MuyGPyS

Release 0.6.6

Benjamin W. Priest

Jul 18, 2023

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 distortion

1.2.2 tensors

Tensor functions

Compute pairwise and crosswise difference tensors for the purposes of kernel construction.

See the following example computing the pairwise and crosswise differences between a batch of training data and their nearest neighbors.

Example

```
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> from MuyGPyS.optimize.batch import sample_batch
>>> from MuyGPyS.gp.tensors import crosswise_tensor, pairwise_tensor
>>> 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, batch_nn_indices = sample_batch(
...         nbrs_lookup, batch_count, train_count
... )
>>> pairwise_diffs = pairwise_tensor(
...         train_features, batch_nn_inidices
... )
>>> crosswise_diffs = crosswise_tensor(
...         train_features,
...         train_features,
...         batch_indices,
...         batch_nn_indices,
... )
)
```

See also the following example computing the crosswise differences between a test dataset and their nearest neighors in the training data.

Example

```
>>> from MuyGPyS.neighbors import NN_Wrapper
>>> from MuyGPyS.gp.tensors import crosswise_tensor, pairwise_tensor
>>> 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_diffs = nbrs_lookup.get_nns(test_features)
>>> test_count, _ = test_features.shape
>>> indices = np.arange(test_count)
>>> nn_indices, _ = nbrs_lookup.get_nns(test_features)
>>> pairwise_diffs = pairwise_tensor(
...         train_features, nn_inidices
... )
>>> crosswise_diffs = crosswise_tensor(
...         test_features,
...         train_features,
```

(continues on next page)

(continued from previous page)

```
...         indices,
...         nn_indices,
... )
```

The helper functions *MuyGPyS.gp.tensors.make_predict_tensors()*, *MuyGPyS.gp.tensors.make_fast_predict_tensors()*, and *MuyGPyS.gp.tensors.make_train_tensors()* wrap these difference tensors and also return the nearest neighbors sets' training targets and (in the latter case) the training targets of the training batch. These functions are convenient as the difference and target tensors are usually needed together.

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

Compute a matrix of differences between data and their nearest neighbors.

Takes full datasets of records of interest `data` and neighbor candidates `nn_data` and produces the differences 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.

See the following example computing the crosswise differences between a batch of training data and their nearest neighbors.

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 list the difference between each feature of each batch element element and its nearest neighbors.

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 *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.

Creates `eps_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.

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.

Return type
eps_tensor

MuyGPyS.gp.tensors.**make_noise_tensor**(*measurement_noise*, *nn_indices*)

Create the heteroscedastic noise tensor for nonuniform noise values for prediction of test data. Can also be used to produce the noise tensor needed during batched training.

Creates `eps_tensor` tensor required by heteroscedastic MuyGPs models.

Parameters

- **test** –
- **measurement_noise** (ndarray) – A matrix of floats of shape `(train_count)` providing the noise corresponding to the response variable at each input value in the data.
- **nn_indices** (ndarray) – The indices of the nearest neighbors of the test points.

Returns
A matrix of floats of shape `(test_count, nn_count)` providing the noise corresponding to the nearest neighbor responses for all observations in the test set.

Return type
eps_tensor

MuyGPyS.gp.tensors.**make_predict_tensors**(*batch_indices*, *batch_nn_indices*, *test_features*, *train_features*, *train_targets*)

Create the difference and target tensors for prediction.

Creates the `crosswise_diffs`, `pairwise_diffs` 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, feature_count)`.
- **train_targets** (`ndarray`) – A matrix of shape `(train_count, feature_count)` whose rows are vector-valued responses for each training element.

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

Returns

- *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.
- *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_train_tensors**(*batch_indices*, *batch_nn_indices*, *train_features*, *train_targets*)

Create the difference 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, feature_count)`.
- **train_targets** (ndarray) – A matrix of shape `(train_count, feature_count)` whose rows are vector-valued responses for each training element.

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

Returns

- *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.
- *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.

MuyGPyS.gp.tensors.**pairwise_tensor**(*data*, *nn_indices*)

Compute a tensor of pairwise differences among sets of nearest neighbors.

Takes a full dataset of records of interest `data` and produces the pairwise differences 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.

1.2.3 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(
...     nu=ScalarHyperparameter("log_sample", (0.1, 2.5)),
...     metric=IsotropicDistortion(
...         l2,
...         length_scale=ScalarHyperparameter(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

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

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

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*)

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

MuyGPyS.gp.kernels.**rbf**

alias of <module 'MuyGPyS.gp.kernels.rbf' from '/home/docs/checkouts/readthedocs.org/user_builds/muygpys/envs/docs/lib/pyth
packages/MuyGPyS/gp/kernels/rbf.py'>

MuyGPyS.gp.kernels.**matern**

alias of <module 'MuyGPyS.gp.kernels.matern' from '/home/docs/checkouts/readthedocs.org/user_builds/muygpys/envs/docs/lib/
packages/MuyGPyS/gp/kernels/matern.py'>

1.2.4 MuyGPS

class MuyGPyS.gp.muygps.**MuyGPS**(*kernel*, *eps=<MuyGPyS.gp.noise.homoscedastic.HomoscedasticNoise object>*, *response_count=1*)

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 homoscedastic ε noise parameter and a vector of σ^2 indicating the scale parameter associated with the posterior variance of each dimension of the response.

σ^2 is the only parameter assumed to be a training target by default, and is treated differently from all other hyperparameters. All other training targets must be manually specified in *k_kwargs*.

Example

```
>>> from MuyGPyS.gp import MuyGPS
>>> k_kwargs = {
...         "kern": "rbf",
...         "metric": "F2",
...         "eps": {"val": 1e-5},
...         "nu": {"val": 0.38, "bounds": (0.1, 2.5)},
...         "length_scale": {"val": 7.2},
... }
>>> muygps = MuyGPS(**k_kwarg)
```

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

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

Parameters

- **kernel** (Union[Matern, RBF]) – The kernel to be used.
- **eps** (Union[NullNoise, HomoscedasticNoise, HeteroscedasticNoise]) – A noise model.
- **response_count** (int) – The number of response dimensions.

apply_new_noise(*new_noise*)

Updates the homo/heteroscedastic noise parameter(s) of a MuyGPs model. To be used when the MuyGPs model has been trained and needs to be used for prediction, or if multiple batches are needed during training of a heteroscedastic model.

Parameters

new_noise (Union[HeteroscedasticNoise, HomoscedasticNoise, NullNoise]) – If homoscedastic, a float to update the nugget parameter. If heteroscedastic, a matrix of shape *(test_count, nn_count)* containing the measurement noise corresponding to the nearest neighbors of each test point.

Returns

A MuyGPs model with updated noise parameter(s).

fast_coefficients(*K*, *train_nn_targets_fast*)

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

To form each row of this matrix, we compute

$$\mathbf{C}_{N^*}(i,:) = (K_{\hat{\theta}}(X_{N^*}, X_{N^*}) + \varepsilon I_k)^{-1} Y(X_{N^*}).$$

Here X_{N^*} is the union of the nearest neighbor of the ith test point and the *nn_count - 1* nearest neighbors of this nearest neighbor, $K_{\hat{\theta}}$ is the trained kernel functor specified by *self.kernel*, εI_k is a diagonal noise matrix whose diagonal is the value of the *self.eps* hyperparameter, and $Y(X_{N^*})$ is the *(train_count,)* vector of responses corresponding to the training features indexed by N^*.

Parameters

- **K** (ndarray) – The full pairwise kernel tensor of shape *(train_count, nn_count, nn_count)*.
- **train_nn_targets_fast** (ndarray) – The nearest neighbor response of each training points of shape *(train_count, nn_count, response_count)*.

Return type

ndarray

Returns

A matrix 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]. For each test point $\mathbf{z}$, we compute

$$\widehat{Y}(\mathbf{z} \mid X) = K_\theta(\mathbf{z}, X_{N^*})\mathbf{C}_{N^*}.$$

Here X_{N^*} is the union of the nearest neighbor of the queried test point $\mathbf{z}$ and the nearest neighbors of that training point, K_θ is the kernel functor specified by *self.kernel*, and $\mathbf{C}_{N^*}$ is the matrix of precomputed coefficients given in Equation (8) of [dunton2022fast].

Parameters

- **Kcross** (ndarray) – A matrix of shape *(batch_count, nn_count)* containing the *1 x nn_count* -shaped cross-covariance vector corresponding to each of the batch elements.
- **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.

This function is designed for use with *MuyGPyS.optimize.chassis.optimize_from_tensors()* and assumes that either *eps* will be passed via a keyword argument or not at all.

Return type
Callable

Returns
A function implementing the posterior mean, where *eps* 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.

This function is designed for use with *MuyGPyS.optimize.chassis.optimize_from_tensors()* and assumes that either *eps* will be passed via a keyword argument or not at all.

Return type
`Callable`

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

`posterior_mean`(*K*, *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 *K* along with *Kcross* and *batch_nn_targets* to predict responses in terms of the posterior mean. Assumes that kernel tensor *K* 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]. For each batch element $\mathbf{x}_i$, we compute

$$\widehat{Y}_{NN}(\mathbf{x}_i \mid X_{N_i}) = K_\theta(\mathbf{x}_i, X_{N_i})(K_\theta(X_{N_i}, X_{N_i}) + \varepsilon I_k)^{-1} Y(X_{N_i}).$$

Here X_{N_i} is the set of nearest neighbors of $\mathbf{x}_i$ in the training data, K_θ is the kernel functor specified by *self.kernel*, εI_k is a diagonal homoscedastic noise matrix whose diagonal is the value of the *self.eps* hyperparameter, and $Y(X_{N_i})$ is the *(nn_count, response_count)* matrix of responses of the nearest neighbors given by the second two dimensions of the *batch_nn_targets* argument.

Parameters

- **K** (`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)* containing the *1 x nn_count* -shaped cross-covariance matrix corresponding to each of the batch elements.
- **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`(*K*, *Kcross*)

Returns the posterior mean 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. For each batch element $\mathbf{x}_i$, we compute

$$Var(\widehat{Y}_{NN}(\mathbf{x}_i \mid X_{N_i})) = K_\theta(\mathbf{x}_i, \mathbf{x}_i) - K_\theta(\mathbf{x}_i, X_{N_i})(K_\theta(X_{N_i}, X_{N_i}) + \varepsilon I_k)^{-1} K_\theta(X_{N_i}, \mathbf{x}_i).$$

Parameters

- **K** (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)* containing the *1 x nn_count* -shaped cross-covariance matrix corresponding to each of the batch elements.

Return type
ndarray

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

set_eps(***eps*)

Reset ε value or bounds.

Uses existing value and bounds as defaults.

Parameters
eps – A hyperparameter dict.

Return type
None

1.2.5 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 = {
...         "eps": {"val": 1e-5},
...         "nu": {"val": 0.67, "bounds": (0.1, 2.5)},
...         "length_scale": {"val": 7.2},
... }
>>> k_kwargs2 = {
...         "eps": {"val": 1e-5},
...         "nu": {"val": 0.38, "bounds": (0.1, 2.5)},
...         "length_scale": {"val": 7.2},
... }
>>> k_args = [k_kwargs1, k_kwargs2]
>>> mmuygps = MMuyGPS("matern", *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:
>>>         K = model.kernel(pairwise_diffs)
>>>         Kcross = model.kernel(crosswise_diffs)
>>>         # do something with K and Kcross...
```

Args

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

apply_new_noise(*new_noise*)

Updates the heteroscedastic noise parameters of a MultivariateMuyGPs model.

Parameters
new_noise – A matrix of shape `(test_count, nn_count, nn_count, response_count)` containing the measurement noise corresponding to the nearest neighbors of each test point and each response.

Returns
A MultivariateMuyGPs model with updated heteroscedastic noise parameters.

fast_coefficients(*pairwise_diffs_fast*, *train_nn_targets_fast*)

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

To form the tensor, we compute

$$\mathbf{C}_{N^*}(i,:,j) = (K_{\hat{\theta}_j}(X_{N^*}, X_{N^*}) + \varepsilon I_k)^{-1} Y(X_{N^*}).$$

Here X_{N^*} is the union of the nearest neighbor of the ith test point and the `nn_count - 1` nearest neighbors of this nearest neighbor, $K_{\hat{\theta}_j}$ is the trained kernel functor corresponding the jth response and specified by `self.models`, εI_k is a diagonal homoscedastic noise matrix whose diagonal is the value of the `self.eps` hyperparameter, and $Y(X_{N^*})$ is the `(train_count, response_count)` matrix of responses corresponding to the training features indexed by N^*.

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 crosswise differences and precomputed coefficient matrix.

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]. For each test point $\mathbf{z}$, we compute

$$\widehat{Y}(\mathbf{z} \mid X) = K_\theta(\mathbf{z}, X_{N^*})\mathbf{C}_{N^*}.$$

Here X_{N^*} is the union of the nearest neighbor of the queried test point $\mathbf{z}$ and the nearest neighbors of that training point, K_θ is the kernel functor specified by `self.kernel`, and $\mathbf{C}_{N^*}$ is the matrix of precomputed coefficients given in Equation (8) of [dunton2022fast].

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.

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 batch element $\mathbf{x}_i$ we compute

$$\widehat{Y}_{NN}(\mathbf{x}_i \mid X_{N_i})_{:,j} = K_\theta^{(j)}(\mathbf{x}_i, X_{N_i})(K_\theta^{(j)}(X_{N_i}, X_{N_i}) + \varepsilon_j I_k)^{-1} Y(X_{N_i})_{:,j}.$$

Here X_{N_i} is the set of nearest neighbors of $\mathbf{x}_i$ in the training data, $K_\theta^{(j)}$ is the kernel functor associated with the jth internal model, corresponding to the jth response dimension, $\varepsilon_j I_k$ is a diagonal homoscedastic noise matrix whose diagonal is the value of the `self.models[j].eps` hyperparameter, and $Y(X_{N_i})_{:,j}$ is the `(batch_count,)` vector of the jth responses of the nearest neighbors given by a slice of the `batch_nn_targets` argument.

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*)

Performs simultaneous posterior variance inference on provided difference tensors.

Return the local posterior variances of each prediction, corresponding to the diagonal elements of a covariance matrix. For each batch element $\mathbf{x}_i$, we compute

$$Var(\widehat{Y}_{NN}(\mathbf{x}_i \mid X_{N_i}))_j = K_\theta^{(j)}(\mathbf{x}_i, \mathbf{x}_i) - K_\theta^{(j)}(\mathbf{x}_i, X_{N_i})(K_\theta^{(j)}(X_{N_i}, X_{N_i}) + \varepsilon I_k)^{-1} K_\theta^{(j)}(X_{N_i}, \mathbf{x}_i).$$

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

Convenience functions for optimizing *MuyGPS* objects

The functions `optimize_from_indices()` and *optimize_from_tensors()* wrap different optimization packages to provide a simple interface to optimize the hyperparameters of *MuyGPS* objects.

Currently, `opt_method="scipy"` wraps `scipy.optimize.opt` multiparameter optimization using L-BFGS-B algorithm using the objective function `MuyGPyS.optimize.objective.loo_crossval()`.

Currently, `opt_method="bayesian"` (also accepts `"bayes"` and `"bayes_opt"`) wraps `bayes_opt.BayesianOptimization`. Unlike the `scipy` version, `BayesianOptimization` can be meaningfully modified by several kwargs. `MuyGPyS` assigns reasonable defaults if no settings are passed by the user. See the BayesianOptimization documentation for details.

`MuyGPyS.optimize.chassis.`**`optimize_from_tensors`**(*muygps*, *batch_targets*, *batch_nn_targets*, *crosswise_diffs*, *pairwise_diffs*, *batch_features=None*, *loss_method='mse'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *sigma_method='analytic'*, *loss_kwargs={}*, *verbose=False*, ***kwargs*)

Find the optimal model using existing difference matrices.

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.chassis import optimize_from_tensors
>>> muygps = optimize_from_tensors(
...         muygps,
...         batch_indices,
...         batch_nn_indices,
...         crosswise_diffs,
...         pairwise_diffs,
...         train_responses,
...         loss_method='mse',
...         obj_method='loo_crossval',
...         opt_method='scipy',
...         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
```

(continues on next page)

(continued from previous page)

```
 success: True
       x: array([0.39963594])
```

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_method** (`str`) – Indicates the loss function to be used.
- **obj_method** (`str`) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.
- **opt_method** (`str`) – Indicates the optimization method to be used. Currently restricted to `"bayesian"` (alternately `"bayes"` or `"bayes_opt"`) and `"scipy"`.
- **sigma_method** (`Optional[str]`) – The optimization method to be employed to learn the `sigma_sq` hyperparameter.
- **loss_kwargs** (`Dict`) – A dictionary of additional keyword arguments to apply to the loss function. Loss function specific.
- **verbose** (`bool`) – If True, print debug messages.
- **kwargs** – Additional keyword arguments to be passed to the wrapper optimizer.

Return type
MuyGPS

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.

`MuyGPyS.optimize.loss.`**cross_entropy_fn**(*predictions*, *targets*)

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.

@NOTE[bwp] I don't remember why we hard-coded eps=1e-6. Might need to revisit.

Parameters

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

Return type
float

Returns
The cross-entropy loss of the prediction.

MuyGPyS.optimize.loss.**get_loss_func**(*loss_method*)

Select a loss function based upon string key.

Currently supports strings "log" or "cross-entropy" for MuyGPyS.optimize.objective.cross_entropy_fn() and "mse" for MuyGPyS.optimize.objective.mse_fn().

Parameters

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

Return type
Callable

Returns
The loss function Callable.

Raises
NotImplementedError – Unrecognized strings will result in an error.

MuyGPyS.optimize.loss.**lool_fn**(*predictions*, *targets*, *variances*, *sigma_sq*)

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

$$l(f(x), y \mid \sigma) = \sum_{i=1}^{b} \sum_{j=1}^{s} \frac{(f(x_i) - y)^2}{\sigma_j} + \log \sigma_j$$

Parameters

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

Return type
float

Returns
The LOOL loss of the prediction.

MuyGPyS.optimize.loss.**lool_fn_unscaled**(*predictions*, *targets*, *variances*)

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 sigma_sq as an argument. The function computes

$$l(f(x), y \mid \sigma) = \sum_{i=1}^{b} \frac{(f(x_i) - y)^2}{\sigma} + \log \sigma$$

Parameters

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

Return type
float

Returns
The LOOL loss of the prediction.

MuyGPyS.optimize.loss.**looph_fn**(*predictions*, *targets*, *variances*, *sigma_sq*, *boundary_scale=1.5*)

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. The function computes

$$l(f(x), y \mid \delta) = \delta^2 \sum_{i=1}^{b} \left(\sqrt{\left(1 + \frac{y_i - f(x_i)}{\sigma_i \delta}\right)^2} - 1 \right) + \log \sigma_i$$

Parameters

- **predictions** (ndarray) – The predicted response of shape (batch_count, response_count).
- **targets** (ndarray) – The expected response of shape (batch_count, response_count).
- **variances** (ndarray) – The unscaled variance of the predicted responses of shape (batch_count, response_count).
- **sigma_sq** (ndarray) – The sigma_sq variance scaling parameter of shape (response_count,).
- **boundary_scale** (float) – The boundary value for the residual beyond which the loss becomes approximately linear. Useful values depend on the scale of the response.

Return type
float

Returns
The sum of pseudo-Huber losses of the predictions.

MuyGPyS.optimize.loss.**mse_fn**(*predictions*, *targets*)

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

Parameters

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

Return type
float

Returns
The mse loss of the prediction.

MuyGPyS.optimize.loss.**pseudo_huber_fn**(*predictions*, *targets*, *boundary_scale=1.5*)

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. Used the form from [wikipedia](https://en.wikipedia.org/wiki/Huber_loss#Pseudo-Huber_loss_function). The function computes

$$l(f(x), y \mid \delta) = \delta^2 \sum_{i=1}^{b} \left(\sqrt{\left(1 + \frac{y_i - f(x_i)}{\delta}\right)^2} - 1 \right)$$

Parameters

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

Return type
float

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_method*, *loss_fn*, *kernel_fn*, *mean_fn*, *var_fn*, *sigma_sq_fn*, *pairwise_diffs*, *crosswise_diffs*, *batch_nn_targets*, *batch_targets*, *batch_features=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.optimize_from_tensors()*, and the format depends on the `opt_method` argument.

Parameters

- **loss_method** (`str`) – Indicates the loss function to be used.
- **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 an epsilon value. The given value is unused if epsilon is fixed.
- **var_fn** (`Callable`) – A function that realizes MuyGPs posterior variance prediction given an epsilon value. The given value is unused if epsilon is fixed.
- **sigma_sq_fn** (`Callable`) – A function that realizes `sigma_sq` optimization given an epsilon value. The given value is unused if epsilon is fixed.
- **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.
- **loss_kwargs** (`Dict`) – A dict listing any additional kwargs to pass to the loss function.

Return type
`Callable`

Returns
A Callable `objective_fn`, whose format depends on `opt_method`.

MuyGPyS.optimize.objective.**make_obj_fn**(*obj_method*, *loss_method*, **args*, ***kwargs*)

Prepare an objective function as a function purely of the hyperparameters to be optimized.

This function is designed for use with *MuyGPyS.optimize.chassis.optimize_from_tensors()*, and the format depends on the `opt_method` argument.

Parameters

- **obj_method** (str) – The name of the objective function to be minimized.
- **opt_method** – The name of the optimization method to be utilized.
- **loss_method** (str) – Indicates the loss function to be used.

Return type
Callable

Returns
A Callable objective_fn, whose format depends on opt_method.

1.3.5 sigma_sq

Convenience functions for optimizing the sigma_sq parameter of *MuyGPS* objects.

Currently only supports an analytic approximation, but will support other methods in the future.

MuyGPyS.optimize.sigma_sq.**mmuygps_analytic_sigma_sq_optim**(*mmuygps*, *pairwise_diffs*, *nn_targets*)

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

We approximate σ^2 by way of averaging over the analytic solution from each local kernel.

$$\sigma^2 = \frac{1}{bk} * \sum_{i \in B} Y_{nn_i}^T K_{nn_i}^{-1} Y_{nn_i}$$

Here Y_{nn_i} and K_{nn_i} are the target and kernel matrices with respect to the nearest neighbor set in scope, where k is the number of nearest neighbors and $b = |B|$ is the number of batch elements considered.

Parameters

- **muygps** – The model to be optimized.
- **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.

Return type
MultivariateMuyGPS

Returns
A new MuyGPs model whose sigma_sq parameter has been optimized.

MuyGPyS.optimize.sigma_sq.**mmuygps_sigma_sq_optim**(*mmuygps*, *pairwise_diffs*, *nn_targets*, *sigma_method='analytic'*)

Optimize the value of the σ^2 scale parameter for each response dimension of a MultivariateMuyGPS object.

The optimization to be applied depends upon the value of sigma_method.

Parameters

- **mmuygps** (*MultivariateMuyGPS*) – The model to be optimized.

- **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.
- **sigma_method** (`Optional[str]`) – The optimization method to apply. Currently only supports `"analytic"` and `None`.

Return type
MultivariateMuyGPS

Returns
A new MultivariateMuyGPs model whose sigma_sq parameter has been optimized.

MuyGPyS.optimize.sigma_sq.**muygps_analytic_sigma_sq_optim**(*muygps*, *pairwise_diffs*, *nn_targets*)

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

We approximate σ^2 by way of averaging over the analytic solution from each local kernel.

$$\sigma^2 = \frac{1}{bk} * \sum_{i \in B} Y_{nn_i}^T K_{nn_i}^{-1} Y_{nn_i}$$

Here Y_{nn_i} and K_{nn_i} are the target and kernel matrices with respect to the nearest neighbor set in scope, where k is the number of nearest neighbors and $b = |B|$ is the number of batch elements considered.

Parameters

- **muygps** (*MuyGPS*) – The model to be optimized.
- **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.

Return type
MuyGPS

Returns
A new MuyGPs model whose sigma_sq parameter has been optimized.

MuyGPyS.optimize.sigma_sq.**muygps_sigma_sq_optim**(*muygps*, *pairwise_diffs*, *nn_targets*, *sigma_method='analytic'*)

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

The optimization to be applied depends upon the value of `sigma_method`.

Parameters

- **muygps** (*MuyGPS*) – The model to be optimized.
- **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.
- **sigma_method** (`Optional[str]`) – The optimization method to apply. Currently only supports `"analytic"` and `None`.

Return type
MuyGPS

Returns
A new MuyGPs model whose sigma_sq parameter has been optimized.

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_method='mse'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *sigma_method='analytic'*, *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.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.examples.regress import do_regress
>>> from MuyGPyS.optimize.objective import mse_fn
>>> train, test = _make_gaussian_data(10000, 1000, 100, 10)
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...         "kern": "rbf",
...         "metric": "F2",
...         "eps": {"val": 1e-5},
...         "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)}
```

(continues on next page)

(continued from previous page)

```
... }
>>> muygps, nbrs_lookup, predictions, variance = do_regress(
...         test['input'],
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> muygps, nbrs_lookup, predictions, variance = do_regress(
...         test['input'],
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> mse = mse_fn(test['output'], 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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed. Currently supports only `"mse"` for regression.
- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to `"bayesian"` and `"scipy"`.

- **sigma_method** (Optional[str]) – The optimization method to be employed to learn the sigma_sq hyperparameter. Currently supports only "analytic" and None. If the value is not None, the returned *MuyGPyS.gp.muygps.MuyGPS* object will possess a sigma_sq member whose value, invoked via muygps.sigma_sq(), is a (response_count,) vector to be used for scaling posterior variances.
- **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_method='mse'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *sigma_method='analytic'*, *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.testing.test_utils import _make_gaussian_data
>>> 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_args = [
...         {
...             "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)}
...             "eps": {"val": 1e-5},
...         },
...         {
```

(continues on next page)

(continued from previous page)

```
...             "length_scale": {"val": 1.5, "bounds": (1e-2, 1e2)}
...             "eps": {"val": 1e-5},
...         },
... ]
>>> mmuygps, nbrs_lookup = make_multivariate_regressor(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         sigma_method="analytic",
...         k_args=k_args,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> mmuygps, nbrs_lookup = make_multivariate_regressor(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         sigma_method="analytic",
...         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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed. Currently supports only `"mse"` for regression.
- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to `"bayesian"` and `"scipy"`.
- **sigma_method** (Optional[str]) – The optimization method to be employed to learn the `sigma_sq` hyperparameter. Currently supports only `"analytic"` and `None`. If

the value is not `None`, the returned `MuyGPyS.gp.muygps.MultivariateMuyGPS` object will possess a `sigma_sq` member whose value, invoked via `mmuygps.sigma_sq()`, is a `(response_count,)` vector to be used for scaling posterior variances.

- **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 epsilon and sigma 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_method='mse'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *sigma_method='analytic'*, *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.testing.test_utils import _make_gaussian_data
>>> 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 = {
...         "kern": "rbf",
...         "metric": "F2",
...         "eps": {"val": 1e-5},
...         "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)}
... }
>>> muygps, nbrs_lookup = make_regressor(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
```

(continues on next page)

(continued from previous page)

```
...         opt_method="bayes",
...         sigma_method="analytic",
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> muygps, nbrs_lookup = make_regressor(
...         train_features,
...         train_responses,
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         sigma_method="analytic",
...         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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed. Currently supports only `"mse"` for regression.
- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to `"bayesian"` and `"scipy"`.
- **sigma_method** (Optional[str]) – The optimization method to be employed to learn the `sigma_sq` hyperparameter. Currently supports only `"analytic"` and `None`. If the value is not `None`, the returned *MuyGPyS.gp.muygps.MuyGPS* object will possess a `sigma_sq` member whose value, invoked via `muygps.sigma_sq()`, is a `(response_count,)` vector to be used for scaling posterior variances.
- **k_kwargs** (Dict) – Parameters for the kernel, possibly including kernel type, distance metric, epsilon and sigma 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_method='lool'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *sigma_method='analytic'*, *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.optimize.objective import mse_fn
>>> train, test = _make_gaussian_data(10000, 1000, 100, 10)
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...         "kern": "rbf",
...         "metric": "F2",
...         "eps": {"val": 1e-5},
...         "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)}
... }
>>> muygps, nbrs_lookup, predictions, precomputed_coefficients_matrix
...         = do_fast_posterior_mean(
...         test['input'],
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument k_kwargs are fixed. Currently supports only "mse" for posterior mean inference.
- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to "loo_crossval".
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to "bayesian" and "scipy".
- **sigma_method** (Optional[str]) – The optimization method to be employed to learn the sigma_sq hyperparameter. Currently supports only "analytic" and None. If the value is not None, the returned *MuyGPyS.gp.muygps.MuyGPS* object will possess a sigma_sq member whose value, invoked via muygps.sigma_sq(), is a (response_count,) vector to be used for scaling posterior variances.
- **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_method='log'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *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.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.examples.classify import do_classify
>>> train, test  = _make_gaussian_dict(10000, 100, 100, 10, categorial=True)
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...     "kernel": RBF(
...         metric=IsotropicDistortion(
...             l2,
...             length_scale=ScalarHyperparameter(1.0, (1e-2, 1e2)),
...         ),
...     ),
...     "eps": HomoscedasticNoise(1e-5),
... )
>>> muygps, nbrs_lookup, surrogate_predictions = do_classify(
...         test['input'],
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="log",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> predicted_labels = np.argmax(surrogate_predictions, axis=1)
>>> true_labels = np.argmax(test['output'], 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_method** (`str`) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by `k_kwargs` are fixed. Currently supports only `"log"` (also known as `"cross_entropy"`) and `"mse"` for classification.
- **obj_method** (`str`) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.

- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to "bayesian" and "scipy".
- **k_kwargs** (Union[Dict, List[Dict], Tuple[Dict, ...]]) – Parameters for the kernel, possibly including kernel type, distance metric, epsilon and sigma 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 "kern" is specified and "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_method='log'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *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.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.examples.classify import make_classifier
>>> train = _make_gaussian_dict(10000, 100, 10, categorial=True)
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...         "kern": "rbf",
...         "metric": "F2",
...         "eps": {"val": 1e-5},
...         "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)},
... }
>>> muygps, nbrs_lookup = make_classifier(
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="log",
```

(continues on next page)

(continued from previous page)

```
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> muygps, nbrs_lookup = make_classifier(
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="log",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed. Currently supports only `"log"` (or `"cross-entropy"`) and `"mse"` for classification.
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to `"bayesian"` and `"scipy"`.
- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.
- **k_kwargs** (Dict) – Parameters for the kernel, possibly including kernel type, distance metric, epsilon and sigma 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_method='mse'*, *obj_method='loo_crossval'*, *opt_method='bayes'*, *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.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.examples.classif import make_multivariate_classifier
>>> train = _make_gaussian_dict(10000, 100, 10, categorial=True)
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_args = [
...         {
...             "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)}
...             "eps": {"val": 1e-5},
...         },
...         {
...             "length_scale": {"val": 1.5, "bounds": (1e-2, 1e2)}
...             "eps": {"val": 1e-5},
...         },
... ]
>>> mmuygps, nbrs_lookup = make_multivariate_classifier(
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_args=k_args,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> mmuygps, nbrs_lookup = make_multivariate_classifier(
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="mse",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_args=k_args,
...         nn_kwargs=nn_kwargs,
```

(continues on next page)

(continued from previous page)

```
...         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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by argument `k_kwargs` are fixed. Currently supports only `"log"` for classification.
- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to `"loo_crossval"`.
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to `"bayesian"` and `"scipy"`.
- **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 epsilon and sigma 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_method='log', obj_method='loo_crossval', opt_method='bayes', 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.testing.test_utils import _make_gaussian_data
>>> from MuyGPyS.examples.regress import do_classify_uq, do_uq
>>> train, test  = _make_gaussian_dict(10000, 100, 100, 10, categorial=True)
>>> nn_kwargs = {"nn_method": "exact", "algorithm": "ball_tree"}
>>> k_kwargs = {
...         "kern": "rbf",
...         "metric": "F2",
...         "eps": {"val": 1e-5},
...         "length_scale": {"val": 1.0, "bounds": (1e-2, 1e2)},
... }
>>> muygps, nbrs_lookup, surrogate_predictions = do_classify(
...         test['input'],
...         train['input'],
...         train['output'],
...         nn_count=30,
...         batch_count=200,
...         loss_method="log",
...         obj_method="loo_crossval",
...         opt_method="bayes",
...         k_kwargs=k_kwargs,
...         nn_kwargs=nn_kwargs,
...         verbose=False,
... )
>>> accuracy, uq = do_uq(surrogate_predictions, test['output'], 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_method** (str) – The loss method to use in hyperparameter optimization. Ignored if all of the parameters specified by `k_kwargs` are fixed. Currently supports only `"log"` (also known as `"cross_entropy"`) and `"mse"` for classification.

- **obj_method** (str) – Indicates the objective function to be minimized. Currently restricted to "loo_crossval".
- **opt_method** (str) – Indicates the optimization method to be used. Currently restricted to "bayesian" and "scipy".
- **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, distance metric, epsilon and sigma 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
```

(continues on next page)

(continued from previous page)

```
>>> 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 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 ε noise parameter.

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.

σ^2 is the only parameter assumed to be a training target by default, and is treated differently from all other hyperparameters. All other training targets must be manually specified in the construction of a MuyGPs_layer object.

Example

```
>>> from MuyGPyS.torch.muygps_layer import MuyGPs_layer
>>> muygps_model = MuyGPS(
...     Matern(
...         nu=ScalarHyperparameter("sample", (0.1, 1)),
...         metric=IsotropicDistortion(
...             l2,
...             length_scale=ScalarHyperparameter(1.0)
...         ),
...     ),
...     eps=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,)
>>> 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 utils import UnivariateSampler, print_results

from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.distortion import IsotropicDistortion, l2
from MuyGPyS.gp.hyperparameter import ScalarHyperparameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.noise import HomoscedasticNoise
from MuyGPyS.gp.tensors import make_predict_tensors
from MuyGPyS.neighbors import NN_Wrapper
from MuyGPyS.optimize.batch import sample_batch
from MuyGPyS.optimize.sigma_sq import muygps_sigma_sq_optim
```

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]:
```
lb = -10.0
ub = 10.0
data_count = 5001
train_step = 10
```

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-5)
```

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

[7]:
```
sim_length_scale = ScalarHyperparameter(1.0)
sim_nu = ScalarHyperparameter(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.

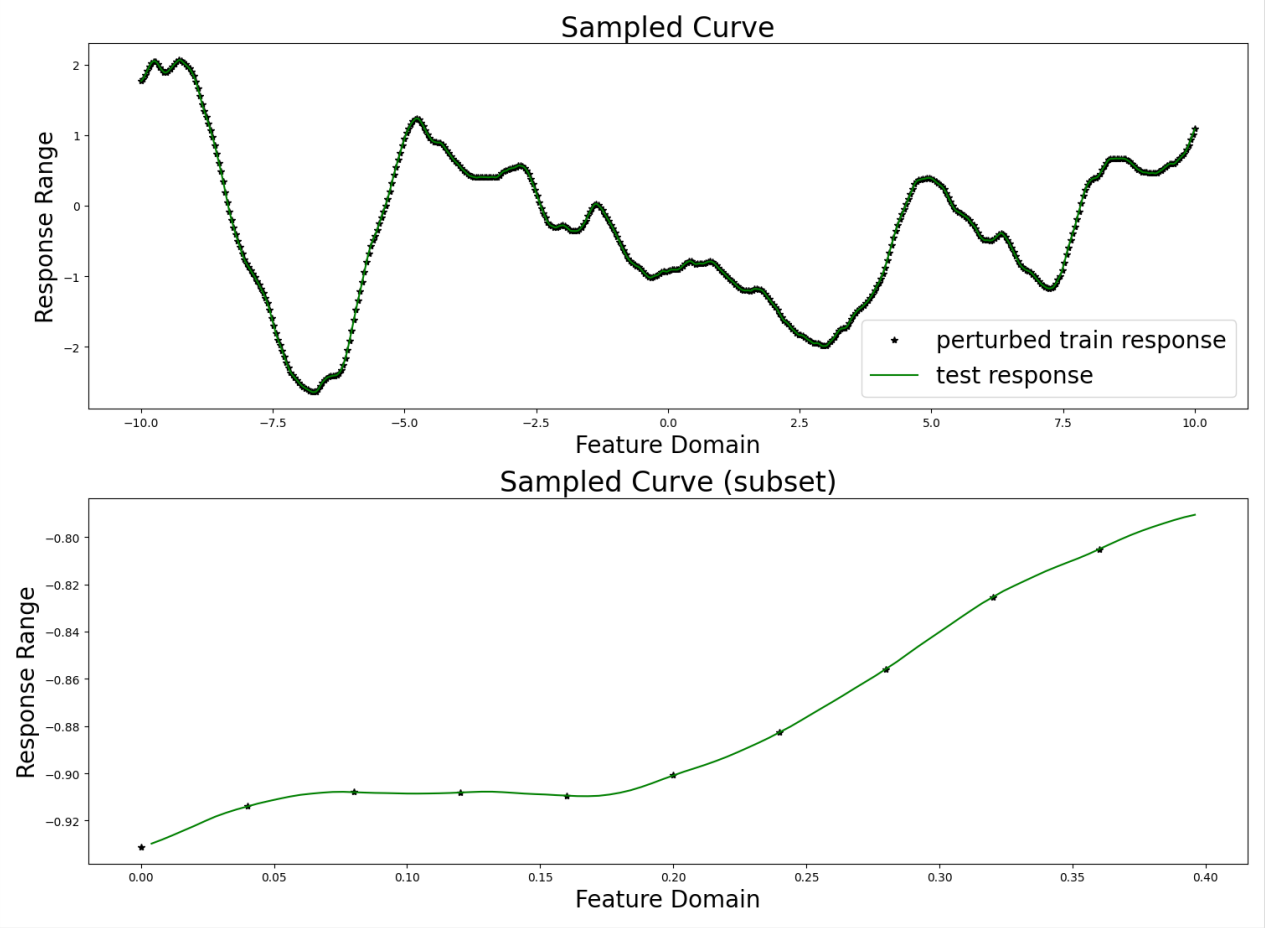
[8]:
```
sampler = UnivariateSampler(
    lb=lb,
    ub=ub,
    data_count=data_count,
    train_step=train_step,
    kernel=Matern(
        nu=sim_nu,
        metric=IsotropicDistortion(
            l2,
            length_scale=sim_length_scale,
        ),
    ),
    eps=nugget_noise,
    measurement_eps=measurement_noise,
)
```

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 and an `eps` noise parameter. The `Matern` object expects a distance function object and a `nu` smoothness parameter. We use an isotropic distance, so `IsotropicDistortion` expects a string 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 `nu` smoothness parameter.

[14]:
```
exp_nu = ScalarHyperparameter("log_sample", (0.1, 5.0))
muygps = MuyGPS(
    kernel=Matern(
        nu=exp_nu,
        metric=IsotropicDistortion(
            l2,
            length_scale=sim_length_scale,
        ),
    ),
    eps=measurement_noise,
)
```

There is one additionally common hyperparameter, the `sigma_sq` scale parameter, that is treated differently than the others. `sigma_sq` cannot be directly set by the user, and always initializes to the value `"unlearned"`. We will show how to train `sigma_sq` below. All hyperparameters other than `sigma_sq` 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, response_count)` coalescing all of the pairwise difference vectors between the same batch of points and their nearest neighbors.

[15]:
```
from MuyGPyS.gp.tensors import crosswise_tensor
batch_crosswise_diffs = 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, response_count)` containing the pairwise differences of each response dimension of the nearest neighbor sets of each sampled batch element.

[16]:
```
from MuyGPyS.gp.tensors import pairwise_tensor
pairwise_diffs = 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.

[17]:
```
Kcross = muygps.kernel(batch_crosswise_diffs)
K = muygps.kernel(pairwise_diffs)
```

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_diffs` and `batch_pairwise_diffs`, the library includes a convenience function `make_train_tensors() <../MuyGPyS/gp/tensors.rst>`__ that bundles these operations.

[19]:
```
from MuyGPyS.gp.tensors import make_train_tensors
(
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    batch_targets,
    batch_nn_targets,
) = make_train_tensors(
    batch_indices,
    batch_nn_indices,
    train_features,
    train_responses,
)
```

We supply a convenient leave-one-out cross-validation utility (`optimize_from_tensors() <../MuyGPyS/gp/optimize.rst>`__) that utilizes these tensors to repeatedly realize kernel tensors during optimization.

This optimization loop wraps a few different batch optimization methods: * `scipy.optimize.minimize <https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.minimize.html>`__ - specifically uses the "L-BFGS-B" algorithm. * `bayes_opt.BayesianOptimization <https://github.com/fmfn/BayesianOptimization>`__ - the `optimize_from_tensors` wrapper only supports batch mode; examine the internals of the function if you would like to use Bayesian optimization interactively.

If we want to use `scipy`-style optimization, we pass the `opt_method="scipy"` kwarg. While it is possible to pass additional kwargs based upon `scipy.optimize.minimize <https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.minimize.html>`__, it is presently unlikely that a user would want to do so.

[20]:
```
from MuyGPyS.optimize import optimize_from_tensors
muygps_scipy = optimize_from_tensors(
    muygps,
    batch_targets,
    batch_nn_targets,
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    loss_method="mse",
    obj_method="loo_crossval",
    opt_method="scipy",
    verbose=True,
)
```

```
parameters to be optimized: ['nu']
bounds: [[0.1 5. ]]
initial x0: [0.49355858]
optimizer results:
      fun: 8.097174300634541e-06
 hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
      jac: array([-6.46382569e-06])
  message: 'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 16
      nit: 7
     njev: 8
   status: 0
  success: True
        x: array([1.29135])
```

Similarly, we can use `bayesian-optimization` style optimization by passing the `opt_method="bayesian"` (alternately `"bayes"` or `"bayes_opt"`) kwarg. There are several additional parameters 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_bayes = optimize_from_tensors(
    muygps,
    batch_targets,
    batch_nn_targets,
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    loss_method="mse",
    obj_method="loo_crossval",
    opt_method="bayesian",
    verbose=True,
    random_state=1,
```

(continues on next page)

(continued from previous page)

```
    init_points=5,
    n_iter=20,
)
```

```
parameters to be optimized: ['nu']
bounds: [[0.1 5. ]]
initial x0: [0.49355858]
|   iter    |  target   |    nu     |
-------------------------------------
| 1         | -6.797e-0 | 0.4936    |
| 2         | -1.027e-0 | 2.143     |
| 3         | -8.11e-05 | 3.63      |
| 4         | -0.004408 | 0.1006    |
| 5         | -7.124e-0 | 1.581     |
| 6         | -1.826e-0 | 0.8191    |
| 7         | -3.023e-0 | 0.6683    |
| 8         | -2.004e-0 | 2.616     |
| 9         | -7.106e-0 | 1.611     |
| 10        | -7.109e-0 | 1.604     |
| 11        | -0.000157 | 4.303     |
| 12        | -0.000242 | 5.0       |
| 13        | -4.259e-0 | 3.122     |
| 14        | -9.146e-0 | 1.169     |
| 15        | -0.000116 | 3.964     |
| 16        | -0.000200 | 4.654     |
| 17        | -2.948e-0 | 2.869     |
| 18        | -5.975e-0 | 3.378     |
| 19        | -1.4e-05  | 2.372     |
| 20        | -3.023e-0 | 0.6682    |
| 21        | -7.951e-0 | 1.911     |
| 22        | -1.184e-0 | 1.001     |
| 23        | -7.761e-0 | 1.351     |
| 24        | -9.814e-0 | 3.798     |
| 25        | -0.000179 | 4.477     |
| 26        | -0.000223 | 4.838     |
=====================================
```

[22]:
```
print(f"scipy.optimize.opt.minimize finds that the optimimal `nu` is {muygps_scipy.
↪kernel.nu()}")
print(f"BayesianOptimization finds that the optimimal `nu` is {muygps_bayes.kernel.nu()}
↪")
```

```
scipy.optimize.opt.minimize finds that the optimimal `nu` is 1.291349997803733
BayesianOptimization finds that the optimimal `nu` is 1.6109672778498811
```

Note here that the returned value for `nu` might be different from the `nu` used by the conventional GP.

As it is a variance scaling parameter that is insensitive to prediction-based optimization, we separately optimize `sigma_sq`. In this case, we invoke *muygps_sigma_sq_optim()*, which approximates `sigma_sq` based upon the mean of the closed-form `sigma_sq` 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.

This is usually performed after optimizing other hyperparameters.

[23]:
```
scipy_K = muygps_scipy.kernel(batch_pairwise_diffs)
muygps_scipy = muygps_sigma_sq_optim(muygps_scipy, batch_pairwise_diffs, batch_nn_
↪targets, sigma_method="analytic")
bayes_K = muygps_bayes.kernel(batch_pairwise_diffs)
muygps_bayes = muygps_sigma_sq_optim(muygps_bayes, batch_pairwise_diffs, batch_nn_
↪targets, sigma_method="analytic")
print(f"scipy-optimized sigma_sq: {muygps_scipy.sigma_sq()}")
print(f"BayesianOptimization-optimized sigma_sq: {muygps_bayes.sigma_sq()}")
```

```
scipy-optimized sigma_sq: [0.16176324]
BayesianOptimization-optimized sigma_sq: [0.26936798]
```

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 `K` 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`.

[24]:
```
test_count, _ = test_features.shape
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

[25]:
```
(
    test_crosswise_diffs,
    test_pairwise_diffs,
    test_nn_targets,
) = make_predict_tensors(
    indices,
    test_nn_indices,
    test_features,
    train_features,
    train_responses,
)
```

We create the kernel tensors for both the scipy and bayes-opt optimized models.

[26]:
```
scipy_Kcross = muygps_scipy.kernel(test_crosswise_diffs)
scipy_K = muygps_scipy.kernel(test_pairwise_diffs)

bayes_Kcross = muygps_bayes.kernel(test_crosswise_diffs)
bayes_K = muygps_bayes.kernel(test_pairwise_diffs)
```

Finally, we use the `MuyGPS.posterior_mean()` and `MuyGPS.posterior_variance()` functions to find the posterior means and variances associated with each training prediction for each model.

[27]:
```
scipy_predictions = muygps_scipy.posterior_mean(
    scipy_K, scipy_Kcross, test_nn_targets
)
scipy_variances = muygps_scipy.posterior_variance(
    scipy_K, scipy_Kcross
)

bayes_predictions = muygps_bayes.posterior_mean(
    bayes_K, bayes_Kcross, test_nn_targets
)
bayes_variances = muygps_bayes.posterior_variance(
    bayes_K, bayes_Kcross
)
```

We here evaluate our prediction performance in terms of RMSE, mean diagonal posterior variance, the mean 95% confidence interval size, and the coverage, which ideally should be near 95%.

The 95% confidence interval are straightforward to compute.

[28]:
```
scipy_confidence_intervals = np.sqrt(scipy_variances) * 1.96
bayes_confidence_intervals = np.sqrt(bayes_variances) * 1.96
```

We compute confidence intervals as the proportion of posterior means that differ from the true response by no more than the confidence interval.

[29]:
```
scipy_coverage = (
    np.count_nonzero(
        np.abs(test_responses - scipy_predictions) < scipy_confidence_intervals
    ) / test_count
)
bayes_coverage = (
    np.count_nonzero(
        np.abs(test_responses - bayes_predictions) < bayes_confidence_intervals
    ) / test_count
)
```

Finally, we print the results using some throwaway convenience functions.

[30]:
```
print_results("scipy", test_responses, scipy_predictions, scipy_variances, scipy_
↪confidence_intervals, scipy_coverage)
print_results("bayes", test_responses, bayes_predictions, bayes_variances, bayes_
↪confidence_intervals, bayes_coverage)
```

```
scipy results:
        RMSE: 0.0005523046981276159
        mean diagonal variance: 7.2036827724115715e-06
        mean confidence interval size: 0.01024846652495427
        coverage: 1.0
bayes results:
        RMSE: 0.0006262325772056444
        mean diagonal variance: 3.135317993317895e-06
        mean confidence interval size: 0.006921679511057385
```

(continues on next page)

(continued from previous page)

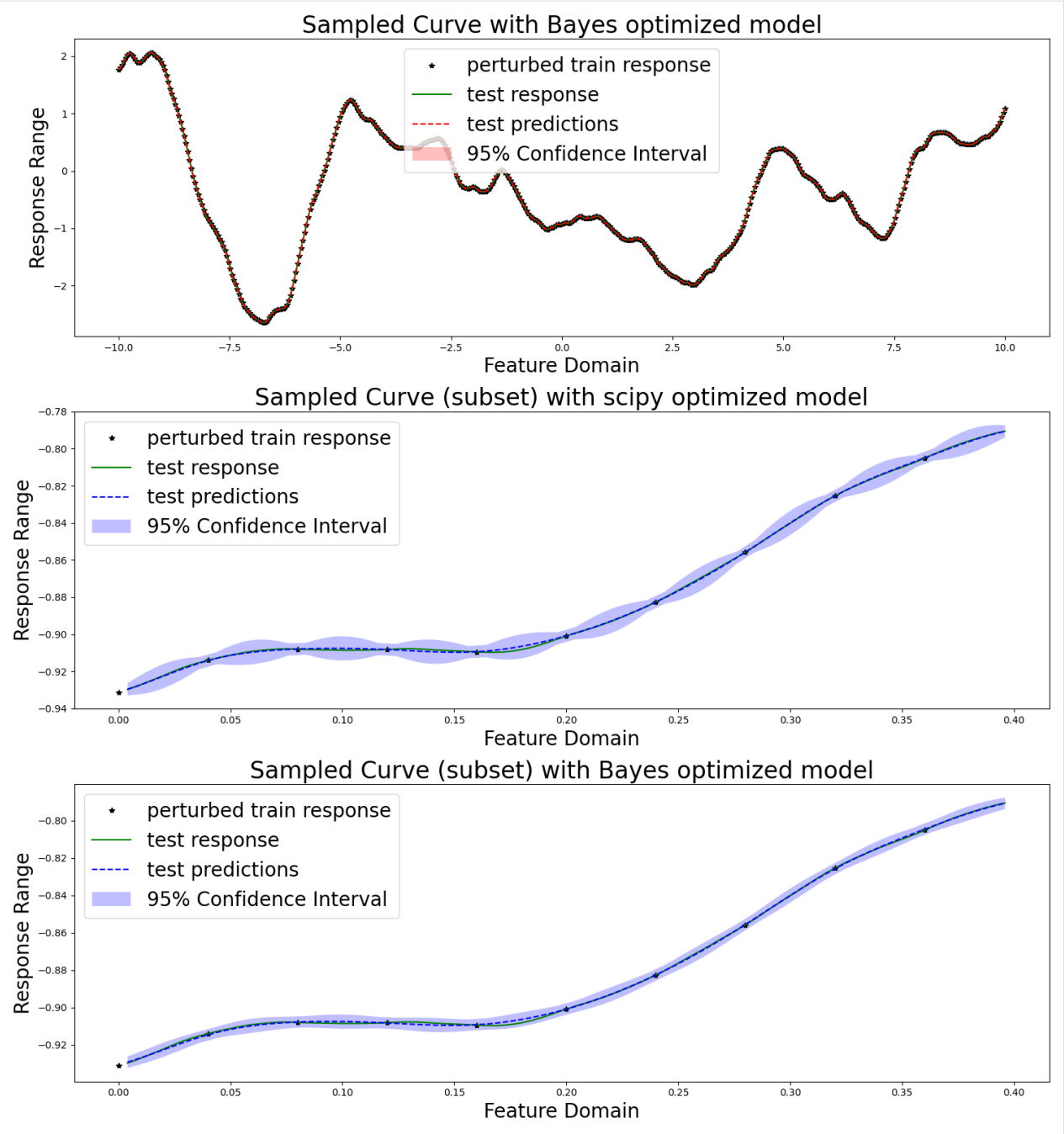
```
        coverage: 1.0
```

These regression examples return 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 `sigma_sq` scaling parameters, of which there will be one scalar associated with each dimension of the response. The kwarg `apply_sigma_sq=True` indicates that this scaling will be performed automatically. This is the default behavior, but will be skipped if `sigma_sq == "unlearned"`.

For a univariate resonse whose variance is obtained with `apply_sigma_sq=False`, the scaled predicted variance is equivalent to multiplying the predicted variances by `muygps.sigma_sq()`.

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.

[31]:
```
sampler.plot_results(scipy_predictions, scipy_confidence_intervals, bayes_predictions,␣
↪bayes_confidence_intervals)
```



[ ]:

1.7 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]:
```
from MuyGPyS.gp.distortion import l2

import numpy as np
import torch
import torchvision
import os
from torch.nn.functional import one_hot
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.

[4]:
```
trans = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5,),(1.0,)),
    ]
)
train_set = torchvision.datasets.MNIST(
    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. In the interest of reducing runtime, we will use vectorized images as our features in this dataset.

[5]:
```
num_classes = 10
num_train_samples = 60000
num_test_samples = 10000
num_pixels = 784
```

We will collect 60,000 training samples and 10,000 test samples. We vectorize the images and one-hot encode the class labels.

[6]:
```
train_features = torch.zeros((num_train_samples,num_pixels))
train_responses = torch.zeros((num_train_samples,num_classes))

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

test_features = torch.zeros((num_test_samples,num_pixels))
test_responses = torch.zeros((num_test_samples,num_classes))

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

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.

[7]:
```
from torch import nn
import random
from torch.optim.lr_scheduler import ExponentialLR
torch.autograd.set_detect_anomaly(True)
np.random.seed(0)
test_count, _ = test_features.shape
train_count, _ = train_features.shape


from MuyGPyS.neighbors import NN_Wrapper
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.

[8]:
```

#We will make use of batching in our hyperparameter training
from MuyGPyS.optimize.batch import sample_batch
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()
```

(continues on next page)

(continued from previous page)

```
    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 `nu` or 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.

[9]:
```
from MuyGPyS.torch import MuyGPs_layer
print('Building Stochastic Variational Deep Kernel MuyGPs model')

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
        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,
        )

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

```
Building Stochastic Variational Deep Kernel MuyGPs model
```

1.7.1 Training a Deep Kernel MuyGPs Model

We instantiate a `SVDKMuyGPs` model with initial guess hyperparameters. We fix a Matérn kernel smoothness parameter of 0.5 and a Guassian homoscedastic noise prior variance of `1e-6`.

[10]:
```
from MuyGPyS.gp import MuyGPS
from MuyGPyS.gp.noise import HomoscedasticNoise
from MuyGPyS.gp.hyperparameter import ScalarHyperparameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.distortion import IsotropicDistortion

model_nu = 0.5
model_length_scale = 1.0
measurement_eps = 1e-6

muygps_model = MuyGPS(
    kernel=Matern(
        nu=ScalarHyperparameter(model_nu),
        metric=IsotropicDistortion(l2,
            length_scale=ScalarHyperparameter(model_length_scale)
        ),
    ),
    eps=HomoscedasticNoise(measurement_eps),
)


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`.

[11]:
```
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.

[12]:
```
ce_loss = nn.CrossEntropyLoss()
# mse_loss = nn.MSELoss()
# l1_loss = nn.L1Loss()
# bce_loss = nn.BCELoss()
```

We construct a standard PyTorch training loop function.

[13]:
```
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

[14]:
```
nbrs_lookup, model_trained = train(nbrs_lookup)
model_trained.eval()
```

```
Iter 1/10 - Loss: 1.5154695510864258
Iter 2/10 - Loss: 1.4770054817199707
Iter 3/10 - Loss: 1.441899299621582
Iter 4/10 - Loss: 1.4263132810592651
Iter 5/10 - Loss: 1.4199936389923096
Iter 6/10 - Loss: 1.40185546875
Iter 7/10 - Loss: 1.3923559188842773
Iter 8/10 - Loss: 1.380581259727478
Iter 9/10 - Loss: 1.372625470161438
Iter 10/10 - Loss: 1.3624086380004883
```

[14]:
```
SVDKMuyGPs(
  (embedding): Sequential(
    (0): Linear(in_features=784, out_features=400, bias=True)
    (1): ReLU()
    (2): Linear(in_features=400, out_features=200, bias=True)
    (3): ReLU()
    (4): Linear(in_features=200, out_features=100, bias=True)
  )
  (GP_layer): MuyGPs_layer()
)
```

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

[15]:
```
from MuyGPyS.examples.muygps_torch import predict_model
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 Low-Level Torch Implementation:")
print(
    (
        torch.sum(
            torch.argmax(predictions,dim=1) == torch.argmax(test_responses,dim=1)
        ) / 10000
    ).numpy()
)
```

```
MNIST Prediction Accuracy Using Low-Level Torch Implementation:
0.9398
```

1.7.2 Training a Deep Kernel MuyGPs Model Using Our Example API Function

Similar to our one-line regression tutorial API, we support a one-line Deep MuyGPs regression API. This snippet performs the same work as above with a singular function execution.

[16]:
```
#Import high-level API function train_deep_kernel_muygps
from MuyGPyS.examples.muygps_torch import train_deep_kernel_muygps

model_nu = 0.5
model_length_scale = 1.0
measurement_eps = 1e-6

muygps_model = MuyGPS(
    kernel=Matern(
        nu=ScalarHyperparameter(model_nu),
        metric=IsotropicDistortion(l2,
            length_scale=ScalarHyperparameter(model_length_scale)
        ),
```

(continues on next page)

(continued from previous page)

```
    ),
    eps=HomoscedasticNoise(measurement_eps),
)

#Use leave-one-out-likelihood loss function to train model
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)

nbrs_lookup, model_trained = train_deep_kernel_muygps(
    model=model,
    train_features=train_features,
    train_responses=train_responses,
    batch_indices=batch_indices,
    nbrs_lookup=nbrs_lookup,
    training_iterations=10,
    optimizer_method=torch.optim.Adam,
    learning_rate=1e-2,
    scheduler_decay=0.97,
    loss_function="ce",
    update_frequency=1,
    verbose=True,
)

model_trained.eval()
```

```
Iter 1/10 - Loss: 1.5164387226
Iter 2/10 - Loss: 1.4815564156
Iter 3/10 - Loss: 1.4386521578
Iter 4/10 - Loss: 1.4218910933
Iter 5/10 - Loss: 1.4173457623
Iter 6/10 - Loss: 1.4027299881
Iter 7/10 - Loss: 1.3904489279
Iter 8/10 - Loss: 1.3764212132
Iter 9/10 - Loss: 1.3704509735
Iter 10/10 - Loss: 1.3603465557
```

[16]:
```
SVDKMuyGPs(
  (embedding): Sequential(
    (0): Linear(in_features=784, out_features=400, bias=True)
    (1): ReLU()
    (2): Linear(in_features=400, out_features=200, bias=True)
    (3): ReLU()
    (4): Linear(in_features=200, out_features=100, bias=True)
  )
  (GP_layer): MuyGPs_layer()
)
```

We similarly report our prediction performance on the test responses using this trained model.

[17]:
```
from MuyGPyS.examples.muygps_torch import predict_model
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 High-Level Training API:")
print(
    (
        torch.sum(
            torch.argmax(predictions,dim=1) == torch.argmax(test_responses,dim=1)
        ) / 10000
    ).numpy()
)
```

```
MNIST Prediction Accuracy Using High-Level Training API:
0.9348
```

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.

1.8 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

from utils import UnivariateSampler

from MuyGPyS._test.gp import benchmark_sample, BenchmarkGP
from MuyGPyS.gp.distortion import IsotropicDistortion, NullDistortion, l2
from MuyGPyS.gp.hyperparameter import ScalarHyperparameter
from MuyGPyS.gp.kernels import Matern
from MuyGPyS.gp.noise import HomoscedasticNoise

np.random.seed(0)
```

(continues on next page)

(continued from previous page)

```
lb = -10.0
ub = 10.0
data_count = 5001
train_step = 10
nugget_var = 1e-14
fixed_length_scale = 1.0
measurement_eps = 1e-3
sampler = UnivariateSampler(
    lb=lb,
    ub=ub,
    data_count=data_count,
    train_step=train_step,
    kernel=Matern(
        nu=ScalarHyperparameter(0.5),
        metric=IsotropicDistortion(
            l2,
            length_scale=ScalarHyperparameter(fixed_length_scale),
        ),
    ),
    eps=HomoscedasticNoise(nugget_var),
    measurement_eps=HomoscedasticNoise(measurement_eps),
)
train_features, test_features = sampler.features()
test_count, _ = test_features.shape
train_count, _ = train_features.shape

train_responses, test_responses = sampler.sample()

sampler.plot_sample()

from MuyGPyS.neighbors import NN_Wrapper
nn_count = 10
nbrs_lookup = NN_Wrapper(train_features, nn_count, nn_method="exact",algorithm="ball_tree
↪")

from MuyGPyS.optimize.batch import sample_batch
batch_count = train_count
batch_indices, batch_nn_indices = sample_batch(
    nbrs_lookup, batch_count, train_count
)

from MuyGPyS.gp import MuyGPS
muygps = MuyGPS(
    kernel=Matern(
        nu=ScalarHyperparameter(0.5),
        metric=IsotropicDistortion(
            metric=l2,
            length_scale=ScalarHyperparameter(
                "log_sample", (0.1, 5.0))
        ),
    ),
```

(continues on next page)

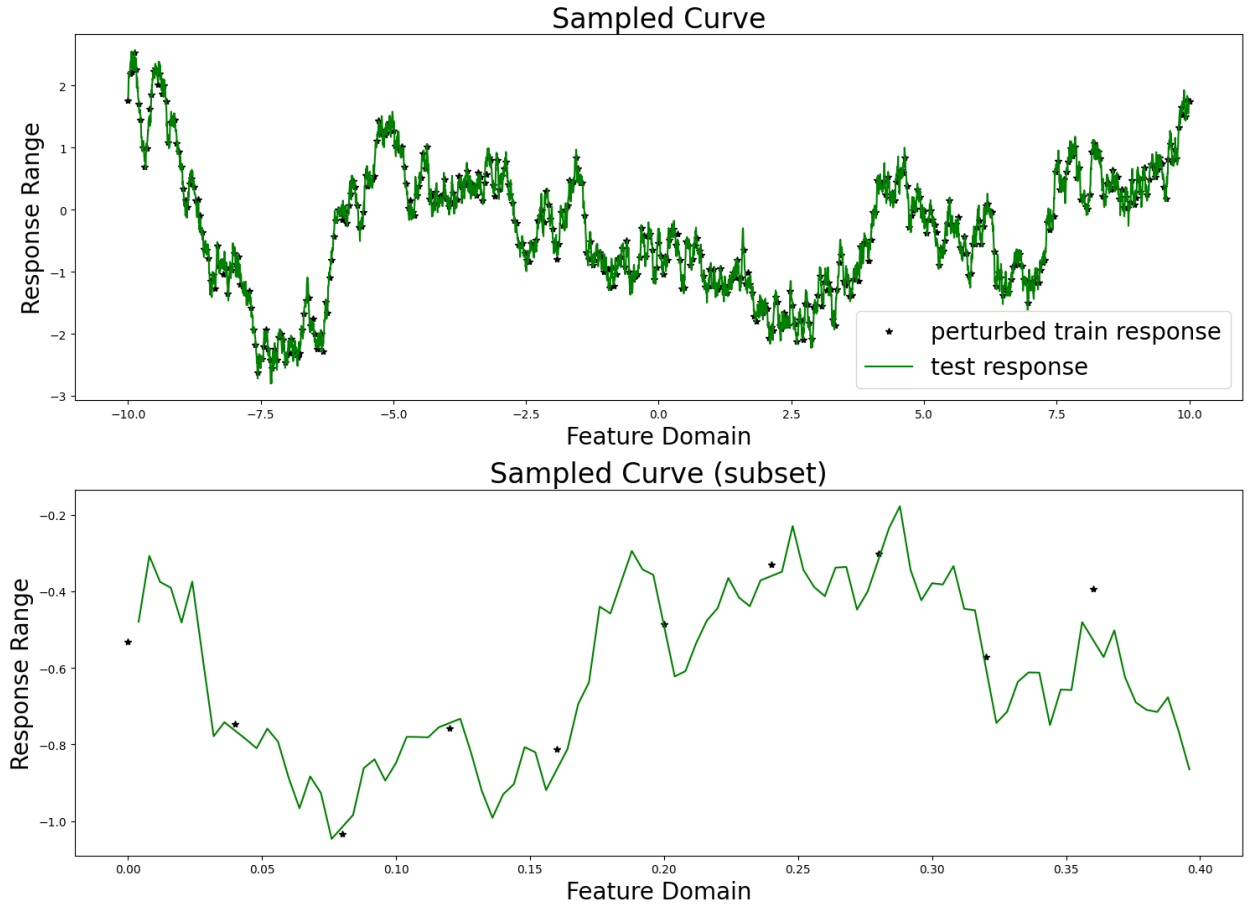
(continued from previous page)

```
    eps=HomoscedasticNoise(measurement_eps),
)

from MuyGPyS.gp.tensors import crosswise_tensor
batch_crosswise_diffs = crosswise_tensor(
    train_features,
    train_features,
    batch_indices,
    batch_nn_indices,
)

from MuyGPyS.gp.tensors import pairwise_tensor
pairwise_diffs = pairwise_tensor(
    train_features, batch_nn_indices
)

Kcross = muygps.kernel(batch_crosswise_diffs)
K = muygps.kernel(pairwise_diffs)

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

from MuyGPyS.gp.tensors import make_train_tensors
(
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    batch_targets,
    batch_nn_targets,
) = make_train_tensors(
    batch_indices,
    batch_nn_indices,
    train_features,
    train_responses,
)

from MuyGPyS.optimize import optimize_from_tensors

muygps = optimize_from_tensors(
    muygps,
    batch_targets,
    batch_nn_targets,
    batch_crosswise_diffs,
    batch_pairwise_diffs,
    loss_method="lool",
    obj_method="loo_crossval",
    opt_method="bayesian",
    verbose=False,
    random_state=1,
    init_points=5,
    n_iter=2,
)
```

(continues on next page)

(continued from previous page)

```
from MuyGPyS.optimize.sigma_sq import muygps_sigma_sq_optim

K = muygps.kernel(batch_pairwise_diffs)
muygps = muygps_sigma_sq_optim(muygps, batch_pairwise_diffs, batch_nn_targets, sigma_
↪method="analytic")
```



1.8.1 Fast Prediction

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

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_diffs` among each nearest neighbor and the `train_nn_targets_fast` or responses of the nearest neighbors in each set. We then create the`K` covariance tensor and form the precomputed coefficients matrix. We then pass the precomputed coefficients matrix, the updated `nn_indices` matrix, and the closest neighbor of each test point to *MuyGPS.fast_posterior_mean()* in order to obtain our predictions.

[3]:
```
from MuyGPyS.gp.tensors import make_fast_predict_tensors, fast_nn_update, make_predict_
↪tensors
nn_indices,_ = nbrs_lookup.get_nns(train_features)
nn_indices = nn_indices.astype(int)
```

[4]:
```
from MuyGPyS.gp.tensors import make_predict_tensors
nn_indices = fast_nn_update(nn_indices)

#find the closest training point to each test point, and its corresponding nearest␣
↪neighbor set

test_neighbors, _ = nbrs_lookup.get_nns(test_features)
closest_neighbor = test_neighbors[:, 0]
closest_set = nn_indices[closest_neighbor, :].astype(int)

#make crosswise distances tensor for prediction
(
    crosswise_diffs,
    pairwise_diffs,
    nn_targets,
) = make_predict_tensors(
    np.arange(test_count),
    closest_set,
    test_features,
    train_features,
    train_responses,
)

K = muygps.kernel(pairwise_diffs)

precomputed_coefficients_matrix = muygps.fast_coefficients(
K,
nn_targets)



#perform GP regression

Kcross = muygps.kernel(crosswise_diffs)
fast_predictions = muygps.fast_posterior_mean(
    Kcross,
    precomputed_coefficients_matrix[closest_neighbor,:,:])
```

1.8.2 Regular 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 workflow uses the compact tensor-making function *make_predict_tensors()* to succinctly create tensors defining the `pairwise_diffs` among each nearest neighbor set, the `crosswise_diffs` 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 `K` covariance tensor and pass them to *MuyGPS.posterior_mean()* in order to obtain our predictions.

[5]:
```
from MuyGPyS.gp.tensors import make_predict_tensors

# make the indices
```

(continues on next page)

(continued from previous page)

```
test_count, _ = test_features.shape
indices = np.arange(test_count)
nn_indices, _ = nbrs_lookup.get_nns(test_features)

# make difference and target tensors
(
    crosswise_diffs,
    pairwise_diffs,
    nn_targets,
) = make_predict_tensors(
    indices,
    nn_indices,
    test_features,
    train_features,
    train_responses,
)

# Make the kernel
Kcross = muygps.kernel(crosswise_diffs)
K = muygps.kernel(pairwise_diffs)

# perform Gaussian process regression

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

1.8.3 Timing Experiment

We compare the prediction time of a regular posterior mean workflow to that of the fast posterior mean workflow. In the regular posterior mean workflow we compute the sum of the time it takes to identify the nearest neighbors of the test features, the time it takes to form the relevant kernel tensors, and the time to 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, the coefficient lookup in the precomputed coefficient matrix, and the dot product to form posterior means.

[6]:
```
from MuyGPyS.optimize.loss import mse_fn
import timeit


test_count, _ = test_features.shape
indices = np.arange(test_count)



def timing_posterior_mean():
    nn_indices, _ = nbrs_lookup.get_nns(test_features)
    (
        crosswise_diffs,
        pairwise_diffs,
        nn_targets,
    ) = make_predict_tensors(
```

(continues on next page)

(continued from previous page)

```
        indices,
        nn_indices,
        test_features,
        train_features,
        train_responses,
    )

    Kcross = muygps.kernel(crosswise_diffs)
    K = muygps.kernel(pairwise_diffs)
    predictions = muygps.posterior_mean(
        K, Kcross, nn_targets
    )

print(f"regular RMSE:")
print(f"\tRMSE: {np.sqrt(mse_fn(predictions, test_responses))}")
print("regular prediction time:")
%timeit timing_posterior_mean()



nn_indices = fast_nn_update(nn_indices)
def timing_fast_posterior_mean():
    test_neighbors, _ = nbrs_lookup.get_nns(test_features)
    closest_neighbor = test_neighbors[:, 0]
    closest_set = nn_indices[closest_neighbor, :].astype(int)

    #make crosswise distances tensor
    (
        crosswise_diffs,
        pairwise_diffs,
        nn_targets,
    ) = make_predict_tensors(
        np.arange(test_count),
        closest_set,
        test_features,
        train_features,
        train_responses,
    )



    Kcross = muygps.kernel(crosswise_diffs)
    fast_predictions = muygps.fast_posterior_mean(
        Kcross,
        precomputed_coefficients_matrix[closest_neighbor,:,:])



print(f"fast prediction RMSE:")
print(f"\tRMSE: {np.sqrt(mse_fn(fast_predictions, test_responses))}")
print("fast prediction time:")
%timeit timing_fast_posterior_mean()
```

(continues on next page)

(continued from previous page)

```
regular RMSE:
        RMSE: 0.1206710068099257
regular prediction time:
64.4 ms ± 731 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
fast prediction RMSE:
        RMSE: 1.658329619936535
fast prediction time:
14.7 ms ± 82.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

1.8.4 Results

We achieve roughly two orders of magnitude speedup using the fast prediction acceleration. The improvement is even more dramatic when the methods are implemented in JAX.

1.9 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

K

L

M

N

O

P

R

S

T

U