(continues on next page)

(continued from previous page)

```
| 6          | 1.707e+03 | 0.8191    |
| 7          | 1.48e+03  | 5.0       |
| 8          | 2.201e+03 | 2.83      |
| 9          | 2.373e+03 | 1.883     |
| 10         | 2.373e+03 | 1.996     |
| 11         | 2.375e+03 | 1.938     |
| 12         | 2.375e+03 | 1.938     |
| 13         | 2.375e+03 | 1.938     |
| 14         | 2.375e+03 | 1.938     |
| 15         | 2.375e+03 | 1.938     |
| 16         | 2.375e+03 | 1.938     |
| 17         | 2.375e+03 | 1.938     |
| 18         | 2.375e+03 | 1.945     |
| 19         | 2.375e+03 | 1.927     |
| 20         | 2.375e+03 | 1.95      |
| 21         | 2.375e+03 | 1.926     |
=====================================
```

As it is a variance scaling parameter that is insensitive to prediction-based optimization, we separately optimize `scale`. In this case, we invoke muygps.optimize_scale(), which approximates `scale` based upon the mean of the closed-form `scale` solutions associated with each of its batched nearest neighbor sets. Note that this method is sensitive to several factors, include `batch_count`, `nn_count`, and the overall size of the dataset, tending to perform better as each of these factors increases. If we had instead used the optimization-free `MuyGPyS.gp.hyperparameter.scale.Scale` class, this function would effectively be a no-op and leave the value of `muygps_optimized.scale` unchanged.

This is usually performed after optimizing other hyperparameters.

[22]:
```
muygps_optimized = muygps_optimized.optimize_scale(batch_pairwise_dists, batch_nn_
↪targets)
```

1.6.5 Inference

With set (or learned) hyperparameters, we are able to use the `muygps` object to predict the response of test data. Several workflows are supported.

See below a simple regression workflow, using the data structures built up in this example. This workflow uses the compact tensor-making function *make_predict_tensors()* to succinctly create tensors defining the `pairwise_dists` among each nearest neighbor set, the `crosswise_dists` between each test point and its nearest neighbor set, and the `nn_targets` or responses of the nearest neighbors in each set. We then create the `Kcross` cross-covariance matrix and `Kin` covariance tensor and pass them to *MuyGPS.posterior_mean()* and *MuyGPS.posterior_variance()* in order to obtain our predictions.

First, we find the indices of the nearest neighbors of all of the test elements and save the results in `test_nn_indices`.

[23]:
```
test_count = test_features.shape[0]
indices = np.arange(test_count)
test_nn_indices, _ = nbrs_lookup.get_nns(test_features)
```

We then use `nn_indices` to make difference and target tensors for the test data. These tensors are similar to those used for batch optimization, except that we do not assume that we know the targets of the

[24]:
```
(
    test_crosswise_dists,
```

(continues on next page)

(continued from previous page)

```
    test_pairwise_dists,
    test_nn_targets,
) = muygps.make_predict_tensors(
    indices,
    test_nn_indices,
    test_features,
    train_features,
    train_responses,
)
```

We create the kernel tensors for the optimized model.

[25]:
```
Kcross = muygps_optimized.kernel(test_crosswise_dists)
Kin = muygps_optimized.kernel(test_pairwise_dists)
```

This regression example returns predictions (posterior means) and variances for each element of the test dataset. These variances are in the form of diagonal and independent variances that encode the uncertaintainty of the model's predictions at each test point. To scale the variances, they should be multiplied by the trained `scale` scaling parameters, of which there will be one scalar associated with each dimension of the response.

We use the `MuyGPS.posterior_mean()` and `MuyGPS.posterior_variance()` functions to find the posterior means and variances associated with each training prediction. The 95% confidence interval sizes are straightforward to compute as $\sigma * 1.96$, where σ is the standard deviation. We compute coverage as the proportion of posterior means that differ from the true response by no more than the confidence interval size. We coverage for the 95% confidence intervals ideally should be near 95%.

[26]:
```
predictions = muygps_optimized.posterior_mean(Kin, Kcross, test_nn_targets)
variances = muygps_optimized.posterior_variance(Kin, Kcross)
confidence_intervals = np.sqrt(variances) * 1.96
coverage = np.count_nonzero(np.abs(test_responses - predictions) < confidence_intervals)␣
↪/ test_count
```

Finally, we use RMSE, the mean diagonal variance and confidence interval size, as well as coverage to analyze our fit.

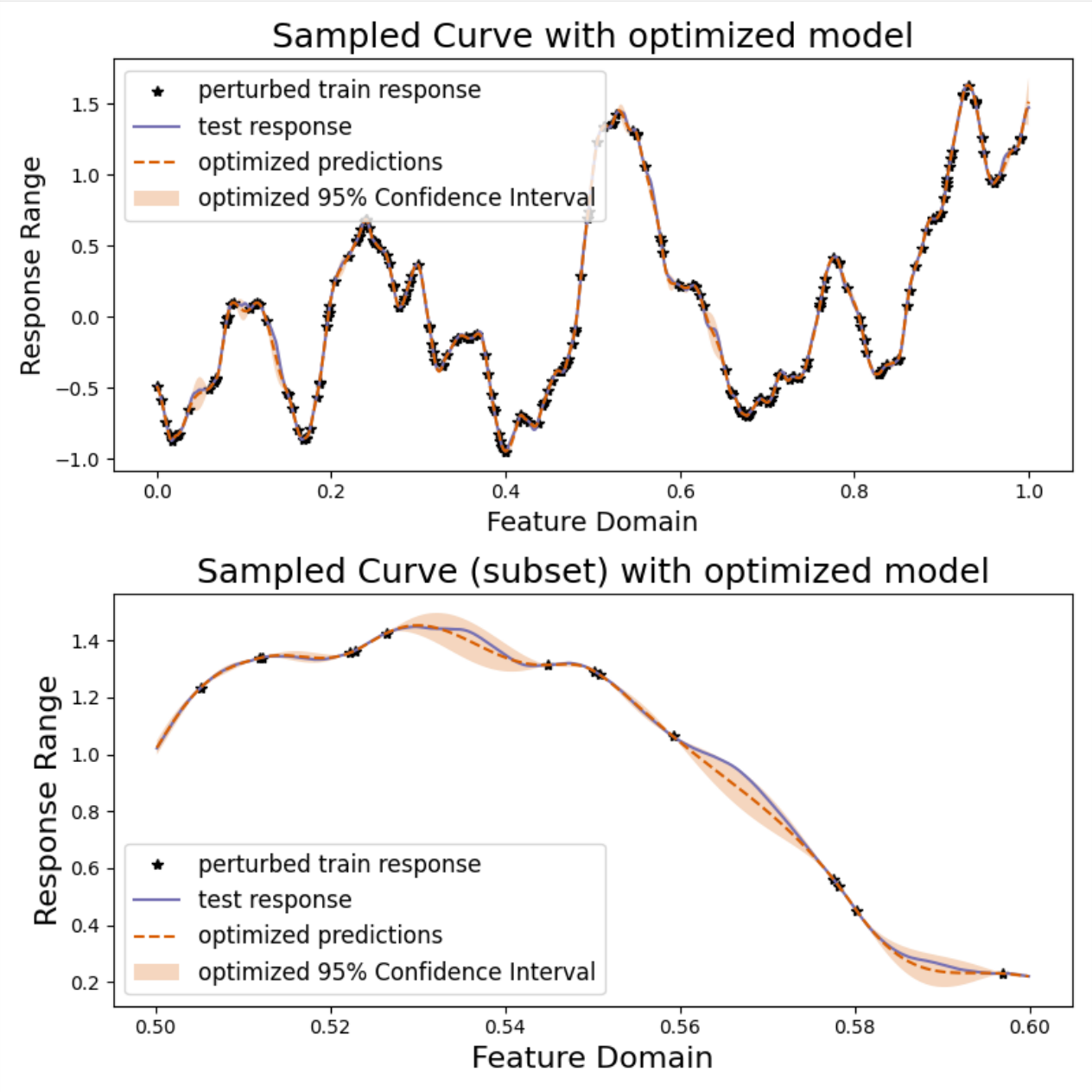
[27]:
```
print_results(
    test_responses, ("optimized", muygps_optimized, predictions, variances, confidence_
↪intervals, coverage)
)
```

[27]:
```
<pandas.io.formats.style.Styler at 0x7fa9dd933490>
```

Note here that the returned value for `smoothness` might be different from the `smoothness` used by the conventional GP. Also, the value of σ^2 is a little different from the "true" value of 1.0. However, our mean predictions have low RMSE and our confidence intervals are low on average while our 95% confidence intervals succeed in covering ~95% of the true responses.

We can also plot our responses and evaluate their performance. We plot below the predicted and true curves, as well as the 95% confidence interval. We plot a smaller subset of the data in the lower curve in order to better scrutinize the 95% confidence interval.

[28]:
```
sampler.plot_results(("optimized", predictions, confidence_intervals))
```



1.7 Illustrating MuyGPs Sparsification, Prediction, and Uncertainty Quantification

This notebook illustrates how MuyGPs conditions predictions on nearest neighbors and visualizes the posterior distributions.

[2]:
```
import matplotlib.pyplot as plt
import numpy as np

from MuyGPyS._test.sampler import UnivariateSampler2D, print_results
from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.deformation import Isotropy, l2, F2
from MuyGPyS.gp.hyperparameter import AnalyticScale, Parameter
from MuyGPyS.gp.kernels import Matern, RBF
from MuyGPyS.gp.noise import HomoscedasticNoise
from MuyGPyS.neighbors import NN_Wrapper
from MuyGPyS.optimize.batch import sample_batch
```

We will set a random seed here for consistency when building docs. In practice we would not fix a seed.

[3]:
```
np.random.seed(0)
```

1.7.1 Sampling a 2D Surface from a Conventional GP

This notebook will use a simple two-dimensional curve sampled from a conventional Gaussian process. We will specify the domain as a simple grid on a one-dimensional surface and divide the observations näively into train and test data.

Feel free to download the source notebook and experiment with different parameters.

First we specify the data size and the proportion of the train/test split.

[4]:
```
points_per_dim = 60
train_ratio = 0.2
```

We use all of these parameters to define a Matérn kernel GP and a sampler for convenience. The `UnivariateSampler2D` class is a convenience class for this tutorial, and is not a part of the library. We will use an anisotropic deformation to ensure that we sample data from the appropriate distribution.

[5]:
```
kernel = Matern(
    smoothness=Parameter(1.5),
    deformation=Isotropy(
        l2,
        length_scale=Parameter(0.2),
    ),
)
```

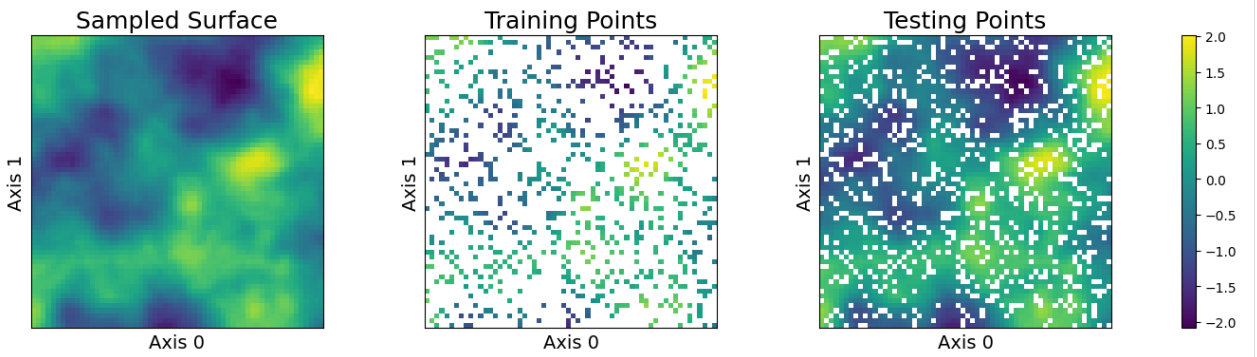
[6]:
```
sampler = UnivariateSampler2D(
    points_per_dim=points_per_dim,
    train_ratio=train_ratio,
    kernel=kernel,
    noise=HomoscedasticNoise(1e-7),
    measurement_noise=HomoscedasticNoise(1e-14),
)
```

Finally, we will sample a curve from this GP prior and visualize it. Note that we perturb the train responses (the values that our model will actual receive) with Gaussian measurement noise. Further note that this is not especially fast, as sampling from a conventional Gaussian process requires computing the Cholesky decomposition of a `(data_count, data_count)` matrix.

[7]:
```
train_features, test_features = sampler.features()
train_count, _ = train_features.shape
test_count, _ = test_features.shape
```

[8]:
```
train_responses, test_responses = sampler.sample()
```

[9]:
```
sampler.plot_sample()
```



1.7.2 Nearest Neighbors Sparsification

MuyGPyS achieves fast posterior inference by restricting the conditioning of predictions on only the most relevant points in the training data. Currently, the library does this by utilizing the k nearest neighbors (KNN), relying upon the intution that nearby points in the input space are more highly correlated than distant points, and that nearby points contribute the overwhelming majority of the weight in the posterior mean. While methods other than nearest neighbors are also worth considering, the library presently only supports KNN.

We will illustrate the intuition behind using KNN. First, we will form a KNN index of the training data for querying. We will use the library's built-in `NN_Wrapper` class, which wraps scikit-learn's exact KNN implementation (used here) and hnswlib's approximate but much faster and more scalable implementation.

[10]:
```
nn_count = 50
nbrs_lookup = NN_Wrapper(train_features, nn_count, nn_method="exact", algorithm="ball_
↪tree")
```

We will use the same Matérn kernel used to simulate this data.

[11]:
```
muygps = MuyGPS(
    kernel=kernel,
    noise=HomoscedasticNoise(1e-7),
)
```

For a given prediction location $\mathbf{z} \in \mathbb{R}^d$, and training set $X \in \mathbb{R}^{n \times d}$ with measured univariate responses $\mathbf{y} \in \mathbb{R}^n$, a conventional zero-mean GP $f \sim \mathcal{GP}(\mathbf{0}, K(\cdot,\cdot))$ predicts the following posterior mean:

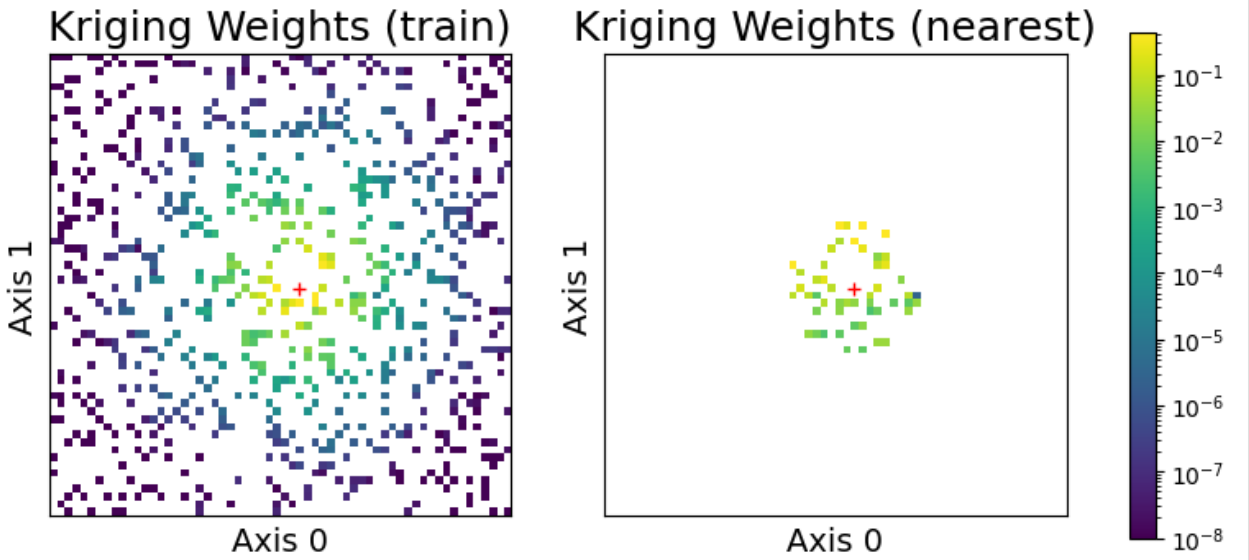
$$E\left[f(\mathbf{z}) \mid X, \mathbf{y}\right] = K(\mathbf{z}, X)K(X, X)^{-1}\mathbf{y}. \tag{1.1}$$

Here $K(\mathbf{z}, X) \in \mathbb{R}^n$ is the cross-covariance between $\mathbf{z}$ and every element of the training data X, and $K(X, X) \in \mathbb{R}^{n \times n}$ is the covariance matrix of X with itself, whose inverse is sometimes called the precision matrix. The product of the cross-covariance with the precision matrix $K(\mathbf{z}, X)K(X, X)^{-1} \in \mathbb{R}^n$ are sometimes called the *kriging weights*. These kriging weights effectively induce a weighted average of the observed responses $\mathbf{y}$. Ergo, if the kriging weights are sparse (and for many practical problems they are), we need only compute the sparse elements of the kriging weights to approximate the posterior mean!

Here we will illustrate our claim by observing the kriging weights for all of the training data for a particular prediction point. We choose a test point, represented by the red plus, and plot the kriging weights of - (left) a version of the problem including all of the data (for illustration purposes) - (center) the posterior mean conditioned on the training data - (right) the posterior mean conditioned only on the nearest neighbors

[12]:
```
test_index = int(test_count / 2) + 20
```

[13]:
```
sampler.plot_kriging_weights(test_index, nbrs_lookup)
```



As we can see, the kriging weights of the GP problem (center plot) isolate most of the weight near the query point (red plus) in space. We can sparsify the kriging weights by only considering the nearest neighbors, represented in the right plot, while maintaining most of the covariance information to predict the point.

1.7.3 Comparing MuyGPs to Conventional GP Posteriors

Here we will compute posterior mean and variances for the data using both a conventional GP approach and MuyGPs.

First, we compute a conventional GP.

[14]:
```
crosswise_dists_full = kernel.deformation.crosswise_tensor(
    test_features,
    train_features,
    np.arange(test_count),
    [np.arange(train_count) for _ in range(test_count)],
)
pairwise_dists_full = kernel.deformation.pairwise_tensor(
    train_features,
```

(continues on next page)

(continued from previous page)

```
    np.arange(train_count),
)
```

[15]:
```
Kcross_full = kernel(crosswise_dists_full)
Kin_full = kernel(pairwise_dists_full)
```

Here we'll stop to note that we have three matrices: the cross-covariance (`Kcross_full`), the covariance (`Kin_full`), and the response vector (`train_responses`). The mean and covariance are computed in terms of dense solves involving these matrices, whose dimensions increase linearly in the data size (resulting in a quadratic increase in storage and a cubic increase in runtime).

[16]:
```
print(f"Kcross_full shape: {Kcross_full.shape}")
print(f"Kin_full shape: {Kin_full.shape}")
print(f"train_responses shape: {train_responses.shape}")
```

```
Kcross_full shape: (2880, 720)
Kin_full shape: (720, 720)
train_responses shape: (720,)
```

We use these matrices to compute the posterior mean and variance, and construct univariate 95% confidence intervals for each individual prediction.

[17]:
```
mean_full = Kcross_full @ np.linalg.solve(Kin_full, train_responses)
covariance_full = 1 - Kcross_full @ np.linalg.solve(Kin_full, Kcross_full.T)
covariance_diag = np.diag(covariance_full)
confidence_interval_full = np.sqrt(covariance_diag) * 1.96
coverage_full = (
    np.count_nonzero(
        np.abs(test_responses - mean_full) < confidence_interval_full
    ) / test_count
)
```

Now we repeat a similar workflow for MuyGPs. This time, we sample nearest neighbors from the previously-constructed index and create distance tensors using `MuyGPyS` convenience functions.

[18]:
```
nn_indices, _ = nbrs_lookup.get_nns(test_features)
(
    crosswise_dists,
    pairwise_dists,
    nn_responses,
) = muygps.make_predict_tensors(
    np.arange(test_count),
    nn_indices,
    test_features,
    train_features,
    train_responses,
)
```

[19]:
```
Kcross = muygps.kernel(crosswise_dists)
Kin = muygps.kernel(pairwise_dists)
```

We now have three tensors, similar to the conventional workflow: `Kcross`, `Kin`, and `nn_responses`. These tensors have the following shapes, which only increase linearly as the data size increases, which drastically improves scalability compared to the conventional GP.

[20]:
```
print(f"Kcross shape: {Kcross.shape}")
print(f"Kin shape: {Kin.shape}")
print(f"nn_responses shape: {nn_responses.shape}")
```

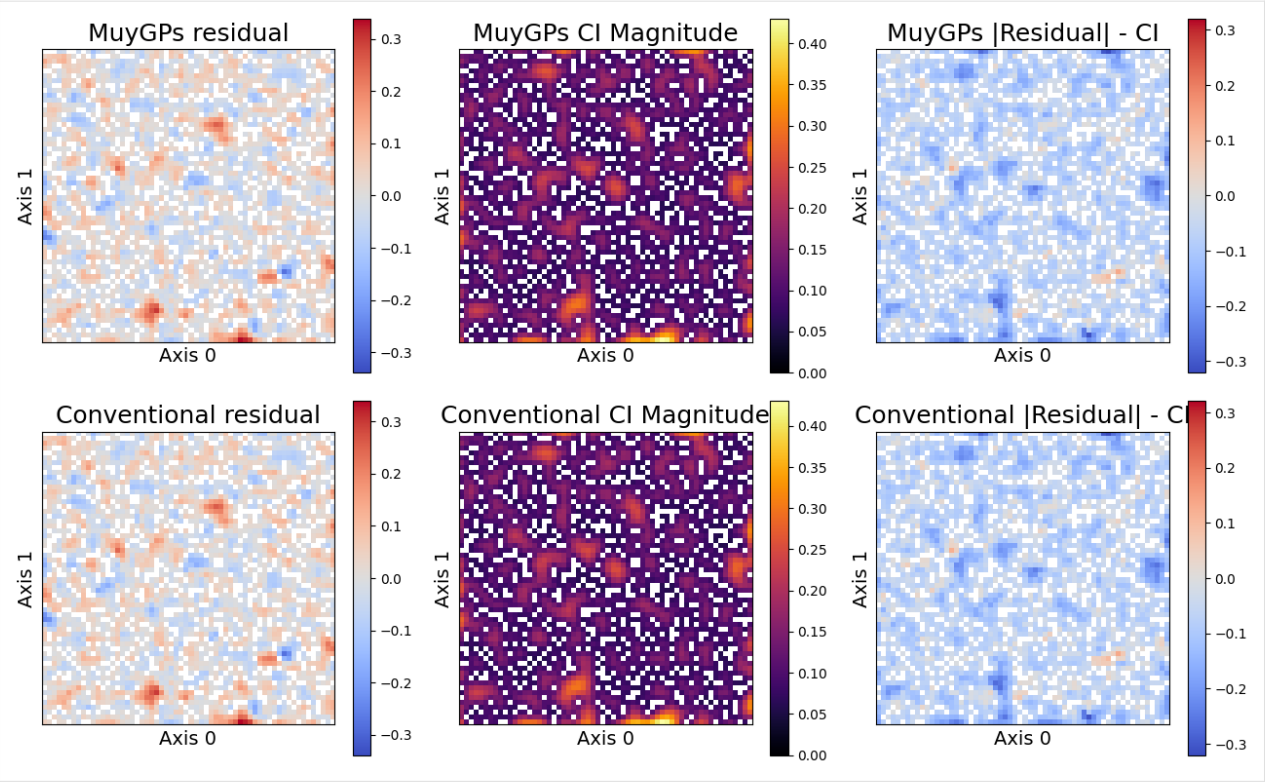
```
Kcross shape: (2880, 50)
Kin shape: (2880, 50, 50)
nn_responses shape: (2880, 50)
```

Here we use MuyGPyS to compute the posterior distribution, similar in form to the conventional GP.

[21]:
```
mean_muygps = muygps.posterior_mean(
    Kin, Kcross, nn_responses
)
variance_muygps = muygps.posterior_variance(
    Kin, Kcross
)
confidence_interval_muygps = np.sqrt(variance_muygps) * 1.96
coverage_muygps = (
    np.count_nonzero(
        np.abs(test_responses - mean_muygps) < confidence_interval_muygps
    ) / test_count
)
```

Finally, we compare our performance. The left column plots the absolute residual of each posterior mean implementation with the true response for the whole test dataset. The center column plots the size of the 95% confidence intervals across the whole dataset. Finally, the right column plots where the residual exceeds the confidence interval. Red points in the right column exceed the confidence interval, which should comprise 5% of the data if the uncertainties are calibrated.

[22]:
```
sampler.plot_errors(
    ("MuyGPs", mean_muygps, confidence_interval_muygps),
    ("Conventional", mean_full, confidence_interval_full),
)
```



We can see that the MuyGPyS posteriors closely matches the conventional GP, while remaining much more scalable. Note especially that the same points exceed the confidence interval for each model. Hopefully, this demonstration has helped to motivate the MuyGPs sparsification approach. For more validation, we directly compare some summary statistics of the two approaches.

[23]:
```
print_results(
    test_responses,
    ("MuyGPyS", muygps, mean_muygps, variance_muygps, confidence_interval_muygps,␣
↪coverage_muygps),
    ("Conventional", muygps, mean_full, covariance_diag, confidence_interval_full,␣
↪coverage_full),
)
```

[23]:
```
<pandas.io.formats.style.Styler at 0x7f3e517d4d30>
```

1.8 Deep Kernels with MuyGPs in PyTorch Tutorial

In this tutorial, we outline how to construct a simple deep kernel model using the PyTorch implementation of MuyGPs.

We use the MNIST classification problem as a benchmark. We will use the deep kernel MuyGPs model to classify images of handwritten digits between 0 and 9. In order to reduce the runtime of the training loop, we will use a fully-connected architecture, meaning we will have to vectorize each image prior to training. We download the training and testing data using the torchvision.datasets API.

First, we will import necessary dependencies. We also force MuyGPyS to use the `"torch"` backend. This can also be done by setting the `MUYGPYS_BACKEND` environment variable to `"torch"`.

[2]:
```
%env MUYGPYS_BACKEND=torch
%env MUYGPYS_FTYPE=32
```

```
env: MUYGPYS_BACKEND=torch
env: MUYGPYS_FTYPE=32
```

[3]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
import os
import torch
import torchvision
from MuyGPyS.examples.muygps_torch import predict_model
from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.deformation import l2, Isotropy
from MuyGPyS.gp.hyperparameter import Parameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.noise import HomoscedasticNoise
from MuyGPyS.neighbors import NN_Wrapper
from MuyGPyS.optimize.batch import sample_batch
from MuyGPyS.torch import MuyGPs_layer
from torch import nn
from torch.nn.functional import one_hot
from torch.optim.lr_scheduler import ExponentialLR
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1707872829.179408       1 tfrt_cpu_pjrt_client.cc:349] TfrtCpuClient created.
```

We set the target directory for torch to download MNIST.

[4]:
```
root = './data'
if not os.path.exists(root):
    os.mkdir(root)
```

We use torch's utilities to download MNIST and transform it into an appropriately normalized tensor.

[5]:
```
trans = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5,),(1.0,)),
    ]
)
train_set = torchvision.datasets.MNIST(
```

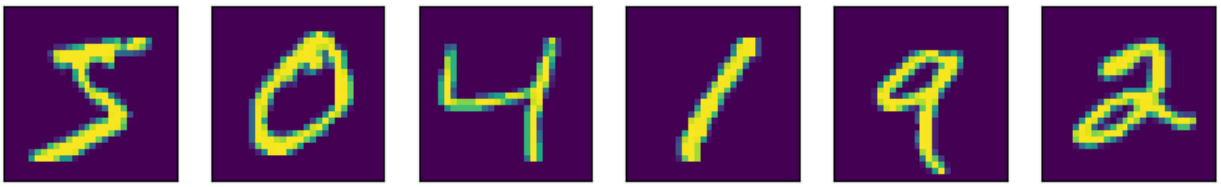
(continues on next page)

(continued from previous page)

```
    root=root, train=True, transform=trans, download=True
)
test_set = torchvision.datasets.MNIST(
    root=root, train=False, transform=trans, download=True
)
```

MNIST is a popular benchmark dataset of hand-written digits, 0-9. Each digit is a 28x28 pixel image, with 784 total pixel features.

[6]:
```
fig, axes = plt.subplots(1, 6, figsize=(10, 3))
for i, ax in enumerate(axes):
    ax.imshow(train_set.data[i, :, :])
    ax.set_xticks([])
    ax.set_yticks([])
plt.show()
```



In the interest of reducing the runtime of this example, we will use vectorized images as our features in this dataset. We will collect 60,000 training samples and 10,000 test samples.

[7]:
```
class_count = len(train_set.classes)
train_count, x_pixel_count, y_pixel_count = train_set.data.shape
test_count, _, _ = test_set.data.shape
feature_count = x_pixel_count * y_pixel_count
```

We vectorize the images and one-hot encode the class labels.

[8]:
```
train_features = torch.zeros((train_count, feature_count))
train_responses = torch.zeros((train_count, class_count))

for i in range(train_count):
    train_features[i,:] = train_set[i][0].flatten()
    train_responses[i,:] = one_hot(
        torch.tensor(train_set[i][1]).to(torch.int64),
        num_classes=class_count,
    )

test_features = torch.zeros((test_count, feature_count))
test_responses = torch.zeros((test_count, class_count))

for i in range(test_count):
    test_features[i,:] = test_set[i][0].flatten()
    test_responses[i,:] = one_hot(
        torch.tensor(test_set[i][1]).to(torch.int64),
        num_classes=class_count,
    )
```

We set up our nearest neighbor lookup structure using the NN_Wrapper data structure in MuyGPs. We then define our batch and construct tensor containing the features and targets of the batched elements and their 30 nearest neighbors. We choose an algorithm that will return the exact nearest neighbors. We set a random seed for reproducability.

[9]:
```
torch.autograd.set_detect_anomaly(True)
np.random.seed(0)
test_count, _ = test_features.shape
train_count, _ = train_features.shape

nn_count = 30
nbrs_lookup = NN_Wrapper(train_features, nn_count, nn_method="exact")
```

We sample a training batch of 500 elements and record their indices and those of their nearest neighbors.

[10]:
```
batch_count = 500
batch_indices, batch_nn_indices = sample_batch(
    nbrs_lookup, batch_count, train_count
)

batch_features = train_features[batch_indices,:]
batch_targets = train_responses[batch_indices, :]
batch_nn_targets = train_responses[batch_nn_indices, :]

if torch.cuda.is_available():
    train_features = train_features.cuda()
    train_responses = train_responses.cuda()
    test_features = test_features.cuda()
    test_responses = test_responses.cuda()
```

We now define a stochastic variational deep kernel MuyGPs class. This class composes a dense neural network embedding with a `MuyGPyS.torch.muygps_layer` Gaussian process layer. Presently, this layer only supports the Matérn kernel with special values of the smoothness parameter set to 0.5, 1.5, 2.5, or ∞. The smoothness values are limited because `torch` does not implement modified bessel functions of the second kind. Future versions of the library will also support other kernel types.

[11]:
```
class SVDKMuyGPs(nn.Module):
    def __init__(
        self,
        muygps_model,
        batch_indices,
        batch_nn_indices,
        batch_targets,
        batch_nn_targets,
    ):
        super().__init__()
        self.embedding = nn.Sequential(
            nn.Linear(784,400),
            nn.ReLU(),
            nn.Linear(400,200),
            nn.ReLU(),
            nn.Linear(200,100),
        )
        self.batch_indices = batch_indices
        self.batch_nn_indices = batch_nn_indices
```

(continues on next page)

(continued from previous page)

```
        self.batch_targets = batch_targets
        self.batch_nn_targets = batch_nn_targets
        self.GP_layer = MuyGPs_layer(
            muygps_model,
            batch_indices,
            batch_nn_indices,
            batch_targets,
            batch_nn_targets,
        )
        self.deformation = self.GP_layer.deformation

    def forward(self,x):
        predictions = self.embedding(x)
        predictions, variances = self.GP_layer(predictions)
        return predictions, variances
```

1.8.1 Training a Deep Kernel MuyGPs Model

We will use a Matérn kernel with a smoothness parameter of 0.5 and a Guassian homoscedastic noise prior variance of `1e-6`.

Presently the torch backend only supports fixed special case Matérn smoothness parameters with values 0.5, 1.5, or 2.5.

An isotropic length scale is the only torch-optimizable parameter.

[12]:
```
muygps_model = MuyGPS(
    kernel=Matern(
        smoothness=Parameter(0.5),
        deformation=Isotropy(
            l2,
            length_scale=Parameter(1.0, (0.1, 2))
        ),
    ),
    noise=HomoscedasticNoise(1e-6),
)
```

We instantiate a `SVDKMuyGPs` model using this MuyGPS model.

[13]:
```
model = SVDKMuyGPs(
    muygps_model = muygps_model,
    batch_indices=batch_indices,
    batch_nn_indices=batch_nn_indices,
    batch_targets=batch_targets,
    batch_nn_targets=batch_nn_targets,
)
if torch.cuda.is_available():
    model = model.cuda()
```

We use the Adam optimizer over 10 training iterations, with an initial learning rate of `1e-2` and decay of `0.97`.

[14]:
```
training_iterations = 10
optimizer = torch.optim.Adam(
    [{'params': model.parameters()}], lr=1e-2
)
scheduler = ExponentialLR(optimizer, gamma=0.97)
```

We will use cross-entropy loss, as it is commonly performant for classification problems. Other losses are available.

[15]:
```
ce_loss = nn.CrossEntropyLoss()
```

We construct a standard PyTorch training loop function.

[16]:
```
def train(nbrs_lookup):
    for i in range(training_iterations):
        model.train()
        optimizer.zero_grad()
        predictions,variances = model(train_features)
        loss = ce_loss(predictions,batch_targets)
        loss.backward()
        optimizer.step()
        scheduler.step()
        if np.mod(i,1) == 0:
            print(f"Iter {i + 1}/{training_iterations} - Loss: {loss.item()}")
            model.eval()
            nbrs_lookup = NN_Wrapper(
                model.embedding(train_features).detach().numpy(),
                nn_count, nn_method="exact"
            )
            batch_nn_indices,_ = nbrs_lookup._get_nns(
                model.embedding(batch_features).detach().numpy(),
                nn_count=nn_count,
            )
            batch_nn_targets = train_responses[batch_nn_indices, :]
            model.batch_nn_indices = batch_nn_indices
            model.batch_nn_targets = batch_nn_targets
        torch.cuda.empty_cache()
    nbrs_lookup = NN_Wrapper(
        model.embedding(train_features).detach().numpy(),
        nn_count,
        nn_method="exact",
    )
    batch_nn_indices,_ = nbrs_lookup._get_nns(
        model.embedding(batch_features).detach().numpy(),
        nn_count=nn_count,
    )
    batch_nn_targets = train_responses[batch_nn_indices, :]
    model.batch_nn_indices = batch_nn_indices
    model.batch_nn_targets = batch_nn_targets
    return nbrs_lookup, model
```

Finally, we execute the training function and evaluate the trained model

[17]:
```
nbrs_lookup, model_trained = train(nbrs_lookup)
```

```
Iter 1/10 - Loss: 1.514917016029358
Iter 2/10 - Loss: 1.4779890775680542
Iter 3/10 - Loss: 1.4398393630981445
Iter 4/10 - Loss: 1.423111081123352
Iter 5/10 - Loss: 1.4219379425048828
Iter 6/10 - Loss: 1.4020675420761108
Iter 7/10 - Loss: 1.3868385553359985
Iter 8/10 - Loss: 1.3743661642074585
Iter 9/10 - Loss: 1.3675148487091064
Iter 10/10 - Loss: 1.3568177223205566
```

Our final model parameters look like the following:

[18]:
```
for n, p in model_trained.named_parameters():
    print(f"{n}, {p.shape if p.shape != torch.Size([]) else p.item()}")
```

```
embedding.0.weight, torch.Size([400, 784])
embedding.0.bias, torch.Size([400])
embedding.2.weight, torch.Size([200, 400])
embedding.2.bias, torch.Size([200])
embedding.4.weight, torch.Size([100, 200])
embedding.4.bias, torch.Size([100])
GP_layer.length_scale, 1.0085885524749756
```

We then compute and report the performance of the predicted test responses using this trained model.

[19]:
```
predictions, variances = predict_model(
    model=model_trained,
    test_features=test_features,
    train_features=train_features,
    train_responses=train_responses,
    nbrs_lookup=nbrs_lookup,
    nn_count=nn_count,
)
print("MNIST Prediction Accuracy Using Hybrid Torch Model:")
print(
    (
        torch.sum(
            torch.argmax(predictions,dim=1) == torch.argmax(test_responses,dim=1)
        ) / 10000
    ).numpy()
)
```

```
MNIST Prediction Accuracy Using Hybrid Torch Model:
0.9361
```

We note that this is quite mediocre performance on MNIST. In the interest of reducing notebook runtime we have used a simple fully-connected neural network model to construct the Gaussian process kernel. To achieve results closer to the state-of-the-art (near 100% accuracy), we recommend using more complex architectures which integrate convolutional kernels into the model.

1.9 Fast Posterior Mean Tutorial

This notebook walks through the fast posterior mean workflow presented in Fast Gaussian Process Posterior Mean Prediction via Local Cross Validation and Precomputation (Dunton et. al 2022) and explains the relevant components of MuyGPyS.

The cell below uses the same code as that found in univariate_regression_tutorial.ipynb. This includes generating the synthetic data from a GP and training two MuyGPs models to fit the data using Bayesian optimization.

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import timeit

from MuyGPyS._test.gp import benchmark_sample, BenchmarkGP
from MuyGPyS._test.sampler import UnivariateSampler, print_fast_results
from MuyGPyS.neighbors import NN_Wrapper
from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.deformation import Isotropy, l2
from MuyGPyS.gp.hyperparameter import AnalyticScale, Parameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.noise import HomoscedasticNoise
from MuyGPyS.gp.tensors import fast_nn_update, make_fast_predict_tensors
```

We will assume that we have already optimized the a MuyGPs model following the *Univariate Regression Tutorial*.

[3]:
```
kernel = Matern(
    smoothness=Parameter(2.0),
    deformation=Isotropy(
        l2,
        length_scale=Parameter(0.05),
    ),
)
```

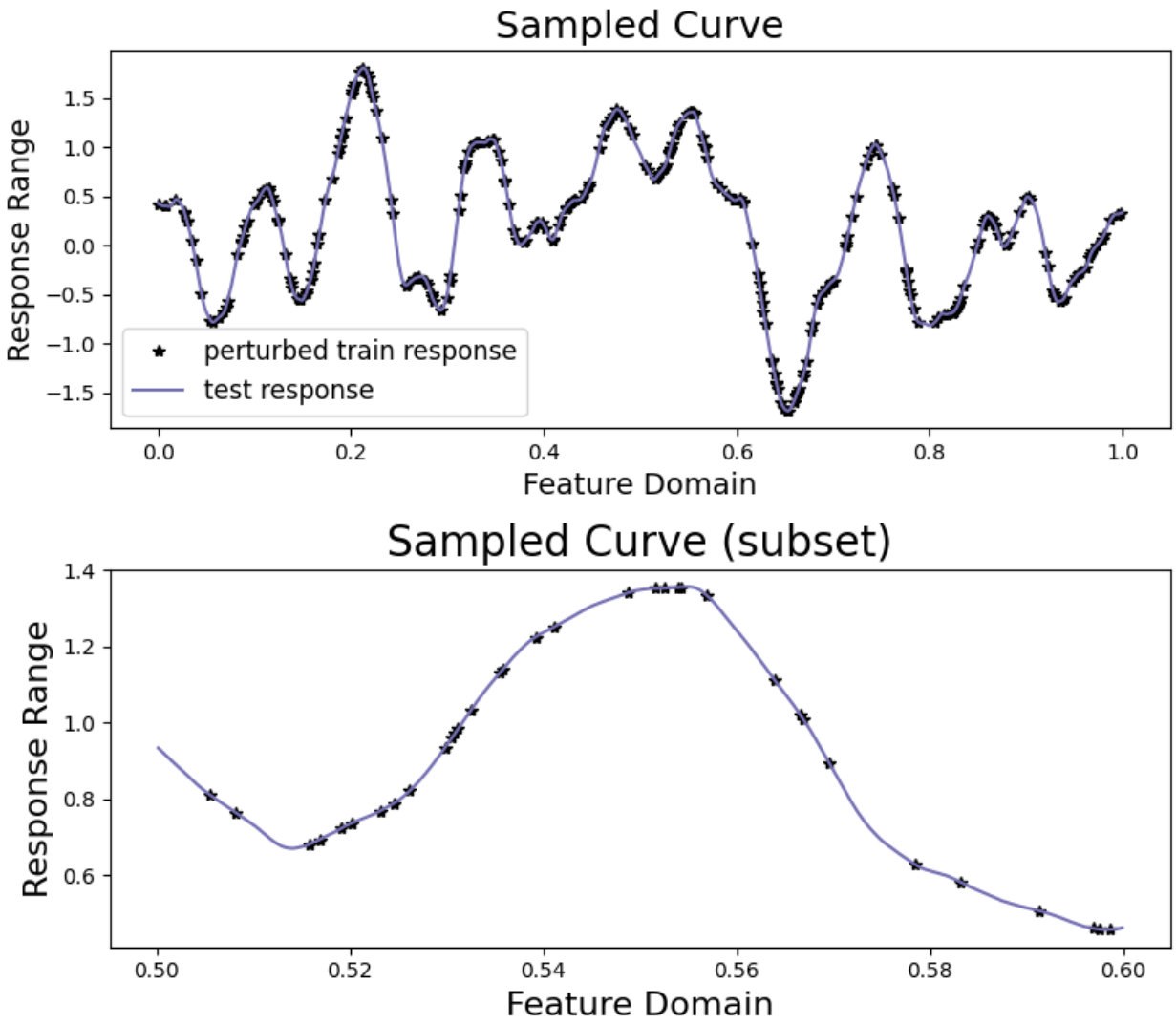
We will use that kernel to simulate a curve and then compare the prediction times for both the conventional regression and the fast kernel regression method.

[4]:
```
np.random.seed(0)
measurement_noise = 1e-4
sampler = UnivariateSampler(
    data_count=3000,
    train_ratio=0.1,
    kernel=kernel,
    noise=HomoscedasticNoise(1e-14),
    measurement_noise=HomoscedasticNoise(measurement_noise),
)
train_features, test_features = sampler.features()
test_count = test_features.shape[0]
train_count = train_features.shape[0]
```

[5]:
```
train_responses, test_responses = sampler.sample()
```

We'll visualize the results.

[6]:
```
sampler.plot_sample()
```



We then prepare a MuyGPS object and a nearest neighbors index. We could use a single MuyGPS object, but in this case we create a second one for the fast regression because a larger noise prior helps to stabilize the computations.

[7]:
```
nbrs_lookup = NN_Wrapper(train_features, nn_count=10, nn_method="exact",algorithm="ball_
↪tree")
muygps = MuyGPS(
    kernel=kernel,
    noise=HomoscedasticNoise(1e-4),
    scale=AnalyticScale(),
)
muygps_fast = MuyGPS(
    kernel=kernel,
    noise=HomoscedasticNoise(1e-1),
    scale=AnalyticScale(),
)
```

1.9.1 Benchmarking Fast Prediction

With set (or learned) hyperparameters, we are able to use the `muygps` object for fast prediction capability.

See below a fast posterior mean workflow, using the data structures built up in this example. This workflow uses the compact tensor-making function *make_fast_predict_tensors()* to succinctly create tensors defining the `pairwise_dists` among each nearest neighbor and the `train_nn_targets_fast` or responses of the nearest neighbors in each set. We then create the `Kin` covariance tensor and form the precomputed coefficients matrix. We then pass the precomputed coefficients matrix, the `nn_indices` matrix of neighborhood indices, and the closest neighbor of each test point to *MuyGPS.fast_posterior_mean()* in order to obtain our predictions.

First we obtain the indices of the nearest neighbors of all of the training datapoints.

[8]:
```
train_nn_indices, _ = nbrs_lookup.get_nns(train_features)
```

We then update these neighborhoods with the index of the corresponding training point so that each neighborhood contains the query point.

[9]:
```
train_nn_indices_fast = fast_nn_update(train_nn_indices)
```

We then compute the pairwise distance tensor and target matrix and use them to construct the corresponding kernel tensor and the precomputed target matrix to be used in the fast kernel regression.

[10]:
```
pairwise_dists_fast, nn_targets_fast = make_fast_predict_tensors(
    train_nn_indices_fast,
    train_features,
    train_responses,
)
Kin_fast = muygps_fast.kernel(
    muygps_fast.kernel.deformation.metric(pairwise_dists_fast)
)
precomputed_coefficients_matrix = muygps_fast.fast_coefficients(Kin_fast, nn_targets_
↪fast)
```

The steps so far have involved only the training data, and can be precomputed before encountering the test data. We now find the closest training point to each test point and return the corresponding enriched training points.

[11]:
```
test_indices = np.arange(test_count)
test_nn_indices, _ = nbrs_lookup.get_nns(test_features)
closest_neighbor = test_nn_indices[:, 0]
closest_set = train_nn_indices_fast[closest_neighbor, :]
```

We use these indices to make the crosswise distance tensor, similar to usual prediction.

[12]:
```
crosswise_dists_fast = muygps.kernel.deformation.crosswise_tensor(
    test_features, train_features, test_indices, closest_set
)
```

Finally, we compute the crosscovariance and perform fast prediction.

[13]:
```
Kcross_fast = muygps_fast.kernel(crosswise_dists_fast)
predictions_fast = muygps_fast.fast_posterior_mean(
    Kcross_fast,
    precomputed_coefficients_matrix[closest_neighbor],
)
```

1.9.2 Comparison with Conventional Prediction

With set (or learned) hyperparameters, we are able to use the `muygps` object to predict the response of test data. Several workflows are supported.

See below a simple posterior mean workflow, using the data structures built up in this example. This is very similar to the prediction workflow found in the *univariate regression tutorial*.

[14]:
```
(
    crosswise_dists,
    pairwise_dists,
    nn_targets,
) = muygps.make_predict_tensors(
    np.arange(test_count),
    test_nn_indices,
    test_features,
    train_features,
    train_responses,
)
Kcross = muygps.kernel(crosswise_dists)
Kin = muygps.kernel(pairwise_dists)
predictions = muygps.posterior_mean(
    Kin, Kcross, nn_targets
)
variances = muygps.posterior_variance(Kin, Kcross)
confidence_intervals = np.sqrt(variances) * 1.96
coverage = np.count_nonzero(np.abs(test_responses - predictions) < confidence_intervals␣
↪/ test_count
```

We compare our two methods in terms of time-to-solution and RMSE. In the conventional workflow we compute the sum of the time it takes to: - identify the nearest neighbors of the test features, - form the relevant kernel tensors, and - solve the posterior means.

In the fast posterior mean case, we compute the sum of the time it takes to: - identify the nearest neighbor of each test point, - lookup coefficients in the precomputed coefficient matrix, and - perform the dot product to form posterior means.

Note that the fast kernel regression method does not compute a variance, and so its posterior variance, confidence intervals, and coverage are nil.

[15]:
```
def timing_posterior_mean():
    test_nn_indices, _ = nbrs_lookup.get_nns(test_features)
    (
        crosswise_dists,
        pairwise_dists,
        nn_targets,
    ) = muygps.make_predict_tensors(
        test_indices,
        test_nn_indices,
        test_features,
        train_features,
        train_responses,
    )
    Kcross = muygps.kernel(crosswise_dists)
    Kin = muygps.kernel(pairwise_dists)
```

(continues on next page)

(continued from previous page)

```
    predictions = muygps.posterior_mean(
        Kin, Kcross, nn_targets
    )

def timing_fast_posterior_mean():
    test_nn_indices_fast, _ = nbrs_lookup.get_nns(test_features)
    closest_neighbor = test_nn_indices_fast[:, 0]
    closest_set = train_nn_indices_fast[closest_neighbor, :].astype(int)
    crosswise_dists = muygps.kernel.deformation.crosswise_tensor(
        test_features,
        train_features,
        test_indices,
        closest_set,
    )
    Kcross = muygps_fast.kernel(crosswise_dists)
    predictsion_fast = muygps_fast.fast_posterior_mean(
        Kcross,
        precomputed_coefficients_matrix[closest_neighbor],
    )
```

[16]:
```
time_conv = %timeit -o timing_posterior_mean()
time_fast = %timeit -o timing_fast_posterior_mean()
```

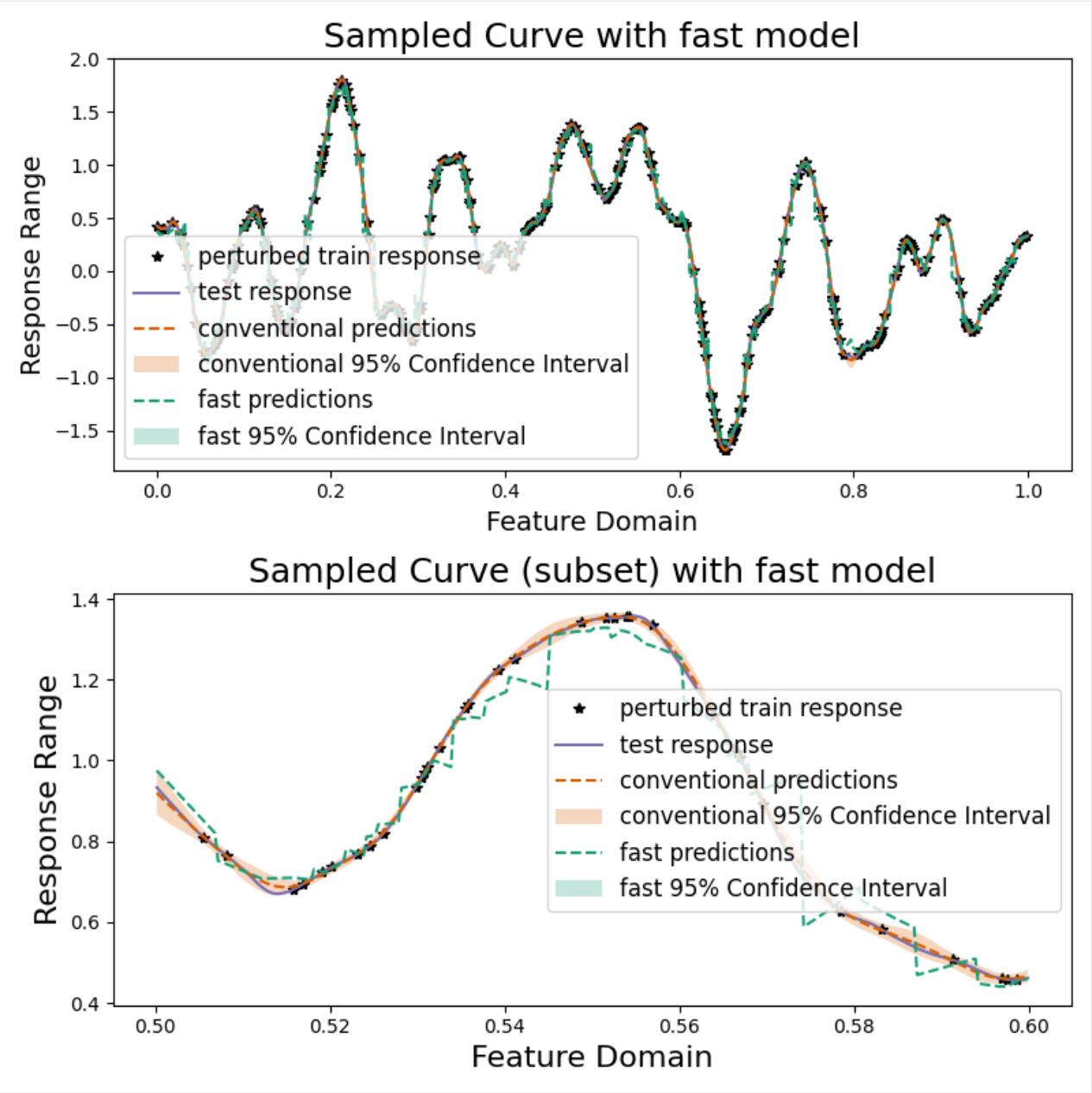
```
169 ms ± 1.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
19.1 ms ± 164 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[17]:
```
nil_vec = np.zeros(test_count)
print_fast_results(
    test_responses,
    ("conventional", time_conv, muygps, predictions),
    ("fast", time_fast, muygps_fast, predictions_fast),
)
```

[17]:
```
<pandas.io.formats.style.Styler at 0x7f1e1e129390>
```

As we can see, we can gain an order of magnitude speed improvement by sacrificing some precision and the posterior variance. We also plot our two methods and compare their results graphically.

[18]:
```
sampler.plot_results(
    ("conventional", predictions, confidence_intervals),
    ("fast", predictions_fast, np.zeros(test_count)),
)
```



1.10 Anisotropic Metric Tutorial

This notebook walks through a simple anisotropic regression workflow and illustrates anisotropic features of `MuyGPyS`.

[2]:
```
import numpy as np

from MuyGPyS._test.sampler import UnivariateSampler2D, print_results
from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.deformation import Anisotropy, Isotropy, l2
from MuyGPyS.gp.hyperparameter import AnalyticScale, Parameter, VectorParameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.noise import HomoscedasticNoise
from MuyGPyS.neighbors import NN_Wrapper
from MuyGPyS.optimize import Bayes_optimize
from MuyGPyS.optimize.batch import sample_batch
from MuyGPyS.optimize.loss import lool_fn
```

```
Matplotlib is building the font cache; this may take a moment.
```

We will set a random seed here for consistency when building docs. In practice we would not fix a seed.

[3]:
```
np.random.seed(0)
```

1.10.1 Sampling a 2D Surface from a Conventional GP

This notebook will use a simple two-dimensional curve sampled from a conventional Gaussian process. We will specify the domain as a simple grid on a one-dimensional surface and divide the observations näively into train and test data.

Feel free to download the source notebook and experiment with different parameters.

First we specify the data size and the proportion of the train/test split.

[4]:
```
points_per_dim = 60
train_ratio = 0.05
```

We will assume that the true data is produced with no noise, so we specify a very small noise prior for numerical stability. This is an idealized experiment with effectively no instrument error.

[5]:
```
nugget_noise = HomoscedasticNoise(1e-14)
```

We will perturb our simulated observations (the training data) with some i.i.d Gaussian measurement noise.

[6]:
```
measurement_noise = HomoscedasticNoise(1e-7)
```

Finally, we will specify a Matérn kernel with hyperparameters. `smoothness` determines how differentiable the GP prior is. The larger `smoothness` grows, the smoother sampled functions will become.

[7]:
```
sim_smoothness = Parameter(1.5)
```

We will use an anisotropic deformation, where displacement along the dimensions are weighted differently. Each dimension has a corresponding `length_scale` parameter.

[8]:
```
sim_length_scale0 = Parameter(0.1)
sim_length_scale1 = Parameter(0.5)
```

We use all of these parameters to define a Matérn kernel GP and a sampler for convenience. The `UnivariateSampler2D` class is a convenience class for this tutorial, and is not a part of the library. We will use an anisotropic deformation to ensure that we sample data from the appropriate distribution.

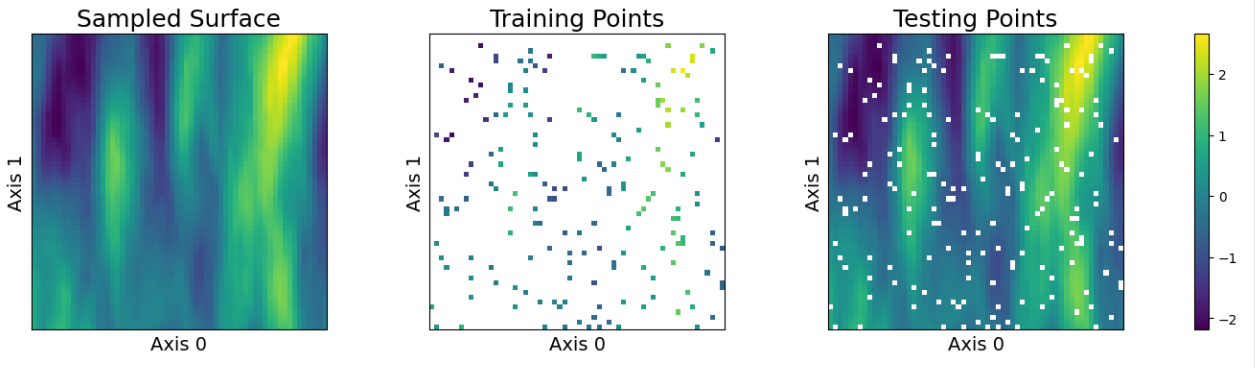
[9]:
```
sampler = UnivariateSampler2D(
    points_per_dim=points_per_dim,
    train_ratio=train_ratio,
    kernel=Matern(
        smoothness=sim_smoothness,
        deformation=Anisotropy(
            l2,
            length_scale=VectorParameter(
                sim_length_scale0,
                sim_length_scale1,
            ),
        ),
    ),
    noise=nugget_noise,
    measurement_noise=measurement_noise,
)
```

Finally, we will sample a curve from this GP prior and visualize it. Note that we perturb the train responses (the values that our model will actual receive) with Gaussian measurement noise. Further note that this is not especially fast, as sampling from a conventional Gaussian process requires computing the Cholesky decomposition of a `(data_count, data_count)` matrix.

[10]:
```
train_features, test_features = sampler.features()
```

[11]:
```
train_responses, test_responses = sampler.sample()
```

[12]:
```
sampler.plot_sample()
```



We can observe that our choice of anisotropy has caused the globular Gaussian features in the sampled surface to "smear" in the direction of the more heavily weighted axis.

1.10.2 Training an Anisotropic Model

We will not belabor the details covered in the *Univariate Regression Tutorial*. We must similarly construct a nearest neighbors index and sample a training batch in order to optimize a model.

For now, we use isotropic nearest neighbors as we do not have a guess as to the anisotropic scaling. Future versions of the library will use learned anisotropy to modify neighborhood structure during optimization.

[13]:
```
nn_count = 30
nbrs_lookup = NN_Wrapper(train_features, nn_count, nn_method="exact", algorithm="ball_
↪tree")
batch_count = sampler.train_count
batch_indices, batch_nn_indices = sample_batch(
    nbrs_lookup, batch_count, sampler.train_count
)
```

We construct a MuyGPs object with a Matérn kernel. For simplicity, we will fix `smoothness` and attempt to optimize the two `length_scale` parameters.

[14]:
```
muygps_anisotropic = MuyGPS(
    kernel=Matern(
        smoothness=sim_smoothness,
        deformation=Anisotropy(
            l2,
            length_scale=VectorParameter(
                Parameter("log_sample", (0.01, 1.0)),
                Parameter("log_sample", (0.01, 1.0)),
            ),
        ),
    ),
    noise=measurement_noise,
    scale=AnalyticScale(),
)
```

We will also create and optimze an isotropic model for comparison.

[15]:
```
muygps_isotropic = MuyGPS(
    kernel=Matern(
        smoothness=sim_smoothness,
        deformation=Isotropy(
            l2,
            length_scale=Parameter("log_sample", (0.01, 1.0)),
        ),
    ),
    noise=measurement_noise,
)
```

We build our difference tensors as usual and use Bayesian optimization. Note that there is a difference between the crosswise and pairwise tensors that we create here, versus those we create for an isotropic kernel. Anisotropic models create *difference* tensors rather than *distance* tensors, which have an extra dimension recording the feature dimension-wise comparisons (in this case, differences) between the items being compared. This is an important distinction, as anisotropic models need to record feature-dimension-wise comparisons to be scaled by trainable parameters, whereas isotropic models do not and collapse differences directory into distances.

[16]:
```
(
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    batch_targets,
    batch_nn_targets,
) = muygps_anisotropic.make_train_tensors(
    batch_indices,
    batch_nn_indices,
    train_features,
    train_responses,
)
```

Keyword arguments for the optimization:

[17]:
```
opt_kwargs = {
    "loss_fn": lool_fn,
    "verbose": True,
    "random_state": 1,
    "init_points": 5,
    "n_iter": 30,
    "allow_duplicate_points": True,
}
```

[18]:
```
muygps_anisotropic = Bayes_optimize(
    muygps_anisotropic,
    batch_targets,
    batch_nn_targets,
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    **opt_kwargs,
)
```

```
parameters to be optimized: ['length_scale0', 'length_scale1']
bounds: [[0.01 1.  ]
 [0.01 1.  ]]
initial x0: [0.07655293 0.30564372]
|   iter    |  target   | length... | length... |
-------------------------------------------------
| 1         | 574.8     | 0.07655   | 0.3056    |
| 2         | 440.2     | 0.4229    | 0.7231    |
| 3         | 301.6     | 0.01011   | 0.3093    |
| 4         | 251.9     | 0.1553    | 0.1014    |
| 5         | 458.9     | 0.1944    | 0.3521    |
| 6         | 389.5     | 0.4028    | 0.5434    |
| 7         | 562.3     | 0.1177    | 0.3752    |
| 8         | 273.3     | 0.8317    | 0.6635    |
| 9         | 580.4     | 0.1578    | 0.6325    |
| 10        | 559.3     | 0.09521   | 0.7611    |
| 11        | 125.1     | 0.4017    | 0.1698    |
| 12        | 585.8     | 0.1279    | 0.694     |
| 13        | 283.7     | 0.438     | 0.3585    |
| 14        | 499.2     | 0.04607   | 0.6306    |
| 15        | 373.6     | 0.5577    | 0.705     |
```

(continues on next page)

(continued from previous page)

```
| 16        | 532.7     | 0.157     | 0.4126    |
| 17        | 586.2     | 0.1351    | 0.6988    |
| 18        | 571.2     | 0.228     | 0.8591    |
| 19        | 540.1     | 0.1101    | 0.9515    |
| 20        | 566.1     | 0.2759    | 0.9977    |
| 21        | 499.1     | 0.4303    | 0.9782    |
| 22        | -555.4    | 1.0       | 0.01      |
| 23        | 321.5     | 1.0       | 1.0       |
| 24        | 401.6     | 0.6919    | 1.0       |
| 25        | 569.5     | 0.2273    | 0.8345    |
| 26        | 580.0     | 0.1583    | 0.6301    |
| 27        | 142.7     | 1.0       | 0.4731    |
| 28        | 247.1     | 0.01      | 1.0       |
| 29        | 218.4     | 0.6718    | 0.4214    |
| 30        | 337.0     | 0.4668    | 0.4931    |
| 31        | 269.9     | 0.9954    | 0.7852    |
| 32        | 420.9     | 0.5595    | 0.8803    |
| 33        | 513.7     | 0.3559    | 0.8713    |
| 34        | 575.9     | 0.1804    | 0.9957    |
| 35        | 516.2     | 0.2717    | 0.6678    |
| 36        | 493.2     | 0.1228    | 0.2604    |
=================================================
```

[19]:
```
print(f"BayesianOptimization finds an optimimal pair of length scales: {muygps_
↪anisotropic.kernel.deformation.length_scale()}")
```

```
BayesianOptimization finds an optimimal pair of length scales: [0.13513486 0.69879261]
```

Note here that these returned length scale values might be a little different than what we used to sample the surface. This can be due to a few factors: 1. optimizer might not have run enough iterations to converge, or 2. there is some mutual unidentifiability between the length scale parameters and the variance `scale` parameter.

However, `length_scale0 < length_scale1` as expected.

We also optimize the isotropic benchmark. Notice that we need to construct new *distance* tensors for the isotropic model.

[20]:
```
(
    batch_crosswise_dists,
    batch_pairwise_dists,
    _,
    _,
) = muygps_isotropic.make_train_tensors(
    batch_indices,
    batch_nn_indices,
    train_features,
    train_responses,
)
```

[21]:
```
muygps_isotropic = Bayes_optimize(
    muygps_isotropic,
    batch_targets,
    batch_nn_targets,
```

(continues on next page)

(continued from previous page)

```
    batch_crosswise_dists,
    batch_pairwise_dists,
    **opt_kwargs,
)
```

```
parameters to be optimized: ['length_scale']
bounds: [[0.01 1.  ]]
initial x0: [0.83917142]
|   iter    |  target   | length... |
-------------------------------------
| 1         | -3.343e+0 | 0.8392    |
| 2         | -3.484e+0 | 0.4229    |
| 3         | -2.103e+0 | 0.7231    |
| 4         | -151.5    | 0.01011   |
| 5         | -903.6    | 0.3093    |
| 6         | 324.5     | 0.1553    |
| 7         | 249.6     | 0.08109   |
| 8         | 72.52     | 0.2193    |
| 9         | 334.5     | 0.1205    |
| 10        | 338.7     | 0.1267    |
| 11        | 339.6     | 0.1385    |
| 12        | 339.9     | 0.1374    |
| 13        | 327.8     | 0.1144    |
| 14        | 340.2     | 0.1361    |
| 15        | 340.3     | 0.1353    |
| 16        | 340.4     | 0.1335    |
| 17        | 338.4     | 0.1415    |
| 18        | 339.8     | 0.1298    |
| 19        | 340.0     | 0.1303    |
| 20        | 338.3     | 0.1417    |
| 21        | 290.4     | 0.1716    |
| 22        | 339.9     | 0.13      |
| 23        | 340.1     | 0.1306    |
| 24        | 338.6     | 0.141     |
| 25        | 340.0     | 0.1303    |
| 26        | 340.2     | 0.1313    |
| 27        | 339.1     | 0.1401    |
| 28        | 340.0     | 0.1302    |
| 29        | 340.4     | 0.1348    |
| 30        | 339.7     | 0.1292    |
| 31        | 339.3     | 0.1395    |
| 32        | 340.3     | 0.1327    |
| 33        | 339.6     | 0.1289    |
| 34        | 340.0     | 0.137     |
| 35        | 340.3     | 0.1352    |
| 36        | 339.4     | 0.1284    |
=====================================
```

[22]:
```
print(f"BayesianOptimization finds that the optimimal isotropic length scale is {muygps_
↪isotropic.kernel.deformation.length_scale()}")
```

```
BayesianOptimization finds that the optimimal isotropic length scale is 0.
↪13350674734009632
```

We see here that when fixed to an isotropic length scale, Bayesian optimization tends to favor the smallest true length scale. We'll see how this affects modeling, prediction, and uncertainty quanlity below.

We separately optimize the `scale` variance scale parameter for each model.

[23]:
```
muygps_anisotropic = muygps_anisotropic.optimize_scale(
    batch_pairwise_diffs, batch_nn_targets
)
muygps_isotropic = muygps_isotropic.optimize_scale(
    batch_pairwise_diffs, batch_nn_targets
)
```

1.10.3 Inference

As in the *Univariate Regression Tutorial*, we must realize difference tensors formed from the testing data and apply them to form Gaussian process predictions for our problem.

[24]:
```
test_count, _ = test_features.shape
indices = np.arange(test_count)
test_nn_indices, _ = nbrs_lookup.get_nns(test_features)
(
    test_crosswise_diffs,
    test_pairwise_diffs,
    test_nn_targets,
) = muygps_anisotropic.make_predict_tensors(
    indices,
    test_nn_indices,
    test_features,
    train_features,
    train_responses,
)
(
    test_crosswise_dists,
    test_pairwise_dists,
    _,
) = muygps_isotropic.make_predict_tensors(
    indices,
    test_nn_indices,
    test_features,
    train_features,
    train_responses,
)
```

As in the *Univariate Regression Tutorial* we will evaluate the prediction performance of our models in terms of RMSE, mean diagonal posterior variance, the mean 95% confidence interval size, and the coverage, which ideally should be near 95%.

[25]:
```
Kcross_anisotropic = muygps_anisotropic.kernel(test_crosswise_diffs)
Kin_anisotropic = muygps_anisotropic.kernel(test_pairwise_diffs)

predictions_anisotropic = muygps_anisotropic.posterior_mean(
    Kin_anisotropic, Kcross_anisotropic, test_nn_targets
)
```

(continues on next page)

(continued from previous page)

```
variances_anisotropic = muygps_anisotropic.posterior_variance(
    Kin_anisotropic, Kcross_anisotropic
)
confidence_intervals_anisotropic = np.sqrt(variances_anisotropic) * 1.96
coverage_anisotropic = (
    np.count_nonzero(
        np.abs(test_responses - predictions_anisotropic) < confidence_intervals_
↪anisotropic
    ) / test_count
)
```

We also evaluate the isotropic model

[26]:
```
Kcross_isotropic = muygps_isotropic.kernel(test_crosswise_dists)
Kin_isotropic = muygps_isotropic.kernel(test_pairwise_dists)

predictions_isotropic = muygps_isotropic.posterior_mean(Kin_isotropic, Kcross_isotropic,␣
↪test_nn_targets)
variances_isotropic = muygps_isotropic.posterior_variance(Kin_isotropic, Kcross_
↪isotropic)

confidence_intervals_isotropic = np.sqrt(variances_isotropic) * 1.96
coverage_isotropic = (
    np.count_nonzero(
        np.abs(test_responses - predictions_isotropic) < confidence_intervals_isotropic
    ) / test_count
)
```

1.10.4 Results comparison

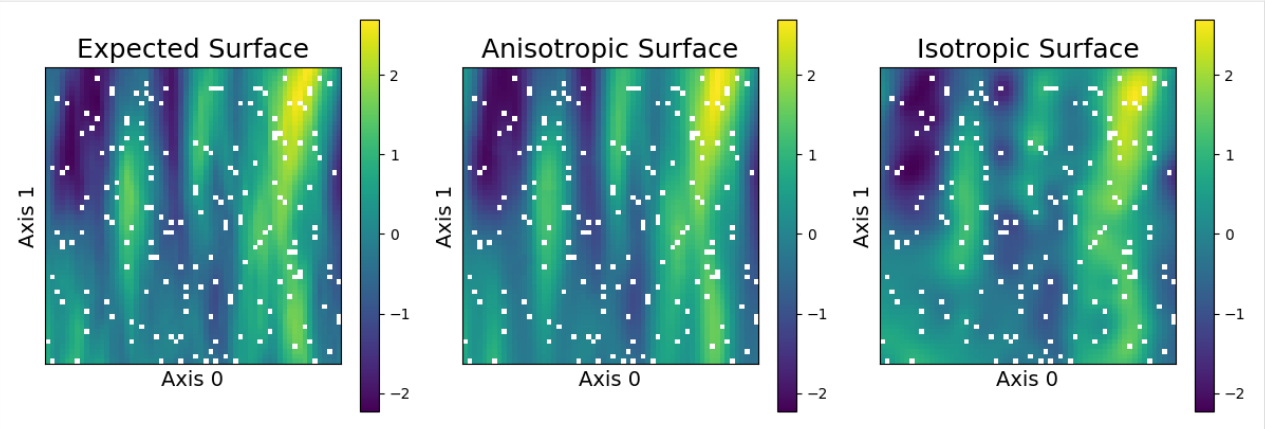
A comparison of our trained models reveals that the anisotropic kernel gets close to the true `(0.1, 0.5)` length scale, whereas the isotropic model has to learn a single parameter that has to split the difference somehow. This results in both a higher RMSE and larger confidence intervals in order to achieve similar coverage.

[27]:
```
print_results(
    test_responses,
    ("anisotropic", muygps_anisotropic, predictions_anisotropic, variances_anisotropic,␣
↪confidence_intervals_anisotropic, coverage_anisotropic),
    ("isotropic", muygps_isotropic, predictions_isotropic, variances_isotropic,␣
↪confidence_intervals_isotropic, coverage_isotropic),
)
```

[27]:
```
<pandas.io.formats.style.Styler at 0x7fe279aa7a30>
```

This dataset is low-dimensional so we can plot our predictions and visually evaluate their performance. We plot below the expected (true) surface, and the surface that our model predicts. Note that they are visually similar and major trends are captured, although there are some differences.

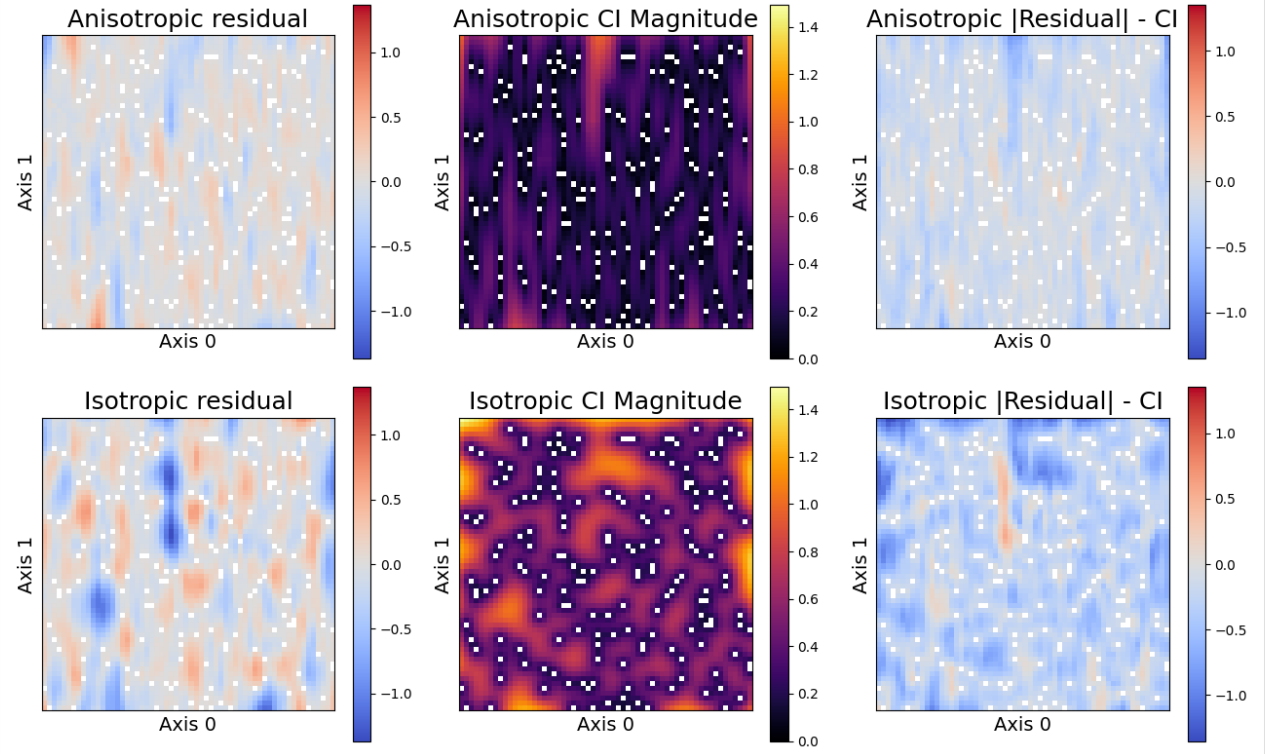
[28]:
```
sampler.plot_predictions(("Anisotropic", predictions_anisotropic), ("Isotropic",␣
↪predictions_isotropic))
```



As we can see, the anisotropic model learns a surface that is much visually closer to what is expected. In particular, the isotropic surface has blobby circular features as it to be expected, as it is unable to differentiate between distances along the different axes.

We will also investigate more details information about the errors. Below we produce three plots that help us to understand our results. The left plot shows the residual, which is the difference between the true values and our expectations. The middle plot shows the magnitude of the 95% confidence interval. The larger the confidence interval, the less certain the model is of its predictions. Finally, the right plot shows the difference between the 95% confidence interval length and the magnitude of the residual. All of the points larger than zero (in red) are not captured by the confidence interval. Hence, this plot shows our coverage distribution.

[29]:
```
sampler.plot_errors(
    ("Anisotropic", predictions_anisotropic, confidence_intervals_anisotropic),
    ("Isotropic", predictions_isotropic, confidence_intervals_isotropic),
)
```



The rightmost columns shows that the anisotropic assumptions both obtains lower residuals, i.e. the posterior means are more accurate. The middle column shows that the the posterior variances (and resulting confidence intervals) are smaller, and therefore the anisotropic model is also more confident in its predictions. Finally, the rightmost plot reveals the uncovered points - all red-scale residuals exceed the confidence interval. Not only does the isotropic model appear to have more uncovered points, they tend to be further outside of the confidence interval than those of the anisotropic model. These results demonstrate the importance of correct model assumptions, both on predictions and uncertainty quantification.

1.11 Loss Function Tutorial

This notebook illustrates the loss functions available in the `MuyGPyS` library. These functions are used to formulate the objective function to be optimized while fitting hyperparameters, and so have a large effect on the outcome of training. We will describe each of these loss functions and plot their behaviors to help the user to select the right loss for their problem.

Each function in this notebook is available for import from `MuyGPyS.optimize.loss`, and is an object of class `MuyGPyS.optimize.loss.LossFn`. It is possible to define new loss functions by creating a new `LossFn` object. View its documentation for more details.

We assume throughout a vector of targets y, a prediction (posterior mean) vector μ, and a posterior variance vector σ for a training batch B with b elements.

[2]:
```
import matplotlib.pyplot as plt
import numpy as np

import cblind as cb
from matplotlib.colors import SymLogNorm, LogNorm
from MuyGPyS.optimize.loss import mse_fn, cross_entropy_fn, lool_fn, pseudo_huber_fn,␣
↪looph_fn
```

[3]:
```
plt.style.use('tableau-colorblind10')
```

[4]:
```
mmax = 3.0
mmin = 0.0
residual_count = 100
ys = np.zeros(residual_count)
residuals = np.linspace(mmin, mmax, residual_count)
smax = 3.0
smin = 1e-1
variance_count = 100
variances = np.linspace(smin, smax, variance_count)
unitary_scale=np.ones(1)
```

1.11.1 Variance-free Loss Functions

`MuyGPyS` features several loss functions that depend only upon the targets y and posterior mean predictions $\hat{\mu}$ of your training batch. These loss functions are situationally useful, although they leave the fitting of variance parameters entirely up to the separate, scale optimization functions and might not be sensitive to certain variance parameters. As they do not require evaluating the posterior variance $\hat{\Sigma}$ or optimizing the variance scale parameter σ^2, these loss functions are generally more efficient to use in practice.

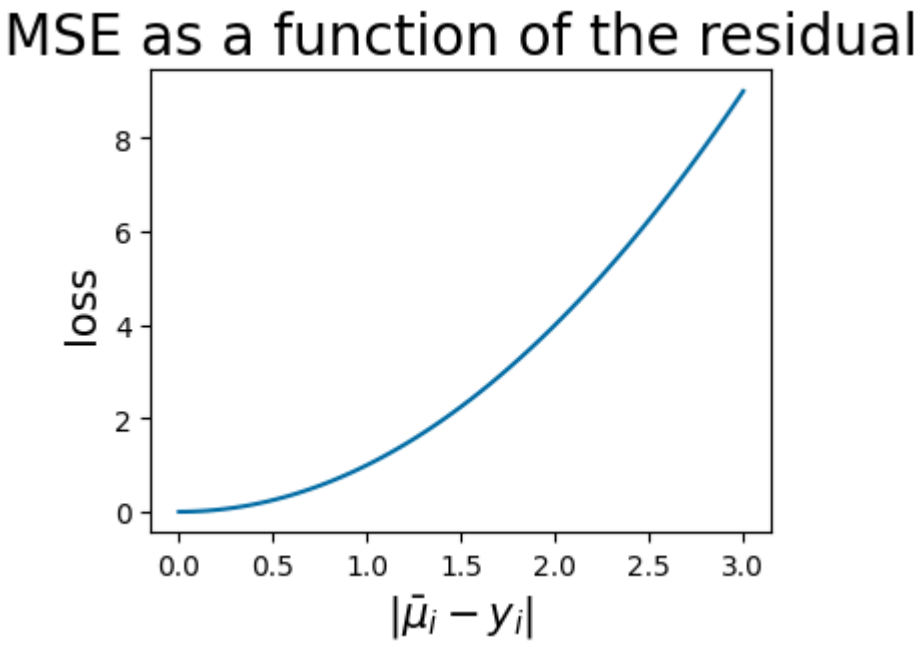
Mean Squared Error (`mse_fn`)

The mean squared error (MSE) or ℓ_2 loss is a classic loss function that computes

$$\ell_{\text{MSE}}(\bar{\mu}, y) = \frac{1}{b}\sum_{i \in B}(\bar{\mu}_i - y_i)^2.$$

The following plot illustrates the MSE as a function of the residual.

[5]:
```
fig, ax = plt.subplots(1, 1, figsize=(4,3))
ax.set_title("MSE as a function of the residual", fontsize=20)
ax.set_ylabel("loss", fontsize=15)
ax.set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
mses = np.array([mse_fn(ys[i], residuals[i]) for i in range(residual_count)])
ax.plot(residuals, mses)
plt.show()
```



Cross Entropy Loss (`cross_entropy_fn`)

The cross entropy loss is a classic classification loss often used in the fitting of neural networks. For targets in $\{0, 1\}$, the library first transforms the predictions to be row-stochastic and then computes

$$\ell_{\text{cross-entropy}}(\bar{\mu}, y) = \sum_{i \in B} y_i \log(\bar{\mu}_i) - (1 - y_i) \log(1 - \bar{\mu}_i)$$

This section is under construction.

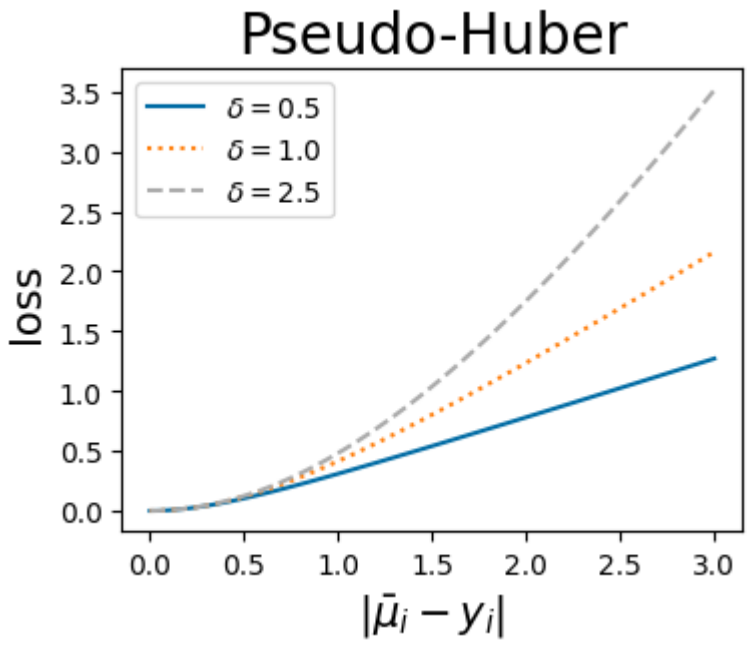
Pseudo-Huber Loss (`pseudo_huber_fn`)

The pseudo-Huber loss is a smooth approximation to the Huber loss, which is approximately quadratic (ℓ_2 loss) for small residuals and approximately linear (ℓ_1 loss) for large residuals. This means that the pseudo-Huber loss is less sensitive to large outliers, which might otherwise force the optimizer to overcompensate in undesirable ways. The pseudo-Huber loss computes

$$\ell_{\text{Pseudo-Huber}}(\bar{\mu}, y \mid \delta) = \sum_{i=1}^{b} \delta^2 \left(\sqrt{1 + \left(\frac{\bar{\mu}_i - y_i}{\delta} \right)^2} - 1 \right),$$

where δ is a parameter that indicates the scale of the boundary between the quadratic and linear parts of the function. The `pseudo_huber_fn` accepts this parameter as the `boundary_scale` keyword argument. Note that the scale of δ depends on the units of y and $\hat{\mu}$. The following plots show the behavior of the pseudo-Huber loss for a few values of δ.

[6]:
```
boundary_scales = [0.5, 1.0, 2.5]
phs = np.array([
    [pseudo_huber_fn(ys[i], residuals[i], boundary_scale=bs) for i in range(residual_
↪count)]
    for bs in boundary_scales
])
fig, ax = plt.subplots(1, 1, figsize=(4, 3))
# for i, ax in enumerate(axes):
ax.set_title(f"Pseudo-Huber", fontsize=20)
ax.set_ylabel("loss", fontsize=15)
ax.set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
ax.plot(residuals, phs[0, :], linestyle="solid", label=f"$\delta = {boundary_scales[0]}$
↪")
ax.plot(residuals, phs[1, :], linestyle="dotted", label=f"$\delta = {boundary_scales[1]}$
↪")
ax.plot(residuals, phs[2, :], linestyle="dashed", label=f"$\delta = {boundary_scales[2]}$
↪")
ax.legend()
plt.show()
```



Coincidence of pseudo-Huber and MSE for relatively small residuals

For large boundary scales relative the residual magnitude, the pseudo-Huber function converges to 1/2 of the ℓ_2 loss or mean squared error, as show by the next figure. This convergence is relative to the scale of the residual, and so the value of the boundary scale is informed by the data distribution and requires the user to reason about the breakpoint where residuals are likely to be unreasonably large. Similarly, when $\delta = 1$ the pseudo-huber loss is parallel to ℓ_1 loss for larger residuals.

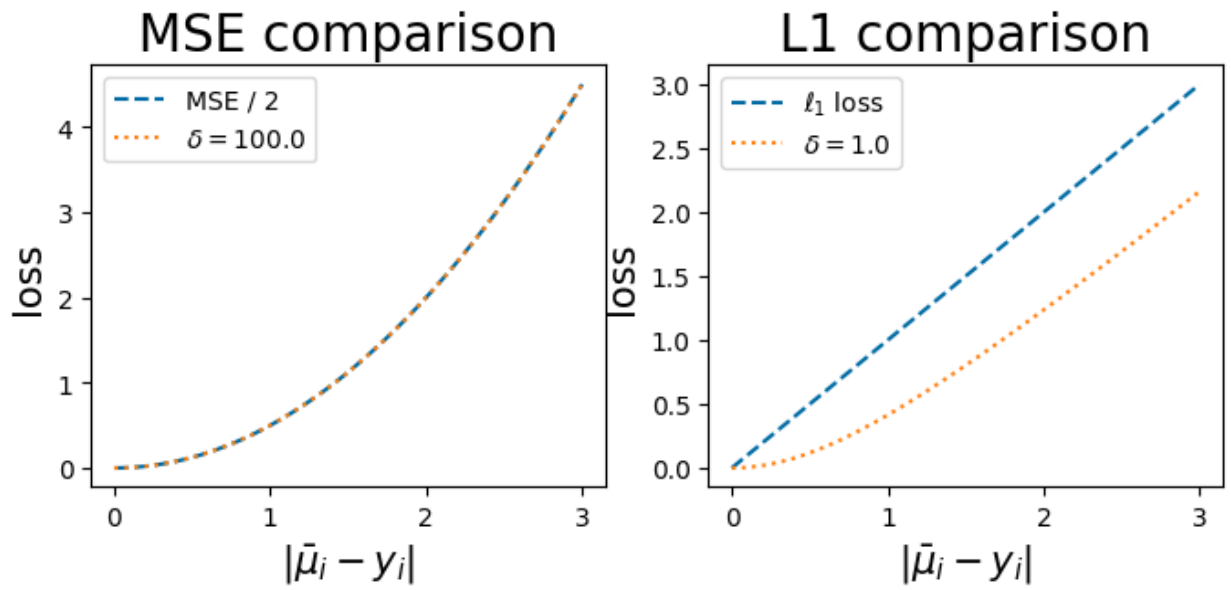
[7]:
```
def l1_fn(
    predictions: np.ndarray,
    targets: np.ndarray,
) -> float:
    return np.sum(np.abs(predictions - targets))

l1s = np.array([l1_fn(ys[i], residuals[i]) for i in range(residual_count)])
```

[8]:
```
big_boundary_scale = 100.0
sml_boundary_scale = 1.0
big_ph = np.array([
    pseudo_huber_fn(ys[i], residuals[i], boundary_scale=big_boundary_scale)
    for i in range(residual_count)
])
sml_ph = np.array([
    pseudo_huber_fn(ys[i], residuals[i], boundary_scale=sml_boundary_scale)
    for i in range(residual_count)
])
fig, axes = plt.subplots(1, 2, figsize=(8,3))
axes[0].set_title("MSE comparison", fontsize=20)
axes[0].set_ylabel("loss", fontsize=15)
```

(continues on next page)

(continued from previous page)

```
axes[0].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert $", fontsize=15)
axes[0].plot(residuals, mses / 2, linestyle="dashed", label=f"MSE / 2")
axes[0].plot(residuals, big_ph, linestyle="dotted", label=f"$\delta = {big_boundary_
↪scale}$")
axes[0].legend()

axes[1].set_title("L1 comparison", fontsize=20)
axes[1].set_ylabel("loss", fontsize=15)
axes[1].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
axes[1].plot(residuals, l1s, linestyle="dashed", label="$\ell_1$ loss")
axes[1].plot(residuals, sml_ph, linestyle="dotted", label=f"$\delta = {sml_boundary_
↪scale}$")
axes[1].legend()
plt.show()
```



1.11.2 Variance-Sensitive Loss Functions

`MuyGPyS` also includes loss functions that explicitly depend upon the posterior variances $\bar{\Sigma}$, which is a diagonal matrix for a univariate MuyGPs model. These loss functions penalize large variances, and so tend to be more sensitive to variance parameters. This comes at increasing the cost of the linear algebra involved in each evaluation of the objective function by a constant factor. This causes an overall increase in compute time per optimization loop, but that is often a worthwhile trade for sensitivity in practice.

$\bar{\Sigma}$ involves multiplying the unscaled `MuyGPS` variance by the σ^2 variance scaling parameter, which at present must by optimized during each evaluation of the objective function.

Leave-One-Out Loss (`lool_fn`)

The leave-one-out-loss or lool scales and regularizes the MSE to make the loss more sensitive to parameters that primarily act on the variance. lool computes

$$\ell_{\text{lool}}(\bar{\mu}, y, \bar{\Sigma}) = \sum_{i \in B} \frac{(\bar{\mu}_i - y_i)^2}{\bar{\Sigma}_{ii}} + \log \bar{\Sigma}_{ii}.$$

The next plot illustrates the loss as a function of both the residual and of σ^2.

[9]:
```
lools = np.array([
    [
        lool_fn(
            ys[i],
            residuals[i],
            variances[variance_count - 1 - j],
            unitary_scale,
        )
        for i in range(residual_count)
    ]
    for j in range(variance_count)
])
```

[10]:
```
variance_vis_values = [0.5, 1.0, 1.5]
variance_vis_points = list()
var_iter = 0
for i, var in enumerate(variances):
    if var_iter >= len(variance_vis_values):
        break
    if var > variance_vis_values[var_iter]:
        variance_vis_points.append([variance_count - 1 - i, var])
        var_iter += 1
```

[11]:
```
residual_vis_values = [0.25, 0.75, 1.25]
residual_vis_points = list()
res_iter = 0
for i, res in enumerate(residuals):
    if res_iter >= len(residual_vis_values):
        break
    if res > residual_vis_values[res_iter]:
        residual_vis_points.append([i, res])
        res_iter += 1
```

[12]:
```
style_count = len(variance_vis_values) + len(residual_vis_values)
colors, linestyles = cb.Colorplots().cblind(style_count)
colors = colors[:style_count]
linestyles = linestyles[:style_count]
linestyles = list(reversed(linestyles))
```

[13]:
```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes[0].set_title("lool", fontsize=20)
axes[0].set_ylabel(r"$\bar{\Sigma}_{ii}$", fontsize=15)
```

(continues on next page)

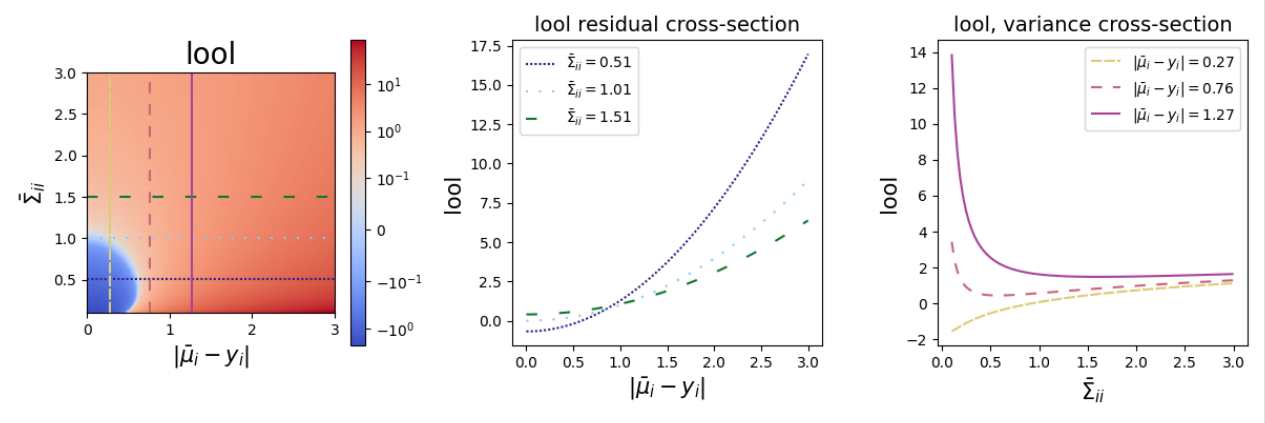
(continued from previous page)

```
axes[0].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
im = axes[0].imshow(
    lools, extent=[mmin, mmax, smin, smax], norm=SymLogNorm(1e-1), cmap="coolwarm"
)
fig.colorbar(im, ax=axes[0])
axes[0].plot(residuals, variance_count * [variance_vis_points[0][1]], color=colors[0],␣
↪linestyle=linestyles[0])
axes[0].plot(residuals, variance_count * [variance_vis_points[1][1]], color=colors[1],␣
↪linestyle=linestyles[1])
axes[0].plot(residuals, variance_count * [variance_vis_points[2][1]], color=colors[2],␣
↪linestyle=linestyles[2])
axes[0].plot(residual_count * [residual_vis_points[0][1]], variances, color=colors[3],␣
↪linestyle=linestyles[3])
axes[0].plot(residual_count * [residual_vis_points[1][1]], variances, color=colors[4],␣
↪linestyle=linestyles[4])
axes[0].plot(residual_count * [residual_vis_points[2][1]], variances, color=colors[5],␣
↪linestyle=linestyles[5])

axes[1].set_title("lool residual cross-section", fontsize=14)
axes[1].set_ylabel("lool", fontsize=15)
axes[1].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert $", fontsize=15)
axes[1].plot(
    residuals,
    lools[variance_vis_points[0][0], :],
    color=colors[0],
    linestyle=linestyles[0],
    label=r"$\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[0][1]:.2f}",
)
axes[1].plot(
    residuals,
    lools[variance_vis_points[1][0], :],
    color=colors[1],
    linestyle=linestyles[1],
    label=r"$\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[1][1]:.2f}",
)
axes[1].plot(
    residuals,
    lools[variance_vis_points[2][0], :],
    color=colors[2],
    linestyle=linestyles[2],
    label=r"$\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[2][1]:.2f}",
)
axes[1].legend()

axes[2].set_title("lool, variance cross-section", fontsize=14)
axes[2].set_ylabel("lool", fontsize=15)
axes[2].set_xlabel(r"$\bar{\Sigma}_{ii}$", fontsize=15)
axes[2].plot(
    variances,
    np.flip(lools[:, residual_vis_points[0][0]]),
    color=colors[3],
    linestyle=linestyles[3],
```

(continues on next page)

(continued from previous page)

```
    label=r"$\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[0][1]:.2f}",
)
axes[2].plot(
    variances,
    np.flip(lools[:, residual_vis_points[1][0]]),
    color=colors[4],
    linestyle=linestyles[4],
    label=r"$\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[1][1]:.2f}",
)
axes[2].plot(
    variances,
    np.flip(lools[:, residual_vis_points[2][0]]),
    color=colors[5],
    linestyle=linestyles[5],
    label=r"$\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[2][1]:.2f}",
)
axes[2].legend()

plt.tight_layout()
plt.show()
```



Notice that the cross-section of the lool surface for a fixed σ is quadratic, while the cross section of the lool surface for a fixed residual is logarithmic. For small enough residuals, this curve inverts and assumes negative values for small σ.

Leave-One-Out Pseudo-Huber (`looph_fn`)

The leave-one-out pseudo-Huber loss (looph) is similar in nature to the lool, but is applied to the pseudo-Huber loss instead of MSE. looph computes

$$\ell_{\text{looph}}(\bar{\mu}, y, \bar{\Sigma} \mid \delta) = \sum_{i=1}^{b} 2\delta^2 \left(\sqrt{1 + \frac{(\bar{\mu}_i - y_i)^2}{\delta^2 \bar{\Sigma}_{ii}}} - 1 \right) + \log \bar{\Sigma}_{ii},$$

where again δ is the boundary scale.

Note that unlike in the pseudo-Huber, here the boundary scale δ is unitless. δ specifies how large the residual must be, in multiples of the standard deviation, for the loss to become approximately linear instead of approximately quadratic. As such, there is no need for most applications to set δ, which the library defaults to `3.0`. This implies that only residuals that are larger than 3 standard deviations are treated as outliers.

The next plots illustrate the looph as a function of the residual, σ, and δ for $\delta \in \{0.5, 3.0\}$. The plots for the smaller δ value illustrates why δ should not be small.

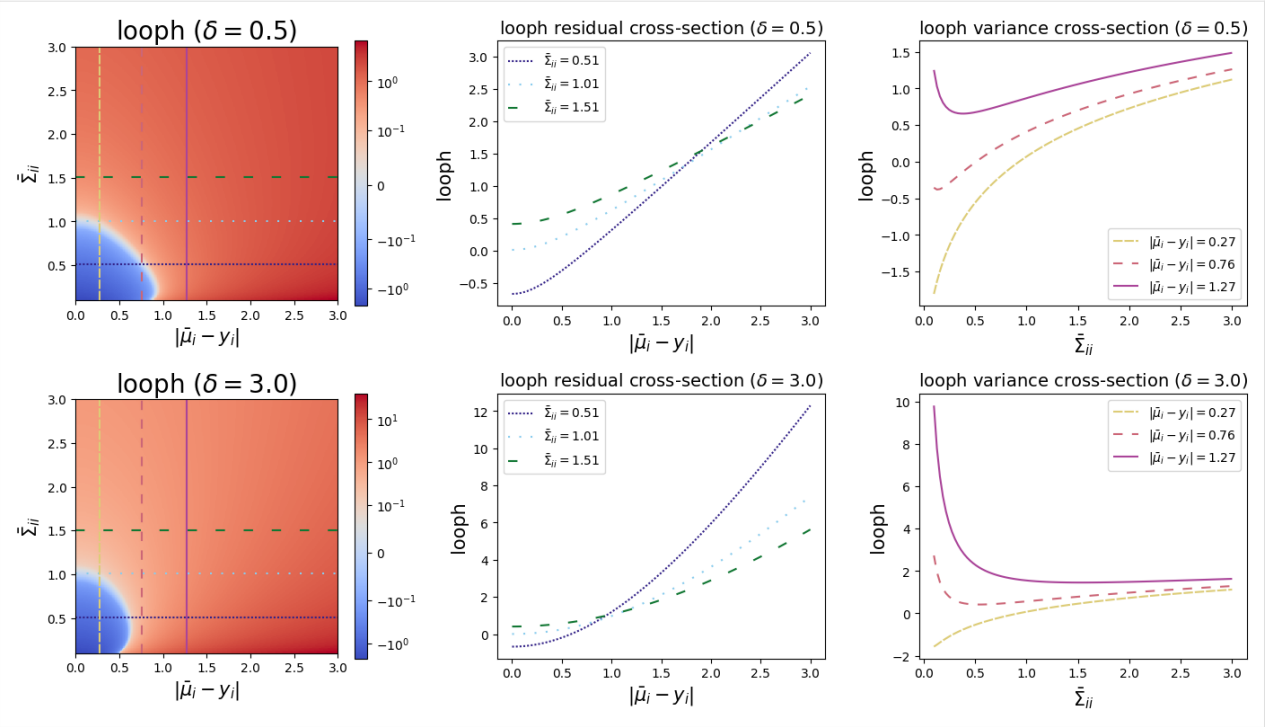
[14]:
```
loo_boundary_scales = np.array([0.5, 3.0])
loophs = np.array([
    [
        [
            looph_fn(
                ys[i],
                residuals[i],
                variances[variance_count - 1 - j],
                unitary_scale,
                boundary_scale=bs
            )
            for i in range(residual_count)
        ]
        for j in range(variance_count)
    ]
    for bs in loo_boundary_scales
])
```

[15]:
```
fig, axes = plt.subplots(2, 3, figsize=(14, 4 * len(loo_boundary_scales)))
for i, bs in enumerate(loo_boundary_scales):
    axes[i, 0].set_title(f"looph ($\delta={bs}$)", fontsize=20)
    axes[i, 0].set_ylabel(r"$\bar{\Sigma}_{ii}$", fontsize=15)
    axes[i, 0].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
    im = axes[i, 0].imshow(
        loophs[i, :, :], extent=[mmin, mmax, smin, smax], norm=SymLogNorm(1e-1), cmap=
↪"coolwarm"
    )
    fig.colorbar(im, ax=axes[i, 0])
    axes[i, 0].plot(residuals, variance_count * [variance_vis_points[0][1]],␣
↪color=colors[0], linestyle=linestyles[0])
    axes[i, 0].plot(residuals, variance_count * [variance_vis_points[1][1]],␣
↪color=colors[1], linestyle=linestyles[1])
    axes[i, 0].plot(residuals, variance_count * [variance_vis_points[2][1]],␣
↪color=colors[2], linestyle=linestyles[2])
    axes[i, 0].plot(residual_count * [residual_vis_points[0][1]], variances,␣
↪color=colors[3], linestyle=linestyles[3])
    axes[i, 0].plot(residual_count * [residual_vis_points[1][1]], variances,␣
↪color=colors[4], linestyle=linestyles[4])
    axes[i, 0].plot(residual_count * [residual_vis_points[2][1]], variances,␣
↪color=colors[5], linestyle=linestyles[5])

    axes[i, 1].set_title(f"looph residual cross-section ($\delta={bs}$)", fontsize=14)
    axes[i, 1].set_ylabel("looph", fontsize=15)
    axes[i, 1].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
    axes[i, 1].plot(
        residuals,
        loophs[i, variance_vis_points[0][0], :],
        color=colors[0],
        linestyle=linestyles[0],
        label=r"$\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[0][1]:.2f}",
```

(continues on next page)

(continued from previous page)

```
    )
    axes[i, 1].plot(
        residuals,
        loophs[i, variance_vis_points[1][0], :],
        color=colors[1],
        linestyle=linestyles[1],
        label=r"$\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[1][1]:.2f}",
    )
    axes[i, 1].plot(
        residuals,
        loophs[i, variance_vis_points[2][0], :],
        color=colors[2],
        linestyle=linestyles[2],
        label=r"$\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[2][1]:.2f}",
    )
    axes[i, 1].legend()

    axes[i, 2].set_title(f"looph variance cross-section ($\delta={bs}$)", fontsize=14)
    axes[i, 2].set_ylabel("looph", fontsize=15)
    axes[i, 2].set_xlabel(r"$\bar{\Sigma}_{ii}$", fontsize=15)
    axes[i, 2].plot(
        variances,
        np.flip(loophs[i, :, residual_vis_points[0][0]]),
        color=colors[3],
        linestyle=linestyles[3],
        label=r"$\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[0][1]:.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(loophs[i, :, residual_vis_points[1][0]]),
        color=colors[4],
        linestyle=linestyles[4],
        label=r"$\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[1][1]:.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(loophs[i, :, residual_vis_points[2][0]]),
        color=colors[5],
        linestyle=linestyles[5],
        label=r"$\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[2][1]:.2f}",
    )
    axes[i, 2].legend()

plt.tight_layout()
plt.show()
```



These plots show us that the looph function can exhibit a more exaggerated upward slope where the residual is in the linear component of the pseudo-Huber curve but is not so large that it still outweighs the variance component of the loss. Note that in practice that both pseudo Huber loss functions may require more training iterations to converge than their alternatives.

Comparison between lool and looph

Here we compare looph to lool for differing the `boundary_scale`s. We see that, similar to the original pseudo-Huber, the looph also converges to lool as the `boundary_scale` grows large. Similarly, looph's loss for a fixed variance becomes linear when $\frac{(\bar{\mu}_i - y_i)^2}{\Sigma_{ii}}$ exceeds the `boundary_scale`.

[16]:
```
compare_boundary_scales = np.array([3.0, 20.0])
compare_loophs = np.array([
    [
        [
            looph_fn(
                ys[i],
                residuals[i],
                variances[variance_count - 1 - j],
                unitary_scale,
                boundary_scale=bs,
            )
            for i in range(residual_count)
        ]
        for j in range(variance_count)
    ]
    for bs in compare_boundary_scales
])
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(2, 3, figsize=(14, 4 * len(compare_boundary_scales)))

for i, bs in enumerate(compare_boundary_scales):
    axes[i, 0].set_title(f"|lool - looph| ($\delta$ = {bs})", fontsize=14)
    axes[i, 0].set_ylabel(r"$\bar{\Sigma}_{ii}$", fontsize=15)
    axes[i, 0].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
    im = axes[i, 0].imshow(
        np.abs(lools - compare_loophs[i, :, :]), extent=[mmin, mmax, smin, smax], cmap=
↪"cb.iris", norm=LogNorm(1e-1)
    )
    fig.colorbar(im, ax=axes[i, 0])
    axes[i, 0].plot(residuals, variance_count * [variance_vis_points[0][1]],␣
↪color=colors[0], linestyle=linestyles[-1])
    axes[i, 0].plot(residuals, variance_count * [variance_vis_points[1][1]],␣
↪color=colors[1], linestyle=linestyles[-1])
    axes[i, 0].plot(residuals, variance_count * [variance_vis_points[2][1]],␣
↪color=colors[2], linestyle=linestyles[-1])
    axes[i, 0].plot(residual_count * [residual_vis_points[0][1]], variances,␣
↪color=colors[3], linestyle=linestyles[-1])
    axes[i, 0].plot(residual_count * [residual_vis_points[1][1]], variances,␣
↪color=colors[4], linestyle=linestyles[-1])
    axes[i, 0].plot(residual_count * [residual_vis_points[2][1]], variances,␣
↪color=colors[5], linestyle=linestyles[-1])

    axes[i, 1].set_title("lool/looph residual cross-section", fontsize=14)
    axes[i, 1].set_ylabel("lool", fontsize=15)
    axes[i, 1].set_xlabel(r"$\vert \bar{\mu}_i - y_i \vert$", fontsize=15)
    axes[i, 1].plot(
        residuals,
        lools[variance_vis_points[0][0], :],
        color=colors[0],
        linestyle=linestyles[0],
        label=r"lool, $\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[0][1]:.2f}",
    )
    axes[i, 1].plot(
        residuals,
        lools[variance_vis_points[1][0], :],
        color=colors[1],
        linestyle=linestyles[0],
        label=r"lool, $\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[1][1]:.2f}",
    )
    axes[i, 1].plot(
        residuals,
        lools[variance_vis_points[2][0], :],
        color=colors[2],
        linestyle=linestyles[0],
        label=r"lool, $\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[2][1]:.2f}",
    )
    axes[i, 1].plot(
        residuals,
        compare_loophs[i, variance_vis_points[0][0], :],
```

(continues on next page)

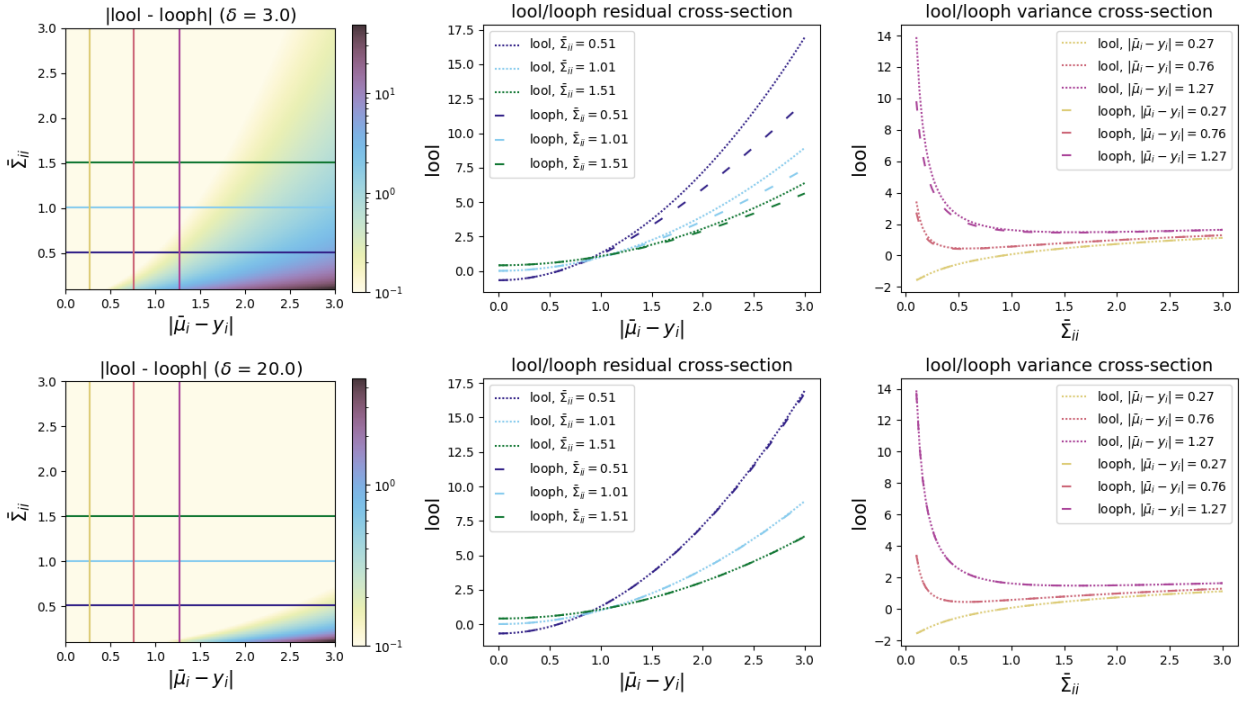
(continued from previous page)

```
        color=colors[0],
        linestyle=linestyles[2],
        label=r"looph, $\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[0][1]:.2f}",
    )
    axes[i, 1].plot(
        residuals,
        compare_loophs[i, variance_vis_points[1][0], :],
        color=colors[1],
        linestyle=linestyles[2],
        label=r"looph, $\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[1][1]:.2f}",
    )
    axes[i, 1].plot(
        residuals,
        compare_loophs[i, variance_vis_points[2][0], :],
        color=colors[2],
        linestyle=linestyles[2],
        label=r"looph, $\bar{\Sigma}_{ii} = $" + f"{variance_vis_points[2][1]:.2f}",
    )
    axes[i, 1].legend()

    axes[i, 2].set_title("lool/looph variance cross-section", fontsize=14)
    axes[i, 2].set_ylabel("lool", fontsize=15)
    axes[i, 2].set_xlabel(r"$\bar{\Sigma}_{ii}$", fontsize=15)
    axes[i, 2].plot(
        variances,
        np.flip(lools[:, residual_vis_points[0][0]]),
        color=colors[3],
        linestyle=linestyles[0],
        label=r"lool, $\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[0][1]:
↪.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(lools[:, residual_vis_points[1][0]]),
        color=colors[4],
        linestyle=linestyles[0],
        label=r"lool, $\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[1][1]:
↪.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(lools[:, residual_vis_points[2][0]]),
        color=colors[5],
        linestyle=linestyles[0],
        label=r"lool, $\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_points[2][1]:
↪.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(compare_loophs[i, :, residual_vis_points[0][0]]),
        color=colors[3],
        linestyle=linestyles[2],
```

(continues on next page)

(continued from previous page)

```
        label=r"looph, $\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_
↪points[0][1]:.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(compare_loophs[i, :, residual_vis_points[1][0]]),
        color=colors[4],
        linestyle=linestyles[2],
        label=r"looph, $\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_
↪points[1][1]:.2f}",
    )
    axes[i, 2].plot(
        variances,
        np.flip(compare_loophs[i, :, residual_vis_points[2][0]]),
        color=colors[5],
        linestyle=linestyles[2],
        label=r"looph, $\vert \bar{\mu}_i - y_i \vert = $" + f"{residual_vis_
↪points[2][1]:.2f}",
    )
    axes[i, 2].legend()

plt.tight_layout()
plt.show()
```



1.12 References

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[muyskens2021muygps] Muyskens, Amanda, Benjamin W. Priest, Imène Goumiri, and Michael Schneider. "MuyGPs: Scalable Gaussian Process Hyperparameter Estimation Using Local Cross-Validation." arXiv preprint arXiV:2104.14581 (2021).

[muyskens2021star] Muyskens, Amanda L., Imène R. Goumiri, Benjamin W. Priest, Michael D. Schneider, Robert E. Armstrong, Jason M. Bernstein, and Ryan Dana. "Star-Galaxy Image Separation with Computationally Efficient Gaussian Process Classification." arXiv preprint arXiv:2105.01106 (2021).

[dunton2022fast] Dunton, Alec M., Benjamin W. Priest, and Amanda Muyskens. "Fast Gaussian Process Posterior Mean Prediction via Local Cross Validation and Precomputation." arXiv preprint arXiv:2205.10879 (2022).

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

F

G

H

I

K

L

M

N

O

P

R

S

T

U