PTA Documentation

Release 0.0.9

Isaac Overcast, et al

Sep 21, 2023

Contents

CHAPTER 1

Installation

`PTA` requires Python >= 3.5. Installation is facilitated by the conda package management system.

1. Download miniconda and run the installer: `bash Miniconda*`
2. Create a separate conda environment to install PTA into:

```
conda create -n PTA
conda activate PTA
```

3. Install:

```
conda install -c conda-forge -c bioconda -c PTA PTA
```

4. Test:

```
PTA -v
```

Installation issues can be reported on the PTA github.

CHAPTER 2

Intro CLI Tutorial

MESS - Massive Eco-Evolutionary Synthesis Simulations

Example CLI:

```
## Create a new params file and populate with default values
MESS -n neutral_model

<edit params-neutral_model.txt>

## Generate 10000 simulations
MESS -p params-neutral_model.txt -s 10000

## Validate format of empirical data from a directory (proper formatting will be␣
↪checked)
MESS -i empirical_dir

## Perform abc model selection (competition model also previously simulated)
## Both neutral and competition models should have been simulated for a similar
## number of replicates, MESS will check for this.
MESS -i empirical_dir --abc params-neutral_model.txt params-competition_model.txt

## Perform random forest model selection (competition model also previously simulated)
MESS -i empirical_dir --RF params-neutral_model.txt params-competition_model.txt

## Estimate parameters of empirical data for a given model
## TODO: Figure out how to specify which parameters to estimate?
MESS -i empirical_dir -p params-neutral_model.txt --estimate_params

## Generate fancy plots through time for a given model. This will
## only run one realization and create several animated gifs <slow>
MESS -p params-neutral_model.txt --fancy-plots
```

The code is on github

CHAPTER 3

Intro API Tutorial

MESS - Massive Eco-Evolutionary Synthesis Simulations

Example API:

```
import MESS

## A mess region contains all universal parameters of the model, metacommunity
## information, and one or more local communities that can be connected
data = MESS.Region("my_first_sim")

## Define the metacommunity model
data.set_metacommunity("logseries")

## Add local communities to the region
loc1 = data.add_local_community("Island1", J=5000, c=0.01)
loc2 = data.add_local_community("Island2", J=1000, c=0.05)

## Define the potentially asymmetric migration matrix
## Migration matrix should be square with dimension equal to # of local communities␣
↪and,
## diagonal elements == 0
data.migration_matrix([[0, 0.05], [0.05, 0]])

## Run the simulation for x number of generations
data.simulate(nsims=100000)
## Alternatively simulate until some proportion of equilibrium is reached
results = data.simulate(lambda=0.7)

print(results)
```

3.1 Inference Procedure

The code is on github

CHAPTER 4

ML Inference

4.1 Supported Ensemble Methods

MESS currently supports three different scikit-learn ensemble methods for classification and parameter estimation:

- RandomForest (*rf*):
- GradientBoosting (*gb*):
- AdaBoost (*ab*):

4.2 Model Selection

4.3 Parameter Estimation

CHAPTER 5

Community Assembly Parameters

The parameters contained in a params file affect the behavior of various parts of the forward-time and backward-time assembly process. The defaults that we chose are fairly reasonable values as as starting point, however, you will always need to modify at least a few of them (for example, to indicate the location of your data), and often times you will want to modify many of the parameters.

Below is an explanation of each parameter setting, the eco-evolutionary process that it affects, and example entries for the parameter into a params.txt file.

5.1 simulation_name

The simulation name is used as the prefix for all output files. It should be a unique identifier for this particular set of simulations, meaning the set of parameters you are using for the current data set. When I run multiple related simulations I usually use names indicating the specific parameter combinations (e.g., filtering_nospeciation, J5000_neutral).

Example: New community simulations are created with the -n options to MESS:

```
## create a new assembly named J1000_neutral
$ MESS -n J1000_neutral
```

5.2 project_dir

A project directory can be used to group together multiple related simulations. A new directory will be created at the given path if it does not already exist. A good name for project_dir will generally be the name of the community/system being studied. The project dir path should generally not be changed after simulations/analysis are initiated, unless the entire directory is moved to a different location/machine.

Example entries into params.txt:

```
/home/watdo/MESS/galapagos          ## [1] create/use project dir called galapagos
galapagos                           ## [1] create/use project dir called galapagos in␣
↪cwd (./)
```

(continues on next page)

(continued from previous page)

5.3 generations

This parameter specifies the amount of time to run forward-time simulations. It can be specified in a number of different ways, but overall time can be considered either in terms of Wright-Fisher (WF) generations or in terms of Lambda. For WF generations you should specify an integer value (or a range of integer values) which will run the forward-time process for WF * J / 2 time-steps (where a time-step is one birth/death/colonization/speciation event). For Lambda you may select either an exact Lambda value (a real value between 0 and 1 exclusive), or you can set *generations* equal to 0, which will draw random Lambda values between 0 and 1 for each simulation.

Example entries into params.txt:

```
0                   ## [2] [generations]: Sample random Lambda values for each␣
↪simulation
100                 ## [2] [generations]: Run each simulation for 100 WF generations
50-100              ## [2] [generations]: Sample uniform between 50-100 WF␣
↪generations for each simulation
```

5.4 community_assembly_model

With this parameter you may specify a neutral or non-neutral scenario for the forward time process. There are currently three different options for this parameter: *neutral*, *filtering*, or *competition*. The *neutral* case indicates full ecological equivalence of all species, so all individuals have an equal probability of death at each time-step. In the *filtering* and *competition* models survival probability is contingent on proximity of species trait values to the environmental optimum, or distance from the local trait mean, respectively. You may also use the wildcard * here and MESS will randomly sample one community assembly model for each simulation.

Example entries into params.txt:

```
neutral             ## [3] [community_assembly_model]: Select the neutral process␣
↪forward-time
filtering           ## [3] [community_assembly_model]: Select the environmental␣
↪filtering process
*                   ## [3] [community_assembly_model]: Randomly choose one of the␣
↪community assembly models
```

5.5 speciation_model

Specify a speciation process in the local community. If *none* then no speciation happens locally. If *point_mutation* then one individual will instantaneously speciate at rate *speciation_prob* for each forward-time step. If *random_fission* then one lineage will randomly split into two lineages at rate *speciation_prob* with the new lineage receiving *n = U~(1, local species abundance)* individuals, and the parent lineage receiving *1 - n* individuals. *protracted* will specify a model of protracted speciation, but this is as yet unimplemented.

Example entries into params.txt:

```
none                ## [4] [speciation_model]: No speciation in the local community
point_mutation      ## [4] [speciation_model]: Point mutation specation process
```

5.6 mutation_rate

Specify the mutation rate for backward-time coalescent simulation of genetic variation. This rate is the per base, per generation probability of a mutation under an infinite sites model.

Example entries into params.txt:

```
2.2e-08              ## [5] [mutation_rate]: Mutation rate scaled per base per␣
↪generation
```

5.7 alpha

Scaling factor for transforming number of demes to number of individuals. `alpha` can be specified as either a single integer value or as a range of values.

Example entries to params.txt file:

```
2000                 ## [6] [alpha]: Abundance/Ne scaling factor
1000-10000           ## [6] [alpha]: Abundance/Ne scaling factor
```

5.8 sequence_length

Length of the sequence to simulate in the backward-time process under an infinite sites model. This value should be specified based on the length of the region sequenced for the observed community data in bp.

Example entries to params.txt file:

```
570                  ## [7] [sequence_length]: Length in bases of the sequence to␣
↪simulate
```

5.9 S_m

S_m specifies the total number of species to simulate in the metacommunity. Larger values will result in more singletons in the local community and reduced rates of multiple-colonization.

Example entries to params.txt file:

```
500                  ## [0] [S_m]: Number of species in the regional pool
100-1000             ## [0] [S_m]: Number of species in the regional pool
```

5.10 J_m

The total number of individuals in the metacommunity.

Example entries to params.txt:

```
0                   ## [9] allow zero low quality bases in a read
5                   ## [9] allow up to five low quality bases in a read
```

5.11 speciation_rate

Example entries to params.txt:

```
2                   ## [2] [speciation_rate]: Speciation rate of metacommunity
```

5.12 death_proportion

Example entries to params.txt:

```
0.7                 ## [3] [death_proportion]: Proportion of speciation rate to be␣
↪extinction rate
```

5.13 trait_rate_meta

Example entries to params.txt:

```
2                   ## [4] [trait_rate_meta]: Trait evolution rate parameter for␣
↪metacommunity
```

5.14 ecological_strength

This parameter dictates the strength of interactions in the environmental filtering and competition models. As the value of this parameter approaches zero, ecological strength is reduced and the assembly process increasingly resembles neutrality (ecological equivalence). Larger values increasingly bias probability of death against individuals with traits farther from the environmental optimum (in the filtering model).

In the following examples the environmental optimum is *3.850979*, and the ecological strength is varied from 0.001 to 100. Column 0 is species ID, column 1 is trait value, column 2 is unscaled probability of death, and column 3 is proportional probability of death. Models with strength of 0.001 and 0.01 are essentially neutral. Strength of 0.1 confers a slight advantage to individuals very close to the local optimum (e.g. species 't97').

Ecological strength of 1 (below, left panel) is noticeably non-neutral (e.g. 't97' survival probability is 10x greater than average). A value of 10 for this parameter generates a _strong_ non-neutral process (below, center panel: 't97' is 100x less likely to die than average, and the distribution of death probabilities is more varied). Ecological strength values >> 10 are _extreme_ and will probably result in degenerate behavior (e.g. strength of 100 (below, right panel) in which several of the species will be effectively immortal, with survival probability thousands of times better than average).

Example entries to params.txt:

```
1                   ## [5] [ecological_strength]: Strength of community assembly␣
↪process on phenotypic change
0.001-1             ## [5] [ecological_strength]: Strength of community assembly␣
↪process on phenotypic change
```

5.15 name

Example entries to params.txt:

```
island1             ## [0] [name]: Local community name
```

5.16 J

Example entries to params.txt:

```
1000-2000           ## [1] [J]: Number of individuals in the local community
```

5.17 m

Example entries to params.txt:

```
0.01                ## [2] [m]: Migration rate into local community
```

5.18 speciation_prob

Example entries to params.txt:

```
0                   ## [3] [speciation_prob]: Probability of speciation per timestep␣
↪in local community
0.0001-0.001        ## [3] [speciation_prob]: Probability of speciation per timestep␣
↪in local community
```

CHAPTER 6

API Documentation

This is the API documentation for MESS, and provides detailed information on the Python programming interface. See the *Intro API Tutorial* for an introduction to using this API to run simulations.

6.1 Simulation model

6.1.1 Region

6.1.2 Metacommunity

6.1.3 Local Community

6.2 Inference Procedure

class PTA.inference.**Ensemble**(*empirical_df*, *sims=''*, *algorithm='rf'*, *verbose=False*)

The Ensemble class is a parent class from which Classifiers and Regressors inherit shared methods. You normally will not want to create an Ensemble class directly, but the methods documented here are inherited by both Classifier() and Regressor() so may be called on either of them.

The base Ensemble class takes care of reading in the empirical dataframe, calculating summary stats, reading the simulated date, and reshaping the sim sumstats to match the stats of the real data.

Attention Ensemble objects should never be created directly. It is a base class that provides functionality to Classifier() and Regressor().

cross_val_predict(*cv=5*, *features=''*, *quick=False*, *verbose=False*)

Perform K-fold cross-validation prediction. For each of the *cv* folds, simulations will be split into sets of *K - (1/K)* training simulations and 1/K test simulations.

Note: CV predictions are not appropriate for evaluating model generalizability, these should only be used for visualization and exploration.

Parameters

- **cv** (*int*) – The number of K-fold cross-validation splits to perform.
- **quick** (*bool*) – If *True* skip feature selection and hyper-parameter tuning, and subset simulations. Runs fast but does a bad job. For testing.
- **verbose** (*bool*) – Report on progress. Depending on the number of CV folds this will be more or less chatty (mostly useless except for debugging).

Returns The array of predicted targets for each set of features when it was a member of the held-out testing set. Also saves the results in the Estimator.cv_preds variable.

cross_val_score(*cv=5*, *quick=False*, *verbose=False*)

Perform K-fold cross-validation scoring. For each of *cv* folds simulations will be split into sets of *K - (1/K)* training simulations and 1/K test simulations.

Parameters

- **cv** (*int*) – The number of K-fold cross-validation splits to perform.
- **quick** (*bool*) – If *True* skip feature selection and hyper-parameter tuning, and subset simulations. Runs fast but does a bad job. For testing.
- **verbose** (*bool*) – Report on progress. Depending on the number of CV folds this will be more or less chatty (mostly useless except for debugging).

Returns The array of scores of the estimator for each K-fold. Also saves the results in the Estimator.cv_scores variable.

dump(*outfile*)

Save the model to a file on disk. Useful for saving trained models to prevent having to retrain them.

Parameters **outfile** (*str*) – The file to save the model to.

feature_importances()

Assuming predict() has already been called, this method will return the feature importances of all features used for prediction.

Returns A pandas.DataFrame of feature importances.

feature_selection(*quick=False*, *verbose=False*)

Access to the feature selection routine. Uses BorutaPy, an all-relevant feature selection method: https://github.com/scikit-learn-contrib/boruta_py http://danielhomola.com/2015/05/08/borutapy-an-all-relevant-feature-selection-method/

Hint Normally you will not run this on your own, but will use it indirectly through the predict() methods.

Parameters

- **quick** (*bool*) – Run fast but do a bad job.
- **verbose** (*bool*) – Print lots of quasi-informative messages.

static load(*infile*)

Load a PTA.inference model from disk. This is complementary to the PTA.inference.Ensemble.dump() method.

Parameters **infile** (*str*) – The file to load a trained model from.

Returns Returns the *PTA.inference.Ensemble* object loaded from the input file.

plot_feature_importance(*cutoff=0.05*, *figsize=(10, 12)*, *layout=None*, *subplots=True*, *legend=False*)

Construct a somewhat crude plot of feature importances, useful for a quick and dirty view of these values. If more than one feature present in the model then a grid-layout is constructed and each individual feature is displayed within a subplot. This function is a thin wrapper around pandas.DataFrame.plot.barh().

Parameters

- **cutoff** (*float*) – Remove any features that do not have greater importance than this value across all plotted features. Just remove uninteresting features to reduce the amount of visual noise in the figures.
- **figsize** (*tuple*) – A tuple specifying figure width, height in inches.
- **layout** (*tuple*) – A tuple specifying the row, column layout of the sub-panels. By default we do our best, and it's normally okay.
- **subplots** (*bool*) – Whether to plot each feature individually, or just cram them all into one huge plot. Unless you have only a few features, setting this option to *False* will look insane.
- **legend** (*bool*) – Whether to plot the legend.

Returns Returns all the matplotlib axes

set_data(*empirical_df*, *verbose=False*)

A convenience function to allow using pre-trained models to make predictions on new datasets without retraining the model. This will calculate summary statistics on input data (recycling metacommunity traits if these were previously input), and reshape the statistics to match the features selected during initial model construction.

This is only sensible if the data from the input community consists of identical axes as the data used to build the model. This will be useful if you have community data from mutiple islands in the same archipelago, different communities that share a common features, and share a metacommunity.

Parameters

- **empirical_df** (*pandas.DataFrame*) – A DataFrame containing the empirical data. This df has a very specific format which is documented here.
- **verbose** (*bool*) – Print progress information.

set_features(*feature_list=''*)

Specify the feature list to use for classification/regression. By default the methods use all features, but if you want to specify exact feature sets to use you may call this method.

Parameters **feature_list** (*list*) – The list of features (summary statistics) to retain for downstream analysis. Items in this list should correspond exactly to summary statistics in the simulations or else it will complain.

set_params(*params={}*)

Allow to directly specify the parameters of the sklearn model, rather than doing a parameter search. Useful if param searching takes a long time and you only want to do it once, and reuse the best parameter set for multiple models.

Parameters **params** (*dict*) – A dictionary of parameter values and settings to pass to the underlying sklearn model. It's up to you to be sure the passed in parameters make sense for whatever model you're using.

set_targets(*target_list=''*)
Specify the target (parameter) list to use for classification/regression. By default the classifier will only consider psi and the regressor will use all targets, but if you want to specify exact target sets to use you may call this method.

Parameters **target_list** (*list*) – The list of targets (model parameters) to retain for downstream analysis. Items in this list should correspond exactly to parameters in the simulations or else it will complain.

6.2.1 Model Selection (Classification)

class PTA.inference.**Classifier**(*empirical_df*, *sims=''*, *algorithm='rf'*, *verbose=False*)
This class wraps all the model selection machinery.

Parameters

- **empirical_df** (*pandas.DataFrame*) – A DataFrame containing the empirical data. This df has a very specific format which is documented here.
- **sims** (*pd.DataFrame/string*) – The path to the file containing all the simulations.
- **algorithm** (*string*) – One of the *Supported Ensemble Methods* to use for parameter estimation.
- **verbose** (*bool*) – Print detailed progress information.

cross_val_predict(*cv=5*, *quick=False*, *verbose=False*)
A thin wrapper around Ensemble.cross_val_predict() that basically just calculates some Classifier specific statistics after the cross validation prodecure. This function will calculate and populate class variables:

- Classifier.classification_report: Mean absolute error

Parameters

- **cv** (*int*) – The number of cross-validation folds to perform.
- **quick** (*bool*) – Whether to downsample to run fast but do a bad job.
- **verbose** (*bool*) – Whether to print progress messages.

Returns A *numpy.array* of model class predictions for each simulation when it was a member of the held-out test set.

plot()
Simple method for visualizing the classification probabilities

plot_confusion_matrix(*ax=''*, *figsize=(8, 8)*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *cbar=False*, *title=''*, *normalize=False*, *outfile=''*)
Plot the confusion matrix for CV predictions. Assumes *Classifier.cross_val_predict()* has been called. If not it complains and tells you to do that first.

Parameters

- **ax** (*matplotlib.pyploat.axis*) – The matplotlib axis to draw the plot on.
- **figsize** (*tuple*) – If not passing in an axis, specify the size of the figure to plot.
- **cmap** (*matplotlib.pyplot.cm*) – Specify the colormap to use.
- **cbar** (*bool*) – Whether to add a colorbar to the figure.
- **title** (*str*) – Add a title to the figure.

- **normalize** (*bool*) – Whether to normalize the bin values (scale to 1/# simulations).
- **outfile** (*str*) – Where to save the figure. This parameter should include the desired output file format, e.g. *.png*, *.svg* or *.svg*.

Returns The *matplotlib.axis* on which the confusion matrix was plotted.

predict(*select_features=False*, *param_search=False*, *by_target=False*, *quick=False*, *force=False*, *verbose=False*)

Predict the community assembly model class probabilities.

Parameters

- **select_features** (*bool*) – Whether to perform relevant feature selection. This will remove features with little information useful for model prediction. Should improve classification performance, but does take time.
- **param_search** (*bool*) – Whether to perform ML classifier hyperparameter tuning. If `False` then classification will be performed with default classifier options, which will almost certainly result in poor performance, but it will run really fast!.
- **by_target** (*bool*) – Whether to predict multiple target variables simultaneously, or each individually and sequentially.
- **quick** (*bool*) – Reduce the number of retained simulations and the number of feature selection and hyperparameter tuning iterations to make the prediction step run really fast! Useful for testing.
- **force** (*bool*) – Force re-running feature selection and hyper-parameter tuning. This is basically here to prevent you from shooting yourself in the foot inside a for loop with *select_features=True* when really what you want (most of the time) is to just run this once, and call predict() multiple times without redoing this.
- **verbose** (*bool*) – Print detailed progress information.

Returns A tuple including the predicted model and the probabilities per model class.

6.2.2 Parameter Estimation (Regression)

class PTA.inference.**Regressor**(*empirical_df*, *sims=''*, *algorithm='rf'*, *verbose=False*)

This class wraps all the parameter estimation machinery.

Parameters

- **empirical_df** (*pandas.DataFrame*) – A DataFrame containing the empirical data. This df has a very specific format which is documented here.
- **sims** (*string*) – The path to the file containing all the simulations.
- **algorithm** (*string*) – The ensemble method to use for parameter estimation.
- **verbose** (*bool*) – Print lots of status messages. Good for debugging, or if you're *really* curious about the process.

cross_val_predict(*cv=5*, *quick=False*, *verbose=False*)

A thin wrapper around Ensemble.cross_val_predict() that basically just calculates some Regressor specific statistics after the cross validation prodecure. This function will calculate and populate class variables:

- Regressor.MAE: Mean absolute error
- Regressor.RMSE: Root mean squared error
- Regressor.vscore: Explained variance score

- Regressor.r2: Coefficient of determination regression score

As well as Regressor.cv_stats which is just a pandas.DataFrame of the above stats.

Parameters

- **cv** (*int*) – The number of cross-validation folds to perform.
- **quick** (*bool*) – Whether to downsample to run fast but do a bad job.
- **verbose** (*bool*) – Whether to print progress messages.

Returns A *numpy.array* of parameter estimates for each simulation when it was a member of the held-out test set.

plot_cv_predictions(*ax=''*, *figsize=(10, 5)*, *figdims=(2, 3)*, *n_cvs=1000*, *title=''*, *targets=''*, *outfile=''*)

Plot the cross validation predictions for this Regressor. Assumes *Regressor.cross_val_predict()* has been called. If not it complains and tells you to do that first.

Parameters

- **ax** (*matplotlib.pyploat.axis*) – The matplotlib axis to draw the plot on.
- **figsize** (*tuple*) – If not passing in an axis, specify the size of the figure to plot.
- **figdims** (*tuple*) – The number of rows and columns (specified in that order) of the output figure. There will be one plot per target parameter, so there should be at least as many available cells in the specified grid.
- **n_cvs** (*int*) – The number of true/estimated points to plot on the figure.
- **title** (*str*) – Add a title to the figure.
- **targets** (*list*) – Specify which of the targets to include in the plot.
- **outfile** (*str*) – Where to save the figure. This parameter should include the desired output file format, e.g. *.png*, *.svg* or *.svg*.

Returns The flattened list of matplotlib axes on which the scatter plots were drawn, one per target.

predict(*select_features=False*, *param_search=False*, *by_target=False*, *quick=False*, *force=True*, *verbose=False*)

Predict parameter estimates for selected targets.

Parameters

- **select_features** (*bool*) – Whether to perform relevant feature selection. This will remove features with little information useful for parameter estimation. Should improve parameter estimation performance, but does take time.
- **param_search** (*bool*) – Whether to perform ML regressor hyperparamter tuning. If `False` then prediction will be performed with default options, which will almost certainly result in poor performance, but it will run really fast!.
- **by_target** (*bool*) – Whether to estimate all parameters simultaneously, or each individually and sequentially. Some ensemble methods are only capable of performing individual parameter estimation, in which case this parameter is forced to `True`.
- **quick** (*bool*) – Reduce the number of retained simulations and the number of feature selection and hyperparameter tuning iterations to make the prediction step run really fast! Useful for testing.

- **force** (*bool*) – Force re-running feature selection and hyper-parameter tuning. This is basically here to prevent you from shooting yourself in the foot inside a for loop with *select_features=True* when really what you want (most of the time) is to just run this once, and call predict() multiple times without redoing this.
- **verbose** (*bool*) – Print detailed progress information.

Returns A *pandas.DataFrame* including the predicted value per target parameter, and 95% prediction intervals if the ensemble method specified for this Regressor supports it.

prediction_interval(*interval=0.95*, *quick=False*, *verbose=False*)

Add upper and lower prediction interval for algorithms that support quantile regression (*rfq*, *gb*).

Hint You normaly won't have to call this by hand, as it is incorporated automatically into the predict() methods. We allow access to in for experimental purposes.

Parameters

- **interval** (*float*) – The prediction interval to generate.
- **quick** (*bool*) – Subsample the data to make it run fast, for testing. The *quick* parameter doesn't do anything for *rfq* because it's already really fast (the model doesn't have to be refit).
- **verbose** (*bool*) – Print information about progress.

Returns A *pandas.DataFrame* containing the model predictions and the prediction intervals.

6.2.3 Classification Cross-Validation

PTA.inference.**classification_cv**(*sims*, *sep=' '*, *algorithm='rf'*, *quick=True*, *verbose=False*)

A convenience function to make it easier and more straightforward to run classification CV. This basically wraps the work of generating the synthetic community (dummy data), selecting which input data axes to retain (determines which summary statistics are used by the ML), creates the Classifier and calls Classifier.cross_val_predict(), and Classifier.cross_val_score().

Feature selection is independent of the real data, so it doesn't matter that we passed in synthetic empirical data here. It chooses features that are only relevant for each summary statistic. Searching for the best model hyperparameters is the same, it is done independently of the observed data.

Parameters

- **sims** (*str*) – A pd.DataFrame or the file containing copious simulations.
- **sep** (*str*) – Separator for loading in a DataFrame, if this was passed.
- **data_axes** (*list*) – A list of the data axis identifiers to prune the simulations with. One or more of 'abundance', 'pi', 'dxy', 'trait'. If this parameter is left blank it will use all data axes.
- **algorithm** (*str*) – One of the supported Ensemble.Regressor algorithm identifier strings: 'ab', 'gb', 'rf', 'rfq'.
- **quick** (*bool*) – Whether to run fast but do a bad job.
- **verbose** (*bool*) – Whether to print progress information.

Returns Returns the trained *PTA.inference.Classifier* with the cross- validation predictions for each simulation in the *cv_preds* member variable and the cross-validation scores per K-fold in the *cv_scores* member variable.

6.2.4 Parameter Estimation Cross-Validation

PTA.inference.**parameter_estimation_cv**(*sims*, *sep=' '*, *data_axes=''*, *algorithm='rf'*, *quick=True*, *verbose=False*)

A convenience function to make it easier and more straightforward to run parameter estimation CV. This basically wraps the work of generating the synthetic community (dummy data), selecting which input data axes to retain (determines which summary statistics are used by the ML), creates the Regressor and calls Regressor.cross_val_predict() and Regressor.cross_val.score().

Feature selection is independent of the real data, so it doesn't matter that we passed in synthetic empirical data here. It chooses features that are only relevant for each summary statistic. Searching for the best model hyperparameters is the same, it is done independently of the observed data.

Parameters

- **simfile** (*str*) – The file containing copious simulations.
- **sep** (*str*) – Separator for loading in a DataFrame, if this was passed.
- **data_axes** (*list*) – A list of the data axis identifiers to prune the simulations with. One or more of 'abundance', 'pi', 'dxy', 'trait'. If this parameter is left blank it will use all data axes.
- **algorithm** (*str*) – One of the supported Ensemble.Regressor algorithm identifier strings: 'ab', 'gb', 'rf', 'rfq'.
- **quick** (*bool*) – Whether to run fast but do a bad job.
- **verbose** (*bool*) – Whether to print progress information.

Returns Returns the trained *PTA.inference.Regressor* with the cross- validation predictions for each simulation in the *cv_preds* member variable and the cross-validation scores per K-fold in the *cv_scores* member variable.

6.2.5 Posterior Predictive Checks

PTA.inference.**posterior_predictive_check**(*empirical_df*, *parameter_estimates*, *ax=''*, *ipyclient=None*, *est_only=False*, *nsims=100*, *outfile=''*, *use_lambda=True*, *force=False*, *verbose=False*)

Currently not working.

Perform posterior predictive simulations. This function will take parameter estimates and perform PTA simulations using these parameter values. It will then plot the resulting summary statistics in PC space, along with the summary statistics of the observed data. The logic of posterior predictive checks is that if the estimated parameters are a good fit to the data, then summary statistics generated using these parameters should resemble those of the real data.

Parameters

- **empirical_df** (*pandas.DataFrame*) – A DataFrame containing the empirical data. This df has a very specific format which is documented here.
- **parameter_estimates** (*pandas.DataFrame*) – A DataFrame containing the the parameter estimates from a PTA.inference.Regressor.predict() call and optional prediction interval upper and lower bounds.
- **ax** (*bool*) – The matplotlib axis to use for plotting. If not specified then a new axis will be created.

- **ipyclient** (*ipyparallel.Client*) – Allow to pass in an ipyparallel client to allow parallelization of the posterior predictive simulations. If no ipyclient is specified then simulations will be performed serially (i.e. SLOW).
- **est_only** (*bool*) – If True, drop the lower and upper prediction interval (PI) and just use the mean estimated parameters for generating posterior predictive simulations. If False, and PIs exist, then parameter values will be sampled uniformly between the lower and upper PI.
- **nsims** (*bool*) – The number of posterior predictive simulations to perform.
- **outfile** (*bool*) – A file path for saving the figure. If not specified the figure is simply not saved to the filesystem.
- **use_lambda** (*bool*) – Whether to generated simulations using time as measured in _lambda or in generations.
- **force** (*bool*) – Force overwrite previously generated simulations. If not force then re-running will append new simulations to previous ones.
- **verbose** (*bool*) – Print detailed progress information.

Returns A *matplotlib.pyplot.axis* containing the plot.

6.3 Stats

6.3.1 Plotting

PTA.plotting.**plot_simulations_hist**(*sims*, *ax=''*, *figsize=(12, 6)*, *feature_set=''*, *nsims=1000*, *bins=20*, *alpha=0.6*, *select=''*, *tol=''*, *title=''*, *outfile=''*, *verbose=False*)

Simple histogram for each summary statistic. Useful for inspecting model performance. Invariant summary statistics will be removed.

Parameters

- **sims** (*str*) –
- **figsize** (*tuple*) –
- **feature_set** (*list*) –
- **nsims** (*int*) –
- **bins** (*int*) – The number of bins per histogram.
- **alpha** (*float*) – Set alpha value to determine transparency [0-1], larger values increase opacity.
- **select** (*int/float*) –
- **tol** (*int/float*) –
- **title** (*str*) –
- **outfile** (*str*) –
- **verbose** (*bool*) –

Returns Return a list of *matplotlib.pyplot.axis* on which the simulated summary statistics have been plotted. This list can be _long_ depending on how many statistics you plot.

PTA.plotting.**plot_simulations_pca**(*sims*, *ax=''*, *figsize=(8, 8)*, *target=''*, *feature_set=''*, *loadings=False*, *nsims=1000*, *select=''*, *tol=''*, *title=''*, *outfile=''*, *colorbar=True*, *verbose=False*)

Plot summary statistics for simulations projected into PC space.

Parameters

- **sims** (*str*) –
- **ax** (*matplotlib.pyplot.axis*) –
- **figsize** (*tuple*) –
- **target** (*str*) –
- **feature_set** (*list*) –
- **loadings** (*bool*) – BROKEN! Whether to plot the loadings in the figure.
- **nsims** (*int*) –
- **select** (*int/float*) –
- **tol** (*int/float*) –
- **title** (*str*) –
- **outfile** (*str*) –
- **verbose** (*bool*) –

Returns Return the *matplotlib.pyplot.axis* on which the simulations are plotted.

The code is on github

Experiment with the MESS model now! Launch the binder instance below and you can open and run the notebooks in the *jupyter-notebooks* directory.

Index