How to incorporate individual measurement uncertainties into Gaussian Process?

Asked 2 years, 3 months ago Modified 9 months ago Viewed 1k times

8

I have a set of observations, `f_i=f(x_i)`, and I want to construct a probabilistic surrogate, `f(x) ~ N[mu(x), sigma(x)]`, where `N` is a normal distribution. Each observed output, `f_i`, is associated with a measurement uncertainty, `sigma_i`. I would like to incorporate these measurement uncertainties into my surrogate, `f_i`, so that `mu(x)` predicts the observations, `f_i(x_i)`, and that the predicted standard deviation, `sigma(x_i)`, envelops the uncertainty in the observed output, `epsilon_i`.

The only way I can think of to accomplish this is through a combination of Monte Carlo sampling and Gaussian Process modeling. It would be ideal to accomplish this with a single Gaussian process, without Monte Carlo samples, but I can not make this work.

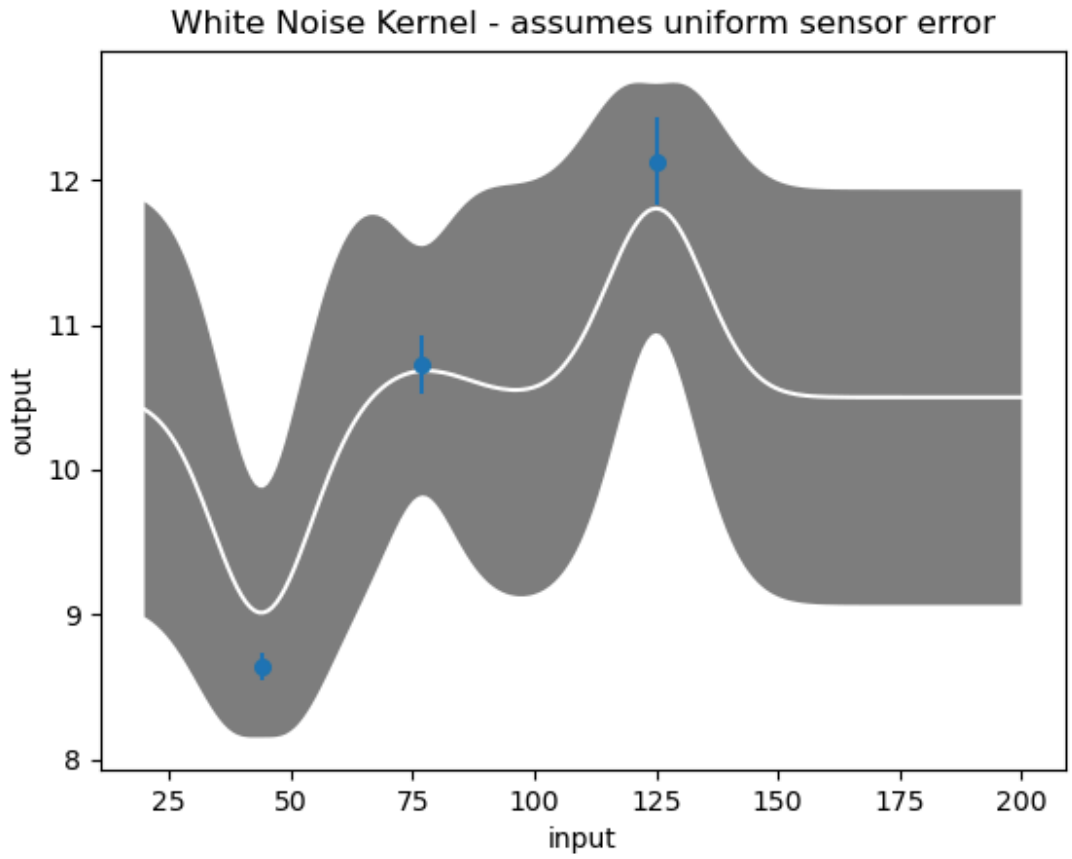
I show three attempts to accomplish my goal. The first two avoid Monte Carlo sampling, but do not predict an average of `f(x_i)` with uncertainty bands that envelop `epsilon(x_i)`. The third approach uses Monte Carlo sampling and accomplishes what I want to do.

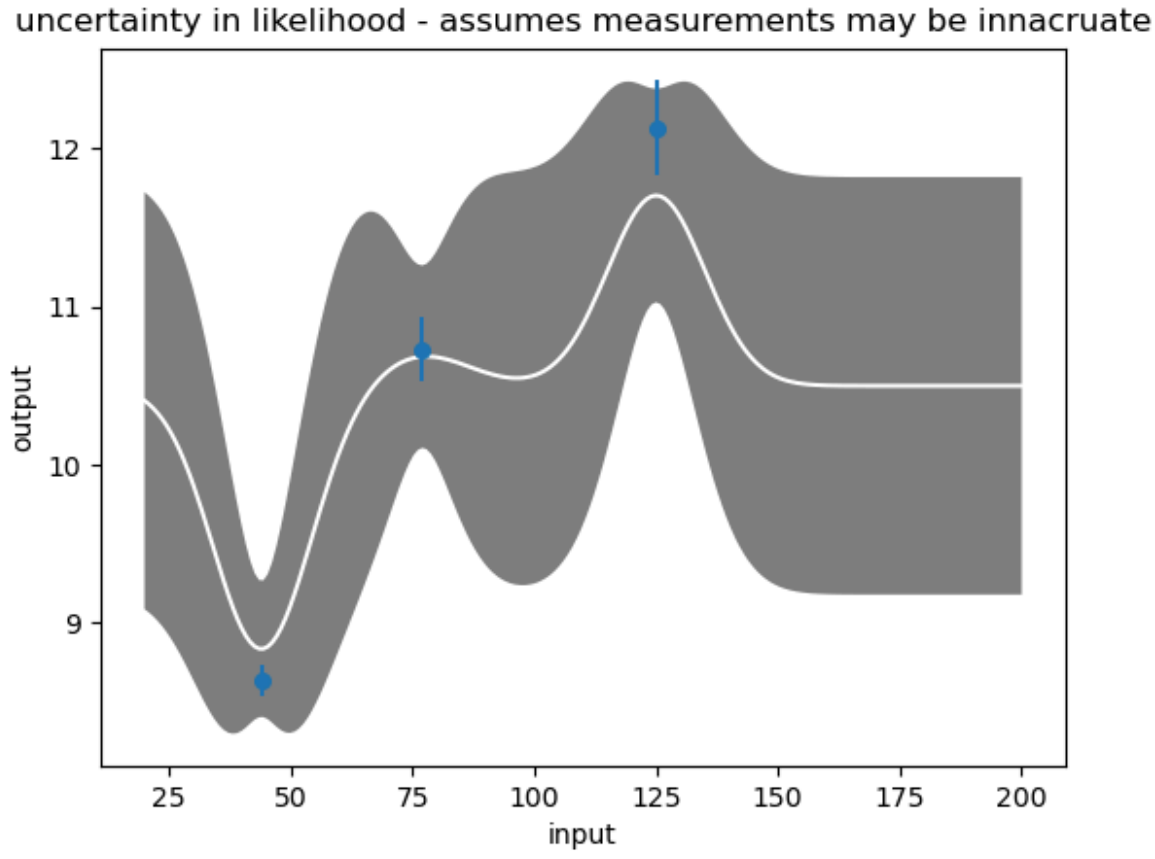
Is there a way to create a Gaussian Process that on average predicts the mean observed output, with uncertainty that will envelop uncertainty in the observed output, without using this Monte Carlo approach?

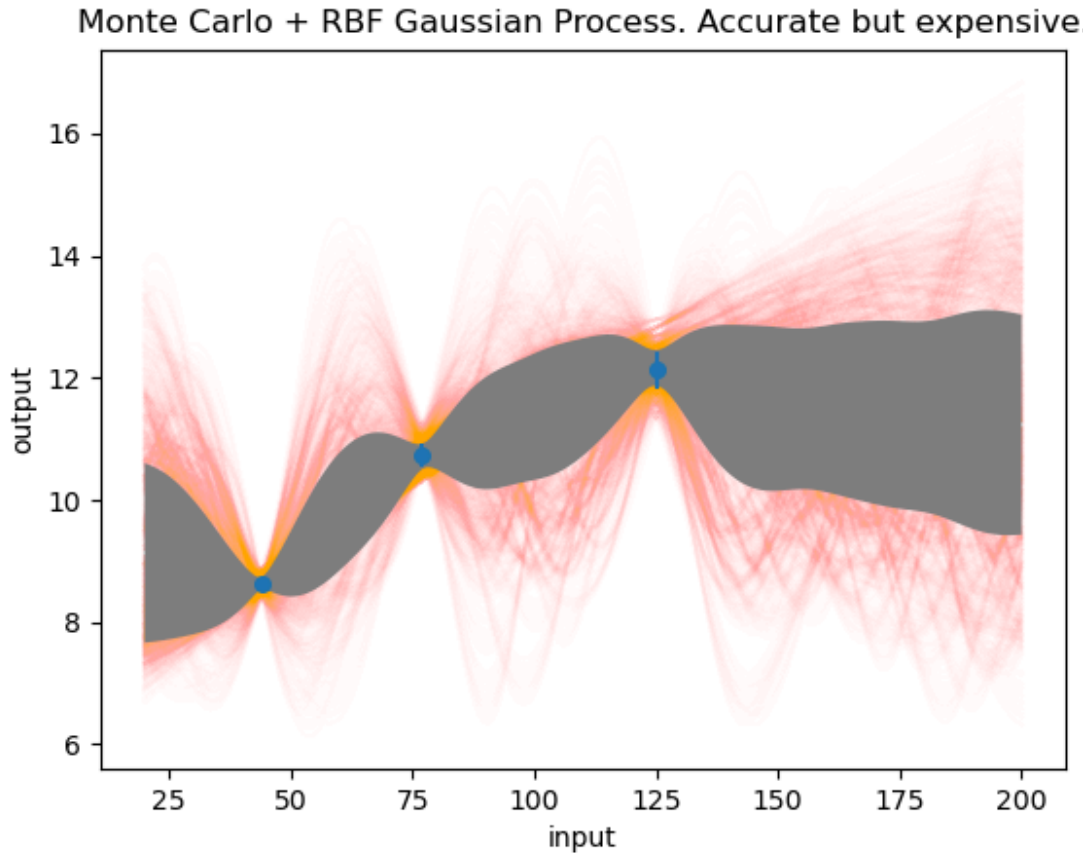
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, Matern, ExpSineSquared,
WhiteKernel

# given a set of inputs, x_i, and corresponding outputs, f_i, I want to make a
surrogate f(x).
# each f_i is measured with a different instrument, that has a different
uncertainty.

# measured inputs
xs = np.array([44, 77, 125])

# measured outputs
fs = [8.64, 10.73, 12.13]

# uncertainty in measured outputs
errs = np.array([0.1, 0.2, 0.3])

# inputs to predict
finex = np.linspace(20, 200, 200)


##############
### approach 1: uncertainty in kernel
# - the kernel is constant and cannot change as a function of the input
# - uncertainty in measurements can be incorporated using a whitenoisekernel
# - the white noise uncertainty can be specified as the average of the
```

```
observation error

# RBF + whitenoise kernel
kernel = 1 * RBF(length_scale=9, length_scale_bounds=(10, 1e3)) +
WhiteKernel(errs.mean(), noise_level_bounds=(errs.mean() - 1e-8, errs.mean() +
1e-8))
gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True)
gaussian_process.fit((np.atleast_2d(xs).T), (fs))
mu, std = gaussian_process.predict((np.atleast_2d(finex).T), return_std=True)
plt.scatter(xs, fs, zorder=3, s=30)
plt.fill_between(finex, (mu - std), (mu + std), facecolor='grey')
plt.plot(finex, mu, c='w')
plt.errorbar(xs, fs, yerr=errs, ls='none')
plt.xlabel('input')
plt.ylabel('output')
plt.title('White Noise Kernel - assumes uniform sensor error')
plt.savefig('gp_whitenoise')
plt.clf()


####################
### Aproach 2: incorporate measurement uncertainty in the likelihood function
# - the likelihood function can be altered throught the alpha parameter
# - this assumes gaussian uncertainty in the measured input
kernel = 1 * RBF(length_scale=9, length_scale_bounds=(10, 1e3))
gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True, alpha=errs)
gaussian_process.fit((np.atleast_2d(xs).T), (fs))
mu, std = gaussian_process.predict((np.atleast_2d(finex).T), return_std=True)
plt.scatter(xs, fs, zorder=3, s=30)
plt.fill_between(finex, (mu - std), (mu + std), facecolor='grey')
plt.plot(finex, mu, c='w')
plt.errorbar(xs, fs, yerr=errs, ls='none')
plt.xlabel('input')
plt.ylabel('output')
plt.title('uncertainty in likelihood - assumes measurements may be innacruate')
plt.savefig('gp_alpha')
plt.clf()

####################
### Aproach 3: Monte Carlo of measurement uncertainty + GP
# - The Gaussian process represents uncertainty in creating the surrogate f(x)
# - The uncertainty in observed inputs can be propogated using Monte Carlo
# - downside: less computationally efficient, no analytic solution for mean or
uncertainty
kernel = 1 * RBF(length_scale=9, length_scale_bounds=(10, 1e3))
posterior_history = np.zeros((finex.size, 100 * 50))
for sample in range(100):
   simulatedSamples = fs + np.random.normal(0, errs)
   gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True)
   gaussian_process.fit((np.atleast_2d(xs).T), (simulatedSamples))
   posterior_sample = gaussian_process.sample_y((np.atleast_2d(finex).T), 50)
   plt.plot(finex, posterior_sample, c='orange', alpha=0.005)
   posterior_history[:, sample * 50 : (sample + 1) * 50] = posterior_sample
plt.plot(finex, posterior_history.mean(1), c='w')
plt.fill_between(finex, posterior_history.mean(1) - posterior_history.std(1),
posterior_history.mean(1) + posterior_history.std(1), facecolor='grey', alpha=1,
zorder=5)
plt.scatter(xs, fs, zorder=6, s=30)
plt.errorbar(xs, fs, yerr=errs, ls='none', zorder=6)
plt.xlabel('input')
plt.ylabel('output')
plt.title('Monte Carlo + RBF Gaussian Process. Accurate but expensive.')
```

```
plt.savefig('gp_monteCarlo')
plt.clf()
```







python **scikit-learn** **gaussian-process** **uncertainty**

Share Improve this question

Follow

edited Oct 24, 2022 at 16:16

asked Oct 21, 2022 at 9:17

kilojoules

10.1k 21 82 157

1 @desertnaut why did you remove the data-science tag? – kilojoules Oct 23, 2022 at 9:29

I'm really sorry I came across this question only now. Super interesting issue. I have some problem in visualizing the outputs – imburningbabe Oct 24, 2022 at 13:06

Let me know if I can make the question more clear – kilojoules Oct 24, 2022 at 16:13

1 I copy-pasted your code and I got this error message -> ConvergenceWarning: The optimal value found for dimension 0 of parameter k2__length_scale is close to the specified lower bound 10.0. Decreasing the bound and calling fit again may find a better value. I see a lot of low level things here on SO, but these are the problems I like :) – imburningbabe Oct 24, 2022 at 17:58

I decided to enforce prior knowledge about what this parameter should be via the minimum bound. The warning appears even if I use 1e-4 as a lower bound. The optimizer seems to want the parameter to be close to zero, except in the white noise case. These are the plots I get when I use 1e-4 as the lower bound: imgur.com/a/2MZZy3l. – kilojoules Oct 24, 2022 at 18:03

2 Answers

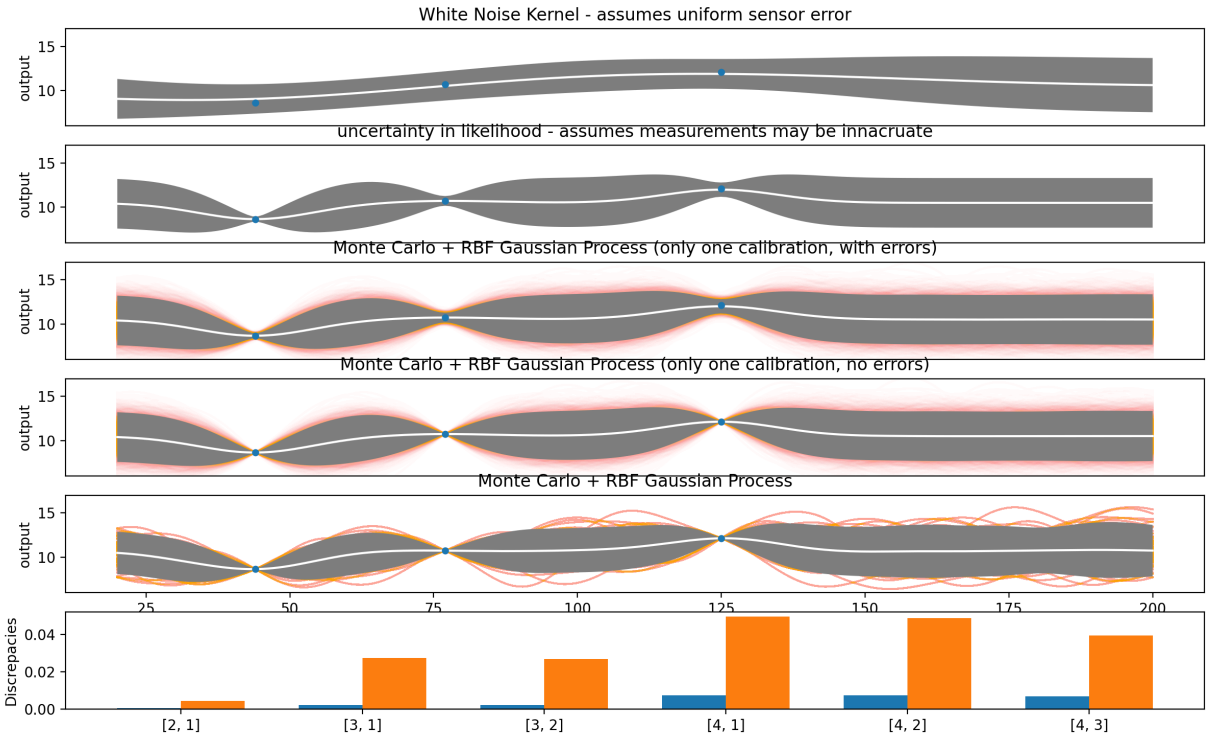
Sorted by: Highest score (default)

As @amance has suggested, you need to square `errs` - see this scikit example. However, the resulting means and standard deviations don't seem to tie up with your Monte-Carlo results, and increasing the number of calibrations and paths does not seem to improve the situation. I do not know how to explain the discrepancy.

Now in more detail. I have added two more Monte-Carlos, to compare them against the corrected `errs**2` approach #1 (white noise kernel counts as approach #0): approach #2 is Monte-Carlo with errors in the kernel (just like #1), and approach #3 is Monte-Carlo with 0 errors. Your original Monte-Carlo is the second plot from the bottom. In the last plot, I compare the means and standard deviations of all approaches (except #0) - the measure is the relative euclidian distance.

As you can see, approaches #1 and #2 yield almost identical means and std devs, and the mean of approach #3 (with 0 errors) is still quite close. Approach #4 yields very different means and std devs, and the paths look very different - there is some sort of clustering going on *even if I comment out errors*. Seems like there is either a problem with the implementation of the code (there are only 5 lines, and nothing obvious to me) or with the module.

Either way, seems like approach #1 is what you need!




```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel

# given a set of inputs, x_i, and corresponding outputs, f_i, I want to make a
surrogate f(x).
# each f_i is measured with a different instrument, that has a different
uncertainty.
xs = np.array([44, 77, 125]) # measured inputs
fs = [8.64, 10.73, 12.13] # measured outputs
errs = np.array([0.1, 0.2, 0.3]) # uncertainty in measured outputs
```

```
finex = np.linspace(20, 200, 200) # inputs to predict
finex_T = [[f] for f in finex]
xs_T = [[x] for x in xs]
calibrations, paths = 100, 100
kernel = RBF(length_scale=9, length_scale_bounds=(10, 1e3))

#######################################
fig, ax = plt.subplots(6)
means, std_devs, titles, posterior_history = [None] * 5, [None] * 5, [None] * 5,
[None] * 5
#######################################
### Approach 1: uncertainty in kernel
# - the kernel is constant and cannot change as a function of the input
# - uncertainty in measurements can be incorporated using a whitenoisekernel
# - the white noise uncertainty can be specified as the average of the
observation error

# RBF + whitenoise kernel
gaussian_process = GaussianProcessRegressor(
  kernel=kernel + WhiteKernel(errs.mean(), noise_level_bounds=(errs.mean() - 1e-
8, errs.mean() + 1e-8)),
  n_restarts_optimizer=9,
  normalize_y=True)
gaussian_process.fit(xs_T, fs)
means[0], std_devs[0] = gaussian_process.predict(finex_T, return_std=True)
titles[0] = 'White Noise Kernel - assumes uniform sensor error'

####################
### Approach 2: incorporate measurement uncertainty in the likelihood function
# - the likelihood function can be altered throught the alpha parameter
# - this assumes gaussian uncertainty in the measured input

gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True, alpha=errs**2)
gaussian_process.fit(xs_T, fs)
means[1], std_devs[1] = gaussian_process.predict(finex_T, return_std=True)
titles[1] = 'uncertainty in likelihood - assumes measurements may be innacruate'

####################
### Test Approach with Errors:
gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True, alpha=errs**2)
gaussian_process.fit(xs_T, fs)
posterior_history[2] = gaussian_process.sample_y(finex_T, calibrations * paths)
titles[2] = 'Monte Carlo + RBF Gaussian Process (only one calibration, with
errors)'

####################
### Test Approach No Errors:
gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True)
gaussian_process.fit(xs_T, fs)
posterior_history[3] = gaussian_process.sample_y(finex_T, calibrations * paths)
titles[3] = 'Monte Carlo + RBF Gaussian Process (only one calibration, no
errors)'

####################
### Approach 3: Monte Carlo of measurement uncertainty + GP
# - The Gaussian process represents uncertainty in creating the surrogate f(x)
# - The uncertainty in observed inputs can be propogated using Monte Carlo
# - downside: less computationally efficient, no analytic solution for mean or
uncertainty

posterior_history[4] = np.zeros((finex.size, calibrations * paths))
for sample in range(calibrations):
   gaussian_process = GaussianProcessRegressor(kernel=kernel,
```

```
n_restarts_optimizer=9, normalize_y=True)
   gaussian_process.fit(xs_T, fs)# + np.random.normal(0, errs**2*0))
   posterior_history[4][:, sample * paths : (sample + 1) * paths] =
gaussian_process.sample_y(finex_T, paths)
titles[4] = 'Monte Carlo + RBF Gaussian Process'

for i in range(2, 5):
  means[i], std_devs[i] = posterior_history[i].mean(1),
posterior_history[i].std(1)

####################
i_j = [[i, j] for i in range(2, 5) for j in range(1, i)]
means_err = [np.linalg.norm(means[i] - means[j]) / np.linalg.norm(means[i] +
means[j]) for i, j in i_j]
str_dev_err = [np.linalg.norm(std_devs[i] - std_devs[j]) /
np.linalg.norm(std_devs[i] + std_devs[j]) for i, j in i_j]

width = 0.35  # the width of the bars
x = np.arange(len(i_j))
rects1 = ax[-1].bar(x - width/2, means_err, width, label='means_err')
rects2 = ax[-1].bar(x + width/2, str_dev_err, width, label='str_dev_err')
ax[-1].set_ylabel('Discrepacies')
ax[-1].set_xticks(x, i_j)

####################
for i, ax_ in enumerate(ax[:-1]):
  if posterior_history[i] is not None:
    ax_.plot(finex, posterior_history[i], c='orange', alpha=0.005, zorder=-8)
  ax_.fill_between(finex, (means[i] - 1.96 * std_devs[i]), (means[i] + 1.96 *
std_devs[i]), facecolor='grey')
  ax_.plot(finex, means[i], c='w')
  ax_.set_title(titles[i])
  ax_.errorbar(xs, fs, errs, linestyle="None", color="tab:blue", marker=".",
markersize=8)
  ax_.set_ylim([6, 17])
  if i == len(ax) - 2:
    ax_.set_xlabel('input')
  else:
    ax_.set_xticks([])
  ax_.set_ylabel('output')

plt.show()
```

Share Improve this answer Follow

answered Oct 29, 2022 at 23:19

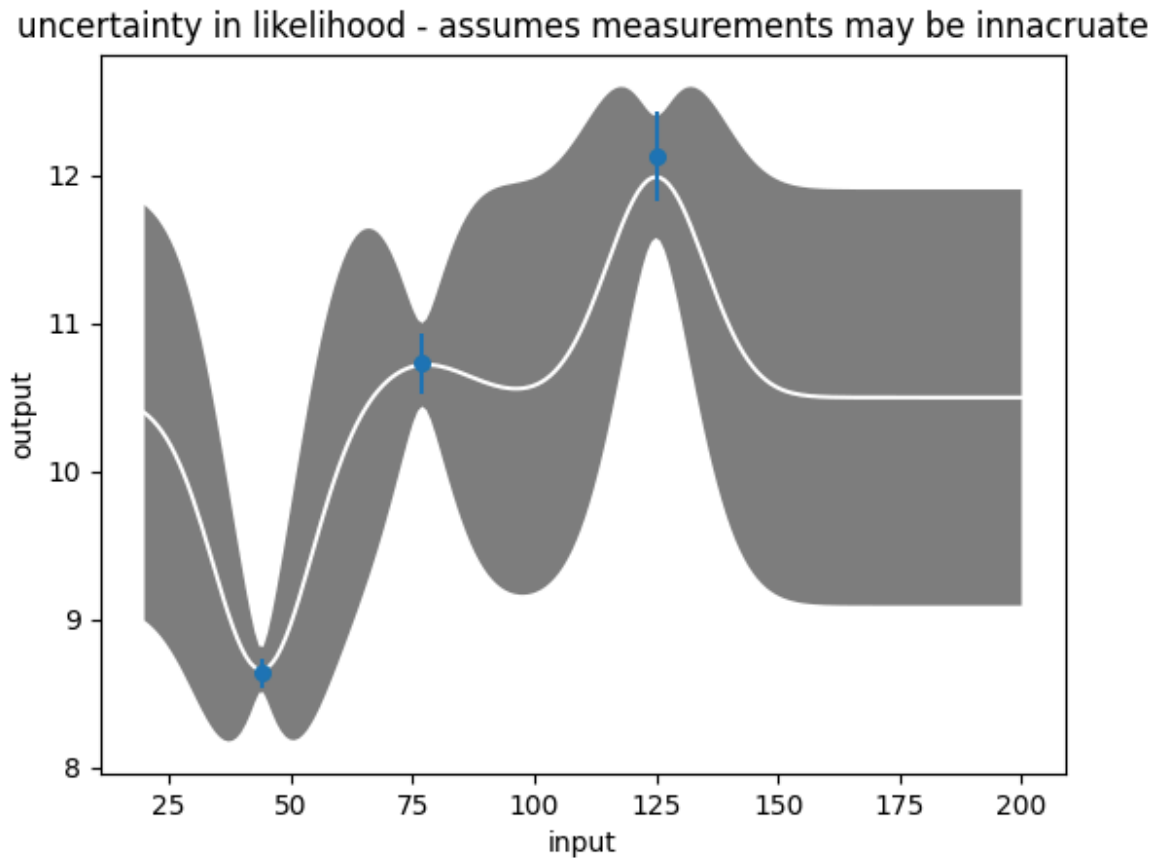
Yulia V

3,559 10 34 66

Using your second approach, only slightly changing Alpha

1

```
kernel = 1 * RBF(length_scale=9, length_scale_bounds=(10, 1e3))
gaussian_process = GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=9, normalize_y=True, alpha=errs**2)
gaussian_process.fit((np.atleast_2d(xs).T), (fs))
mu, std = gaussian_process.predict((np.atleast_2d(finex).T), return_std=True)
```



Share Improve this answer Follow edited Oct 27, 2022 at 17:42

answered Oct 27, 2022 at 16:16

amance

1,770 7 15

It's confusing to me that the mean of the Gaussian Process does not intersect with the mean of the measurements. I would expect the mean of our posterior to incorporate this prior knowledge. In this approach, it seems like we are somehow anticipating a deterministic bias in the sensor as well as the calibrated uncertainty distribution. – kilojoules Oct 28, 2022 at 17:33

1 @kilojoules - the errors you have specified are different from 0, and this gives the regressor the liberty to slightly misfit the target. If you set the errors to 0, then the target will be on the mean curve. – Yulia V Oct 29, 2022 at 23:48

@kilojoules this slight misfit is a healthy sign actually, perfect fit if the error is not 0 would imply overfit. – Yulia V Oct 30, 2022 at 9:43