ODElib Documentation

Contents

Module `ODElib`

Sub-modules

- ODElib.Framework
- ODElib.Statistics

Module ODElib.Framework

Functions

Function rawstats

```
def rawstats(pdseries)
```

calculates raw median and standard deviation of posterior

Classes

Class ModelFramework

```
class ModelFramework(ODE, parameter_names, state_names, dataframe=None,
state_summations=None, t_end=5, t_steps=1000, random_seed=0, **kwargs)
```

The ModelFramweork class acts to facilitate and expedite the analysis of different ODEs given some experimental data. Specifically, this class uses a Markov Chain Monte Carlo (MCMC) implementation to fit and generate posterior distributions of those parameters.

Parameters

`ODE` **: function** A callable ODE function with the arguments y, t, and ps. y is the argument for an array of state variables that must match the function output. t is the argument for the time array. ps is the argument for an array of parameters.

`parameter_names` **: list of str** A list of strings should be specified containing the names of each parameter. The order of the parameters MUST match the unpacking order of the ps argument in the ODE function.

`state_names` **: list of str** A list of strings should be specified containing the names of each state variable. The order of the state variable names MUST match the unpacking order of the y argument in the ODE function.

`dataframe` **: pandas.DataFrame, optional** A dataframe specifying the data for the model to be fit to. Dataframe columns must be specifed in one of two ways: 1) Dataframes contain the columns 'organism', 'time', 'abundance', and 'variance' or 2) Dataframes contain the columns 'organism', 'time', 'abundance', and 'replicate'. Option 1 assumes the user has appropriately calculated the variance and mean abundance at each timepoint for each species, while option 2 will automatically calculate the variance and means. organism names must corrospond to the state_names so fittings can be matched to the appropriate datapoints

`state_summations` **: dict, optional** A dictionary mapping a representative name to the summation of ODE state variables.

`t_end` **: int** Final timepoint of integration. If a dataframe is passed, the final timepoint will be set to the final timepoint in the dataframe.

`t_steps` **: int** Number of timesteps to calculate during integration

`random_seed` **: int** Random seed to be used by samplers

Methods

Method MCMC

```
def MCMC(self, chain_inits=1, iterations_per_chain=1000, cpu_cores=1, static_parameters=[],
print_report=True, fitsurvey_samples=1000, sd_fitdistance=3.0)
```

Launches Markov Chain Monte Carlo

A Markov Chain Monte Carlo fitting protocol is used to find best fits. Note that chains can only be computed by a single CPU, therefore, increasing the number of cpu_cores for a single chain with many iterations will not improve performance.

Parameters

`chain_inits` **: list of dicts or dataframe** list of dictionaries mapping parameters to their values or dataframe with parameter values as columns. Values will be used as the intial values for the Markov Chains, where the length of the list/dataframe implies the number of chains to start

`iterations_per_chain` **: int** number of iterations to perform during MCMC chain. Default = 1000

`cpu_cores` **: int** number of cores used in fitting, Default = 1

`print_report` **: bool** Print a basic

`static_parameters` **: list, optional** A list of parameters that do not change during MCMC fitting

`fitsurvey_samples` **: int** The number of samples to take from multidimensional to search for good inital fits. Default = 1000

`sd_fitdistance` **: float** The number of standard deviations away from data that is acceptable as an initial fit. Default = 3.0

Returns

pandas.DataFrame Data containing results from all markov chains

Method `copy`

```
def copy(self, overwrite={})
```

Creates a copy of the current ModelFramework. All attributes are copied automatically. Typically used for creating instances for parallel operations.

Parameters

`overwrite` **: dict** A dictionary containing new initial states or parameters

Returns

ModelFramework

Method `explore_equilibriums`

```
def explore_equilibriums(self, samples=1000, cpu_cores=1, **parameter_mapping)
```

Launch

Parameters

`samples` **: int** Number of samples to search

`cpu_cores` **: int** number of cpu cores to use

`**kwargs` parameters mapped to tuples. Tuples should include three values: mean, standard deviation, and a boolean for tinylog transformation. Tinylog transformation is defined as `np.power(10,-(pos_norm(loc=mu,scale=sigma)`. Otherwise, only a pos_norm distribution is sampled, where pos_norm is a normal distribution with the lower bound always truncated at zero.

Returns

list list of outputs from func

Method `find_inits`

```
def find_inits(self, var_dist={}, set_best=True, step=1, **kwargs)
```

get the initial state variable values for integration

Parameters

`var_dist` **: tulple, optional** a mapping of state varaible names to a tuple of a scipy distribution and a boolean. If the boolean is true, samples drawn from the specified distribution will be exponentiated

`virus_init` **: int, optional** ignore v0 in data and set the viral initial value

Return

numpy array a numpy array of initial values for integration

Method `fit_survey`

```
def fit_survey(self, samples=1000, cpu_cores=1)
```

samples prior distribuiton with LHS scheme for fits

Method `get_AIC`

```
def get_AIC(self, chi)
```

Method `get_Rsqrd`

```
def get_Rsqrd(self, mod_dict)
```

Method `get_adjRsqrd`

```
def get_adjRsqrd(self, mod_dict, Rsqrd=None)
```

Method `get_chi`

```
def get_chi(self, mod_dict)
```

goodness of fit test

Method `get_fitstats`

```
def get_fitstats(self, prediction_dict={})
```

return dictionary of adjusted R-squared, Chi, and AIC of current parameters

Method `get_inits`

```
def get_inits(self, as_dict=False)
```

returns the initial values used in integration

Parameters

`as_dict` **: bool** If True, return inital states for ODE as a dictionary mapping state names to the initial values. Otherweise, return as an array according to the index of state_names

Method `get_model`

```
def get_model(self)
```

return the ODE function used for integration

Method `get_numstatevar`

```
def get_numstatevar(self)
```

returns the number of state varaibles

Method `get_parameters`

```
def get_parameters(self, as_dict=False, **kwargs)
```

return the parameters needed for integration

Parameters

`as_dict` **: bool, optional** If true, return dict with parameter names mapped to values
`kwargs` **: optional** pass a mapping of parameters to be packages for value return

Return

parameters numpy array of parameters ready for odeint or dict of parameters

Method `get_pnames`

```
def get_pnames(self)
```

returns the names of the parameters used in the current model

Method `get_snames`

```
def get_snames(self, after_summation=True, predict_obs=False)
```

returns the names of the state variables used in the current model

Parameters

`after_summation` **: bool** Return the state variables as defined by the state_summations argument. If state_summations was not specified, state names of the ODE are returned. If False, the state names of the ODE are returned regardless of the definitons in state_summations
`predict_obs` **: bool** Only return state names that have observations in the data

Method `gradient`

```
def gradient(self, parameter_name, p_range, intialstates=None, seed_equilibrium=True,
aggregate_enpoints=False, print_status=True)
```

Iterativly launches numerical simulations with different Srs

Parameters

`Srs` **: array** An array indicating the Sr of each simulation
`model` **: function** Model used in numerical integration
`initvalues` **: list of tuples** a list of tuples, where tuple[0]= member name and tuple[1]= inital value in simulation
`traits` **: dict of arrays** A dictionary mapping trait names to arrays. Note that this must be compatible with the respective model
`t_final` **: int** How long the simulation should run
`steps` **: int** How many steps should be taken per t
`seed` **: bool, optional**

Method `integrate`

```
def integrate(self, inits=None, parameters=None, predict_obs=False, as_dataframe=True,
sum_subpopulations=True)
```

allows option to return model solutions at sample times

Parameters

`inits` **: numpy.array, optional** ignore h0 and v0 in data and set the initial values for integration
`parameters` **: numpy.array, optional** ignore stored parameters and use specified

`predict_obs` **: bool** If True, only time points in df will be returned
`as_dataframe` **: bool** If True, integration results are returned as a dataframe. This operation is expensive and should not be used when being called iterativly
`sum_subpopulations` **: bool** apply state_summations after integration. If False, integration results are returned without summation. Default True

Returns

integration results as a dataframe or dictionary

Method `plot`

```
def plot(self, states=None, overlay={})
```

Method `reset_dataframe`

```
def reset_dataframe(self, df)
```

refreshes datastrucutres with new dataframe

Method `search_initparamfits`

```
def search_initparamfits(self, samples=1000, cpu_cores=1, **kwargs)
```

search parameter space for good initial parameter values

Parameters

`samples` **: int** Number of samples to search
`cpu_cores` **: int** number of cpu cores to use
`**kwargs` parameters mapped to tuples. Tuples should include three values: mean, standard deviation, and a boolean for tinylog transformation. Tinylog transformation is defined as `np.power(10,-(pos_norm(loc=mu,scale=sigma)`. Otherwise, only a pos_norm distribution is sampled, where pos_norm is a normal distribution with the lower bound always truncated at zero.

Returns

list list of outputs from func

Method `set_inits`

```
def set_inits(self, **kwargs)
```

set parameters for the model

Parameters

`**kwargs` key word arguments, where keys are parameters and args are parameter values. Alternativly, pass **dict

Raises

ValueError ValueError is raised if a initial condition for a summation variable is passed and the summation of the ODE state variables are not equal

Method `set_parameters`

```
def set_parameters(self, **kwargs)
```

set parameters for the model

Parameters

`**kwargs` key word arguments, where keys are parameters and args are parameter values. Alternativly, pass **dict

Class `parameter`

```
class parameter(init_value=None, stats_gen=None, hyperparameters=None,
name=None)
```

Parameter used in ModelFramework class

The parameter class is used by the ModelFramework class to initialize parameters. The goal of this class is to easily maintain a parameter value, the underlying distribution of the parameter, and hyperparameters for defining the distribution. Moreover, this class is responsible for defining the random walks during MCMC fittings, therefore, any parameter-specifc random walks can be fully customized.

Parameters

`initials` **: float** initial value of the parameter. If not specified, one is drawn. If a parameter needs to be an array, it must be specified here.

`stats_gen` **: scipy.stats.rv_continuous or scipy.stats.rv_discrete** An instance of a scipy.stats.rv_continuous or scipy.stats.rv_discrete that can be call typical statistical functions (pdf/pmf, cdf, ppf, ect.).

`hyperparameters` **: dict** A dictionary mapping hyperparameter names (indicated by the scipy distribution) to their values

`name` **: str** the name of the parameter

Methods

Method `copy`

```
def copy(self)
```

Method `fit`

```
def fit(self, data)
```

fits distribution to data and assigns hyperparameters

Method `get_figure`

```
def get_figure(self, samples=1000, logspace=False)
```

returns a matplotlib Figure

Method `has_distribution`

```
def has_distribution(self)
```

Return true if the parameter has a distribution

Method `pdf`

```
def pdf(self, val=None)
```

Method `rwalk`

```
def rwalk(self, std=0.05)
```

randomly walk the parameter

Module ODElib.Statistics

Sub-modules

- ODElib.Statistics.Samplers
- ODElib.Statistics.distributions
- ODElib.Statistics.stats

Module ODElib.Statistics.Samplers

Functions

Function MetropolisHastings

```
def MetropolisHastings(modelframework, nits=1000, burnin=None, static_parameters=set(),
print_progress=True)
```

allows option to return model solutions at sample times

Parameters

`nits` **: int** number of iterations
`burnin` **: int** number of iterations to ignore initially, Defaults to half of nits
`static_parameters` **: list-like, optional** specify parameters that you do not want to change during the markov chain

Returns

`tupple` **: pall, likelihoods, iterations** host and virus counts

Function sample_lhs

```
def sample_lhs(parameter_dict, samples)
```

Sample parameter space using a Latin Hyper Cube sampling scheme

Parameters

`parameter_dict` **: dict** parameter names mapped to a ODElib.parameters objects assigned distribuitons (and hyperparameters if applicable)
`samples` **: int** number of LHS samples to be taken

Returns

DataFrame DataFrame storing each LHS sample, organized into columns by the parameter name. Note that arrays are stored within the rows

Notes

Module ODElib.Statistics.distributions

Functions

Function Positive_Normal

```
def Positive_Normal(loc, scale)
```

normal distribution for positive values only

Classes

Class `discrete_norm`

```
class discrete_norm(a=0, b=inf, name=None, badvalue=None, moment_tol=1e-08,
values=None, inc=1, longname=None, shapes=None, extradoc=None, seed=None)
```

Normal distribution

Ancestors (in MRO)

- scipy.stats._distn_infrastructure.rv_discrete
- scipy.stats._distn_infrastructure.rv_generic

Class `gamma_gen`

```
class gamma_gen(momtype=1, a=None, b=None, xtol=1e-14, badvalue=None, name=None,
longname=None, shapes=None, extradoc=None, seed=None)
```

Gamma Distribution

Ancestors (in MRO)

- scipy.stats._distn_infrastructure.rv_continuous
- scipy.stats._distn_infrastructure.rv_generic

Module `ODElib.Statistics.stats`

Functions

Function `AIC`

```
def AIC(chi, num_parameters)
```

calcualte Akaike information criterion (AIC) for the model fit

Function `Rsqrd`

```
def Rsqrd(C_dict, O_dict)
```

calculate R^2

Function `chi`

```
def chi(O, C, S)
```

calculate reduced chi squared

Parameters

`O` **: numpy.ndarray** observed values
`C` **: numpy.ndarray** calculated values
`S` **: numpy.ndarray** variance

Returns:

chi fit of calculated values, lower values indicate a better fit

Function `get_adjusted_rsquared`

```
def get_adjusted_rsquared(Rsqrd, num_samples, num_parameters)
```

calculate adjusted R^2

Function `predict_logsigma`

```
def predict_logsigma(sigma, mean)
```

This function predicts the log transformed standard deviation from the mean and standard devation of untransformed data

Parameters

`sigma` **: numpy.ndarray** standard deviations, calculated without transformations
`mean` **: numpy.ndarray** mean, calculated without transformation

Returns

numpy.ndarray an array containing the variance as if it was calculated in log space

---