vittles Documentation

Release 0.173

Contributors

Dec 08, 2020

Contents

"Variational inference tools to leverage estimator sensitivity" or `vittles`.

This is a library (very much still in development) intended to make sensitivity analysis easier for optimization problems.

The purpose is to automate much of the boilerplate required to perform optimization and sensitivity analysis for statistical problems that employ optimization or estimating equations.

For additional background and motivation, see the following papers:

Covariances, Robustness, and Variational Bayes
Ryan Giordano, Tamara Broderick, Michael I. Jordan
https://arxiv.org/abs/1709.02536

A Swiss Army Infinitesimal Jackknife
Ryan Giordano, Will Stephenson, Runjing Liu, Michael I. Jordan, Tamara Broderick
https://arxiv.org/abs/1806.00550

Evaluating Sensitivity to the Stick Breaking Prior in Bayesian Nonparametrics
Runjing Liu, Ryan Giordano, Michael I. Jordan, Tamara Broderick
https://arxiv.org/abs/1810.06587

CHAPTER 1

Installation

To get the latest released version, just use pip:

```
$ pip install vittles
```

However, `vittles` is under active development, and you may want to install a newer version from github:

```
$ pip install git+git://github.com/rgiordan/vittles
```

CHAPTER 2

API

2.1 Sensitivity Functions

2.1.1 Hyperparameter sensitivity linear approximation

class vittles.sensitivity_lib.**HyperparameterSensitivityLinearApproximation**(*objective_fun*, *opt_par_value*, *hyper_par_value*, *validate_optimum=False*, *hessian_at_opt=None*, *cross_hess_at_opt=No*
hyper_par_objective_fun
grad_tol=1e-08)

Linearly approximate dependence of an optimum on a hyperparameter.

Suppose we have an optimization problem in which the objective depends on a hyperparameter:

$$\hat{\theta} = \text{argmin}_\theta f(\theta, \lambda).$$

The optimal parameter, $\hat{\theta}$, is a function of λ through the optimization problem. In general, this dependence is complex and nonlinear. To approximate this dependence, this class uses the linear approximation:

$$\hat{\theta}(\lambda) \approx \hat{\theta}(\lambda_0) + \frac{d\hat{\theta}}{d\lambda}|_{\lambda_0}(\lambda - \lambda_0).$$

In terms of the arguments to this function, θ corresponds to `opt_par`, λ corresponds to `hyper_par`, and f corresponds to `objective_fun`.

Methods

set_base_values:	Set the base values, λ_0 and $\theta_0 := \hat{\theta}(\lambda_0)$, at which the linear approximation is evaluated.
get_dopt_dhyper:	Return the Jacobian matrix $\frac{d\hat{\theta}}{d\lambda}\|_{\lambda_0}$ in flattened space.
get_hessian_at_opt:	Return the Hessian of the objective function in the flattened space.
predict_opt_par_from_hyper_par:	Use the linear approximation to predict the value of `opt_par` from a value of `hyper_par`.

__init__(*objective_fun*, *opt_par_value*, *hyper_par_value*, *validate_optimum=False*, *hessian_at_opt=None*, *cross_hess_at_opt=None*, *hyper_par_objective_fun=None*, *grad_tol=1e-08*)

Parameters

objective_fun [*callable*] The objective function taking two positional arguments, - `opt_par`: The parameter to be optimized (*numpy.ndarray* (N,)) - `hyper_par`: A hyperparameter (*numpy.ndarray* (N,)) and returning a real value to be minimized.

opt_par_value [*numpy.ndarray* (N,)] The value of `opt_par` at which `objective_fun` is optimized for the given value of `hyper_par_value`.

hyper_par_value [*numpy.ndarray* (M,)] The value of `hyper_par` at which `opt_par` optimizes `objective_fun`.

validate_optimum [*bool*, optional] When setting the values of `opt_par` and `hyper_par`, check that `opt_par` is, in fact, a critical point of `objective_fun`.

hessian_at_opt [*numpy.ndarray* (N,N), optional] The Hessian of `objective_fun` at the optimum. If not specified, it is calculated using automatic differentiation.

cross_hess_at_opt [*numpy.ndarray* (N, M)] Optional. The second derivative of the objective with respect to `input_val` then `hyper_val`. If not specified it is calculated at initialization.

hyper_par_objective_fun [*callable*, optional] The part of `objective_fun` depending on both `opt_par` and `hyper_par`. The arguments must be the same as `objective_fun`: - `opt_par`: The parameter to be optimized (*numpy.ndarray* (N,)) - `hyper_par`: A hyperparameter (*numpy.ndarray* (N,)) This can be useful if only a small part of the objective function depends on both `opt_par` and `hyper_par`. If not specified, `objective_fun` is used.

grad_tol [*float*, optional] The tolerance used to check that the gradient is approximately zero at the optimum.

get_opt_par_function()
Return a differentiable function returning the optimal value.

predict_opt_par_from_hyper_par(*new_hyper_par_value*)
Predict `opt_par` using the linear approximation.

Parameters

new_hyper_par_value: 'numpy.ndarray' (M,) The value of `hyper_par` at which to approximate `opt_par`.

2.1.2 Hyperparameter sensitivity Taylor series approximation

class vittles.sensitivity_lib.**ParametricSensitivityTaylorExpansion**(*estimating_equation*, *input_val0*, *hyper_val0*, *order*, *hess_solver*, *forward_mode=True*, *max_input_order=None*, *max_hyper_order=None*, *force=False*)

Evaluate the Taylor series of an optimum on a hyperparameter.

This is a class for computing the Taylor series of eta(eps) = argmax_eta objective(eta, eps) using forward-mode automatic differentation.

Note: This class is experimental and should be used with caution.

__init__(*estimating_equation*, *input_val0*, *hyper_val0*, *order*, *hess_solver*, *forward_mode=True*, *max_input_order=None*, *max_hyper_order=None*, *force=False*)

Parameters

estimating_equation [*callable*] A vector-valued function function of two arguments, (*input*, *output*), where the length of the vector is the same as the length of *input*, and which is (approximately) the zero vector when evaluated at (*input_val0*, *hyper_val0*).

input_val0 [*numpy.ndarray* (N,)] The value of `input_par` at the optimum.

hyper_val0 [*numpy.ndarray* (M,)] The value of `hyper_par` at which `input_val0` was found.

order [*int*] The maximum order of the Taylor series to be calculated.

hess_solver [*function*] A function that takes a single argument, *v*, and returns

$$\frac{\partial G}{\partial \eta}^{-1} v,$$

where $G(\eta, \epsilon)$ is the estimating equation, and the partial derivative is evaluated at $(\eta, \epsilon) =$ (*input_val0*, *hyper_val0*).

forward_mode [*bool*] Optional. If *True* (the default), use forward-mode automatic differentiation. Otherwise, use reverse-mode.

max_input_order [*int*] Optional. The maximum number of nonzero partial derivatives of the objective function gradient with respect to the input parameter. If *None*, calculate partial derivatives of all orders.

max_hyper_order [*int*] Optional. The maximum number of nonzero partial derivatives of the objective function gradient with respect to the hyperparameter. If *None*, calculate partial derivatives of all orders.

force: 'bool' Optional. If *True*, force the instantiation of potentially expensive reverse mode derivative arrays. Default is *False*.

evaluate_input_derivs(*dhyper*, *max_order=None*)
Return a list of the derivatives dkinput / dhyperk dhyper^k

evaluate_taylor_series(*new_hyper_val*, *add_offset=True*, *max_order=None*, *sum_terms=True*)
Evaluate the derivative `d^k input / d hyper^k` in the direction dhyper.

Parameters

new_hyper_val: 'numpy.ndarray' (N,) The new hyperparameter value at which to evaluate the Taylor series.

add_offset: 'bool' Optional. Whether to add the initial constant input_val0 to the Taylor series.

max_order: 'int' Optional. The order of the Taylor series. Defaults to the `order` argument to `__init__`.

sum_terms: 'bool' If `True`, add the terms in the Taylor series. If `False`, return the terms as a list.

Returns

The Taylor series approximation to `input_val(new_hyper_val)` if `add_offset` is `True`, or to `input_val(new_hyper_val) - input_val0` if `False`. If `sum_terms` is `True`, then a vector of the same length as `input_val` is returned. Otherwise, an array of shape `max_order + 1, len(input_val)` is returned containing the terms of the Taylor series approximation.

evaluate_taylor_series_terms(*new_hyper_val*, *add_offset=True*, *max_order=None*)
Return the terms in a Taylor series approximation.

classmethod optimization_objective(*objective_function*, *input_val0*, *hyper_val0*, *order*, *hess0=None*, *forward_mode=True*, *max_input_order=None*, *max_hyper_order=None*, *force=False*)

Parameters

objective_function [*callable*] The optimization objective as a function of two arguments (eta, eps), where eta is the parameter that is optimized and eps is a hyperparameter.

hess0 [*numpy.ndarray* (N, N)] Optional. The Hessian of the objective at (`input_val0`, `hyper_val0`). If not specified it is calculated at initialization.

The remaining arguments are the same as for the '__init__' method.

print_terms(*k=None*)
Print the derivative terms in the Taylor series.

Parameters

k: integer Optional. Which term to print. If unspecified, all terms are printed.

2.1.3 Optimum checking

class vittles.bivariate_sensitivity_lib.**OptimumChecker**(*estimating_equation*, *solver*, *input_base*, *hyper_base*)

__init__(*estimating_equation*, *solver*, *input_base*, *hyper_base*)
Estimate the error in sensitivity due to incomplete optimization.

Parameters

estimating_equation [*callable*] A function taking arguments *(input, hyper)* and returning a vector, typically the same length as the input. The idea is that *estimating_equation(input_base, hyper_base) = [0, ..., 0]*.

solver [*callable*] A function of a single vector variable *v* solving $H^{-1}v$, where *H* is the Hessian of the estimating equation with respect to the input variable at *input_base*, *hyper_base*.

input_base [*numpy.ndarray*] The base value of the parameter to be optimized

hyper_base [*numpy.ndarray*] The base value of the hyperparameter.

correction(*hyper_new*, *dinput_dhyper=None*, *newton_step=None*)
Return the first-order correction to the change in dinput_dhyper as you take a Newton step.

evaluate(*hyper_new*, *dinput_dhyper=None*, *newton_step=None*)
Return the first-order approximation to the change in dinput_dhyper as you take a Newton step.

get_dinput_dhyper(*dhyper*)
Return the first directional derivative of the optimum with respect to the hyperparameter in the direction *dhyper*.

get_newton_step()
Return a Netwon step towards the optimum.

class vittles.bivariate_sensitivity_lib.**CrossSensitivity**(*estimating_equation*, *solver*, *input_base*, *hyper1_base*, *hyper2_base*, *term_ii=True*, *term_i1=True*, *term_i2=True*, *term_12=True*)
Calculate a second-order derivative of an optimum with resepct to two hyperparameters.

Given an estimating equation $G(\theta, \epsilon_1, \epsilon_2)$, with $G(\hat{\theta}(\epsilon_1, \epsilon_2), \epsilon_1, \epsilon_2) = 0$, this class evaluates a directional derivatives

$$\frac{d^2\hat{\theta}}{d\epsilon_1 d\epsilon_2}\Delta\epsilon_1\Delta\epsilon_2.$$

__init__(*estimating_equation*, *solver*, *input_base*, *hyper1_base*, *hyper2_base*, *term_ii=True*, *term_i1=True*, *term_i2=True*, *term_12=True*)
Initialize self. See help(type(self)) for accurate signature.

2.1.4 Linear response covariances

class vittles.lr_cov_lib.**LinearResponseCovariances**(*objective_fun*, *opt_par_value*, *validate_optimum=False*, *hessian_at_opt=None*, *factorize_hessian=True*, *grad_tol=1e-08*)
Calculate linear response covariances of a variational distribution.

Let $q(\theta|\eta)$ be a class of probability distribtions on θ where the class is parameterized by the real-valued vector η. Suppose that we wish to approximate a distribution $q(\theta|\eta^*) \approx p(\theta)$ by solving an optimization problem $\eta^* = \mathrm{argmin} f(\eta)$. For example, f might be a measure of distance between $q(\theta|\eta)$ and $p(\theta)$. This class uses the sensitivity of the optimal η^* to estimate the covariance $\mathrm{Cov}_p(g(\theta))$. This covariance estimate is called the "linear response covariance".

In this notation, the arguments to the class mathods are as follows. f is `objective_fun`, η^* is `opt_par_value`, and the function `calculate_moments` evaluates $\mathbb{E}_{q(\theta|\eta)}[g(\theta)]$ as a function of η.

Methods

set_base_values:	Set the base values, η^* that optimizes the objective function.
get_hessian_at_opt:	Return the Hessian of the objective function evaluated at the optimum.
get_hessian_cholesky_at_opt:	Return the Cholesky decomposition of the Hessian of the objective function evaluated at the optimum.
get_lr_covariance:	Return the linear response covariance of a given moment.

__init__(*objective_fun*, *opt_par_value*, *validate_optimum=False*, *hessian_at_opt=None*, *factorize_hessian=True*, *grad_tol=1e-08*)

Parameters

objective_fun: Callable function A callable function whose optimum parameterizes an approximate Bayesian posterior. The function must take as a single argument a numeric vector, `opt_par`.

opt_par_value: The value of `opt_par` at which `objective_fun` is optimized.

validate_optimum: Boolean When setting the values of `opt_par`, check that `opt_par` is, in fact, a critical point of `objective_fun`.

hessian_at_opt: Numeric matrix (optional) The Hessian of `objective_fun` at the optimum. If not specified, it is calculated using automatic differentiation.

factorize_hessian: Boolean If `True`, solve the required linear system using a Cholesky factorization. If `False`, use the conjugate gradient algorithm to avoid forming or inverting the Hessian.

grad_tol: Float The tolerance used to check that the gradient is approximately zero at the optimum.

get_lr_covariance(*calculate_moments*)

Get the linear response covariance of a vector of moments.

Parameters

calculate_moments: Callable function A function that takes the folded `opt_par` as a single argument and returns a numeric vector containing posterior moments of interest.

Returns

Numeric matrix If `calculate_moments(opt_par)` returns $\mathbb{E}_q[g(\theta)]$ then this returns the linear response estimate of $\mathrm{Cov}_p(g(\theta))$.

get_lr_covariance_from_jacobians(*moment_jacobian1*, *moment_jacobian2*)

Get the linear response covariance between two vectors of moments.

Parameters

moment_jacobian1: 2d numeric array. The Jacobian matrix of a map from a value of `opt_par` to a vector of moments of interest. Following standard notation for Jacobian matrices, the rows should correspond to moments and the columns to elements of a flattened `opt_par`.

moment_jacobian2: 2d numeric array. Like `moment_jacobian1` but for the second vector of moments.

Returns

Numeric matrix If `moment_jacobian1(opt_par)` is the Jacobian of $\mathbb{E}_q[g_1(\theta)]$ and `moment_jacobian2(opt_par)` is the Jacobian of $\mathbb{E}_q[g_2(\theta)]$ then this returns the linear response estimate of $\mathrm{Cov}_p(g_1(\theta), g_2(\theta))$.

get_moment_jacobian(*calculate_moments*)

The Jacobian matrix of a map from `opt_par` to a vector of moments of interest.

Parameters

calculate_moments: Callable function A function that takes the folded `opt_par` as a single argument and returns a numeric vector containing posterior moments of interest.

Returns

Numeric matrix The Jacobian of the moments.

2.2 Helper Functions

2.2.1 Sparse Hessians

class vittles.sparse_hessian_lib.**SparseBlockHessian**(*objective_function*, *sparsity_array*)

Efficiently calculate block-sparse Hessians.

The objective function is expected to be of the form

$$x = (x_1, ..., x_G) \text{ (or some permutation thereof)}$$

$$f(x) = \sum_{g=1}^{G} f_g(x_g)$$

Each x_g is expected to have the same dimension. Consequently, the Hessian matrix of `f` with respect to `x`, is block diagonal with `G` blocks, up to a permutation of the order of `x`. The purpose of this class is to efficiently calculate this Hessian when the block structure (i.e., the partition of `x`) is known.

__init__(*objective_function*, *sparsity_array*)

In terms of the class description, `objective_function = f`, `opt_par = x`, and `sparsity_array` describes the partition of `x` into $(x_1, ..., x_G)$.

Parameters

objective_function [*callable*] An objective function of which to calculate a Hessian. The argument should be

- `opt_par`: *numpy.ndarray* (N,) The optimization parameter.

sparsity_array [*numpy.ndarray* (G, M)] An array containing the indices of rows and columns of each block. The Hessian should contain `G` dense blocks, each of which is `M` by `M`. Each row of `sparsity_array` should contain the indices of the corresponding block. There must be no repeated indices, and each block must be the same size.

get_block_hessian(*opt_par*, *print_every=0*)

Get the block Hessian at `opt_par` and `weights`.

Returns

hessian [*scipy.sparse.coo_matrix* (N, N)] The block-sparse Hessian given by and `sparsity_array`.

get_global_hessian(*opt_par*, *global_inds=None*, *print_every=0*)
Get the dense Hessian terms for the global parameters, which are, by default, indexed by any indices not in `_sparsity_array`.

2.2.2 System solvers

vittles.solver_lib.**get_cg_solver**(*mat_times_vec*, *dim*, *cg_opts={}*)
Return a function solving $h^{-1}v$ for a matrix *h* using conjugate gradient.

Parameters

mat_times_vec [*callable*] A function of a single vector argument, *v* that returns the product *h v* for some invertible matrix *h*.

dim [*int*] The dimension of the vector *v*.

cg_opts [*dict*] Optional, a dictionary of named arguments for the solver *sp.sparse.linalg.cg*.

Returns

solve [*callable*] A function of a single vector argument, *v* that returns the conjugate gradient approximation to $h^{-1}v$.

vittles.solver_lib.**get_cholesky_solver**(*h*)
Return a function solving $h^{-1}v$ for a matrix *h*.

Parameters

h [A dense or sparse matrix.]

Returns

solve [*callable*] A function of a single vector argument, *v* that returns $h^{-1}v$.

vittles.solver_lib.**get_dense_cholesky_solver**(*h*, *h_chol=None*)
Return a function solving $h^{-1}v$ with a dense Cholesky factorization.

Parameters

h [*numpy.ndarray*] A dense symmetric positive definite matrix.

h_chol [A Cholesky factorization] Optional, a Cholesky factorization created with *sp.linalg.cho_factor*. If this is specified, the argument *h* is ignored. If *None* (the default), the factoriztion of *h* is calculated.

Returns

solve [*callable*] A function of a single vector argument, *v* that returns $h^{-1}v$.

vittles.solver_lib.**get_sparse_cholesky_solver**(*h*)
Return a function solving $h^{-1}v$ for a sparse *h*.

Parameters

h [A sparse invertible matrix.]

Returns

solve [*callable*] A function of a single vector argument, *v* that returns $h^{-1}v$.

CHAPTER 3

Examples

3.1 Maximum Likelihood and Weight Sensitivity.

[1]:
```
import autograd
from autograd import numpy as np

import copy

# This contains functions that are used in several paragami examples.
import example_utils

import matplotlib.pyplot as plt
%matplotlib inline

import paragami
import vittles

# Use the original scipy for functions we don't need to differentiate.
import scipy as osp

import time
```

3.1.1 Define a Model.

For illustration, let's consider a simple example: a Gaussian maximum likelihood estimator.

$$x_n \overset{iid}{\sim} \mathcal{N}(\mu, \Sigma), \text{ for } n = 1, ..., N.$$

The "model parameters" are μ and Σ, and we will estimate them using maximum likelihood estimation (MLE). Let the data be denoted by $X = (x_1, ..., x_N)$. For a given set of data weights $W = (w_1, ..., w_N)$, we can define the loss

$$\ell(X, W, \mu, \Sigma) = \frac{1}{2} \sum_{n=1}^{N} w_n \left((x_n - \mu)^T \Sigma^{-1} (x_n - \mu) + \log |\Sigma| \right).$$

The loss on the original dataset is given when $W_1 = (1, ..., 1)$, so we will take the MLE to be

$$\hat{\mu}, \hat{\Sigma} = \text{argmax} \ell(X, W_1, \mu, \Sigma).$$

We will consider approximating the effect of varying W on the optimal value in the sensitivity section below.

Of course, this example has a closed-form optimum as a function of the weights, W:

$$\hat{\mu}(W) = \frac{1}{\sum_{n=1}^{N} w_n} \sum_{n=1}^{N} w_n x_n \qquad \hat{\Sigma}(W) = \frac{1}{\sum_{n=1}^{N} w_n} \sum_{n=1}^{N} w_n \left(x_n - \hat{\mu}(W)\right) \left(x_n - \hat{\mu}(W)\right)^T \tag{3.1}$$

(3.2)

However, for expository purposes let us treat it as a generic optimization problem.

Specify Parameters and Draw Data.

[2]:
```
np.random.seed(42)

num_obs = 1000

# True values of parameters
true_sigma = \
    np.eye(3) * np.diag(np.array([1, 2, 3])) + \
    np.random.random((3, 3)) * 0.1
true_sigma = 0.5 * (true_sigma + true_sigma.T)

true_mu = np.array([0, 1, 2])

# Data
x = np.random.multivariate_normal(
    mean=true_mu, cov=true_sigma, size=(num_obs, ))

# Original weights.
original_weights = np.ones(num_obs)
```

Write out the log likelihood and use it to specify a loss function.

[3]:
```
# The loss function is the weighted negative of the log likelihood.
def get_loss(norm_param_dict, x, weights):
    return np.sum(
        -1 * weights * example_utils.get_normal_log_prob(
            x, norm_param_dict['sigma'], norm_param_dict['mu']))

true_norm_param_dict = dict()
true_norm_param_dict['sigma'] = true_sigma
true_norm_param_dict['mu'] = true_mu

print('Loss at true parameter: {}'.format(
    get_loss(true_norm_param_dict, x, original_weights)))
```

```
Loss at true parameter: 2392.751922600241
```

3.1.2 Flatten and Fold for Optimization.

Note that we have written our loss, `get_loss` as a function of a *dictionary of parameters*, `norm_param_dict`.

We can use `paragami` to convert such a dictionary to and from a flat, unconstrained parameterization for optimization.

Define a `paragami` pattern that matches the input to `get_loss`.

[4]:
```
norm_param_pattern = paragami.PatternDict()
norm_param_pattern['sigma'] = paragami.PSDSymmetricMatrixPattern(size=3)
norm_param_pattern['mu'] = paragami.NumericArrayPattern(shape=(3, ))
```

"Flatten" the dictionary into an unconstrained vector.

[5]:
```
norm_param_freeflat = norm_param_pattern.flatten(true_norm_param_dict, free=True)
print('The flat parameter has shape: {}'.format(norm_param_freeflat.shape))
```

```
The flat parameter has shape: (9,)
```

Optimize using `autograd`.

We can use this flat parameter to optimize the likelihood directly without worrying about the PSD constraint on Σ.

[6]:
```
print('\nNext, wrap the loss to be a function of the flat parameter.')

# Later it will be useful to change the weights used for optimization.
optim_weights = original_weights
get_freeflat_loss = paragami.FlattenFunctionInput(
    original_fun=lambda param_dict: get_loss(param_dict, x, optim_weights),
    patterns=norm_param_pattern,
    free=True)

print('Finally, use the flattened function to optimize with autograd.\n')
get_freeflat_loss_grad = autograd.grad(get_freeflat_loss)
get_freeflat_loss_hessian = autograd.hessian(get_freeflat_loss)

# Initialize with zeros.
init_param = np.zeros(norm_param_pattern.flat_length(free=True))
mle_opt = osp.optimize.minimize(
    method='trust-ncg',
    x0=init_param,
    fun=get_freeflat_loss,
    jac=get_freeflat_loss_grad,
    hess=get_freeflat_loss_hessian,
    options={'gtol': 1e-12, 'disp': False})
```

```

Next, wrap the loss to be a function of the flat parameter.
Finally, use the flattened function to optimize with autograd.

```

"Fold" to inspect the result.

We can now "fold" the optimum back into its original shape for inspection and further use.

[7]:
```
norm_param_flat0 = copy.deepcopy(mle_opt.x)
norm_param_opt = norm_param_pattern.fold(mle_opt.x, free=True)

for param in ['sigma', 'mu']:
    print('Parmeter {}\nOptimal:\n{}\n\nTrue:\n{}\n\n'.format(
        param, norm_param_opt[param], true_norm_param_dict[param]))
```

```
Parmeter sigma
Optimal:
[[ 1.06683522  0.07910048  0.04229475]
 [ 0.07910048  1.89297797 -0.02650233]
 [ 0.04229475 -0.02650233  2.92376984]]

True:
[[1.03745401 0.07746864 0.03950388]
 [0.07746864 2.01560186 0.05110853]
 [0.03950388 0.05110853 3.0601115 ]]


Parmeter mu
Optimal:
[-0.04469438  1.03094019  1.8551187 ]

True:
[0 1 2]
```

3.1.3 Sensitivity Analysis for Approximate LOO.

Suppose we are interested in how our estimator would change if we left out one datapoint at a time. The leave-one-out (LOO) estimator for the n^{th} datapoint is given by $W_{(n)} = (1, ..., 1, 0, 1, ..., 1)$, where the 0 occurs in the n^{th} place, and

$$\hat{\mu}_{(n)}, \hat{\Sigma}_{(n)} = \text{argmax}\ell(X, W_{(n)}, \mu, \Sigma).$$

In full generality, one must re-optimize to get $\hat{\mu}_{(n)}, \hat{\Sigma}_{(n)}$. This can be expensive. To avoid the cost of repeatedly re-optimizing, we can form a linear approximation to the dependence of the optimal model parameters on the weights.

Specify a flattented objective function that also depends on the weights.

[8]:
```
get_freeflat_hyper_loss = paragami.FlattenFunctionInput(
    original_fun=lambda param_dict, weights: get_loss(param_dict, x, weights),
    patterns=norm_param_pattern,
    free=True,
    argnums=0)
```

Define a `HyperparameterSensitivityLinearApproximation` object.

[9]:
```
weight_sens = \
    vittles.HyperparameterSensitivityLinearApproximation(
        objective_fun=           get_freeflat_hyper_loss,
        opt_par_value=           norm_param_flat0,
        hyper_par_value=  original_weights)
```

Use `weight_sens.predict_opt_par_from_hyper_par` to predict the effect of different weights on the optimum.

[10]:
```
def get_loo_weight(n):
    weights = np.ones(num_obs)
    weights[n] = 0
    return weights

n_loo = 10
print('Approximate the effect of leaving out observation {}.'.format(n_loo))
loo_weights = get_loo_weight(n_loo)
norm_param_flat1 = \
    weight_sens.predict_opt_par_from_hyper_par(loo_weights)
approx_loo_norm_param_opt = norm_param_pattern.fold(norm_param_flat1, free=True)

for param in ['sigma', 'mu']:
    print('Parameter  {}\nApproximate LOO:\n{}\nOriginal optimum:\n{}\n\n'.format(
          param, approx_loo_norm_param_opt[param], norm_param_opt[param]))
```

```
Approximate the effect of leaving out observation 10.
Parameter  sigma
Approximate LOO:
[[ 1.06789931  0.07906974  0.04205564]
 [ 0.07906974  1.89118719 -0.0359618 ]
 [ 0.04205564 -0.0359618   2.90264779]]
Original optimum:
[[ 1.06683522  0.07910048  0.04229475]
 [ 0.07910048  1.89297797 -0.02650233]
 [ 0.04229475 -0.02650233  2.92376984]]


Parameter  mu
Approximate LOO:
[-0.04475159  1.02902066  1.85020216]
Original optimum:
[-0.04469438  1.03094019  1.8551187 ]
```

The approximation `predict_opt_par_from_hyper_par` is fast, so we can easily approximate LOO estimators for each datapoint.

[11]:
```
approx_loo_params = []
tic = time.time()
for n_loo in range(num_obs):
    loo_weights = get_loo_weight(n_loo)
    norm_par_flat1 = \
        weight_sens.predict_opt_par_from_hyper_par(loo_weights)
```

(continues on next page)

(continued from previous page)

```
    approx_loo_params.append(norm_param_pattern.fold(norm_par_flat1, free=True))
approx_loo_time_per_n = (time.time() - tic) / num_obs
```

Compare with the re-optimizing.

We can calculate the exact optimum for a few datapoints to compare with the approximation.

[12]:
```
num_opt = 20

opt_loo_params = []
opt_n = np.arange(num_opt)
tic = time.time()
for n_loo in opt_n:
    loo_weights = get_loo_weight(n_loo)
    optim_weights = loo_weights
    # Start at the previous optimum to speed up optimization.
    # Note that you generally need an accurate optimum to measure
    # the effect of small changes.
    loo_mle_opt = osp.optimize.minimize(
        method='trust-ncg',
        x0=mle_opt.x,
        fun=get_freeflat_loss,
        jac=get_freeflat_loss_grad,
        hess=get_freeflat_loss_hessian,
        options={'gtol': 1e-12, 'disp': False})
    opt_loo_params.append(norm_param_pattern.fold(loo_mle_opt.x, free=True))
opt_loo_time_per_n = (time.time() - tic) / num_opt
```

The approximate LOO estimator is much faster.

[13]:
```
print('Approximate LOO time per datapoint:\t{0:.5f} seconds'.format(approx_loo_time_
↪per_n))
print('Re-optimization LOO time per datapoint:\t{0:.5f} seconds'.format(opt_loo_time_
↪per_n))
```

```
Approximate LOO time per datapoint:     0.00010 seconds
Re-optimization LOO time per datapoint: 0.12403 seconds
```

A quantity one might consider for LOO analysis is the excess loss on the left-out datapoint.

$$\text{Excess loss}_n = \ell(x_n, 1, \hat{\mu}, \hat{\Sigma}) - \ell(x_n, 1, \hat{\mu}_{(n)}, \hat{\Sigma}_{(n)}).$$

We can compare the excess loss as estimated with the approximation and by re-optimizing.

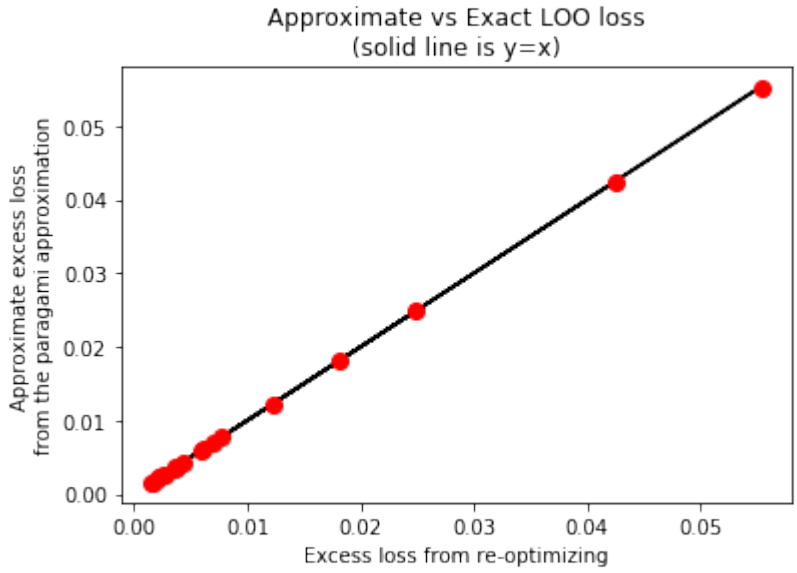
[14]:
```
def obs_n_excess_loss(norm_param, n):
    return \
        get_loss(norm_param, x[n, :], 1) - \
        get_loss(norm_param_opt, x[n, :], 1)

opt_loo_loss = [ obs_n_excess_loss(opt_loo_params[n], n) for n in opt_n ]
approx_loo_loss = [ obs_n_excess_loss(approx_loo_params[n], n) for n in opt_n ]

plt.plot(opt_loo_loss, opt_loo_loss, 'k')
plt.plot(opt_loo_loss, approx_loo_loss, 'ro', markersize=8)
plt.xlabel('Excess loss from re-optimizing')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Approximate excess loss\nfrom the paragami approximation')
plt.title('Approximate vs Exact LOO loss\n(solid line is y=x)')
```

[14]:
```
Text(0.5, 1.0, 'Approximate vs Exact LOO loss\n(solid line is y=x)')
```

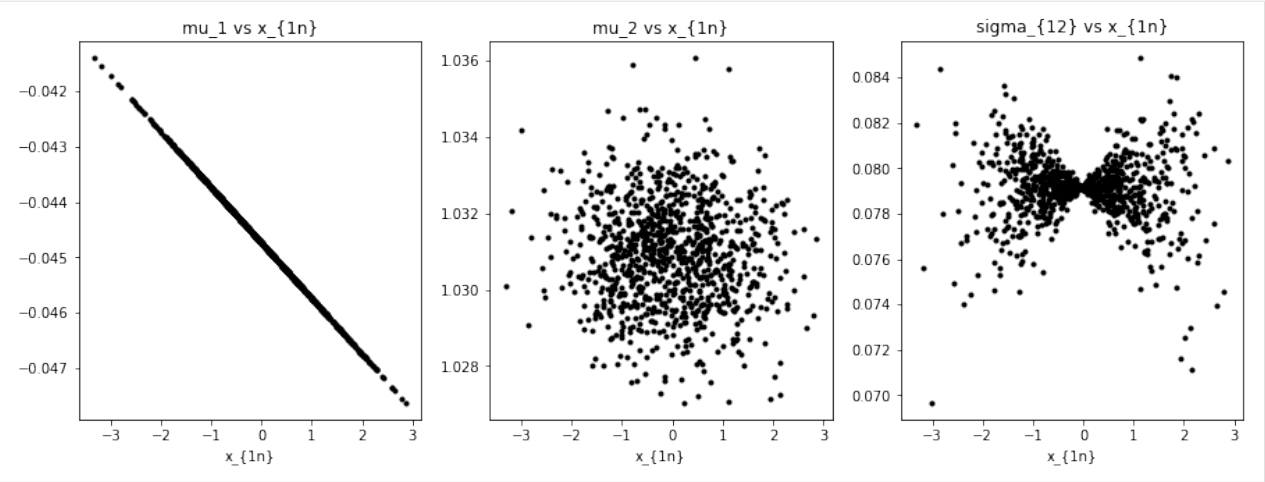


Fast tools lead to fun exploratory data analysis!

We can graph LOO sensitivity of various parameters vs x_{1n}.

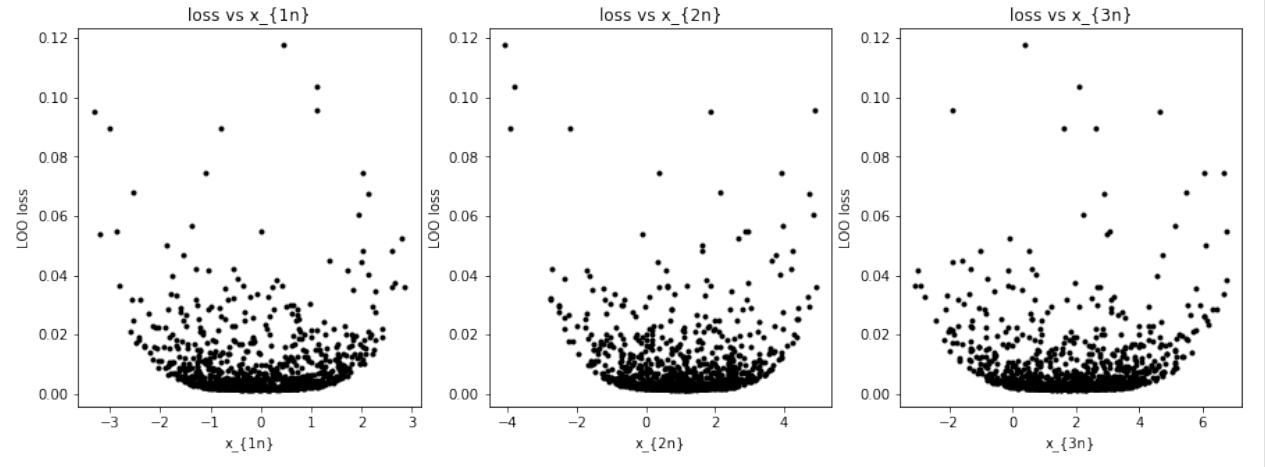
[15]:
```
mu_1_loo = [ param['mu'][0] for param in approx_loo_params ]
mu_2_loo = [ param['mu'][1] for param in approx_loo_params ]
sigma_12_loo = [ param['sigma'][0, 1] for param in approx_loo_params ]

plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.plot(x[:, 0], mu_1_loo, 'k.')
plt.title('mu_1 vs x_{1n}')
plt.xlabel('x_{1n}')

plt.subplot(1, 3, 2)
plt.plot(x[:, 0], mu_2_loo, 'k.')
plt.title('mu_2 vs x_{1n}')
plt.xlabel('x_{1n}')

plt.subplot(1, 3, 3)
plt.plot(x[:, 0], sigma_12_loo, 'k.')
plt.title('sigma_{12} vs x_{1n}')
plt.xlabel('x_{1n}')
```

[15]:
```
Text(0.5, 0, 'x_{1n}')
```



We can examine the excess loss versus the data.

[16]:
```
def obs_n_loss(norm_param, n):
    return \
        get_loss(norm_param, x[n, :], 1) - \
        get_loss(norm_param_opt, x[n, :], 1)

loo_loss = [ obs_n_loss(approx_loo_params[n], n) for n in range(num_obs) ]

# Plot the LOO loss versus each dimension of the data.
plt.figure(figsize=(15, 5))
for dim in range(3):
    plt.subplot(1, 3, dim + 1)
    plt.plot(x[:, dim], loo_loss, 'k.')
    plt.title('loss vs x_{{{}n}}'.format(dim + 1))
    plt.xlabel('x_{{{}n}}'.format(dim + 1))
    plt.ylabel('LOO loss')
```



CHAPTER 4

Release History

4.1 None yet.

Python Module Index

v

Index

Symbols

C

E

G

H

L

O

P

S

V