mrtool

Release 0.0.1

Jan 26, 2021

Contents

MRTool (Meta-Regression Tool) package is designed to solve general meta-regression problem. The most common features include,

- linear and log prediction function,
- spline extension for covariates,
- direct Gaussian, Uniform and Laplace prior on fixed and random effects,
- shape constraints (monotonicity and convexity) for spline.

Advanced features include,

- spline knots ensemble,
- automatic covariate selection.

CHAPTER 1

Installation

This package uses data class, therefore require `python>=3.7`.

Required packages include,

- basic scientific computing suite, Numpy, Scipy and Pandas,
- main optimization engine, IPOPT,
- customized packages, LimeTr and XSpline,
- testing tool, Pytest.

After install the required packages, clone the repository and install MRTool.

```
git clone https://github.com/ihmeuw-msca/MRTool.git
cd MRTool && python setup.py install
```

CHAPTER 2

Getting Started

To build and run a model, we only need four steps,

1. create `MRData` object and load data from data frame
2. configure the `CovModel` with covariates and priors
3. create `MRModel` object with data object and covriate models and fit the model
4. predict or create draws with new data and model result

In the following, we will list a set of examples to help user get familiar with the syntax.

- *simple linear model*

CHAPTER 3

Important Concepts

To correctly setup the model and solve problems, it is very important to understand some key *concepts*. We introduce them under three categories,

- How can we match the data generating mechansim?
- How can we incorporate prior knowledge?
- How do the underlying optimization algorithms work?

3.1 Examples and Demos

In this part of the documentation, we will organize all useful examples and demos.

3.1.1 Example: Simple Linear Model

In the following, we will go through a simple example of how to solve a linear mixed effects model. Consider the following setup,

$$y_{ij} = (\beta_0 + u_{0i}) + x\beta_1 + \epsilon_{ij}$$

where y is the measurement, x is the covariate, β_0 and β_1 is the fixed effects, u_0 is the random intercept and ϵ is the measurement error. And i is index for study, j is index for observation within study.

Assume our data frame looks like,

y	x	y_se	study_id
0.20	0.0	0.1	A
0.29	0.1	0.1	A
0.09	0.2	0.1	B
0.14	0.3	0.1	C
0.40	0.4	0.1	D

and our goal is to obtain the fixed effects and random effects for each study.

Create Data Object

The first step is to create a `MRData` object to carry the data information.

```
from mrtool import MRData

data = MRData()
data.load_df(
    df,
    col_obs='y',
    col_covs=['x'],
    col_obs_se='y_se',
    col_study_id='study_id'
)
```

Notice that the `MRData` will automatically create an `intercept` in the covariate list.

Configure Covariate Models

The second step is to create covariate models.

```
from mrtool import LinearCovModel

cov_intercept = LinearCovModel('intercept', use_re=True)
cov_x = LinearCovModel('x')
```

Create Model and Fit Model

The third step is to create the model to group data and covariate models. And use the optimization routine to find result.

```
from mrtool import MRBRT

model = MRBRT(
    data,
    [cov_intercept, cov_x]
)
model.fit_model()
```

You could get the fixed effects and random effects by calling `model.beta_soln` and `model.re_soln`.

Predict and Create Draws

The last step is to predict and create draws.

```
# first create data object used for predict
# the new data frame has to provide the same covariates as in the fitting
data_pred = MRData()
data_pred.load_df(
    df_pred,
    col_covs=['x']
```

(continues on next page)

(continued from previous page)

```
)

# create point prediction
y_pred = model.predict(data_pred)

# sampling solutions
beta_samples, gamma_samples = model.sample_soln(sample_size=1000)

# create draws
y_draws = model.create_draws(
    data_pred,
    beta_samples,
    gamma_samples
)
```

Here `y_pred` is the point prediction and `y_draws` contains `1000` draws of the outcome.

3.2 Concepts

In `MRTool` there are many important concepts and definitions. We list them here under the topics of **data generating mechanisms**, **priors** and **optimization**.

3.2.1 Data Generating Mechanism

During the modeling process, the first question that needs to be answered is how is the data generated and the data generating mechanism is about using given information to create predictive model.

Range Exposure

Very often, data is being collected over cohorts or different groups of people, and therefore one data point can be interpreted as an average.

For example, if we are interested in the relation between smoking and relative risk of getting lung cancer, one data point is measured by the relative risk between the smoking and the non-smoking group. Within the smoking group, subjects have different exposures to smoking. So what the data point measures is the average relative risk for the corresponding range of exposures.

If we denote x as the exposure and $f(x)$ as the function between the outcome and exposure, one measurement y over a range of exposures $x \in [a, b]$ can be expressed as,

$$y = \frac{1}{b-a}\int_a^b f(x)\,\mathrm{d}x.$$

A special case is when the function f is linear, $f(x) = \beta x$, and the expression can be simplified as,

$$y = \frac{1}{b-a}\int_a^b f(x)\,\mathrm{d}x = \frac{1}{2}(a+b)\beta.$$

It is equivalent to use the midpoint of the exposures as the covariate.

Sample Code

In the code, you could communicate with the program that you have a range exposure by inputting a pair of covariates instead of one.

```
cov_model = CovModel('exposure', alt_cov=['exposure_start', 'exposure_end'])
```

Relative Risk 1: Binary

Relative risk (RR) is the most common measurement type for the applications of `MRTool`. Here we take a chance to introduce the basic concepts regarding relative risk, and how we build different types of relative risk models in `MRTool`.

Relative risk is the probability ratio of a certain outcome between exposed and unexposed group. For more information please check the wiki page. Here we use smoking and lung cancer as a risk-outcome pair to explain the idea.

Imagine the experiment is conducted with two groups, smoking (e) and non-smoking (u) group. We record the probability of getting lung cancer among the two groups, P_e, P_u and the relative risk can be expressed as,

$$RR = \frac{P_e}{P_u}.$$

To implement meta-analysis on the effect of smoking, we often convert the collected relative risks from different studies (longitudinal or not) to log space, for the convenience of removing the sign restriction,

$$\ln(RR) = \ln(P_e) - \ln(P_u).$$

To setup the binary model, we simply parametrize the log relative risk with an intercept,

$$\ln(RR) = \mathbf{1}(\beta + u),$$

where β is the fixed effect for intercept and u is the random effect. When β is significantly greater than zero, we say that it is harmful. For other risk outcome pair, there is possibility that β is significantly less than zero, in which case we will call it protective.

Very often instead of only considering smoking vs non-smoking (binary), we also want to study the effects under different exposure to smoking. The most common assumption is log linear, please check *Relative Risk 2: Log Linear* for the details.

Sample Code

To setup the problem, we will only need `LinearCovModel`.

```
from mrtool import MRData, LinearCovModel, MRBRT

data = MRData()
# `intercept` is automatically added to the data
# no need to pass it in `col_covs`
data.load_df(
    df=df,
    col_obs='ln_rr',
    col_obs_se='ln_rr_se',
    col_study_id='study_id'
)
cov_model = LinearCovModel('intercept', use_re=True)
model = MRBRT(data, cov_models=[cov_model])
```

Relative Risk 2: Log Linear

When analyzing relative risk across different exposure levels, the most widely used assumption is that the model is log linear. We parametrize the log risk as a linear function of exposure,

$$\ln(RR) = \ln(R_e) - \ln(R_u) = x_a(\beta + u) - x_r(\beta + u) = (x_a - x_r)(\beta + u),$$

where x is the exposure, β, u are the fixed and random effects, and a, r refer to "alternative" and "reference" groups. They are consistent with previous notation, "exposed" and "unexposed".

Remark 1: **No intercept!**

Notice that in this model, we do NOT include the intercept to model the log risk. It is not possible to infer the absolute position of the risk curve using relative risk data, only the relative position.

To see this, first assume that we have intercept in the log risk formulation, $\ln(R) = (\beta_0 + u_0) + x(\beta_1 + u_1)$, when we construct the log relative risk,

$$\begin{aligned}\ln(RR) =& \ln(R_e) - \ln(R_u)\\ =& (\beta_0 + u_0) + x_a(\beta_1 + u_1) - ((\beta_0 + u_0) + x_r(\beta_1 + u_1))\\ =& (x_a - x_r)(\beta_1 + u_1)\end{aligned}$$

the intercept cancels and we returns to the original formula.

Remark 2: **No intercept! Again!**

The other possible use of the intercept is to directly model the log relative risk, instead of log risk,

$$\ln(RR) = (\beta_0 + u_0) + (x_a - x_r)(\beta_1 + u_1).$$

This does NOT work due to the fact that when x_a is equal to x_r, we expect the log relative risk is zero.

Compare to *Relative Risk 1: Binary*, where we use the intercept to model the log relative risk,

- In the binary model, we directly model the log relative risk instead of log risk.
- In the binary model, we never have the case when the exposures for two groups are the same.

Sample Code

To setup the problem, we will only need `LinearCovModel`, just as in *Relative Risk 1: Binary*.

If there is already a column in the data frame corresponding to the exposure differences, we can simply use it as the covariate.

```
from mrtool import MRData, LinearCovModel, MRBRT

data = MRData()
data.load_df(
    df=df,
    col_obs='ln_rr',
    col_obs_se='ln_rr_se',
    col_covs=['exposure_diff']
    col_study_id='study_id'
)
cov_model = LinearCovModel('exposure_diff', use_re=True)
model = MRBRT(data, cov_models=[cov_model])
```

Otherwise if you pass in the exposure for the "alternative" and "reference" group, the `LinearCovModel` will setup the model for you.

```
data.load_df(
    df=df,
    col_obs='ln_rr',
    col_obs_se='ln_rr_se',
    col_covs=['exposure_alt', 'exposure_ref']
    col_study_id='study_id'
)
cov_model = LinearCovModel(alt_cov='exposure_alt', ref_cov='exposure_ref', use_
↪re=True)
```

3.2.2 Priors

3.2.3 Optimization

3.3 API Reference

3.3.1 mrtool.core package

data

data module for *mrtool* package.

class MRData(*obs=<factory>*, *obs_se=<factory>*, *covs=<factory>*, *study_id=<factory>*, *data_id=<factory>*)

Bases: `object`

Data for simple linear mixed effects model.

get_covs(*covs*)

Get covariate matrix.

Parameters **covs** (*Union[List[str], str]*) – List of covariate names or one covariate name.

Returns Covariates matrix, in the column fashion.

Return type np.ndarray

get_study_data(*studies*)

Get study specific data.

Parameters **studies** (*Union[List[Any], Any]*) – List of studies or one study.

Returns MRData: Data object contains the study specific data.

Return type *MRData*

has_covs(*covs*)

If the data has the provided covariates.

Parameters **covs** (*Union[List[str], str]*) – List of covariate names or one covariate name.

Returns If has covariates return *True*.

Return type bool

has_studies(*studies*)
If the data has provided study_id

Parameters **Union[List[Any], Any]** (*studies*) – List of studies or one study.

Returns If has studies return *True*.

Return type bool

is_cov_normalized(*covs=None*)
Return true when covariates are normalized.

Return type bool

is_empty()
Return true when object contain data.

Return type bool

load_df(*data*, *col_obs=None*, *col_obs_se=None*, *col_covs=None*, *col_study_id=None*, *col_data_id=None*)
Load data from data frame.

load_xr(*data*, *var_obs=None*, *var_obs_se=None*, *var_covs=None*, *coord_study_id=None*)
Load data from xarray.

normalize_covs(*covs=None*)
Normalize covariates by the largest absolute value for each covariate.

num_covs
Number of covariates.

num_obs
Number of observations.

num_points
Number of data points.

num_studies
Number of studies.

reset()
Reset all the attributes to default values.

to_df()
Convert data object to data frame.

Return type DataFrame

cov_model

Covariates model for *mrtool*.

class CovModel(*alt_cov*, *name=None*, *ref_cov=None*, *use_re=False*, *use_re_mid_point=False*, *use_spline=False*, *use_spline_intercept=False*, *spline_knots_type='frequency'*, *spline_knots=array([0., 0.33333333, 0.66666667, 1.])*, *spline_degree=3*, *spline_l_linear=False*, *spline_r_linear=False*, *prior_spline_derval_gaussian=None*, *prior_spline_derval_gaussian_domain=(0.0, 1.0)*, *prior_spline_derval_uniform=None*, *prior_spline_derval_uniform_domain=(0.0, 1.0)*, *prior_spline_der2val_gaussian=None*, *prior_spline_der2val_gaussian_domain=(0.0, 1.0)*, *prior_spline_der2val_uniform=None*, *prior_spline_der2val_uniform_domain=(0.0, 1.0)*, *prior_spline_funval_gaussian=None*, *prior_spline_funval_gaussian_domain=(0.0, 1.0)*, *prior_spline_funval_uniform=None*, *prior_spline_funval_uniform_domain=(0.0, 1.0)*, *prior_spline_monotonicity=None*, *prior_spline_monotonicity_domain=(0.0, 1.0)*, *prior_spline_convexity=None*, *prior_spline_convexity_domain=(0.0, 1.0)*, *prior_spline_num_constraint_points=20*, *prior_spline_maxder_gaussian=None*, *prior_spline_maxder_uniform=None*, *prior_spline_normalization=None*, *prior_beta_gaussian=None*, *prior_beta_uniform=None*, *prior_beta_laplace=None*, *prior_gamma_gaussian=None*, *prior_gamma_uniform=None*, *prior_gamma_laplace=None*)

Bases: `object`

Covariates model.

attach_data(*data*)
Attach data.

create_constraint_mat()
Create constraint matrix. :returns: Return linear constraints matrix and its uniform prior. :rtype: tuple{numpy.ndarray, numpy.ndarray}

create_design_mat(*data*)
Create design matrix. :param data: The data frame used for storing the data :type data: mrtool.MRData

Returns Return the design matrix for linear cov or spline.

Return type tuple{numpy.ndarray, numpy.ndarray}

create_regularization_mat()
Create constraint matrix. :returns: Return linear regularization matrix and its Gaussian prior. :rtype: tuple{numpy.ndarray, numpy.ndarray}

create_spline(*data*, *spline_knots=None*)
Create spline given current spline parameters. :type data: *MRData* :param data: The data frame used for storing the data :type data: mrtool.MRData :type spline_knots: `Optional[ndarray]` :param spline_knots: Spline knots, if `None` determined by frequency or domain. :type spline_knots: np.ndarray, optional

Returns The spline object.

Return type xspline.XSpline

create_x_fun(*data*)

create_z_mat(*data*)

has_data()
Return `True` if there is one data object attached.

num_constraints

num_regularizations

num_x_vars

num_z_vars

class LinearCovModel(**args*, ***kwargs*)
Bases: *mrtool.core.cov_model.CovModel*

Linear Covariates Model.

create_x_fun(*data*)
Create design function for the fixed effects.

create_z_mat(*data*)
Create design matrix for the random effects.

Parameters **data** (*mrtool.MRData*) – The data frame used for storing the data

Returns Design matrix for random effects.

Return type numpy.ndarray

class LogCovModel(**args*, ***kwargs*)
Bases: *mrtool.core.cov_model.CovModel*

Log Covariates Model.

create_constraint_mat(*threshold=1e-06*)
Create constraint matrix. Overwrite the super class, adding non-negative constraints.

create_x_fun(*data*)
Create design functions for the fixed effects.

Parameters **data** (*mrtool.MRData*) – The data frame used for storing the data

Returns Design functions for fixed effects.

Return type tuple{function, function}

create_z_mat(*data*)
Create design matrix for the random effects.

Parameters **data** (*mrtool.MRData*) – The data frame used for storing the data

Returns Design matrix for random effects.

Return type numpy.ndarray

num_constraints

num_z_vars

model

Model module for mrtool package.

class LimeTr
Bases: object

class MRBRT(*data*, *cov_models*, *inlier_pct=1.0*)
Bases: object

MR-BRT Object

attach_data(*data=None*)
Attach data to cov_model.

check_input()
Check the input type of the attributes.

create_c_mat()
Create the constraints matrices.

create_draws(*data*, *beta_samples*, *gamma_samples*, *random_study=True*, *sort_by_data_id=False*)
Create draws for the given data set.

Parameters

- **data** (MRData) – MRData object contains predict data.
- **beta_samples** (*np.ndarray*) – Samples of beta.
- **gamma_samples** (*np.ndarray*) – Samples of gamma.
- **random_study** (*bool, optional*) – If *True* the draws will include uncertainty from study heterogeneity.
- **sort_by_data_id** (*bool, optional*) – If *True*, will sort the final prediction as the order of the original data frame that used to create the *data*. Default to False.

Returns Returns outcome sample matrix.

Return type np.ndarray

create_gprior()
Create direct gaussian prior.

create_h_mat()
Create the regularizer matrices.

create_lprior()
Create direct laplace prior.

create_uprior()
Create direct uniform prior.

create_x_fun(*data=None*)
Create the fixed effects function, link with limetr.

create_z_mat(*data=None*)
Create the random effects matrix, link with limetr.

extract_re(*study_id*)
Extract the random effect for a given dataset.

Return type ndarray

fit_model(***fit_options*)
Fitting the model through limetr.

Parameters

- **x0** (*np.ndarray*) – Initial guess for the optimization problem.
- **inner_print_level** (*int*) – If non-zero printing iteration information of the inner problem.
- **inner_max_iter** (*int*) – Maximum inner number of iterations.
- **inner_tol** (*float*) – Tolerance of the inner problem.
- **outer_verbose** (*bool*) – If *True* print out iteration information.
- **outer_max_iter** (*int*) – Maximum outer number of iterations.
- **outer_step_size** (*float*) – Step size of the outer problem.

- **outer_tol** (*float*) – Tolerance of the outer problem.
- **normalize_trimming_grad** (*bool*) – If *True*, normalize the gradient of the outer trimming problem.

get_cov_model(*name*)
Choose covariate model with name.

Return type *CovModel*

get_cov_model_index(*name*)
From cov_model name get the index.

Return type int

predict(*data*, *predict_for_study=False*, *sort_by_data_id=False*)
Create new prediction with existing solution.

Parameters

- **data** (MRData) – MRData object contains the predict data.
- **predict_for_study** (*bool, optional*) – If *True*, use the random effects information to prediction for specific study. If the *study_id* in *data* do not contain in the fitting data, it will assume the corresponding random effects equal to 0.
- **sort_by_data_id** (*bool, optional*) – If *True*, will sort the final prediction as the order of the original data frame that used to create the *data*. Default to False.

Returns Predicted outcome array.

Return type np.ndarray

sample_soln(*sample_size=1*, *sim_prior=True*, *sim_re=True*, *max_iter=100*, *print_level=0*)
Sample solutions.

Parameters

- **sample_size** (*int, optional*) – Number of samples.
- **sim_prior** (*bool, optional*) – If *True*, simulate priors.
- **sim_re** (*bool, optional*) – If *True*, simulate random effects.
- **max_iter** (*int, optional*) – Maximum number of iterations. Default to 100.
- **print_level** (*int, optional*) – Level detailed of optimization information printed out during sampling process. If 0, no information will be printed out.

Returns Return beta samples and gamma samples.

Return type Tuple[np.ndarray, np.ndarray]

summary()
Return the summary data frame.

Return type Tuple[DataFrame, DataFrame]

class MRBeRT(*data*, *ensemble_cov_model*, *ensemble_knots*, *cov_models=None*, *inlier_pct=1.0*)
Bases: object

Ensemble model of MRBRT.

create_draws(*data*, *beta_samples*, *gamma_samples*, *random_study=True*, *sort_by_data_id=False*)
Create draws. For function description please check *create_draws* for *MRBRT*.

Return type ndarray

fit_model(*x0=None*, *inner_print_level=0*, *inner_max_iter=20*, *inner_tol=1e-08*, *outer_verbose=False*, *outer_max_iter=100*, *outer_step_size=1.0*, *outer_tol=1e-06*, *normalize_trimming_grad=False*, *scores_weights=array([1., 1.])*, *slopes=array([2., 10.])*, *quantiles=array([0.4, 0.4])*)
Fitting the model through limetr.

predict(*data*, *predict_for_study=False*, *sort_by_data_id=False*, *return_avg=True*)
Create new prediction with existing solution.

Parameters **return_avg** (*bool*) – When it is *True*, the function will return an average prediction based on the score, and when it is *False* the function will return a list of predictions from all groups.

Return type ndarray

sample_soln(*sample_size=1*, *sim_prior=True*, *sim_re=True*, *max_iter=100*, *print_level=0*)
Sample solution.

Return type Tuple[List[ndarray], List[ndarray]]

score_model(*scores_weights=array([1., 1.])*, *slopes=array([2., 10.])*, *quantiles=array([0.4, 0.4])*)
Score the model by there fitting and variation.

summary()
Create summary data frame.

Return type Tuple[DataFrame, DataFrame]

create_knots_samples(*data*, *alt_cov_names=None*, *ref_cov_names=None*, *l_zero=True*, *num_splines=50*, *num_knots=5*, *width_pct=0.2*, *return_settings=False*)
Create knot samples for relative risk application.

Parameters

- **data** (MRData) – Data object.
- **alt_cov_names** (*List[str], optional*) – Name of the alternative exposures, if *None* use *['b_0', 'b_1']*. Default to *None*.
- **ref_cov_names** (*List[str], optional*) – Name of the reference exposures, if *None* use *['a_0', 'a_1']*. Default to *None*.
- **l_zero** (*bool, optional*) – If *True*, assume the exposure min is 0. Default to *True*.
- **num_splines** (*int, optional*) – Number of splines. Default to 50.
- **num_knots** (*int, optional*) – Number of the spline knots. Default to 5.
- **width_pct** (*float, optional*) – Minimum percentage distance between knots. Default to 0.2.
- **return_settings** (*bool, optional*) – Returns the knots setting if *True*. Default to *False*.

Returns Knots samples.

Return type np.ndarray

score_sub_models_datafit(*mr*)
score the result of mrbert

score_sub_models_variation(*mr*, *ensemble_cov_model_name*, *n=1*)
score the result of mrbert

Return type float

utils

utils module of the *mrtool* package.

class Matrix
Bases: object

class Polyhedron
Bases: object

get_generators()

class RepType
Bases: object

INEQUALITY = None

avg_integral(*mat*, *spline=None*, *use_spline_intercept=False*)
Compute average integral.

Parameters

- **mat** (*numpy.ndarray*) – Matrix that contains the starting and ending points of the integral or a single column represents the mid-points.
- **spline** (*xspline.XSpline | None, optional*) – Spline integrate over with, when *None* treat the function as linear.
- **use_spline_intercept** (*bool, optional*) – If *True* use all bases from spline, otherwise remove the first bases.

Returns Design matrix when spline is not *None*, otherwise the mid-points.

Return type numpy.ndarray

col_diff_mat(*n*)
column difference matrix

combine_cols(*cols*)
Combine column names into one list of names.

Parameters **cols** (*list{str | list{str}}*) – A list of names of columns or list of column names.

Returns Combined names of columns.

Return type list{str}

empty_array()

expand_array(*array*, *shape*, *value*, *name*)
Expand array when it is empty.

Parameters

- **array** (*np.ndarray*) – Target array. If array is empty, fill in the value. And When it is not empty assert the shape agrees and return the original array.
- **shape** (*Tuple[int]*) – The expected shape of the array.
- **value** (*Any*) – The expected value in final array.
- **name** (*str*) – Variable name of the array (for error message).

Returns Expanded array.

Return type np.ndarray

get_cols(*df*, *cols*)
Return the columns of the given data frame. :param df: Given data frame. :type df: pandas.DataFrame :param cols: Given column name(s), if is *None*, will return a empty data frame. :type cols: str | list{str} | None

Returns The data frame contains the columns.

Return type pandas.DataFrame | pandas.Series

input_cols(*cols*, *append_to=None*, *default=None*)
Process the input column name. :param cols: The input column name(s). :type cols: str | list{str} | None :param append_to: A list keep track of all the column names. :type append_to: list{str} | None, optional :param default: Default value when *cols* is *None*. :type default: str | list{str} | None, optional

Returns The name of the column(s)

Return type str | list{str}

input_gaussian_prior(*prior*, *size*)
Process the input Gaussian prior

Parameters

- **prior** (*numpy.ndarray*) – Either one or two dimensional array, with first group refer to mean and second group refer to standard deviation.
- **size** (*int, optional*) – Size the variable, prior related to.

Returns Prior after processing, with shape (2, size), with the first row store the mean and second row store the standard deviation.

Return type numpy.ndarray

input_laplace_prior(*prior*, *size*)
Process the input Gaussian prior

Parameters

- **prior** (*numpy.ndarray*) – Either one or two dimensional array, with first group refer to mean and second group refer to standard deviation.
- **size** (*int, optional*) – Size the variable, prior related to.

Returns Prior after processing, with shape (2, size), with the first row store the mean and second row store the standard deviation.

Return type numpy.ndarray

input_uniform_prior(*prior*, *size*)
Process the input Gaussian prior

Parameters

- **prior** (*numpy.ndarray*) – Either one or two dimensional array, with first group refer to mean and second group refer to standard deviation.
- **size** (*int, optional*) – Size the variable, prior related to.

Returns Prior after processing, with shape (2, size), with the first row store the mean and second row store the standard deviation.

Return type numpy.ndarray

is_cols(*cols*)
Check variable type fall into the column name category. :param cols: Column names candidate. :type cols: str | list{str} | None

Returns if *col* is either str, list{str} or None

Return type bool

is_gaussian_prior(*prior*, *size=None*)
Check if variable satisfy Gaussian prior format

Parameters **prior** (*numpy.ndarray*) – Either one or two dimensional array, with first group refer to mean and second group refer to standard deviation.

Keyword Arguments **size** (*int | None, optional*) – Size the variable, prior related to.

Returns True if satisfy condition.

Return type bool

is_laplace_prior(*prior*, *size=None*)
Check if variable satisfy Gaussian prior format

Parameters **prior** (*numpy.ndarray*) – Either one or two dimensional array, with first group refer to mean and second group refer to standard deviation.

Keyword Arguments **size** (*int | None, optional*) – Size the variable, prior related to.

Returns True if satisfy condition.

Return type bool

is_numeric_array(*array*)
Check if an array is numeric.

Parameters **array** (*np.ndarray*) – Array need to be checked.

Returns True if the array is numeric.

Return type bool

is_uniform_prior(*prior*, *size=None*)
Check if variable satisfy uniform prior format

Parameters **prior** (*numpy.ndarray*) – Either one or two dimensional array, with first group refer to lower bound and second group refer to upper bound.

Keyword Arguments **size** (*int | None, optional*) – Size the variable, prior related to.

Returns True if satisfy condition.

Return type bool

mat_to_fun(*alt_mat*, *ref_mat=None*)

mat_to_log_fun(*alt_mat*, *ref_mat=None*, *add_one=True*)

nonlinear_trans(*score*, *slope=6.0*, *quantile=0.7*)

ravel_dict(*x*)
Ravel dictionary.

Return type dict

sample_knots(*num_intervals*, *knot_bounds=None*, *interval_sizes=None*, *num_samples=1*)
Sample knots given a set of rules.

Parameters

- **num_intervals** (int) – Number of intervals (number of knots minus 1).

- **knot_bounds** (Optional[ndarray]) – Bounds for the interior knots. Here we assume the domain span 0 to 1, bound for a knot should be between 0 and 1, e.g. `[0.1, 0.2]`. `knot_bounds` should have number of interior knots of rows, and each row is a bound for corresponding knot, e.g. `knot_bounds=np.array([[0.0, 0.2], [0.3, 0.4], [0.3, 1.0]])`, for when we have three interior knots.
- **interval_sizes** (Optional[ndarray]) – Bounds for the distances between knots. For the same reason, we assume elements in *interval_sizes* to be between 0 and 1. For example, `interval_distances=np.array([[0.1, 0.2], [0.1, 0.3], [0.1, 0.5], [0.1, 0.5]])` means that the distance between first (0) and second knot has to be between 0.1 and 0.2, etc. And the number of rows for `interval_sizes` has to be same with `num_intervals`.
- **num_samples** (int) – Number of knots samples.

Returns Return knots sample as array, with *num_samples* rows and number of knots columns.

Return type np.ndarray

sample_simplex(*n*, *N=1*)
sample from n dimensional simplex

sizes_to_indices(*sizes*)
Converting sizes to corresponding indices. :param sizes: An array consist of non-negative number. :type sizes: numpy.dnarray

Returns List the indices.

Return type list{range}

to_list(*obj*)
Convert objective to list of object.

Parameters **obj** (*Any*) – Object need to be convert.

Returns If *obj* already is a list object, return *obj* itself, otherwise wrap *obj* with a list and return it.

Return type List[Any]

3.3.2 mrtool.cov_selection package

Cov Finder

class CovFinder(*data*, *covs*, *pre_selected_covs=None*, *normalized_covs=True*, *num_samples=1000*, *laplace_threshold=1e-05*, *power_range=(-8, 8)*, *power_step_size=0.5*, *inlier_pct=1.0*, *alpha=0.05*, *beta_gprior=None*, *beta_gprior_std=1.0*, *bias_zero=False*, *use_re=None*)
Bases: object

Class in charge of the covariate selection.

create_model(*covs*, *prior_type='Laplace'*, *laplace_std=None*)
Create Gaussian or Laplace model.

Parameters

- **covs** (*List[str]*) – A list of covariates need to be included in the model.
- **prior_type** (*str*) – Indicate if use Gaussian or Laplace model.
- **laplace_std** (*float*) – Standard deviation of the Laplace prior. Default to None.

Returns Created model object.

Return type *MRBRT*

fit_gaussian_model(*covs*)
Fit Gaussian model.

Parameters **covs** (*List[str]*) – A list of covariates need to be included in the model.

Returns the fitted model object.

Return type *MRBRT*

fit_laplace_model(*covs*, *laplace_std*)
Fit Laplace model.

Parameters

- **covs** (*List[str]*) – A list of covariates need to be included in the model.
- **laplace_std** (*float*) – The Laplace prior std.

Returns the fitted model object.

Return type *MRBRT*

fit_pre_selected_covs()
Fit the pre-selected covariates.

static is_significance(*var_samples*, *var_type='beta'*, *alpha=0.05*)

Return type ndarray

loose_gamma_uprior = array([1., 1.])

select_covs(*verbose=False*)

select_covs_by_laplace(*laplace_std*, *verbose=False*)

summary_gaussian_model(*gaussian_model*)
Summary the gaussian model. Return the mean standard deviation and the significance indicator of beta.

Parameters **gaussian_model** (MRBRT) – Gaussian model object.

Returns Mean, standard deviation and indicator of the significance of beta solution.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray]

update_beta_gprior(*covs*, *mean*, *std*)
Update the beta Gaussian prior.

Parameters

- **covs** (*List[str]*) – Name of the covariates.
- **mean** (*np.ndarray*) – Mean of the priors.
- **std** (*np.ndarray*) – Standard deviation of the priors.

zero_gamma_uprior = array([0., 0.])

3.3.3 mrtool.evidence_score package

scorelator

class Scorelator(*ln_rr_draws*, *exposures*, *exposure_domain=None*, *ref_exposure=None*, *score_type='area'*)
Bases: object

Evaluate the score of the result. Warning: This is specifically designed for the relative risk application. Haven't been tested for others.

static annotate_between_curve(*annotation*, *x*, *y_lower*, *y_upper*, *ax*, *mark_area=False*)
Annotate between the curve.

Parameters

- **annotation** (*str*) – the annotation between the curve.
- **x** (*np.ndarray*) – independent variable.
- **y_lower** (*np.ndarray*) – lower bound of the curve.
- **y_upper** (*np.ndarray*) – upper bound of the curve.
- **ax** (*Axes*) – axis of the plot.
- **mark_area** (*bool, optional*) – If True mark the area. Default to False.

get_evidence_score(*lower_draw_quantile=0.025*, *upper_draw_quantile=0.975*, *path_to_diagnostic=None*)
Get evidence score.

Parameters

- **lower_draw_quantile** (*float, optional*) – Lower quantile of the draw for the score.
- **upper_draw_quantile** (*float, optioanl*) – Upper quantile of the draw for the score.
- **path_to_diagnostic** (*Union[str, Path, None], optional*) – Path of where the picture is saved, if None the plot will not be saved. Default to None.

Returns Evidence score.

Return type float

normalize_ln_rr_draws()
Normalize log relative risk draws.

area_between_curves(*lower*, *upper*, *ind_var=None*, *normalize_domain=True*)
Compute area between curves.

Parameters

- **lower** (*np.ndarray*) – Lower bound curve.
- **upper** (*np.ndarray*) – Upper bound curve.
- **ind_var** (*Union[np.ndarray, None], optional*) – Independent variable, if *None*, it will assume sample points are evenly spaced. Default to None.
- **normalize_domain** (*bool, optional*) – If *True*, when *ind_var* is *None*, will normalize domain to 0 and 1. Default to True.

Returns Area between curves.

Return type float

seq_area_between_curves(*lower*, *upper*, *ind_var=None*, *normalize_domain=True*)
Sequence areas_between_curves.

Args: Please check the inputs for area_between_curves.

Returns Return areas defined from the first two points of the curve to the whole curve.

Return type np.ndarray

Python Module Index

m

Index

L

M

N

P

R

S

T

U

Z