mastercurves

Release 0.2

Kyle R. Lennon

Feb 20, 2023

INTRODUCTION

mastercurves is a Python package for automatically superimposing data sets to create a master curve, using Gaussian process regression and maximum a posteriori estimation.

Check out the *Quick Start* section for more information on getting started, including an *Installation* guide.

Note: This project is under active development. Check back for updates!

CHAPTER ONE

CONTENTS

1.1 mastercurves

mastercurves is a Python package for automatically superimposing data sets to create a master curve, using Gaussian process regression and maximum a posteriori estimation.

1.1.1 Citing

This package is based on the methodology outlined in A Data-Driven Method for Automated Data Superposition with Applications in Soft Matter Science. If you use this software, please cite this paper! You can use this BibTex citation:

```
@misc{https://doi.org/10.48550/arxiv.2204.09521,
    doi = {10.48550/ARXIV.2204.09521},
    url = {https://arxiv.org/abs/2204.09521},
    author = {Lennon, Kyle R. and McKinley, Gareth H. and Swan, James W.},
    title = {A Data-Driven Method for Automated Data Superposition with Applications in␣
↪Soft Matter Science},
    publisher = {arXiv},
    year = {2022},
}
```

1.1.2 News

- mastercurves is in the Python Package Index! You can now quickly install it using pip.

1.1.3 Installation

The easiest way to install and use this package is via pip:

```
$ pip install mastercurves
```

You can also use pip to download updates:

```
$ pip install mastercurves --upgrade
```

1.1.4 Contributing

Questions, comments, or suggestions can be raised as issues on GitHub or emailed directly to krlennon[at]mit.edu.

1.2 Automated Data Superposition

Most methods for automated superposition of discrete data sets have two main components. First, they define a continuous interpolant for the data. Then, they define and minimize an objective function based on these interpolants.

The method used in this package (outlined here) uses Gaussian process regression to build statistical interpolants (that come with mean and variance estimates). The objective is the negative logarithm of the posterior probability that one data set is observed from the interpolant for another data set. Therefore, this method is an application of maximum *a posteriori* estimation.

1.2.1 Gaussian Process Regression

Gaussian Process Regression (GPR) is a machine learning method for building a continuous model of a data set. GPR treats each data point in a data set as observations y of (not necessarily independent) random variables. These random variables may be concatenated to form a random vector, or a stochastic process parameterized by the independent coordinate x. If each random variable is assumed to follow a Gaussian distribution, then we call this a Gaussian Process:

$$y(x) \sim GP(\mu(x), K(x, x'; \theta))$$

The Gaussian Process model is defined by a mean function $\mu(x)$, which is usually assumed to be zero, and a covariance kernel *K(x,x';theta)*, which defines the pointwise variance at x and pairwise covariance between x and x'. We typically choose a form of the covariance kernel that encodes prior expectations about the data, such as some degree of smoothness, scale, or uncorrelated variance. The default kernel in this package is:

$$K(x, x'; \theta) = A\left(1 + \frac{(x - x')^2}{2\alpha l^2}\right)^{-\alpha} + B + \sigma^2 \delta_{x,x'}$$

where the set of parameters $\theta = [A, \alpha, l, B, \sigma]$ are called *hyperparameters*. The values of these hyperparameters are determined by maximizing the marginal log-likelihood of the data given the covariance kernel, with some prior specified over the hyperparameters.

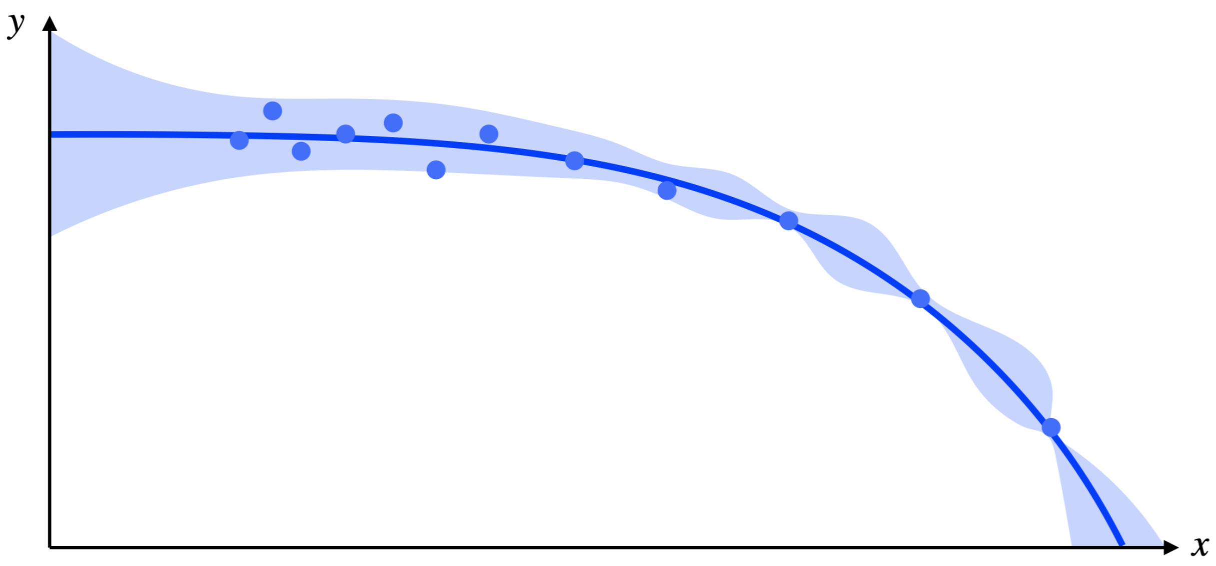
Once the covariance kernel is set and hyperparameters are optimized, the *posterior* mean $m(x)$ and variance $s^2(x)$ of the Gaussian Process model are fixed:

$$m(x^*) = \underline{K}(x^*, \underline{x})\mathbf{K}^{-1}\underline{y}$$

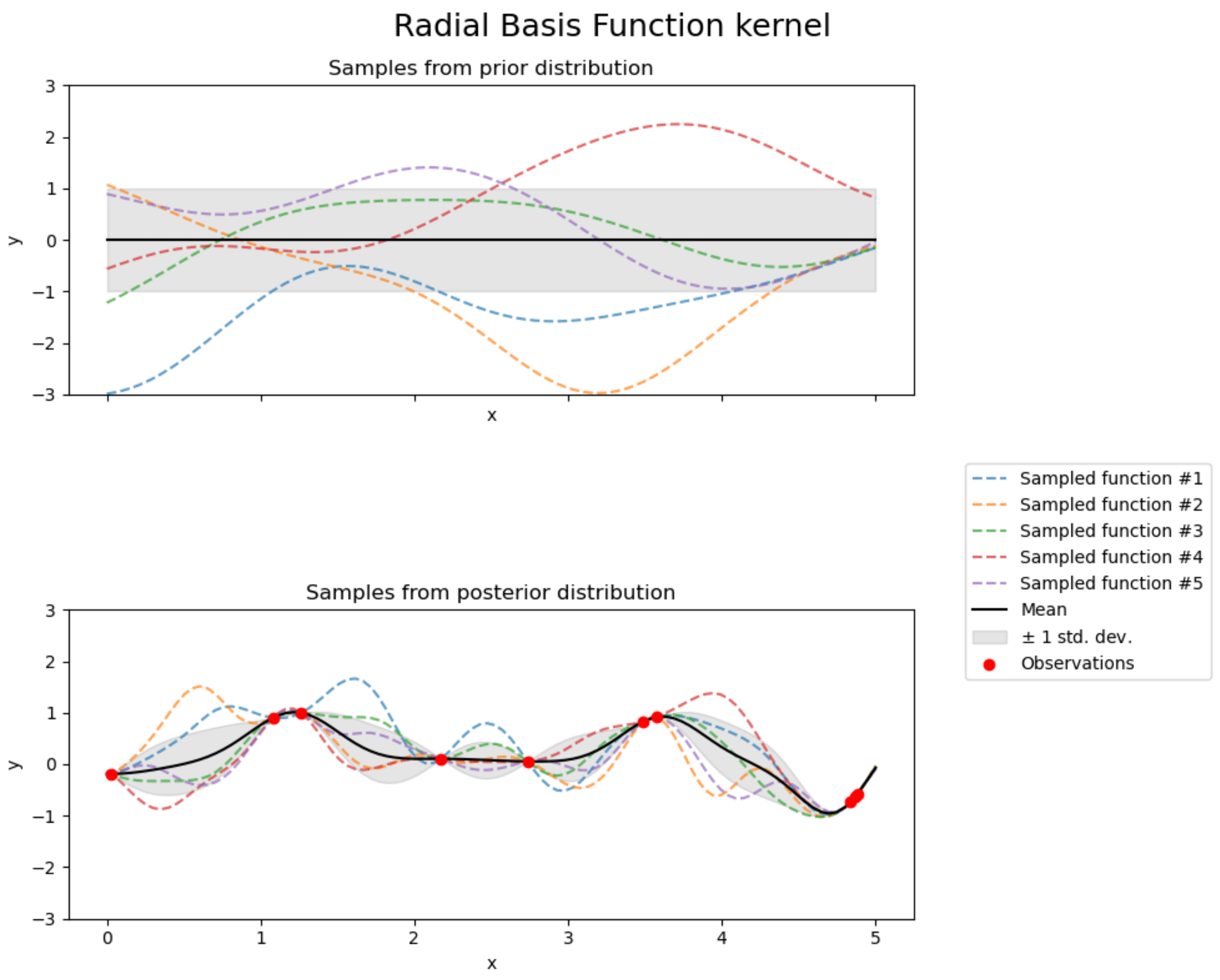
$$s^2(x^*) = K(x^*, x^*; \theta) - \underline{K}(x^*, \underline{x})\mathbf{K}^{-1}\mathbf{K}^T$$

where x^* is an arbitrary coordinate of interest, $\underline{K}(x^*, \underline{x})$ is a vector obtained by evaluating the $K(x^*, x'; \theta)$ with x' taking on each value present in the data set, $\mathbf{K}$ is a matrix obtained by evaluating $K(x, x'; \theta)$ with both x and x' taking on each value present in the data set, and $\underline{y}$ is a vector consisting of each value of y present in the data set.

A trained Gaussian Process model predicts a smooth interpolant with uncertainty bounds, which looks qualitatively like the example presented below.



It is typical for GPR to be explained abstractly as specifying a *prior distribution* of functions that could represent a data set, and applying the data set to obtained the *posterior distribution* of functions drawn from that prior that are most likely to fit the data. For instance, the prior and posterior for a Radial Basis Function kernel (the first term in our kernel) with an example data set are presented below (taken from sklearn).



1.2.2 Maximum A Posteriori Estimation

Consider the problem of shifting a single data point (x_i, y_i) by a horizontal scale factor a and a vertical shift factor b. Say that this data point belongs to a data set at state j, and we'd like to choose a and b so that this data point overlaps with a GP model from state k. We begin by assuming that this shifted data point is an observation from the GP:

$$by_i \sim N(m_k(ax_i), s_k^2(ax_i))$$

The *negative log-likelihood (NLL)* of this observation is:

$$-\ln p(x_i, y_i|a, b) = \frac{1}{2}\left(\frac{by_i - m_k(ax_i)}{s_k(ax_i)}\right)^2 + \ln s_k(ax_i) + \ln\sqrt{2\pi}$$

Now, consider generalizing this task to an entire data set $\mathcal{D}_j = \{(x_i, y_i)\}$. The NLL for this entire set is the sum of the NLLs of the individual data points:

$$-\ln p(\mathcal{D}_j|a, b) = \sum_{(x_i,y_i)\in\mathcal{D}_j} -\ln p(x_i, y_i|a, b)$$

Finally, to obtain the *posterior* probability of certain shift factors given the data, we apply Bayes' theorem:

$$p(a, b|\mathcal{D}_j) \propto p(\mathcal{D}_j|a, b)p(a, b)$$

with a suitable choice of the *prior* over the shift factors, $p(a, b)$. We typically employ either a uniform prior or a Gaussian prior, though others may be implemented within this package.

The maximum *a posteriori* estimate for the shift factors is that which minimizes the negative logarithm of the posterior:

$$\{\hat{a}, \hat{b}\} = \arg\min_{a,b} [-\ln p(a, b|\mathcal{D}_j)]$$

By repeating this estimation for each consecutive pair of data sets, we may construct the master curve.

1.3 Quick Start

1.3.1 Installation

To use mastercurves, first install it using pip:

```
$ pip install mastercurves
```

1.3.2 Creating a master curve

First import the package, then use the `MasterCurve()` constructor to initialize a master curve object:

```
from mastercurves import MasterCurve
mc = MasterCurve()
```

1.3.3 Adding data to a master curve

Next, collect data into three lists: `x_data`, `y_data`, and `states`. The elements of `x_data` and `y_data` should be arrays containing the x- and y-coordinates for a single state (i.e. one data set, which will be superposed with data sets comprising the other elements of `x_data` and `y_data`). The elements of `states` should be numeric values labeling the corresponding states.

When the data is ready, add it to the master curve:

```
mc.add_data(x_data, y_data, states)
```

1.3.4 Defining the coordinate transformations

Then, add coordinate transformations to the master curve. If only horizontal shifting by a scale factor is required (the typical case for time-temperature superposition), this can be done as follows:

```
from mastercurves.transforms import Multiply
mc.add_htransform(Multiply())
```

1.3.5 Superposing the data

The master curve is now ready for superposition:

```
mc.superpose()
```

1.3.6 Plotting the results

Once superposition is complete, you can generate plots of the raw data, data with Gaussian process interpolants, and the superposed mastercurve!

```
mc.plot()
```

1.4 Tutorial: Creating a Master Curve

This tutorial will explore the fundamentals of using the `mastercurves` package to create a master curve from synthetic data describing the one-dimensional diffusion of an instantaneous point source. The tutorial is based on the example in the introduction from A Data-Driven Method for Automated Data Superposition with Applications in Soft Matter Science, and on this issue in the package repository.

1.4.1 Import Packages

Using the `mastercurvse` package will always require importing `numpy` and `matplotlib.pyplot`. It also requires importing the `mastercurves` package itself or some modules from the package. We'll import the essential modules for this tutorial: `mastercurves.MasterCurve` and `mastercurve.transforms.Multiply`.

```
import numpy as np
import matplotlib.pyplot as plt
from mastercurves import MasterCurve
from mastercurves.transforms import Multiply
```

1.4.2 Creating Synthetic Data

Now, let's generate some synthetic data from the diffusion equation. We'll ultimately work with the logarithm of this data, so let's first define an array of positive x coordinates:

```
x = np.linspace(0.1,2)
```

We'll also sample from a few positive values of the time:

```
t_states = [1, 2, 3, 4]
```

We also need a function to compute the concentration at different (x, t) from the diffusion equation:

```
diffusion_eq = lambda x, t, M, D : (M/np.sqrt(4*np.pi*D*t))*np.exp(-(x**2)/(4*D*t))
```

Now, we can create our synthetic data. We'll just assume that $M = 1$ and $D = 1$ in dimensionless units for now:

```
x_data = [x for t in t_states]
c_data = [diffusion_eq(x, t, 1, 1) for t in t_states]
```

Lastly, we'll take the logarithm of our data. The `mastercurves` package can work with the raw data itself for certain cases, but performance is much better (and the package is more flexible) when working with the logarithm of data that will be shifted by `Multiply` transforms (check out the Method section of the associated paper to learn why). So, we should almost always take the logarithm of the data before developing the master curve.

```
x_data = [np.log(xi) for xi in x_data]
c_data = [np.log(ci) for ci in c_data]
```

1.4.3 Creating the Master Curve

We're now ready to create a master curve and superpose the data. The first step is to initialize a `MasterCurve` object. Because our synthetic data is noiseless, we'll create a `MasterCurve` with no fixed noise:

```
mc = MasterCurve(fixed_noise = 0)
```

The next step is to add the data to the `MasterCurve`:

```
mc.add_data(x_data, c_data, t_states)
```

In the diffusion equation, there is both a dynamic concentration scale and a dynamic length scale. This means that we'll need to shift both the horizontal and vertical axes by a time-dependent multiplicative shift factor to superpose the data. We can add these `Multiply` transforms (note that we could pass the argument `scale="log"` to these transforms to indicate that we're working with the logarithm of our data, but this is not necessary since `"log"` is the default scale):

```
mc.add_htransform(Multiply())
mc.add_vtransform(Multiply())
```

Finally, we'll superpose the data using these transforms:

```
mc.superpose()
```

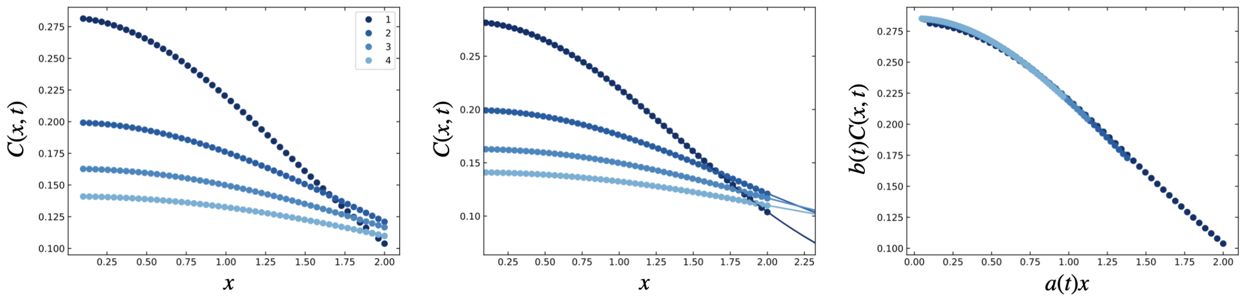
1.4.4 Plotting the Master Curve

We can use the built-in `plot` method to graphically display the data, Gaussian process fits, and master curve:

```
fig1, ax1, fig2, ax2, fig3, ax3 = mc.plot(colormap = lambda i: plt.cm.Blues_r(i/1.5))
```

We've passed a `colormap` argument to this method to define a custom colormap to more closely match the figures in the paper. The value of this argument can be any colormap from `matplotlib.pyplot.cm`.

By default, the `plot` method will display the data on a logarithmic scale. Here, we'll adjust to a linear scale to more closely mimic the figures in the paper (using the `ax.set_xscale` and `ax.set_yscale` methods). You can see the results below, which show the raw data (left), data with Gaussian process fits (center), and master curve (right).



1.4.5 Analyzing the Shift Factors

An important feature of the `mastercurves` package is that we may analyze the shift factors used to superpose the data. These shift factors are stored as attributes of the `MasterCurve` object. We can grab them directly from the object:

```
a = mc.hparams[0]
b = mc.vparams[0]
```

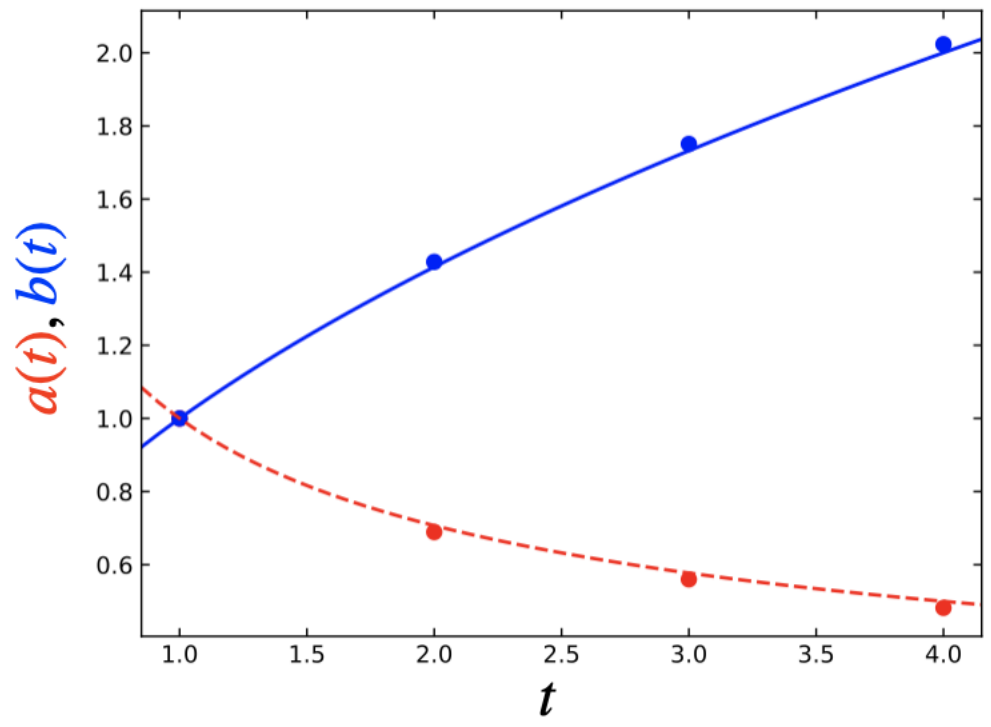
Note that we take the zeroth (0) element of the `hparams` and `vparams` attributes. This is because these attributes store the shift factors for each transformation added to the `MasterCurve`, and there may be more than one transformation. These shift factors are stores as a list, with each element containing the inferred shift parameters for each transform. We only have one horizontal and one vertical transform here, so `mc.hparams` and `mc.vparams` each have only one element. For `mc.hparams`, that element is the list of horizontal shift factors for each state (or the vertical shift factors for `mc.vparams`).

We can plot these shift factors against the state coordinate, t:

```
fig, ax = plt.subplots(1,1)
ax.plot(t_states, a, 'ro')
ax.plot(t_states, b, 'bo')
```

Based on the diffusion equation, we expect that these shift factors should follow specific trends with t, namely that they should vary with the inverse square root and square root of time, respectively. We'll check this by plotting those relationships:

As we see from the plot below, the inferred shift factors indeed closely match the expected behavior!



1.5 Tutorial: Adjusting the Gaussian Process Kernel

This tutorial will outline the steps for changing the covariance kernel used in the Gaussian process fits to data, and for changing bounds on the kernel hyperparameters. We'll use the previous tutorial on the diffusion equation as a base, and make a few modifications to show how to change the kernel.

1.5.1 Changing the Gaussian Process Kernel

`mastercurves` provides a method, `mastercurves.MasterCurve.set_gp_kernel`, which allows the user to manually adjust the Gaussian process kernel if they believe that the default kernel is insufficient for a particular data set. The first step in changing the kernel is to import the desired kernels from `scikit-learn`. See the documentation for `scikit-learn` for a list of supported kernels.

Here, we'll consider a simple modification to the previous tutorial, where we change the `RationalQuadratic` component of the kernel to an `RBF`. We'll need to import the `RBF`, `ConstantKernel`, and `WhiteKernel` kernels:

```
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, ConstantKernel
```

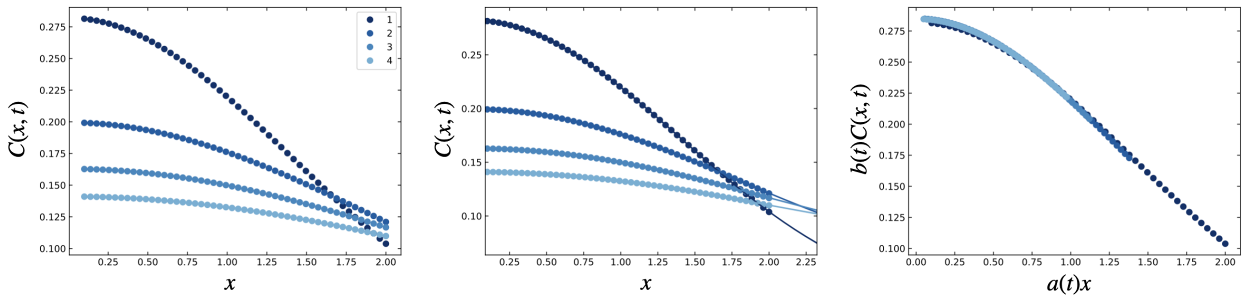
We can now define our new composite kernel:

```
kernel = (RBF() * ConstantKernel() + ConstantKernel() + WhiteKernel())
```

To implement this kernel, we just need to add one line setting the kernel before we call `superpose`:

```
mc.set_gp_kernel(kernel)
mc.superpose()
```

We can see below that the results are not substantially affected by the change in kernel. This is a good sign, as the master curve should be determined by the data, rather than the interpolant selected for the data. The default kernel has been selected to provide flexibility so that many different shapes are well represented by the Gaussian process models.



1.5.2 Changing the Hyperparameter Bounds

The `mastercurves` package uses the default bounds from `scikit-learn` on the hyperparameters for each kernel function. However, you may run into warnings when superposing your data, like the following (from this issue in the package repository):

```
ConvergenceWarning: The optimal value found for dimension 0 of parameter k1__k1__k1__k1__
↪alpha is close to the specified upper bound 100000.0. Increasing the bound and calling␣
↪fit again may find a better value.
```

An issue like this one may arise from data varies across many orders of magnitude, or has sharp kinks. The `scikit-learn` package is warning-heavy, and results are not typically adversely affected by these types of warnings. However, if you see this warning and suboptimal performance of the package, you can address the warning by defining a kernel with custom bounds.

The process for changing hyperparameter bounds is similar to that for changing the kernel itself. First, import all relevant kernels from `scikit-learn` to redefine the kernel. If you'd like to change the bounds on the default kernel, you should import the following:

```
from sklearn.gaussian_process.kernels import RationalQuadratic, WhiteKernel,␣
↪ConstantKernel
```

Next, define the kernel function, and pass custom bounds to any of the constituent kernels that you'd like. To address the above warning, we have to change the bounds for the `alpha` hyperparameter, which belongs to the `RationalQuadratic` kernel:

```
kernel = (RationalQuadratic(alpha_bounds=(1E-5,1E6)) * ConstantKernel() +␣
↪ConstantKernel() + WhiteKernel())
```

See the documentation for `scikit-learn` to learn more about the hyperparameters associated with each kernel. Note that you may also change the default value for each hyperparameter in a similar manner.

Once the kernel is defined, the steps to adding it to a `MasterCurve` and superposing the data are the same as before:

```
mc.set_gp_kernel(kernel)
mc.superpose()
```

1.6 Time-Temperature Superposition

This demo uses creep compliance data digitized from Plazek (1965). The data is pre-processed to account for changes in compliance scale, so only horizontal shifting by a scale factor is required to superpose the data sets taken at different temperatures.

1.6.1 Import Packages

We'll need a few auxillary packages to run this demo:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as tck
import csv
```

We can either import the mastercurves package directly, or we can import only the modules that we need. Doing the latter is a bit cleaner:

```
from mastercurves import MasterCurve
from mastercurves.transforms import Multiply
```

1.6.2 Loading the Data

Not all of the data sets contain the same number of points, so we'll process the file using the CSV package.

```
T = [97, 100.6, 101.8, 104.5, 106.7, 109.6, 114.5, 125, 133.8, 144.9]
T.reverse()
ts = [[] for i in range(len(T))]
Js = [[] for i in range(len(T))]
with open("data/tts_plazek.csv") as file:
    reader = csv.reader(file)
    next(reader)
    next(reader)
    for row in reader:
        for k in range(len(T)):
            if not row[2*k] == "":
                ts[k] += [float(row[2*k])]
                Js[k] += [float(row[2*k+1])]
```

Notice that we reverse the order of the temperature array, so that the data is organized from high to low temperature. This is because we expect lower temperature data to be shifted leftwards (with $a < 1$). This is the default assumption for `Multiply()` shifts, although it is possible to allow for $a > 1$ as well, by changing the bounds of the transform.

As is typical during the creation of master curves, it will be easiest to shift the logarithm of the data. We'll take the logarithm of both the time and compliance coordinates before adding the data to the master curve.

```
for k in range(len(T)):
    ts[k] = np.log(np.array(ts[k]))
    Js[k] = np.log(np.array(Js[k]))
```

1.6.3 Creating the Master Curve

Once the data is pre-processed, only a few steps are needed to create a master curve. First, initialize a `MasterCurve` object:

```
mc = MasterCurve()
```

Next, add the data and the horizontal, multiplicative transform to the master curve:

```
mc.add_data(ts, Js, T)
mc.add_htransform(Multiply(scale="log"))
```

Note that we have passed the argument `scale="log"` to the `Multiply()` constructor, because the data added to the `MasterCurve` object (i.e. `ts` and `Js`) is the logarithm of the measured variables. `scale="log"` is the default for a `Multiply()` object, but we've shown it explicitly declared here for clarity.

We're now ready to superpose the data!

```
mc.superpose()
```

1.6.4 Plotting the Results

Plotting the master curve is also simple. Now that we've superposed the data, let's first change the reference state to that defined in Plazek (1965). We need to know the current shift factors to do this, so we'll obtain them as follows:

We need the index `0` because `mc.hparams` is a list whose elements are the parameters for each transformation applied to the data. We have only one transformation here, so this list has only one element (this is usually the case). This element (which we store in `a`) is itself a list, whose elements are the horizontal shift factor for each temperature.

We can use these shift factors to change the reference state:

```
mc.change_ref(97)
mc.change_ref(100, a_ref=10**(1.13))
```

Notice that we change the reference state here in two parts. Our method has shifted curves leftwards, but Plazek shifted them rightwards. Changing the reference to the lowest temperature accounts for the difference in direction (by making sure uncertainty propagation moves in the correct direction). The second change of reference simply rescales all of the shift factors (and uncertainties) so that the lowest temperature shift matches that found by Plazek.

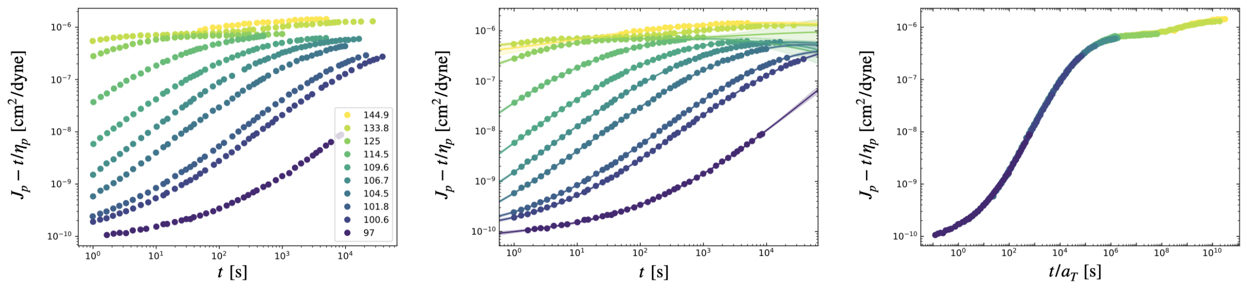
Now, we can plot the raw data, GP models, and master curve:

```
fig1, ax1, fig2, ax2, fig3, ax3 = mc.plot(log=True, colormap=plt.cm.viridis_r)
```

Lastly, let's clean up the plots a little:

```
ax2.tick_params(which="both",direction="in",right=True,top=True)
ax3.tick_params(which="both",direction="in",right=True,top=True)
ax3.xaxis.set_major_locator(tck.LogLocator(base=10.0, numticks=14))
ax3.xaxis.set_minor_locator(tck.LogLocator(base=10.0, subs=[0.1,0.2,0.3,0.4,0.5,0.6,0.7,
↪0.8,0.9,1],numticks=14))
ax3.set_xticklabels(["", "", "", r"$10^{0}$", "", r"$10^{2}$", "", r"$10^{4}$", "", r"
↪$10^{6}$", "", r"$10^{8}$", "", r"$10^{10}$"])
plt.show()
```

The results are shown below!



1.7 Sensitivity to Noisy Data

Here, we'll build on the previous demo (Time-Temperature Superposition) to investigate the sensitivity of the method to noise in the data.

This demo uses creep compliance data digitized from Plazek (1965). The data is pre-processed to account for changes in compliance scale, so only horizontal shifting by a scale factor is required to superpose the data sets taken at different temperatures.

1.7.1 Adding Noise to Data

Most of the code needed to run this example is the same as for the TTS demo. We'll slightly modify how we pre-processs the data, by adding Gaussian White Noise. We'll set this up by choosing a relative noise level (ratio of the noise to the signal), and seeding the random number generator (so that we all get the same result):

```
noise = 0.2
np.random.seed(3)
```

Next, we'll add some synthetic noise to the creep compliance, and take the logarithm as before:

```
for k in range(len(T)):
    ts[k] = np.log(np.array(ts[k]))
    Js[k] = np.log(np.array(Js[k])*(1 + noise*np.random.randn(len(Js[k]))))
```

1.7.2 Creating and Plotting the Master Curve

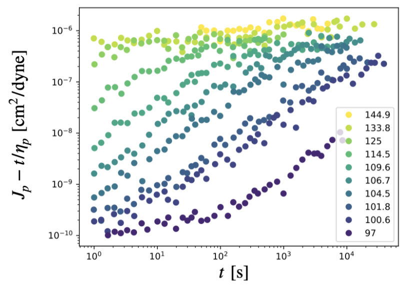
The steps needed to create the master curve and superpose the data are the same as before:

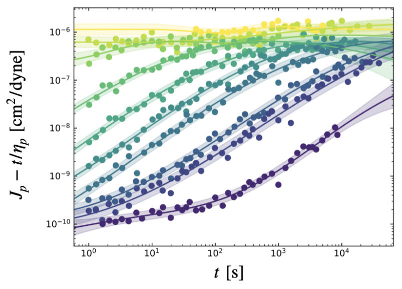
```
# Build a master curve
mc = MasterCurve()
mc.add_data(ts, Js, T)

# Add transformations
mc.add_htransform(Multiply())

# Superpose
mc.superpose()
```

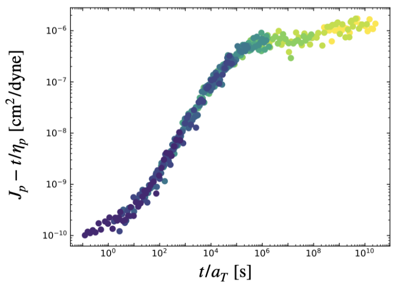
Plotting the master curve is also the same as before. We'll change the reference state, plot, and then beautify the plots (shown in the previous demo, but not here):

```
a = mc.hparams[0]
mc.change_ref(97)
mc.change_ref(100, a_ref=10**(1.13))
figs_and_axes = mc.plot(log=True, colormap=plt.cm.viridis_r)
```

The results are shown below!







1.7.3 Comparing Shift Factors

Let's see how the added noise has affected the shift factors. First, we'll obtain the uncertainties in the shift factors:

```
da = mc.huncertainties[0]
```

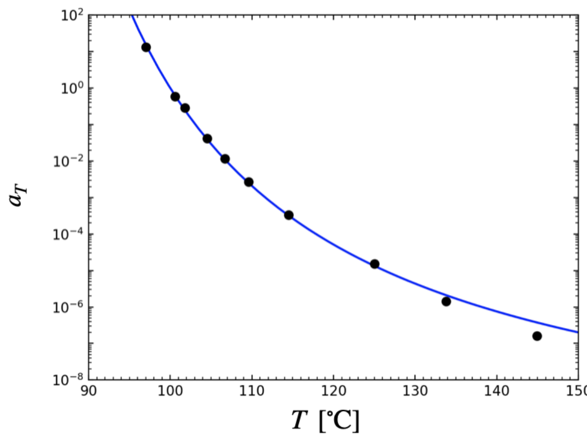
and use these to plot error bars on the shift factors vs. temperature:

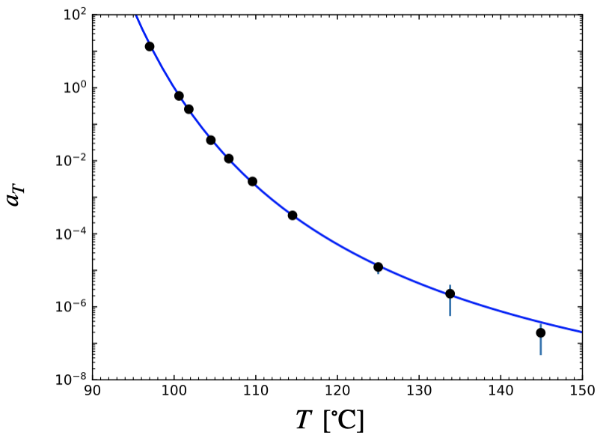
```
fig, ax = plt.subplots(1,1)
ax.errorbar(np.array(T), a, yerr=da, ls='none', marker='o', mfc='k', mec='k')
```

We can also compare the shift factors to the WLF equation with coefficients found by Plazek for this data set:

```
Tv = np.linspace(90,150)
ax.semilogy(Tv, 10**(-10.7*(Tv - 100)/(29.9 + Tv - 100)), 'b')
plt.show()
```

The shift factors agree with the WLF equation very closely. We also see that the shift factors for data with no added noise (left) are very lose to those with added noise (right). The noise doesn't substantially affect the performance of the method, but is instead directly transduced to larger uncertainties in the shift factors (as seen in the larger error bars in the plot on the right).





1.8 MasterCurve

class mastercurves.mc.**MasterCurve**(*fixed_noise=0.04*)

Class definition for a master curve, consisting of multiple data sets superimposed.

A *MasterCurve* object will contain all of the data used to construct a master curve, along with the coordinate transformations, parameters for coordinate transformations (such as shift factors), associated uncertainties, Gaussian process models, and transformed data.

Attributes:

xdata (list[array_like]): list whose elements are the independent coordinates for a data set at a given state

ydata (list[array_like]): list whose elements are the dependent coordinates for a data set at a given state

states (list[float]): list whose elements are the value of the state defining each data set

htransforms (list[Transform]): list whose elements are the coordinate transforms performed on the independent coordinate (i.e. horizontally)

hparam_names (list[string]): list whose elements are the names of the parameters for the horizontal transforms

hbounds (list[tuple]): list whose elements are the upper and lower bounds for the horizontal transformation parameters

hshared (list[bool]): list whose elements indicate whether the corresponding element of hparam_names is a parameter whose value is shared across all states (True) or takes on an independent value for each state (False)

hparams (list[list[float]]): list whose elements are the values of the horizontal transformation parameters at each state

huncertainties (list[float]): list whose elements are the uncertainties associated with the values in hparams

vtransforms (list[Transform]): list whose elements are the coordinate transforms performed on the dependent coordinate (i.e. vertically)

vparam_names (list[string]): list whose elements are the names of the parameters for the vertical transforms

vbounds (list[tuple]): list whose elements are the upper and lower bounds for the vertical transformation parameters

vshared (list[bool]): list whose elements indicate whether the corresponding element of vparam_names is a parameter whose value is shared across all states (True) or takes on an independent value for each state (False)

vparams (list[list[float]]): list whose elements are the values of the vertical transformation parameters at each state

vuncertainties (list[float]): list whose elements are the uncertainties associated with the values in vparams

kernel (sklearn.gaussian_process.kernels.Kernel): kernel function for the Gaussian process model

gps (list[sklearn.gaussian_process.GaussianProcessRegressor]): list whose elements are Gaussian process models for each state `xtransformed` (`list[array_like]`): list whose elements are the transformed independent coordinates for a data set at a given state

`ytransformed` (`list[array_like]`): list whose elements are the transformed dependent coordinates for a data set at a given state

`__init__`(*fixed_noise=0.04*)

Initialize a MasterCurve object.

Args:

`fixed_noise` (`float`): the fixed noise level for the Gaussian process models, corresponding to experimental uncertainty not evident in the data. By default, the noise level is 0.04.

`add_data`(*xdata_new*, *ydata_new*, *states_new*)

Add a data set or data sets to the master curve.

Args:

`xdata_new` (`array_like` or `list[array_like]`): array(s) corresponding to the dependent coordinates for given states

`ydata_new` (`array_like` or `list[array_like]`): array(s) corresponding to the independent coordinates for given states

`states_new` (`float` or `list[float]`): values of the state parameter corresponding to the data in `xdata_new` and `ydata_new`

`add_htransform`(*htransform*)

Add a horizontal transformation (or series of sequential transformations) to the master curve.

Args:

`htransform` (`Transform`): object of a Transform class, which implements the coordinate transformation and stores information about transformation parameters

`add_vtransform`(*vtransform*)

Add a vertical transformation (or series of sequential transformations) to the master curve.

Args:

`vtransform` (`Transform`): object of a Transform class, which implements the coordinate transformation and stores information about transformation parameters

`change_ref`(*ref_state*, *a_ref=1*, *b_ref=1*)

Change the reference state for the master curve.

Args:

`ref_state` (`float`): the new reference state, which may or may not be one of the current states

`a_ref` (`float`): if `ref_state` is not a current state, must be provided. This is the new reference's horizontal shift with respect to the current reference. Defaults to 1.

`b_ref` (`float`): if `ref_state` is not a current state, must be provided. This is the new reference's vertical shift with respect to the current reference. Defaults to 1.

`clear`()

Clear the master curve's data (useful for memory management).

`hpopt`(*lamh=None*, *lamv=None*, *npoints=100*, *alpha=0.5*, *folds=10*)

Perform hyperparameter optimization on the prior (regularization) using MCCV.

Args:

`lamh` (`list[tuple[float]]`): ranges (tuples) for the horizontal hyperparameter search. If not searching this parameter, then the entry should be None.

lamv (list[tuple[float]]): ranges (tuples) for the vertical hyperparameter search. If not searching this parameter, then the entry should be None.

npoints (int): number of grid points to search. Default is 100.

alpha (float): keep rate for MCCV. Default is 0.5.

folds (int): number of MCCV folds. Default is 10.

Returns:

lamh_opt (list[float]): optimal horizontal hyperparameters at each state

lamv_opt (list[float]): optimal vertical hyperparameters at each state

output_table(*file=None*)

Write a csv file with a table of all parameters (and return as a data frame).

Args:

file (string): (optional) path to the file to which the data frame will be written

Returns:

df (pandas.DataFrame): data frame containing the transformation parameters

plot(*log=True*, *colormap=matplotlib.pyplot.cm.tab10*, *colorby='index'*)

Plot the data, GPs, and master curve.

Args:

log (bool): whether the data represents the logarithm of the measured quantity. Defaults to True.

colormap (matplotlib.colors.Colormap): colormap for plotting. Defaults to the tab10 colormap.

colorby (string): how to color the data. Options are index for coloring by index, or state for coloring by the value of the state. Defaults to index.

Returns:

fig1, ax1 (matplotlib.Figure and matplotlib.axes.Axes): the figure and axes objects displaying the raw (untransformed data)

fig2, ax2 (matplotlib.Figure and matplotlib.axes.Axes): the figure and axes objects displaying the untransformed data and GP models

fig3, ax3 (matplotlib.Figure and matplotlib.axes.Axes): the figure and axes objects displaying the superposed data (i.e. the master curve)

set_gp_kernel(*kernel*)

Set the kernel function for the Gaussian Processes used to fit the data.

Args:

kernel (sklearn.gaussian_process.kernels.Kernel): the (potentially composite) kernel function

superpose(*lamh=None*, *lamv=None*)

Optimize the transformations to superpose the data sets onto a single master curve.

Args:

lamh (list[float]): hyperparameters for the horizontal shifts, corresponding to each state in the master curve. Defaults to None, meaning a uniform (unregularized) prior.

lamv (list[float]): hyperparameters for the vertical shifts, corresponding to each state in the master curve. Defaults to None, meaning a uniform (unregularized) prior.

Returns:

loss_min (float): the minimum loss computed during superposition

1.9 Transforms

This package comes with two built-in coordinate transformations, `Multply` and `PowerLawAge`. Users can define custom coordinate transformations by following the structure of these class definitions.

1.9.1 Multiply

class `mastercurves.transforms.multiply.`**`Multiply`**(*bounds=(0.01, 1)*, *scale='log'*, *prior='uniform'*)

Class definition for a multiplicative shift.

Attributes:

`bounds` (`tuple[float]`): the bounds for the shift factor

`default` (`float`): default value of the shift factor (1)

`prior` (`p, lam -> float`): the prior distribution over the shift factor. Either Gaussian or uniform.

`scale` (`string`): coordinate scale, either log or linear

`shared` (`bool`): `False`, since the shift factors are not shared between states

`type` (`string`): *`Multiply`*

`__init__`(*bounds=(0.01, 1)*, *scale='log'*, *prior='uniform'*)

Initialize the Multiply object.

Args:

`bounds` (`tuple[float]`): the upper and lower bounds for the shift factors. Defaults to (1E-2,1).

`scale` (`string`): either "log" for a shift in the logarithm of a variable or "linear" for the shift in the variable. Defaults to "log".

`prior` (`string`): either "uniform" for a uniform prior over the shift factors or "Gaussian" for a Gaussian prior. Defaults to "uniform".

`backward`(*param*, *state*, *data*)

Run a forward shift on the data (from the reference state to the current state).

Args:

`param` (`float`): value of the shift factor for this state

`state` (`float`): value of the state parameter for this data set

`data` (`array_like`): coordinates to be shifted

Returns:

`transformed` (`array_like`): the transformed coordinates

`forward`(*param*, *state*, *data*)

Run a forward shift on the data (from the current state to the reference state).

Args:

`param` (`float`): value of the shift factor for this state

`state` (`float`): value of the state parameter for this data set

`data` (`array_like`): coordinates to be shifted

Returns:

`transformed` (`array_like`): the transformed coordinates

1.9.2 PowerLawAge

class `mastercurves.transforms.powerlawage.`**`PowerLawAge`**(*tref*, *scale='log'*)

Class definition for a power law aging shift.

Attributes:

`bounds` (`tuple[float]`): the bounds for the aging exponent

`default` (`float`): default value for the aging exponent (1.1)

`param` (`string`): name of the aging exponent ("mu")

`prior` (`p, lam -> float`): the prior distribution over the aging exponent. Only a uniform prior currently supported.

`scale` (`string`): coordinate scale, either log or linear

`shared` (`bool`): attr:*True*, since the aging exponent is state-independent

`tref` (`float`): the reference time

`type` (`string`): *`PowerLawAge`*

`__init__`(*tref*, *scale='log'*)

Initialize the PowerLawAge object.

Args:

`tref` (`float`): the reference time

`scale` (`string`): the scale of the time axis ("log" or "linear"). Defaults to "log"

`backward`(*param*, *state*, *data*)

Run a backward shift of the data (from effective time to real time).

Args:

`param` (`float`): value of the aging exponent "mu"

`state` (`float`): value of the state parameter (the wait time)

`data` (:attr:'array_like`): the effective coordinate (either xi or log(xi))

Returns:

`transformed` (`array_like`): the transformed real time coordinate (either t or log(t))

`forward`(*param*, *state*, *data*)

Run a forward shift of the data (from real time to effective time).

Args:

`param` (`float`): value of the aging exponent "mu"

`state` (`float`): value of the state parameter (the wait time)

`data` (:attr:'array_like`): the time coordinate (either t or log(t))

Returns:

`transformed` (`array_like`): the transformed effective time coordinate (either xi or log(xi))

INDEX