GP_model_checking_test_cases_rjf_upgrade1

February 10, 2019

1 Gaussian process model checking: test cases

Here we test model checking diagnostics patterned after Bastos-O'Hagan:

```
Leonardo S. Bastos and Anthony O'Hagan,
<a href="https://doi.org/10.1198/TECH.2009.08019"> <i>Diagnostics for Gaussian Process Emulators
Technometrics <b>51</b>, 425 (2009).
```

The diagnostic functions are from the gsum module written by Jordan Melendez.

Last revised 15-Dec-2018 by Dick Furnstahl [furnstahl.1@osu.edu], building on the original notebook by Jordan Melendez and modifications by Daniel Phillips.

1.1 Overview of B&O Model Checking Implementation

Bastos & O'Hagan provide a versatile set of diagnostic tools for testing whether or not a Gaussian process (GP) is a reasonable emulator for an expensive simulator. Our use case is slightly different than theirs. We don't necessarily care about our GPs matching some underlying simulator. Rather, given a set of curves from a hierarchy of simulators, we wish to answer the following questions: 1. Can they reasonably be assumed to be drawn from the same underlying Gaussian process? 2. If so, which Gaussian process? 3. The underlying GP is later used as a model discrepancy, so how can we test its performance against experiment?

These three questions may or may not be decided by diagnostics discussed in B&O, but to find out we must implement their methods! This notebook tests our adaptations of their methods.

1.2 Modules to import

(rjf note: imports in the original notebook that were moved to gsum have been removed.)

```
In [1]: # standard python: see online documentation
        import numpy as np
        import scipy as sp

        # For plotting we use matplotlib; other choices are possible
        import matplotlib as mpl
        import matplotlib.pyplot as plt

        # special imports for python programming: see online documentation
```

```
from itertools import cycle

# scikit-learn machine learning  https://scikit-learn.org/stable/modules/classes.html
from sklearn.gaussian_process import GaussianProcessRegressor
   # see https://scikit-learn.org/stable/modules/gaussian_process.html
   #  for documentation. Main excerpt:
   # The GaussianProcessRegressor implements Gaussian processes (GP) for
   #  regression purposes. For this, the prior of the GP needs to be specified.
   #  The prior mean is assumed to be constant and zero (for normalize_y=False)
   #  or the training datas mean (for normalize_y=True).
   #  The priors covariance is specified by passing a kernel object.
   #  The hyperparameters of the kernel are optimized during fitting of
   #  GaussianProcessRegressor by maximizing the log-marginal-likelihood (LML)
   #  based on the passed optimizer. If the initial hyperparameters should be kept
   #  fixed, None can be passed as optimizer.
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C, WhiteKernel
   # RBF is a particular GP kernel (radial-basis function kernel, aka
   #  squared-exponential kernel).
   # ConstantKernel: "Can be used as part of a product-kernel where it scales the
   #  magnitude of the other factor (kernel) or as part of a sum-kernel, where it
   #  modifies the mean of the Gaussian process."
   # WhiteKernel: "The main use-case of this kernel is as part of a sum-kernel where
   #  it explains the noise-component of the signal. Tuning its parameter corresponds
   #  to estimating the noise-level."

# gsum is the package written by Jordan Melendez
import gsum
from gsum import rbf, default_attributes, cholesky_errors, mahalanobis
from gsum import lazy_property, pivoted_cholesky
from gsum import ConjugateGaussianProcess, ConjugateStudentProcess
from gsum import Diagnostic, GraphicalDiagnostic
```

In [2]:
```
# set rcParams here
mpl.rcParams['figure.dpi'] = 120
```

In [3]:
```
%matplotlib inline
```

1.3 Test case

This test case will generate toy data from the same given GP by sampling a few curves and selecting a set of training points from each curve. Then we fit a GP to the data.

1.3.1 Class definition for model checking of a toy model

In [4]:
```
from math import ceil

class Toy_model:
    """
    Toy model for GP model checking based on Bastos & O'Hagan.
```

```
 Uses functions from the gsum package by Jordan Melendez.
To explore a new toy model:
  1. Make an instance of this class, using default or specified parameters, e.g., ::
       tm1 = Toy_model(name='rjf_no_shift_seed_6_samples_3',
                        x_min=0, x_max=20, x_num=21, data_skip=3, data_offset=2,
                        n_samples=3, n_ref=1000,
                        basevar=1.0, varshiftfactor=1.0, baselengthscale=3.0,
                        lengthscaleshift=0.0,
                        seed=6, nugget_sd = 1e-4,
                        vlines=True, print_level=1
                       )
      There are print methods to output information on the toy model.
  2. Set up the kernels and gps for the models, e.g., ::
       tm1.setup_toy_model()
  3. Fit the data using the test data and compute means and covariancs, e.g., ::
       tm1.fit_toy_model()
  4. Run one or more of the methods to do model checking, e.g., ::
       ``tm1.model_checking_with_test_data_global(plots='Md and pivoted cholesky')
     If you don't specify `plots`, you get the full plotzilla graph.

Parameters
----------
name : str
    Description/label of the particular toy model.
x_min, x_max : float
xnum : int
    Mesh points for the GPs: x_num points from x_min to x_max.  Used by setup_X_full
data_skip, data_offset, data_pts : int
    Training data points: every data_skip points starting with data_offset point.
    Used by setup_mask to create the mask for the training data.
n_samples : int
    Draw n_samples curves
n_ref : int
    Number of diagnostic samples (should this ever change?)
basevar, varshiftfactor, baselengthscale, lengthscaleshift : float
    DP characterization of the GP(s) used to sample the toy data.
    Specifies hyperparameters (hps) var and length scale for each GP.
    Run self.setup_toy_gp_hps to create the toy_gp_hps array.

seed : int
    self.toy_gp_seeds = seed + np.arange(n_samples)  # array of n_samples values
    Here specified as ascending integers starting from an initial seed,
     but they could be specified by hand or randomized

nugget_sd : float
    Nugget as standard deviation

    # Vertical lines (True) or a histogram (False) for the md and kl plots
```

```python
        self.vlines = vlines

    print_level : int
        How much information to print using print_information(print_level).
          0 --- don't print anything
          1 --- all parameters and sample information

    Methods
    -------
    print_parameter_list()
       Print the parameters of the toy model.
    print_sample_table()
       Print the characteristics of the sampled GPs.
    print_information(level=1):
       Print information about the toy model based on level.


    Notes
    -----
    1. Need to ensure that toy models are completely updated when changes
        are made to the parameters.

    """
    def __init__(self,
                 name = '[unnamed]',
                 x_min = 0,
                 x_max = 20,
                 x_num = 41,
                 data_skip = 5,
                 data_offset = 0,    # should be less than data_skip
                 n_samples = 4,
                 n_ref = 1000,
                 basevar = 1.0,
                 varshiftfactor = 1.0,
                 baselengthscale = 3.0,
                 lengthscaleshift = 0.0,
                 seed = 2,
                 nugget_sd = 1e-4,
                 vlines = True,
                 print_level = 1
                ):
        """Specify the toy model by the range and number of points, which
            points are used for training, how many gp curves are sampled and
            with what gp hyperparameters, and what seed and nugget are used.
        """
        self.name = name

        # set up mesh points for the GPs (x_num points from x_min to x_max)
```

```
        self.x_min = x_min
        self.x_max = x_max
        self.x_num = x_num
        self.setup_X_full()

        # training data points (every data_skip points starting with data_offset point)
        self.data_skip = data_skip
        self.data_offset = data_offset
        self.data_pts = ceil(x_num/data_skip)
        self.setup_mask()

        self.n_samples = n_samples   # draw n_samples curves
        self.n_ref = n_ref       # number of diagnostic samples (should this ever change

        # DP characterization of the GP(s) used to sample the toy data.
        #  Specifies hyperparameters (hps) var and length scale for each GP
        self.basevar = basevar
        self.varshiftfactor = varshiftfactor
        self.baselengthscale = baselengthscale
        self.lengthscaleshift = lengthscaleshift
        self.setup_toy_gp_hps()  # create the toy_gp_hps array

        self.seed = seed
        self.toy_gp_seeds = seed + np.arange(n_samples)  # array of n_samples values
        # Here specified as ascending integers starting from an initial seed,
        #  but they could be specified by hand or randomized

        self.nugget_sd = nugget_sd  # Check if we are sensitive to the value

        # Vertical lines (True) or a histogram (False) for the md and kl plots
        self.vlines = vlines

        self.print_level = print_level
        self.print_information(self.print_level)

    def print_parameter_list(self):
        """Print the parameters of the toy model."""
        print('  * mesh for the GPS: {0:d} points from {1:.2f} to {2:.2f}'\
                   .format(self.x_num, self.x_min, self.x_max))
        print('  * training data: every {0:d} points starting with point {1:d};'\
                   .format(self.data_skip, self.data_offset),\
              ' total {0:d} pts'.format(self.data_pts))
        print('  * nugget_sd: {0:.2e}'.format(self.nugget_sd))

    def print_sample_table(self):
        """Print the characteristics of the sampled GPs."""
        self.setup_toy_gp_hps()   # make sure that the hyperparameters are set
        gps_cycle = cycle(np.arange(len(self.toy_gp_hps)))   # go through gps cyclically
```

```python
        print('\n sample #   variance  length scale  seed   color')
        for i in range(self.n_samples):
            gp_index = next(gps_cycle)
            print('   {0:2d}        {1:.2f}        {2:.1f}        {3:2d}'\
                   .format(i, self.toy_gp_hps[gp_index][0], self.toy_gp_hps[gp_index][1]
                   self.toy_gp_seeds[i]))

    def print_information(self, level=1):
        """Print information about the toy model based on level (0-1)"""
        if (level==1):
            print('\n\n', '*'*72)
            print(' Information for toy model: \'{:s}\' '.format(self.name))
            self.print_parameter_list()
            self.print_sample_table()

    def setup_X_full(self):
        """Creates the array of x points (X_full)."""
        self.X_full = np.atleast_2d(np.linspace(self.x_min, self.x_max, self.x_num)).T

    def setup_mask(self):
        """ Creates the mask array for the training data.  Has a True entry if
             corresponding point is in training data, otherwise the entry is False.
        """
        self.mask = np.array([(i-self.data_offset) % self.data_skip == 0 \
                              for i in range(len(self.X_full))])

    def setup_toy_gp_hps(self):
        """Set up the hyperparameters to be used for the different GPs in the toy model.
           Currently based on the DP scheme of specifying baseline values for the
           variance and length scale and then factors to multiply/divide the variance
           and add/substract to the length scale, for a total of three different GPs.
           More generally, one could extend the setup so that
                set toy_gp_hps = [ [var0, ls0], [var1, ls1], ...].
        """
        _bv = self.basevar; _bls = self.baselengthscale;
        _vs = self.varshiftfactor; _ls = self.lengthscaleshift
        self.toy_gp_hps = [ [_bv, _bls],
                            [_bv*_vs, _bls+_ls],
                            [_bv/_vs, _bls-_ls] ]

    def setup_toy_model(self):
        """Set up array of kernels and corresponding GPs for the toy model.
           Currently each kernel is the sum of an RBF kernel scaled by the variance
           and a noise kernel (which uses the nugget_sd nugget).
        """
        self.setup_toy_gp_hps()   # make sure that the hyperparameters are set
        self.toy_gp_kernel = []
        self.toy_gp = []
```

```
        for i in range(len(self.toy_gp_hps)):
            self.toy_gp_kernel.append( C(self.toy_gp_hps[i][0], (1e-3, 1e3)) \
                                * RBF(self.toy_gp_hps[i][1], (1e-2, 1e2)) \
                                + WhiteKernel(self.nugget_sd**2) )
            self.toy_gp.append( GaussianProcessRegressor(kernel=self.toy_gp_kernel[i], o

        # kernel with starting hyperparameters
        self.base_gp_kernel = C(self.basevar, (1e-3, 1e3)) * RBF(self.baselengthscale, (

        # Generate full toy data and split into training and test data.
        #  Sample the gps at all of the X points.
        self.toy_data_full = []       # toy data at X_full
        self.toy_data_training = []   # the points used to train
        self.toy_data_test = []       # the remaining points
        gps_cycle = cycle(np.arange(len(self.toy_gp_hps)))   # sample from gps cyclicall
        for i in range(self.n_samples):
            self.toy_data_full.append( self.toy_gp[next(gps_cycle)].sample_y(
                                                    self.X_full, n_samples=1,
                                                    random_state=self.toy_gp_seeds[i
            self.toy_data_training.append( self.toy_data_full[i][:, self.mask] )
            self.toy_data_test.append( self.toy_data_full[i][:, ~self.mask])
        self.toy_data_full = np.concatenate(self.toy_data_full)
        self.toy_data_training = np.concatenate(self.toy_data_training)
        self.toy_data_test = np.concatenate(self.toy_data_test)

        self.X_training = self.X_full[self.mask]
        self.X_test = self.X_full[~self.mask]


    def fit_toy_model(self):
        """Fit GP hyperparameters for the training data"""
        self.my_gp = ConjugateGaussianProcess(self.base_gp_kernel)
        self.my_gp.fit(self.X_training, self.toy_data_training, noise_sd=self.nugget_sd)

        # compute the mean and covariance of the fitted GP at the training set points
        self.fitmean_training = self.my_gp.mean(self.X_training)
        self.fitcov_training = self.my_gp.cov(self.X_training)

        # compute the values of the fitted GP at all the data points
        self.m_test, self.K_test = self.my_gp.predict(self.X_test,
                                                return_cov=True, pred_noise=True)
        # print(np.diag(K_pred))
        self.sd_test = np.sqrt(np.diag(self.K_test))

        # compute the mean and covariance of the overall GP at the set X_full
        self.fitmean_test = self.my_gp.mean(self.X_test)
        self.fitcov_test = self.my_gp.cov(self.X_test,self.X_test)
```

```
def plot_toy_data_and_fits(self, **kwargs):
    """ Plot the gps, test data, and fits"""
    fig = plt.figure(figsize=(12,4))
    ax1 = fig.add_subplot(1,2,1)
    ax1.plot(self.X_full.ravel(), self.toy_data_full.T);
    ax1.plot(self.X_training.ravel(), self.toy_data_training.T,
             ls='', marker='o', fillstyle='none', markersize=10, c='gray');


    ax2 = fig.add_subplot(1,2,2)
    # Plot the underlying process
    ax2.plot(self.X_training.ravel(), self.my_gp.mean(), ls='--', c='gray')
    ax2.plot(self.X_training.ravel(), self.my_gp.mean() + self.my_gp.sd(), ls=':', c
    ax2.plot(self.X_training.ravel(), self.my_gp.mean() - self.my_gp.sd(), ls=':', c

    # Now the true data
    ax2.plot(self.X_full.ravel(), self.toy_data_full.T);
    ax2.plot(self.X_training.ravel(), self.toy_data_training.T, ls='', marker='o',
             fillstyle='none', markersize=10, c='gray');

    # The predicted interpolants and their errors
    ax2.plot(self.X_test.ravel(), self.m_test.T, c='k', ls='--', label='test');
    for m in self.m_test:
        ax2.fill_between(self.X_test.ravel(), m + 2*self.sd_test, m - 2*self.sd_test
                         color='gray', alpha=0.25)
    # self.ax2.legend();
    return fig, ax1, ax2

def model_checking_with_training_data_only(self, plots='plotzilla'):
    """Perform the model checking and generate plots using training data only.
       The fit mean and covariance are calculated in fit_toy_model.
    """
    np.random.seed(self.seed)
    gp = ConjugateGaussianProcess(self.base_gp_kernel)
    gpmc = Diagnostic(self.fitmean_training, self.fitcov_training)
    gd = GraphicalDiagnostic(gpmc, self.toy_data_training, nref=self.n_ref)
    if plots == 'plotzilla':
        gd.plotzilla(self.X_training, gp, vlines=self.vlines);
    elif plots == 'Md and pivoted cholesky':
        self.plot_Md_and_pivoted_cholesky(self.X_training, gd, vlines=self.vlines);

def model_checking_with_test_data_global(self, plots='plotzilla'):
    """Perform global model checking and generate plots using test data.
       The fit mean and covariance are calculated in fit_toy_model.
    """
    np.random.seed(self.seed)
    gp = ConjugateGaussianProcess(self.base_gp_kernel)
```

```
        gpmc_test = Diagnostic(self.fitmean_test,
                               self.fitcov_test
                               + self.nugget_sd**2 * np.eye(self.fitcov_test.shape[0]))
        gd_test = GraphicalDiagnostic(gpmc_test, self.toy_data_test, nref=self.n_ref)
        if plots == 'plotzilla':
            gd_test.plotzilla(self.X_test, gp, vlines=self.vlines);
        elif plots == 'Md and pivoted cholesky':
            self.plot_Md_and_pivoted_cholesky(self.X_test, gd_test, vlines=self.vlines);

    def model_checking_with_test_data_interpolants(self, plots='plotzilla'):
        """Perform model checking with interpolants and generate plots using test data.
           m_test and K_test are calculated in fit_toy_model.
        """
        np.random.seed(self.seed)
        gp = ConjugateGaussianProcess(self.base_gp_kernel)
        gp.fit(self.X_training, self.toy_data_training, noise_sd=self.nugget_sd)
        self.mean_est, self.cov_est = gp.predict(self.X_test, return_cov=True, pred_nois
        gpmc = Diagnostic(np.zeros(self.m_test.shape[1]), self.K_test \
                          + self.nugget_sd**2 * np.eye(self.K_test.shape[0]))
        gd = GraphicalDiagnostic(gpmc, self.toy_data_test - self.m_test, nref=self.n_ref
        if plots == 'plotzilla':
            gd.plotzilla(self.X_test, gp, predict=True, vlines=self.vlines);
        elif plots == 'Md and pivoted cholesky':
            self.plot_Md_and_pivoted_cholesky(self.X_test, gd, vlines=self.vlines);

    def plot_Md_and_pivoted_cholesky(self, X, gd=None, predict=False, vlines=True):
        """Make a plot with a subset of the plotzilla plots.  Here three plots are
            shown: Mahalanobis distance, pivoted Cholesky errors, and the corresponding
            pivoted Cholesky QQ plot.
        """
        if gd is None:
            pass
        fig, axes = plt.subplots(1, 3, figsize=(12, 3))
        gd.md(vlines=vlines, ax=axes[0])
        gd.pivoted_cholesky_errors(axes[1])
        gd.pivoted_cholesky_errors_qq(axes[2])
        fig.tight_layout()
        return fig, axes
```

1.3.2 Ok, let's roll!

```
In [5]: from copy import deepcopy  # when we want to duplicate a toy model

        # Initiate a toy model (note that we only need to specify values different from defaults

        # this one has the same gps for all the samples
        tm1 = Toy_model(name='rjf_no_shift_seed_6_samples_3',
```

```
                    x_min=0, x_max=40, x_num=41, data_skip=4, data_offset=0,
                    n_samples=3, n_ref=1000,
                    basevar=1.0, varshiftfactor=1.0, baselengthscale=3.0, lengthscaleshift=0.
                    seed=4, nugget_sd = 1e-4,
                    vlines=True, print_level=1
                   )

# this one changes the variances by varshiftfactor (all else the same)
tm2 = deepcopy(tm1)
tm2.name = 'rjf_var_shift_2_seed_6_samples_3'
tm2.varshiftfactor = 2.0
tm2.print_information()

# this one changes the length scales by lengthscaleschift
tm3 = deepcopy(tm1)
tm3.name = 'rjf_scale_shift_0p3_seed_6_samples_3'
tm3.lengthscaleshift=0.3
tm3.print_information()
```

```
***********************************************************************
Information for toy model: 'rjf_no_shift_seed_6_samples_3'
 * mesh for the GPS: 41 points from 0.00 to 40.00
 * training data: every 4 points starting with point 0;  total 11 pts
 * nugget_sd: 1.00e-04

sample #   variance  length scale  seed   color
   0         1.00       3.0         4
   1         1.00       3.0         5
   2         1.00       3.0         6


***********************************************************************
Information for toy model: 'rjf_var_shift_2_seed_6_samples_3'
 * mesh for the GPS: 41 points from 0.00 to 40.00
 * training data: every 4 points starting with point 0;  total 11 pts
 * nugget_sd: 1.00e-04

sample #   variance  length scale  seed   color
   0         1.00       3.0         4
   1         2.00       3.0         5
   2         0.50       3.0         6


***********************************************************************
Information for toy model: 'rjf_scale_shift_0p3_seed_6_samples_3'
 * mesh for the GPS: 41 points from 0.00 to 40.00
```

```
 * training data: every 4 points starting with point 0;  total 11 pts
 * nugget_sd: 1.00e-04

sample #   variance  length scale  seed   color
   0         1.00       3.0         4
   1         1.00       3.3         5
   2         1.00       2.7         6
```

```
In [6]: # Set up the kernels and gps for the models
        tm1.setup_toy_model()
        tm2.setup_toy_model()
        tm3.setup_toy_model()

        # Fit the data using the test data and compute means and covariancs
        tm1.fit_toy_model()
        tm2.fit_toy_model()
        tm3.fit_toy_model()
```

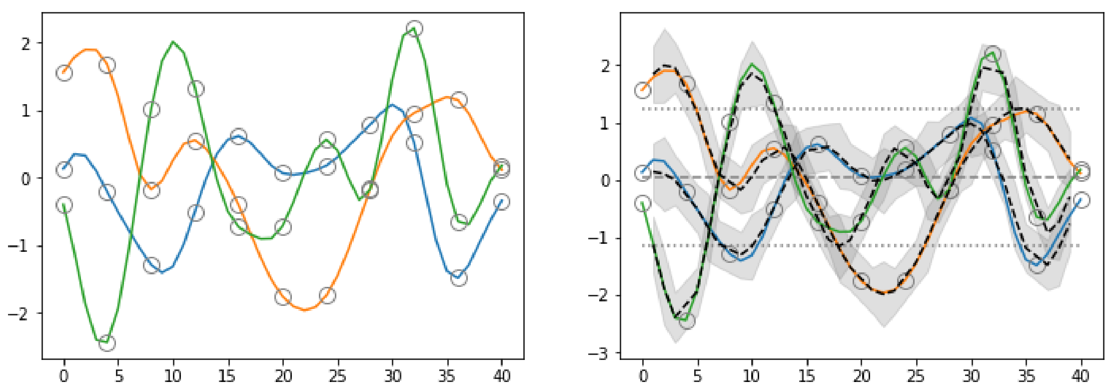
1.3.3 Aside: Do the nugget size check

Here we are varying the nugget for tm3 and checking whether it makes a difference

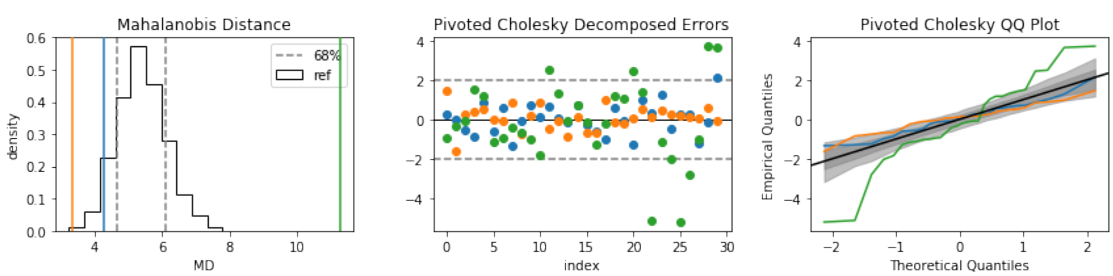
```
In [7]: # make 6 deep (new and complete) copies of tm3 for testing impact of nugget size
        tm_nugget_test = [deepcopy(tm3) for _ in range(6)]
        factors = 2**(np.arange(6))  # start with tm1 nugget and multiply by 2 each time
        for i, tm in enumerate(tm_nugget_test):
            tm.nugget_sd = factors[i] * tm1.nugget_sd
            tm.name = tm1.name + '_nugget_{:.1e}'.format(tm.nugget_sd)
            #tm.print_information()

        # Plot the data and fits for the original model tm3
        tm3.print_information()
        tm3.plot_toy_data_and_fits();
        plt.show()     # this waits until the plot is finished before proceeding

        # set up kernels and fits for the nugget test models
        #  and then generate model checking plotzilla for global test data
        for tm in tm_nugget_test:
            #tm.print_information()
            tm.setup_toy_model()
            tm.fit_toy_model()

            print('\n *** TESTING: nugget_sd is {:.1e} ***'.format(tm.nugget_sd))
            print(' Condition number for covariance matrix is {:.1e}'.format(
                  np.linalg.cond(
                   tm.fitcov_test + tm.nugget_sd**2 * np.eye(tm.fitcov_test.shape[0]) )))
            #tm.model_checking_with_training_data_only(plots='Md and pivoted cholesky')
```

```
print('  Plots for global test data:')
tm.model_checking_with_test_data_global(plots='Md and pivoted cholesky')
plt.show()  # this waits until the plot is finished before proceeding
print('  Plots for interpolant test data:')
tm.model_checking_with_test_data_interpolants(plots='Md and pivoted cholesky')
plt.show()  # this waits until the plot is finished before proceeding
```

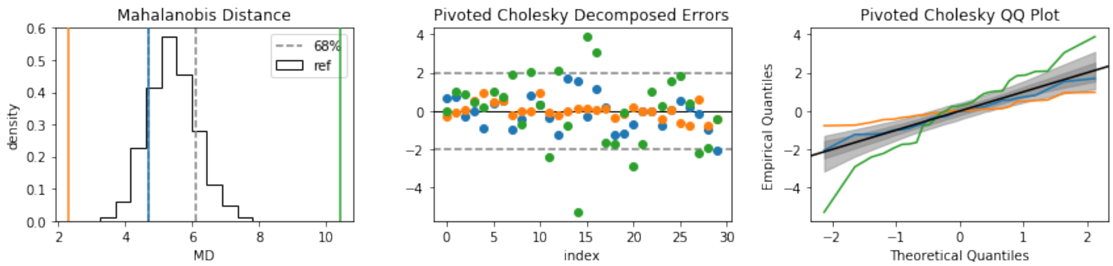
```
************************************************************************
Information for toy model: 'rjf_scale_shift_0p3_seed_6_samples_3'
 * mesh for the GPS: 41 points from 0.00 to 40.00
 * training data: every 4 points starting with point 0;  total 11 pts
 * nugget_sd: 1.00e-04

sample #   variance  length scale  seed   color
   0         1.00         3.0        4
   1         1.00         3.3        5
   2         1.00         2.7        6
```



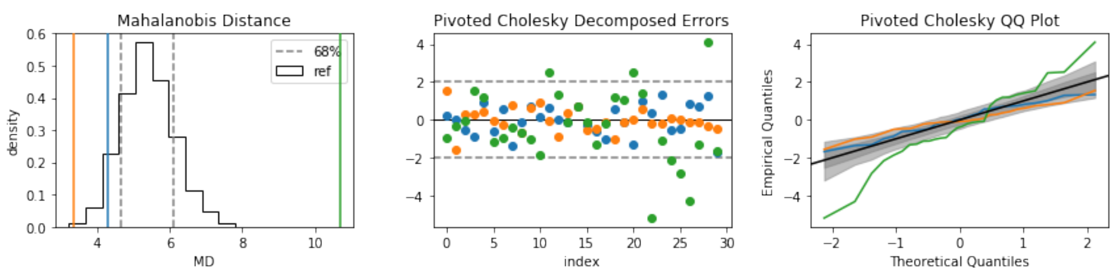
```
*** TESTING: nugget_sd is 1.0e-04 ***
Condition number for covariance matrix is 7.0e+08
 Plots for global test data:
```



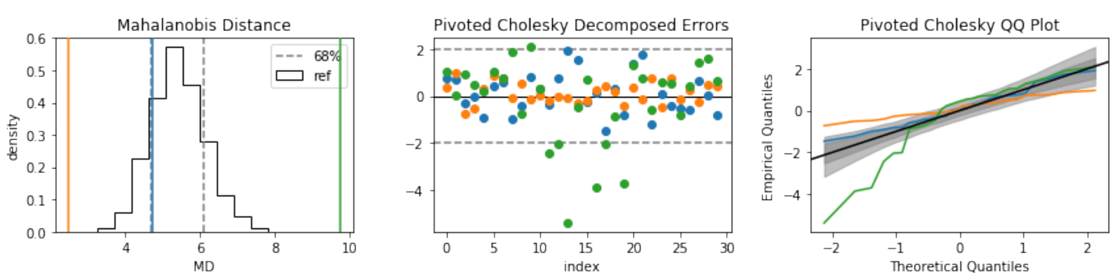
Plots for interpolant test data:



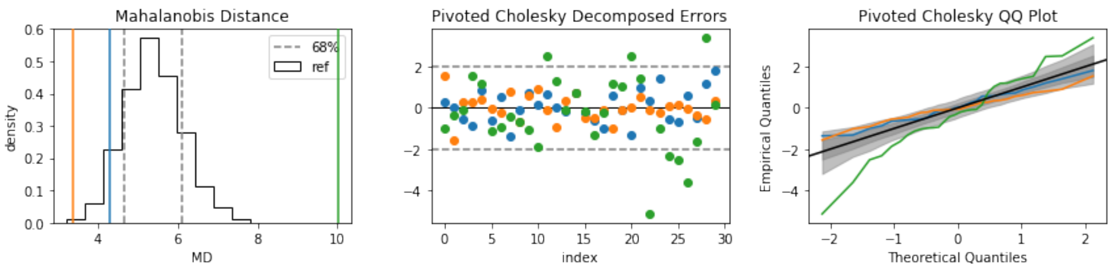
*** TESTING: nugget_sd is 2.0e-04 ***
Condition number for covariance matrix is 1.9e+08
Plots for global test data:



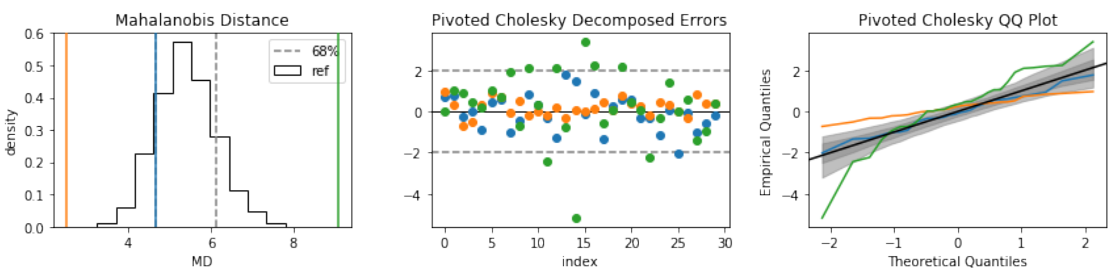
Plots for interpolant test data:



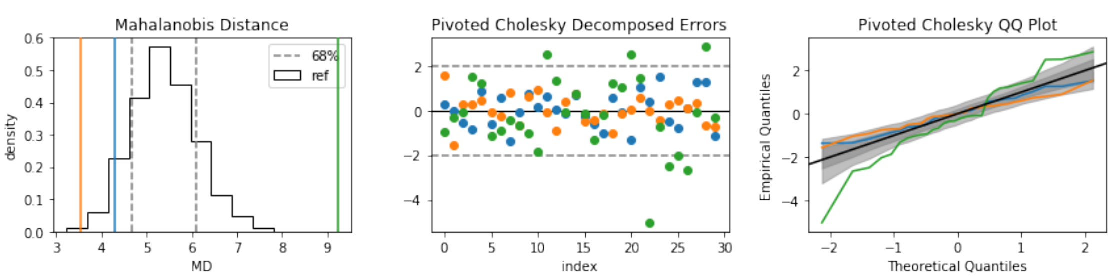
```
*** TESTING: nugget_sd is 4.0e-04 ***
Condition number for covariance matrix is 4.8e+07
 Plots for global test data:
```



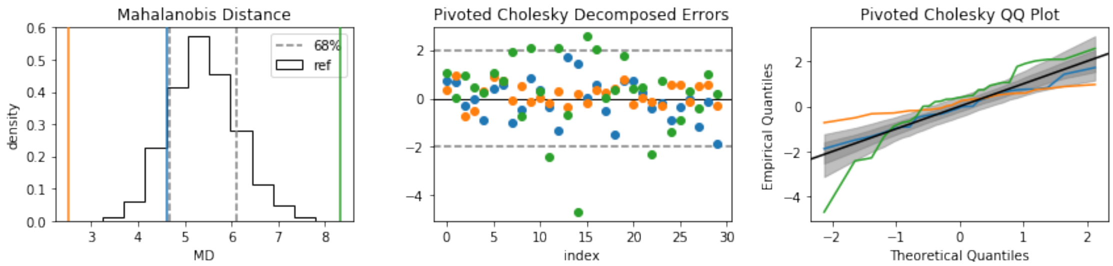

```
 Plots for interpolant test data:
```



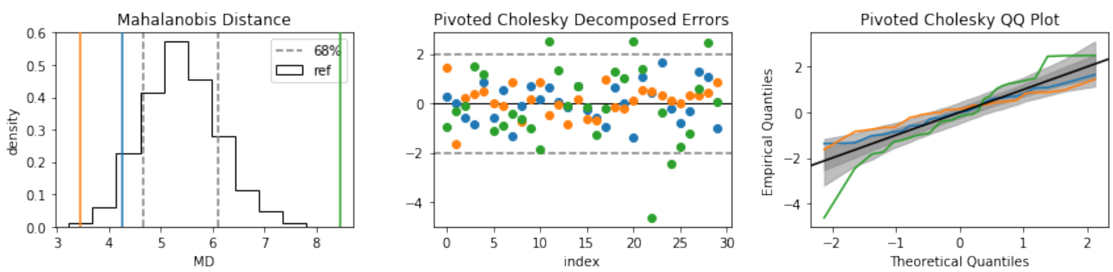

```
*** TESTING: nugget_sd is 8.0e-04 ***
Condition number for covariance matrix is 1.2e+07
 Plots for global test data:
```



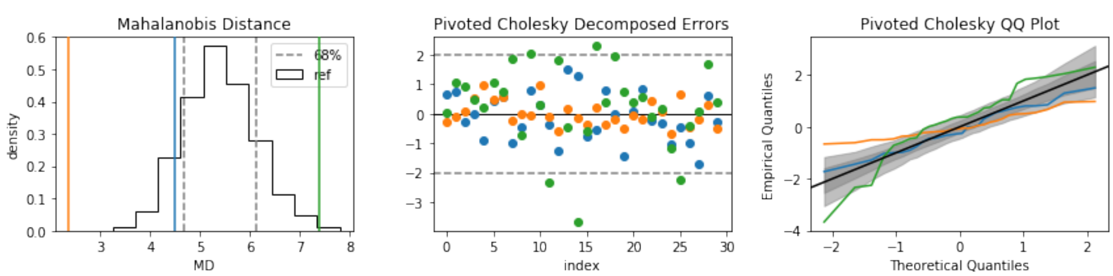
Plots for interpolant test data:



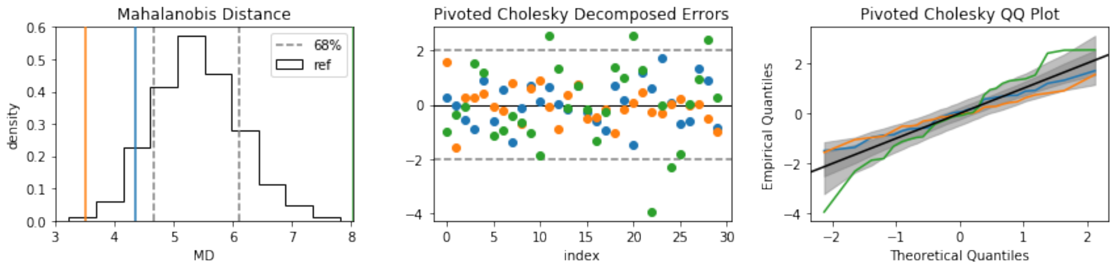
*** TESTING: nugget_sd is 1.6e-03 ***
Condition number for covariance matrix is 3.1e+06
Plots for global test data:



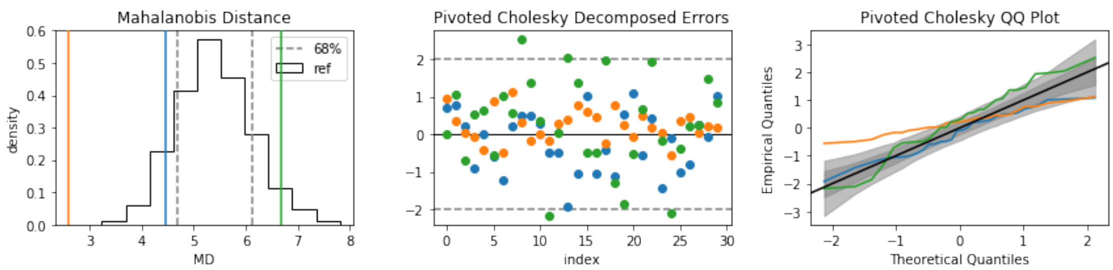
Plots for interpolant test data:



*** TESTING: nugget_sd is 3.2e-03 ***
Condition number for covariance matrix is 7.5e+05
Plots for global test data:



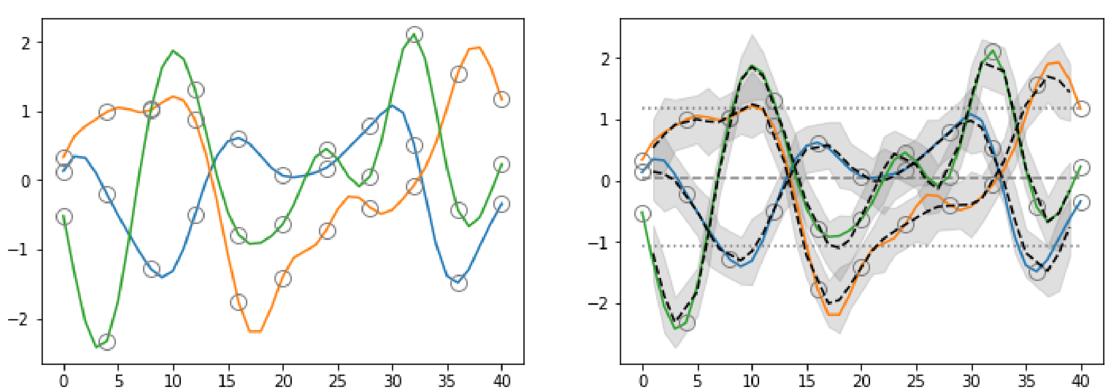
Plots for interpolant test data:



1.3.4 Plot the toy data and fits

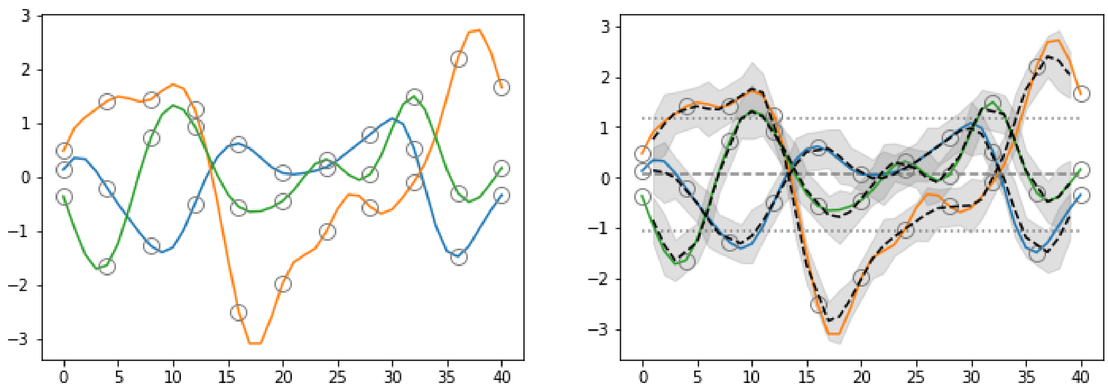
```
In [8]: print(tm1.name)
        tm1.plot_toy_data_and_fits();
```

```
rjf_no_shift_seed_6_samples_3
```



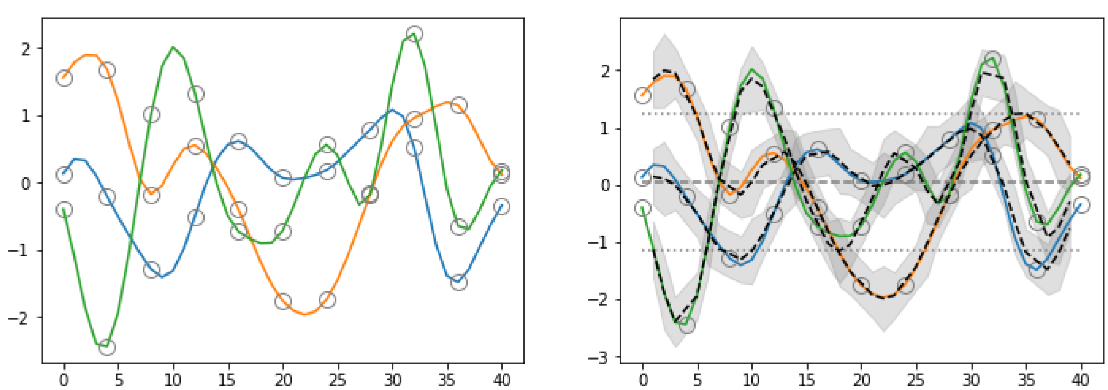
```
In [9]: print(tm2.name)
        tm2.plot_toy_data_and_fits();
```

```
rjf_var_shift_2_seed_6_samples_3
```



```
In [10]: print(tm3.name)
         tm3.plot_toy_data_and_fits();
```

```
rjf_scale_shift_0p3_seed_6_samples_3
```

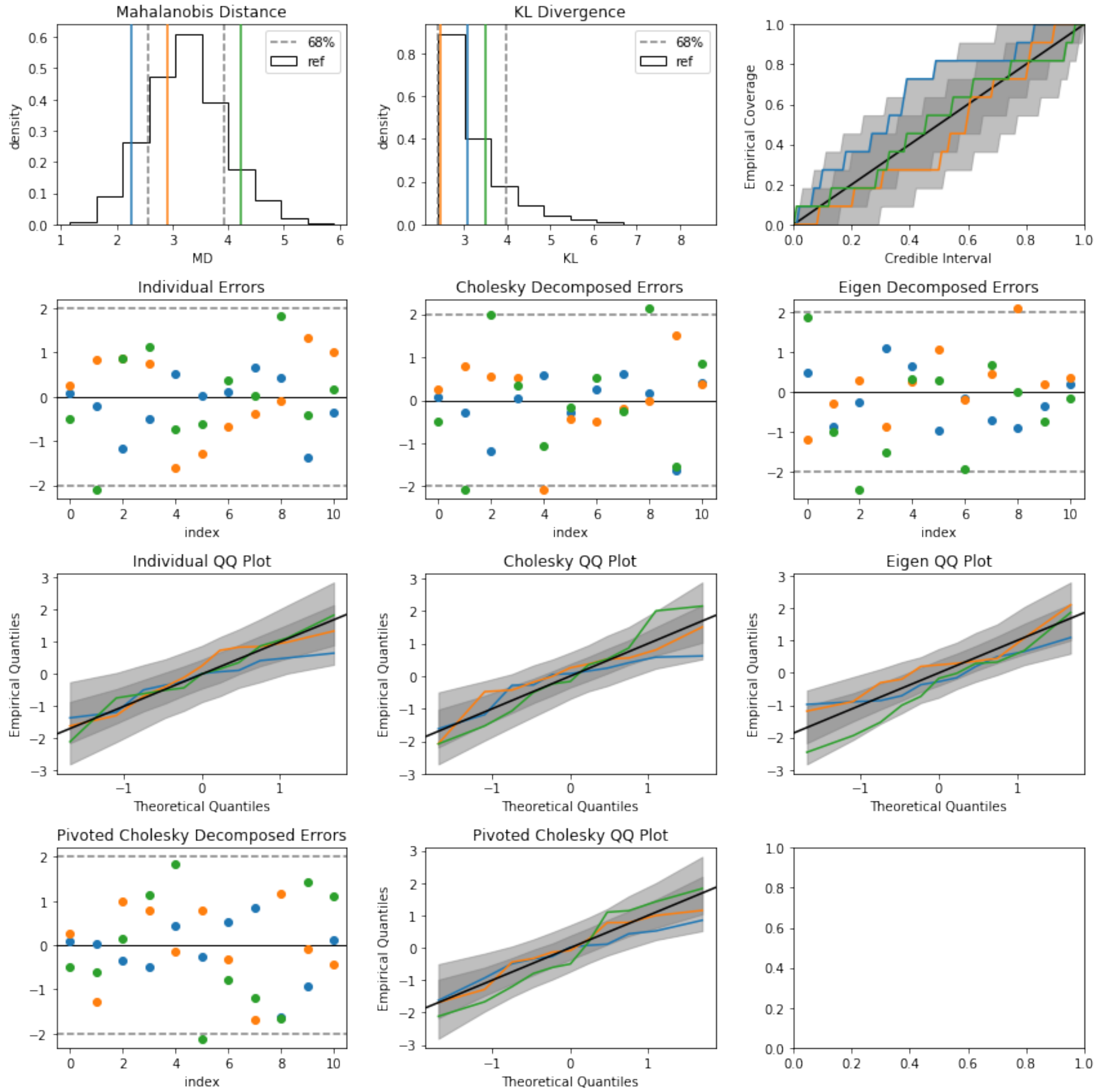


Comments:

1.3.5 Model checking with the training data only

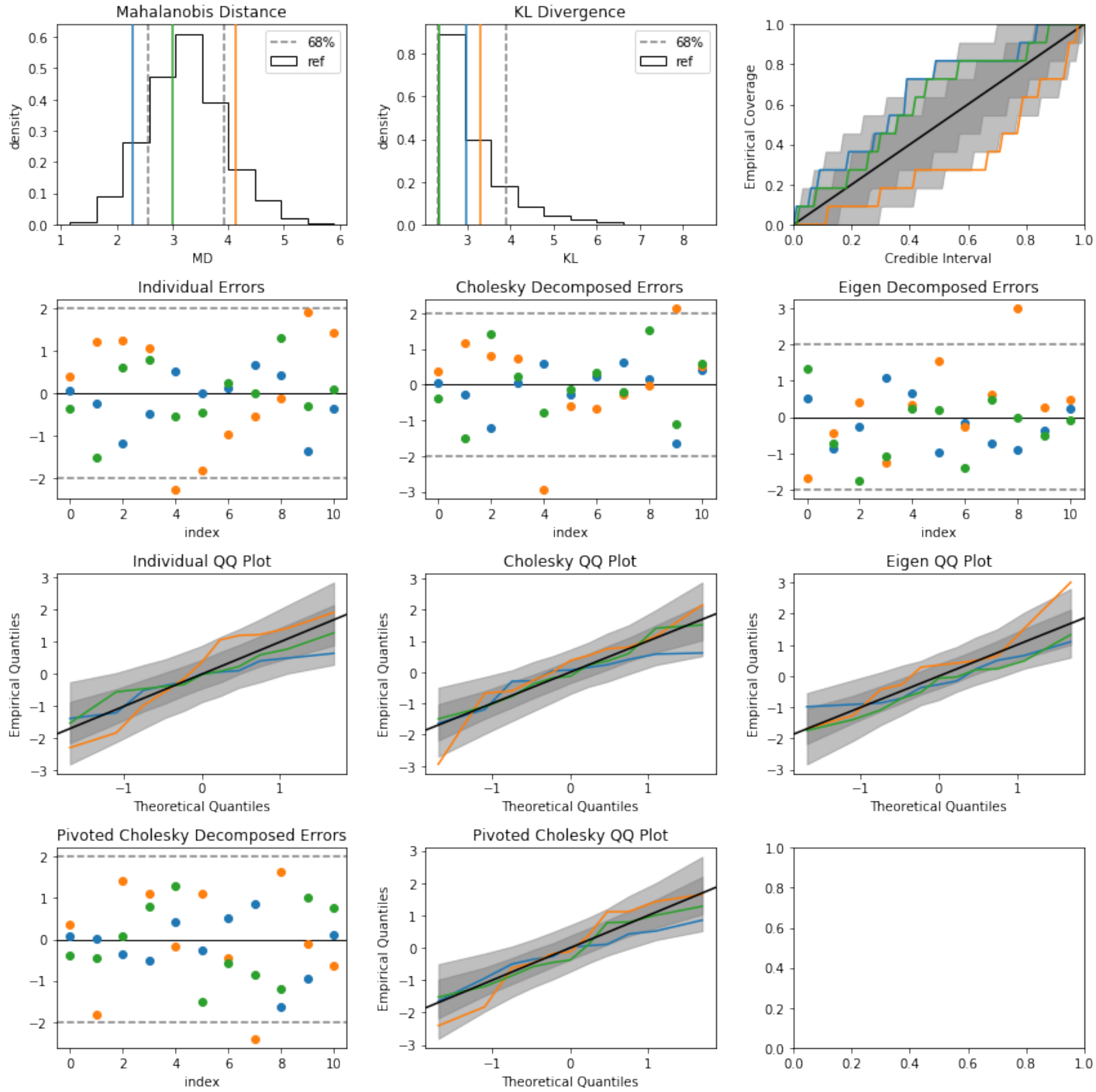
```
In [11]: print(tm1.name)
         tm1.model_checking_with_training_data_only()
```

```
rjf_no_shift_seed_6_samples_3
```



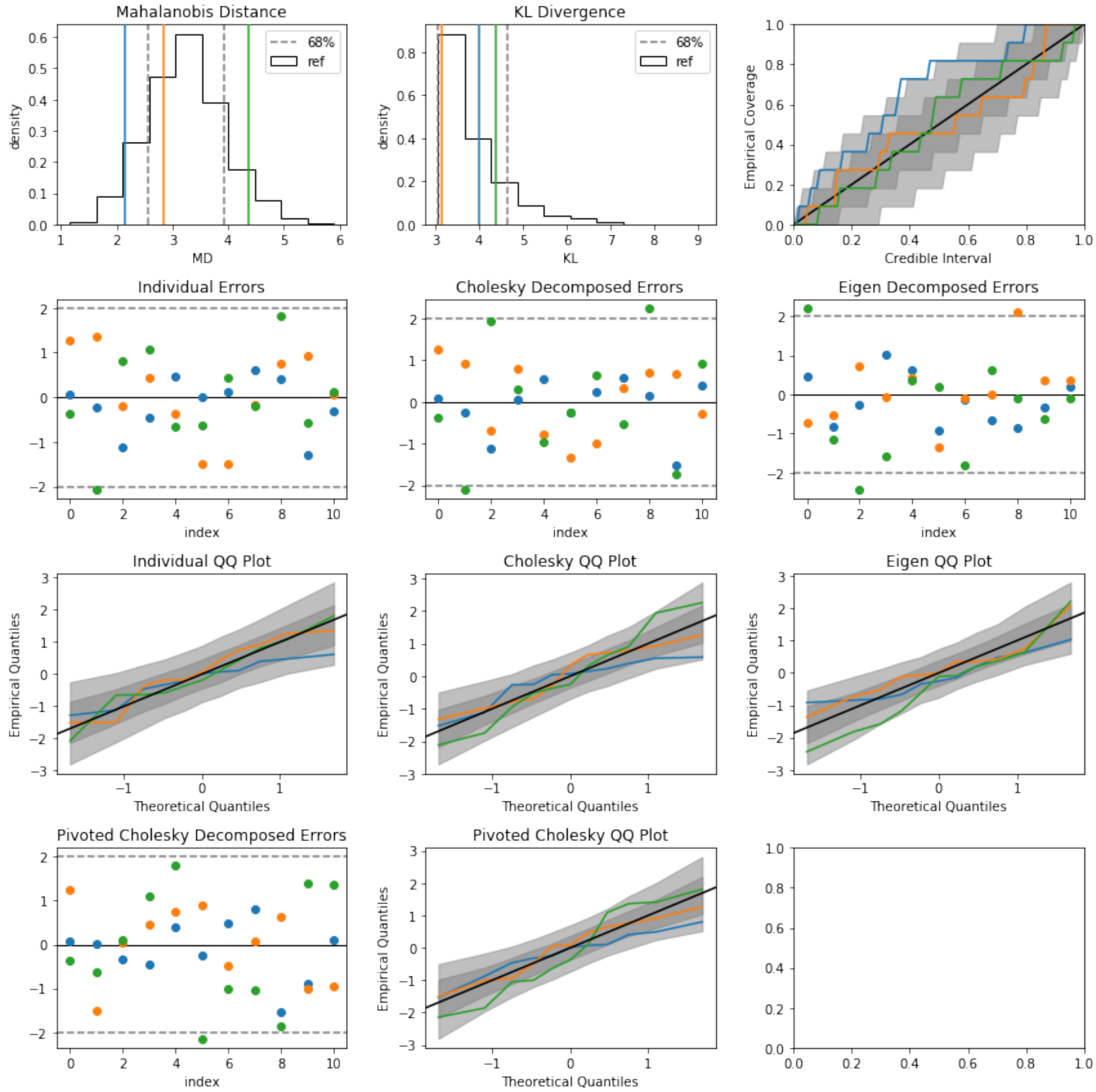
In [12]:
```
print(tm2.name)
tm2.model_checking_with_training_data_only()
```

```
rjf_var_shift_2_seed_6_samples_3
```




```
In [13]: print(tm3.name)
         tm3.model_checking_with_training_data_only()
```

```
rjf_scale_shift_0p3_seed_6_samples_3
```

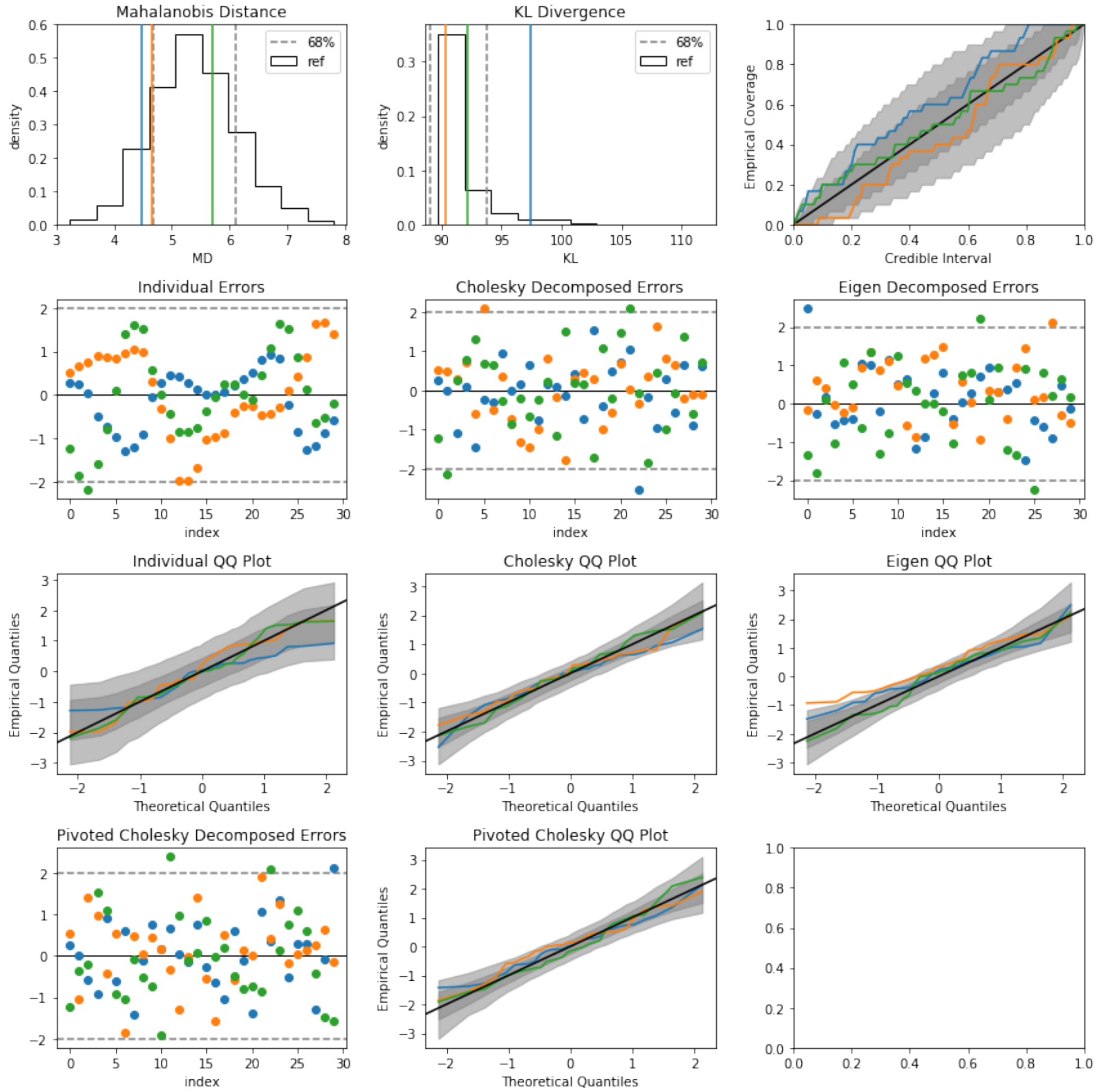


Comments:

1.3.6 Use the test dataset

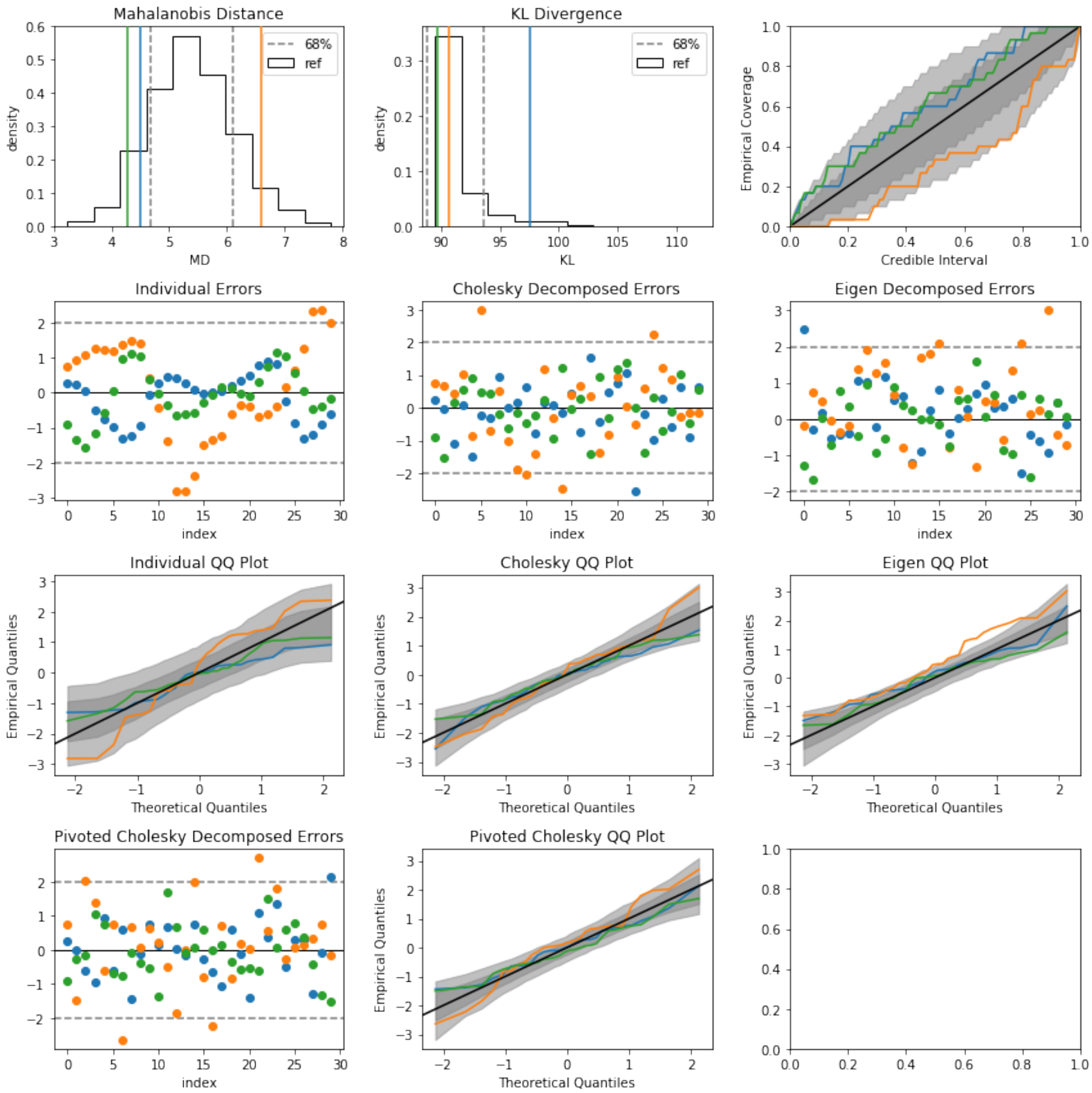
```
In [14]: print(tm1.name)
         tm1.model_checking_with_test_data_global()
```

```
rjf_no_shift_seed_6_samples_3
```



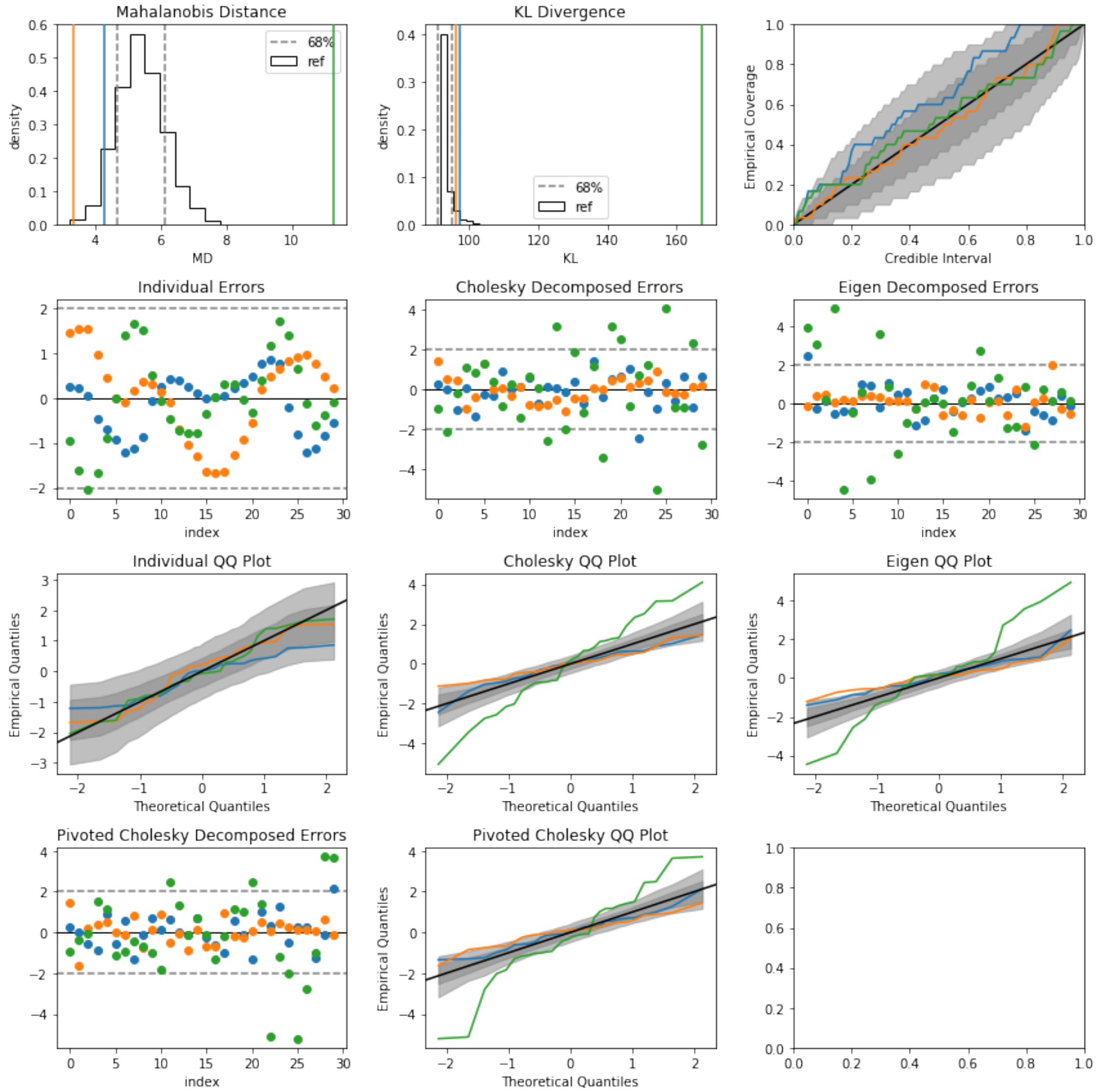

```
In [15]: print(tm2.name)
         tm2.model_checking_with_test_data_global()
```

```
rjf_var_shift_2_seed_6_samples_3
```



In [16]:
```
print(tm3.name)
tm3.model_checking_with_test_data_global()
```

```
rjf_scale_shift_0p3_seed_6_samples_3
```



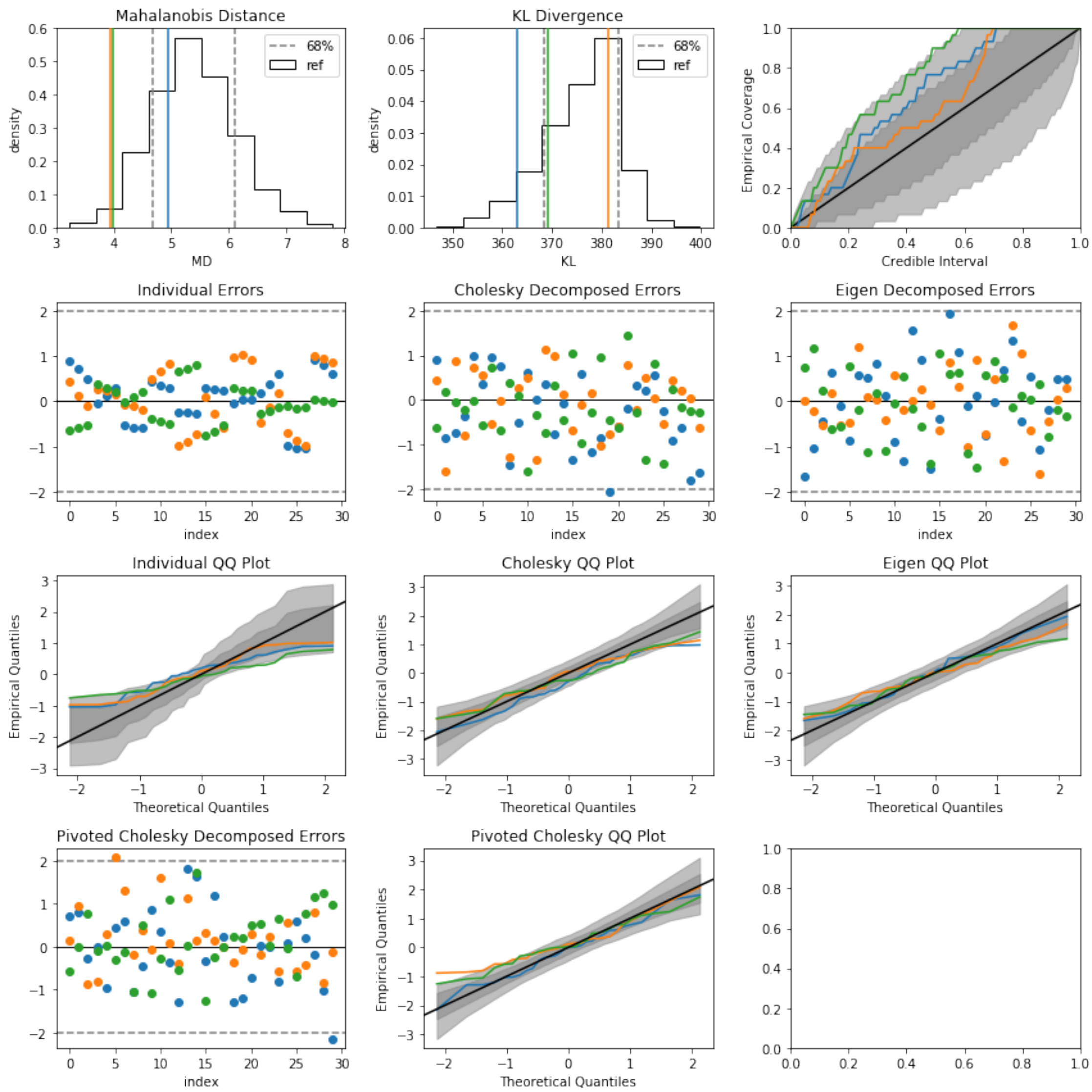
Comments:

1.3.7 Model checking with the interpolants

What if we performed the same model checking with the interpolants? This time, we are comparing each colored curve to the process defined by the thin gray bands around that curve. One potential clever way to combine the diagnostics from interpolated processes relies on the fact that the only thing that is different about the interpolating processes is their mean function that interpolates the data. If we subtract the means off the process and the data, then we are back to the simple iid case.

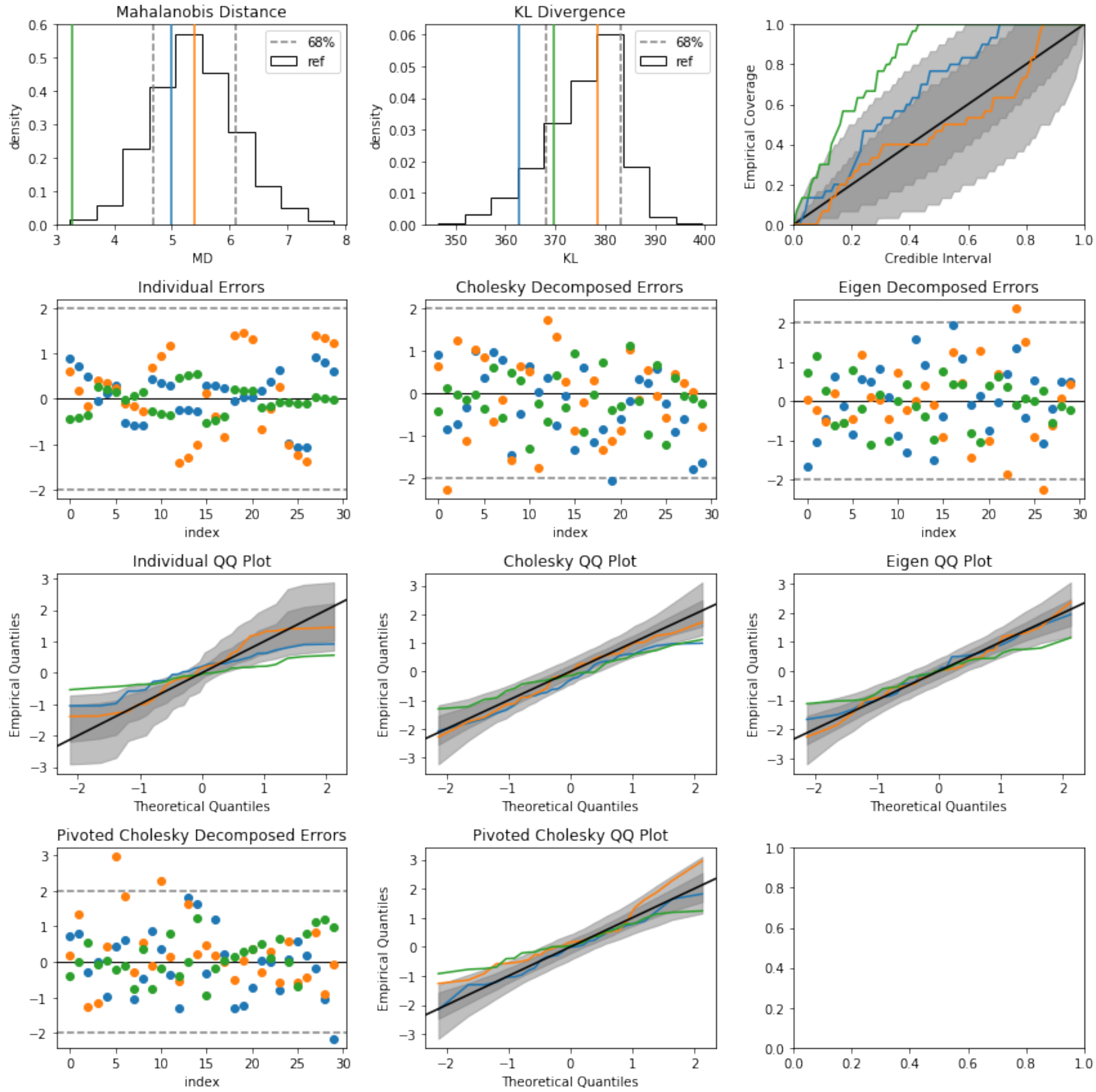
```
In [17]: print(tm1.name)
         tm1.model_checking_with_test_data_interpolants()
```

```
rjf_no_shift_seed_6_samples_3
```



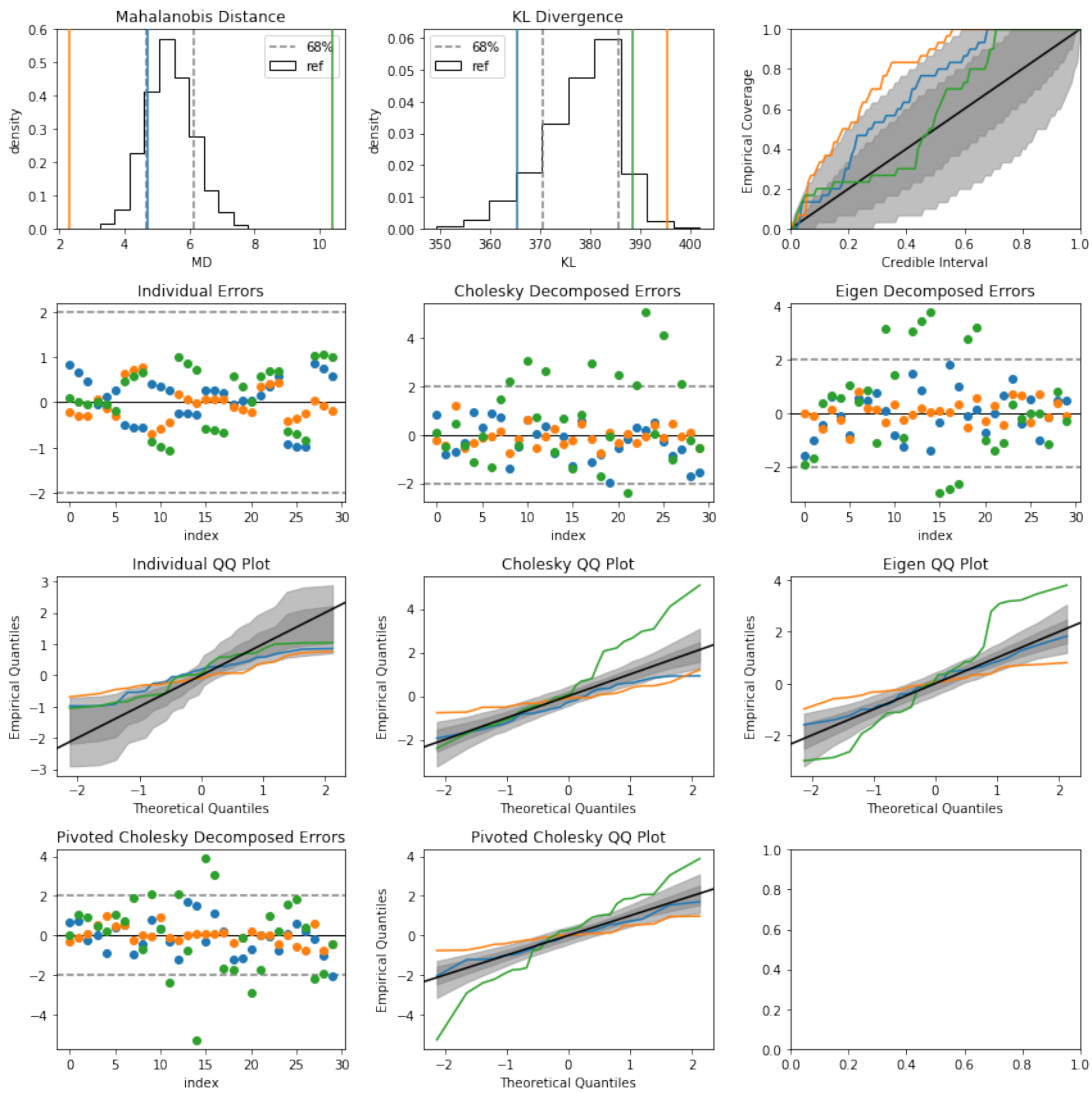
In [18]:
```
print(tm2.name)
tm2.model_checking_with_test_data_interpolants()
```

```
rjf_var_shift_2_seed_6_samples_3
```




```
In [19]: print(tm3.name)
         tm3.model_checking_with_test_data_interpolants()
```

```
rjf_scale_shift_0p3_seed_6_samples_3
```



Comments:

In [20]:
```
# *** RUN THIS CELL AFTER YOU HAVE SAVED THE NOTEBOOK ***
# save as a pdf file
output_directory = './saved_notebooks'
output_filename = 'test_file'

!jupyter nbconvert GP_model_checking_test_cases_rjf_upgrade1.ipynb --to pdf \
                  --output-dir=$output_directory --output $output_filename \
                  >/dev/null 2>&1
```