
CDR Documentation

Release 0.4.1

Cory Shain

Jul 22, 2021

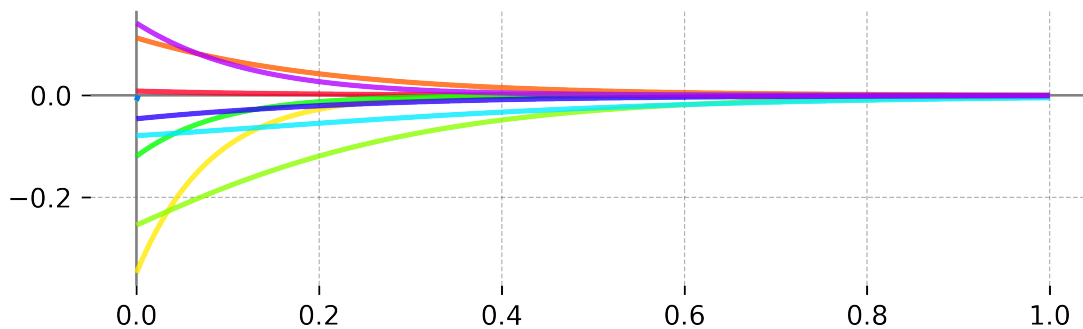
CONTENTS:

1	Introduction	3
1.1	Installation	3
1.2	Basic Usage	4
1.3	References	5
2	Getting Started	7
2.1	Configuring CDR(NN) Experiments	7
2.2	Running CDR executables	8
2.3	Training CDR(NN) Models	8
2.4	Evaluating CDR(NN) Models	9
2.5	CDR(NN) Model Comparison	9
3	CDR Configuration Files	11
3.1	Section: [data]	11
3.2	Section: [global_settings]	12
3.3	Section: [cdr_settings]	13
3.4	Section: [irf_name_map]	20
3.5	Sections: [model_CDR_*]	20
4	CDR Model Formulas	23
4.1	Basic Overview	23
4.2	CDR Model Formulas (not CDRNN)	23
4.3	CDRNN Model Formulas	33
5	IRF Trees in CDR	35
5.1	Factorization	35
5.2	IRF composition	36
5.3	Parameter tying	36
5.4	CDR tree summaries	37
6	CDR Package API	39
6.1	cdr.backend module	39
6.2	cdr.base module	39
6.3	cdr.config module	39
6.4	cdr.data module	40
6.5	cdr.cdrbase module	46
6.6	cdr.cdrbayes module	46
6.7	cdr.cdrmle module	46
6.8	cdr.cdrnnbase module	46
6.9	cdr.cdrnnbayes module	46
6.10	cdr.cdrnnmle module	46

6.11	cdr.formula module	46
6.12	cdr.io module	57
6.13	cdr.kwargs module	58
6.14	cdr.opt module	59
6.15	cdr.plot module	59
6.16	cdr.signif module	62
6.17	cdr.synth module	63
6.18	cdr.util module	65
7	Indices	69
	Python Module Index	71
	Index	73

Documentation for the `cdr` Python package.

INTRODUCTION



In many real world time series, events trigger “ripples” in a dependent variable that unfold slowly and overlap in time (temporal diffusion). Recovering the underlying dynamics of temporally diffuse effects is challenging when events and/or responses occur at irregular intervals. Continuous-time deconvolutional regression (CDR) is a regression technique for time series that directly models temporal diffusion of effects (Shain & Schuler, 2018, 2021). CDR uses machine learning to estimate continuous-time impulse response functions (IRFs) that mediate between predictors (event properties) and responses. Given data and a model template specifying the functional form(s) of the IRF kernel(s), CDR finds IRF parameters that optimize some objective function. This approach can be generalized to account for non-stationary, non-linear, non-additive, and context-dependent response functions by implementing the IRF as a deep neural network (Shain, 2021).

The `cdr` package documented here provides Python implementations of both CDR and CDRNN. This package provides (1) an API for programming with CDR(NN) and (2) executables that allow users to train and evaluate CDR(NN) models out of the box, without needing to write any code. Source code and bug reporting tools are available on [Github](#).

1.1 Installation

Install [anaconda](#) and clone the CDR repository (<https://github.com/coryshain/cdr>) to the directory of your choice. From the CDR repository root, run the following commands to create a new conda environment:

```
conda env create -f conda_cdr.yml
conda activate cdr
```

(Optional) Run the following from the CDR repository root to install CDR system-wide (otherwise, simply run CDR from the repository root):

```
python setup.py install
```

The `cdr` environment must first be activated anytime you want to use the CDR codebase:

```
conda activate cdr
```

1.2 Basic Usage

Once the `cdr` package is installed system-wide as described above (and the `cdr` conda environment is activated via `conda activate cdr`), the `cdr` package can be imported into Python as shown:

```
import cdr
```

Most users will not need to program with CDR(NN), but will instead run command-line executables for model fitting and criticism. These can be run as:

```
python -m cdr.bin.<EXECUTABLE-NAME> ...
```

For documentation of available CDR(NN) executables, run:

```
python -m cdr.bin.help( <SCRIPT-NAME> ) *
```

CDR(NN) models are defined using configuration (`*.ini`) files (fully described in *CDR Configuration Files*), which can be more convenient than shell arguments for specifying many settings at once, and which provide written documentation of the specific settings used to generate any given result. For convenience, we have provided a utility to initialize a new `*.ini` file, which can be run like this:

```
python -m cdr.bin.create_config > PATH.ini
```

This will initialize a CDR-oriented config file. To initialize a CDRNN-oriented config file, add the flag `-t cdrnn`. To initialize a plotting-oriented config file (which defines visualizations to run for an already fitted model), add the flag `-t plot`. To include annotation comments in the output file (which can help with customizing it), add the flag `-a`. The `PATH.ini` file can then be modified as needed to set up the analysis.

CDR model formula syntax resembles R-style model formulas ($DV \sim IV1 + IV2 + \dots$) and is fully described in *CDR Model Formulas*. The core novelty is the `C(preds, IRF)` call (C for “convolve”), in which the first argument is a ‘+’-delimited list of predictors and the second argument is a call to an impulse response kernel (e.g. `Exp`, `Normal`, `ShiftedGammaShapeGT1`, see *CDR Model Formulas* for complete list). For example, a model with the following specification:

```
formula = DV ~ C(a + b + c, Normal())
```

will fit a CDR model that convolves predictors `a`, `b`, `c` using `Normal` IRFs with trainable location and scale parameters.

CDRNN models do not require user-specification of a kernel family, so their formula syntax is simpler:

```
formula = DV ~ a + b + c
```

The response shape to all variables (and all their interactions) will be fitted jointly.

Once a model file has been written (e.g. `model.ini`), the model(s) defined in it can be trained by running:

```
python -m cdr.bin.train model.ini
```

IRF estimates will be incrementally dumped into the output directory specified by `model.ini`, and learning curves can be inspected in Tensorboard:


```
python -m tensorboard.main --logdir=<PATH-TO-CDR-OUTPUT>
```

1.3 References

Shain, Cory and Schuler, William (2018). Deconvolutional time series regression: A technique for modeling temporally diffuse effects. *EMNLP18*.

Shain, Cory and Schuler, William (2021). Continuous-time deconvolutional regression for psycholinguistic modeling. *Cognition*.

Shain, Cory (2021). CDRNN: Discovering complex dynamics in human language processing. *ACL21*.

GETTING STARTED

The `cdr` package provides utilities that support easy training and evaluation of CDR models. Most of these utilities depend on a config file `*.ini` containing metadata needed to construct the CDR model(s). At training time, a copy of the config file is placed in the output directory for the experiment, which can be useful for record keeping. The approach of using config files rather than command line arguments to define experiments has several advantages:

- Users only need to define the experiment once, and settings can be reused for arbitrarily many models
- The config file contains a permanent record of the settings used in the experiment, facilitating later reproduction
- The config file can be shared across many utilities (training, evaluation, statistical testing, plotting, etc.)

This guide describes how to write the config file and use it to train and evaluate CDR(NN) models.

2.1 Configuring CDR(NN) Experiments

A set of CDR experiments is defined in a config file `*.ini`, which can then be passed to CDR executables. For example, once a config file `foo.ini` has been created, the models defined in it can be trained using the included `train` executable, like so::

```
python -m cdr.bin.train foo.ini
```

The config file must at minimum contain the sections `[data]` (with pointers to training/evaluation data) and `[global_settings]` (with the location of the output directory). A section `[cdr_settings]` can be (optionally) provided to override default settings. In addition, the config file should contain at least one model section, consisting of the prefix `model_CDR_` (for CDR) or `model_CDRNN_` (for CDRNN) followed by a custom name for the model. For example, to define a CDR model called `readingtimes`, use the section heading `[model_CDR_readingtimes]`. The identifier `CDR_readingtimes` will then be used by a number of utilities to designate this model (the reason CDR must be included in the prefix is that this repository distributes with unofficial support for running LM, LME, and GAM models from the same interface). The data and settings configurations will be shared between all models defined a single config file. Each model section heading should contain at least the field `formula`. The value provided to `formula` must be a valid CDR model string. Additional fields may be provided to override inherited CDR settings on a model-specific basis. Experiments involving distinct datasets require distinct config files.

For more details on the available configuration parameters, see [CDR Configuration Files](#). For more details on CDR model formulae, see [CDR Model Formulas](#). To jump-start a new config file, run:

```
python -m cdr.bin.create_config > PATH.ini
```

Add the flag `-t cdrnn` to generate a CDRNN-oriented config, and add the flag `-a` to include annotations that can help you fill in the fields correctly.

2.2 Running CDR executables

A number of executable utilities are contained in the module `cdr.bin` and can be executed using Python's `-m` flag. For example, the following call runs the `train` utility on all models specified in a config file `config.ini`:

```
python -m cdr.bin.train config.ini
```

Usage help for each utility can be viewed by running::

```
python -m cdr.bin.<UTIL-NAME> -h
```

for the utility in question. Or usage help for all utilities can be printed at once using::

```
python -m cdr.bin.help
```

The following sections go into more detail about training and evaluation utilities that will likely be useful to most users.

2.3 Training CDR(NN) Models

First, gather some data and write a config file defining your experiments. Note that CDR utilities assume the data will be saved in a **space-delimited text file**, a requirement that facilitates interoperability with the [Modelblocks repository](<https://github.com/modelblocks/modelblocks-release>), which can be used to generate and preprocess data for many psycholinguistic experiments. Once the config file has been written, training a CDR model is simple using the `train` utility, which takes as its principle argument a path to the config file. For example, if the config file is named `experiment.ini`, all models defined in it can be trained with the call::

```
python -m dstr.bin.train experiment.ini
```

To restrict training to some subset of models, the `-m` flag is used. For example, the following call trains models A and B only::

```
python -m cdr.bin.train experiment.ini -m CDR_A CDR_B
```

CDR periodically saves training checkpoints to the model's output directory (every **save_freq** training epochs, where **save_freq** can be defined in the config file or left at its default of 1). This allows training to be interrupted and resumed. To save space, checkpoints overwrite each other, so the output directory will contain only the most recent checkpoint. If the `train` utility discovers a checkpoint in the model's output directory, it loads it and resumes training from the saved state. This feature can also be used to add training epochs after termination in the event of model non-convergence. Simply increase the number of training iterations specified in the config file and rerun `train`, and the model will continue training until it hits the new maximum number of epochs.

CDR uses the history of losses by training iteration to automatically diagnose convergence and stop training, but it's still a good idea to visually inspect learning curves to make sure nothing horrible has happened during the training process (e.g. large loss spikes, which signal a problem and can confuse the convergence checker). CDR learning curves can be visualized in Tensorboard. To run Tensorboard for a CDR model called `CDR_A` saved in experiment directory `EXP`, run:

```
python -m tensorboard.main --logdir=EXP/CDR_A
```

then open the returned address (usually `http://<USERNAME>:6006`) in a web browser.

2.4 Evaluating CDR(NN) Models

This package provides several utilities for inspecting and evaluating fitted CDR models. The principal evaluation utility is `predict`. The following generates predictions on test data from the model `CDR_A` defined in `experiment.ini`:

```
python -m cdr.bin.predict experiment.ini -m CDR_A -p test
```

This call will save files containing elementwise predictions, errors, and likelihoods, along with a performance summary. For more details on usage, run::

```
python -m cdr.bin.predict -h
```

Once `predict` has been run for multiple models, statistical model comparison (permutation test) can be performed using `compare`, as shown::

```
python -m cdr.bin.compare experiment.ini -p test
```

The above call will permutation test pairwise differences in mean squared error on test data for all unique pairs of models defined in `experiment.ini`.

In addition to these core utilities, `convolve` convolves the input predictors using the fitted CDR data transform and saves the data table, and `make_plots` generates IRF plots with basic customization as permitted by the command line arguments.

2.5 CDR(NN) Model Comparison

Two or more CDR(NN) models can be statistically compared using bootstrap testing (specifically, paired permutation tests). This repository provides utilities to support this. To compare models `CDR_A` and `CDR_B` using a mean squared error test statistic on the test partition designated in the `*.ini` file, run:

```
python -m cdr.bin.pt experiment.ini -m CDR_A CDR_B -M err -p test
```

Models can also be compared using a likelihood difference test statistic:

```
python -m cdr.bin.pt experiment.ini -m CDR_A CDR_B -M loglik -p test
```

If you specified ablations within the model definition using the `ablate` keyword (see *CDR Configuration Files*), you can test all single-variable nested ablations in a config file as follows:

```
python -m cdr.bin.pt experiment.ini -a -M loglik -p test
```

A separate utility (`ct`) is also provided to permutation test the difference in *correlation* between true and predicted responses:

```
python -m cdr.bin.ct experiment.ini -a -p test
```

The properties of each of these test statistics govern which is most appropriate to use. Error-based tests are useful for comparing qualitatively different models whose likelihood functions are either not available or not directly comparable (e.g. CDR vs. LME). Likelihood-based tests are useful for comparing CDR models with non-normal error (e.g. \sinh - $\operatorname{arcsinh}$). Correlation-based tests are useful when the primary emphasis is on the strength of linear relationship between predictions and response, abstracting away from shift and scale.

Full usage for these bootstrapping utilities can be viewed by running:

```
python -m cdr.bin.help pt ct
```

CDR CONFIGURATION FILES

The CDR utilities in this module read config files that follow the INI standard. Basic information about INI file syntax can be found e.g. [here](#). This reference assumes familiarity with the INI protocol.

CDR configuration files contain the sections and fields described below.

3.1 Section: [data]

The [data] section supports the following fields:

REQUIRED

- **X_train**: `str`; Path to training data (impulse matrix)
- **y_train**: `str`; Path to training data (response matrix)
- **series_ids**: space-delimited list of `str`; Names of columns used to define unique time series

Note that, unlike e.g. linear models, CDR does not require synchronous predictors (impulses) and responses, which is why separate data objects must be provided for each of these components. If the predictors and responses are synchronous, this is fine. The `X_train` and `y_train` fields can point to the same file. The system will treat each unique combination of values in the columns given in `series_ids` as constituting a unique time series.

OPTIONAL

- **X_dev**: `str`; Path to dev data (impulse matrix)
- **y_dev**: `str`; Path to dev data (response matrix)
- **X_test**: `str`; Path to test data (impulse matrix)
- **y_test**: `str`; Path to test data (response matrix)
- **history_length**: `int`; Length of history window in timesteps (default: 128)
- **filters**: `str`; List of filters to apply to response data (;-delimited).

All variables used in a filter must be contained in the data files indicated by the `y_*` parameters in the [data] section of the config file. The variable name is specified as an INI field, and the condition is specified as its value. Supported logical operators are `<`, `<=`, `>`, `>=`, `==`, and `!=`. For example, to keep only data points for which column `foo` is less or equal to 100, the following filter can be added:

```
filters = foo <= 100
```

To keep only data points for which the column `foo` does not equal `bar`, the following filter can be added:

```
filters = foo != bar
```

Filters can be conjunctively combined:

```
filters = foo > 5; foo <= 100
```

Count-based filters are also supported, using the designated `nunique` suffix. For example, if the column `subject` is being used to define a random effects level and we want to exclude all subjects with fewer than 100 data points, this can be accomplished using the following filter:

```
filters = subjectsnunique > 100
```

More complex filtration conditions are not supported automatically in CDR but can be applied to the data by the user as a preprocess.

Several CDR utilities (e.g. for prediction and evaluation) are designed to handle train, dev, and test partitions of the input data, but these partitions must be constructed in advance. This package also provides a `partition` utility that can be used to partition input data by applying modular arithmetic to some subset of the variables in the data. For usage details run:

```
python -m cdr.bin.partition -h
```

IMPORTANT NOTES

- The files indicated in `X_*` must contain the following columns:
 - **time**: Timestamp associated with each observation
 - A column for each variable in `series_ids`
 - A column for each predictor variable indicated in the model formula
- The file in `y_*` must contain the following columns:
 - **time**: Timestamp associated with each observation
 - A column for the response variable in the model formula
 - A column for each variable in `series_ids`
 - A column for each random grouping factor in the in the model formula
 - A column for each variable used for data filtration (see below)
- Data in `y_*` may be filtered/partitioned, but data in `X_*` **must be uncensored** unless independent reason exists to assume that certain observations never have an impact on the response.

3.2 Section: [global_settings]

The `[global_settings]` section supports the following fields:

- **outdir**: `str`; Path to output directory where checkpoints, plots, and Tensorboard logs should be saved (default: `./cdr_model/`). If it does not exist, this directory will be created. At runtime, the `train` utility will copy the config file to this directory as `config.ini`, serving as a record of the settings used to generate the analysis.
- **use_gpu_if_available**: `bool`; If available, run on GPU. If `False`, always runs on CPU even when system has compatible GPU.

3.3 Section: [cdr_settings]

The [cdr_settings] section supports the following fields:

3.3.1 All models

- **outdir**: `str`; Path to output directory, where logs and model parameters are saved. **Default**: `./cdr_model/`
- **use_distributional_regression**: `bool`; Whether to model all parameters of the predictive distribution as dependent on IRFs of the impulses (distributional regression). If `False`, only the mean depends on the predictors (other parameters of the predictive distribution are treated as constant). **Default (CDR)**: `False`; **Default (CDRNN)**: `True`
- **predictive_distribution_map**: `str` or `None`; Map defining predictive distribution. Can be a space-delimited list of distribution names (one per response variable), a space-delimited list of `';`-delimited tuples matching response variables to distribution names (e.g. `response;Bernoulli`), or `None`, in which case the predictive distribution will be inferred as `Normal` for continuous variables and `Categorical` for categorical variables. **Default**: `None`
- **center_inputs**: `bool`; DISCOURAGED UNLESS YOU HAVE A GOOD REASON, since this can distort rate estimates. Center inputs by subtracting training set means. Can improve convergence speed and reduce vulnerability to local optima. Only affects fitting – prediction, likelihood computation, and plotting are reported on the source values. **Default**: `False`
- **rescale_inputs**: `bool`; Rescale input features by dividing by training set standard deviation. Can improve convergence speed and reduce vulnerability to local optima. Only affects fitting – prediction, likelihood computation, and plotting are reported on the source values. **Default (CDR)**: `False`; **Default (CDRNN)**: `True`
- **standardize_response**: `bool`; Standardize (Z-transform) the response variable implicitly during training using training set mean and variance. Can improve convergence speed and reduce vulnerability to local optima. Only affects fitting – prediction, likelihood computation, and plotting are reported on the source values. **Default**: `True`
- **history_length**: `int`; Length of the history (backward) window (in timesteps). **Default**: `128`
- **future_length**: `int`; Length of the future (forward) window (in timesteps). Note that causal IRF kernels cannot be used if **future_length** > 0. **Default**: `0`
- **asymmetric_error**: `bool`; Whether to model numeric responses by default with an (asymmetric) SinhArcsinh transform of the Normal distribution. Otherwise, defaults to a Normal distribution. Only affects response variables whose distributions have not been explicitly specified using **predictive_distribution_map**. **Default**: `False`
- **constraint**: `str`; Constraint function to use for bounded variables. One of `['abs', 'square', 'softplus']`. **Default**: `softplus`
- **scale_loss_with_data**: `bool`; Whether to multiply the scale of the LL loss by N, where N is num batches. This turns the loss into an expectation over training set likelihood. **Default**: `True`
- **scale_regularizer_with_data**: `bool`; Whether to multiply the scale of all weight regularization by $B * N$, where B is batch size and N is num batches. If **scale_loss_with_data** is true, this approach ensures a stable regularization strength (relative to the loss) across datasets and batch sizes. **Default**: `False`
- **n_iter**: `int`; Number of training iterations. If using variational inference, this becomes the *expected* number of training iterations and is used only for Tensorboard logging, with no impact on training behavior. **Default**: `100000`
- **minibatch_size**: `int` or `None`; Size of minibatches to use for fitting (full-batch if `None`). **Default**: `1024`

- **eval_minibatch_size**: `int`; Size of minibatches to use for prediction/evaluation. **Default**: 10000
- **n_samples_eval**: `int`; Number of posterior predictive samples to draw for prediction/evaluation. Ignored for evaluating CDR MLE models. **Default**: 1000
- **optim_name**: `str` or `None`; Name of the optimizer to use. Must be one of:
 - 'SGD'
 - 'Momentum'
 - 'AdaGrad'
 - 'AdaDelta'
 - 'Adam'
 - 'FTRL'
 - 'RMSProp'
 - 'Nadam' **Default**: Nadam
- **max_global_gradient_norm**: `float` or `None`; Maximum allowable value for the global norm of the gradient, which will be clipped as needed. If `None`, no gradient clipping. **Default (CDR)**: `None`; **Default (CDRNN)**: 1.0
- **epsilon**: `float`; Epsilon parameter to use for numerical stability in bounded parameter estimation. **Default (CDR)**: 1e-05; **Default (CDRNN)**: 0.01
- **optim_epsilon**: `float`; Epsilon parameter to use if **optim_name** in ['Adam', 'Nadam'], ignored otherwise. **Default**: 1e-08
- **learning_rate**: `float`; Initial value for the learning rate. **Default (CDR)**: 0.001; **Default (CDRNN)**: 0.01
- **learning_rate_min**: `float`; Minimum value for the learning rate. **Default**: 0.0
- **lr_decay_family**: `str` or `None`; Functional family for the learning rate decay schedule (no decay if `None`). **Default**: `None`
- **lr_decay_rate**: `float`; coefficient by which to decay the learning rate every `lr_decay_steps` (ignored if `lr_decay_family==None`). **Default**: 0.0
- **lr_decay_steps**: `int`; Span of iterations over which to decay the learning rate by `lr_decay_rate` (ignored if `lr_decay_family==None`). **Default**: 25
- **lr_decay_iteration_power**: `float`; Power to which the iteration number t should be raised when computing the learning rate decay. **Default**: 1
- **lr_decay_staircase**: `bool`; Keep learning rate flat between `lr_decay_steps` (ignored if `lr_decay_family==None`). **Default**: `False`
- **loss_filter_n_sds**: `float` or `None`; How many moving standard deviations above the moving mean of the loss to use as a cut-off for stability (suppressing large losses). If `None`, or 0, no loss filtering. **Default (CDR)**: `None`; **Default (CDRNN)**: 1000.0
- **ema_decay**: `float` or `None`; Decay factor to use for exponential moving average for parameters (used in prediction). **Default**: 0.999
- **convergence_n_iterates**: `int` or `None`; Number of timesteps over which to average parameter movements for convergence diagnostics. If `None` or 0, convergence will not be programmatically checked (reduces memory overhead, but convergence must then be visually diagnosed). **Default (CDR)**: 500; **Default (CDRNN)**: 100
- **convergence_stride**: `int`; Stride (in iterations) over which to compute convergence. If larger than 1, iterations within a stride are averaged with the most recently saved value. Larger values increase the receptive field of the slope estimates, making convergence diagnosis less vulnerable to local perturbations but also increasing the number of post-convergence iterations necessary in order to identify convergence. **Default**: 1

- **convergence_alpha**: float or None; Significance threshold above which to fail to reject the null of no correlation between convergence basis and training time. Larger values are more stringent. **Default**: 0.5
- **regularizer_name**: str or None; Name of global regularizer; can be overridden by more regularizers for more specific parameters (e.g. l1_regularizer, l2_regularizer). If None, no regularization. **Default**: None
- **regularizer_scale**: str or float; Scale of global regularizer; can be overridden by more regularizers for more specific parameters (ignored if regularizer_name==None). **Default**: 0.0
- **coefficient_regularizer_name**: str, "inherit" or None; Name of coefficient regularizer (e.g. l1_regularizer, l2_regularizer); overrides **regularizer_name**. If 'inherit', inherits **regularizer_name**. If None, no regularization. **Default**: inherit
- **coefficient_regularizer_scale**: str, float or "inherit"; Scale of coefficient regularizer (ignored if regularizer_name==None). If 'inherit', inherits **regularizer_scale**. **Default**: inherit
- **intercept_regularizer_name**: str, "inherit" or None; Name of intercept regularizer (e.g. l1_regularizer, l2_regularizer); overrides **regularizer_name**. If 'inherit', inherits **regularizer_name**. If None, no regularization. **Default**: inherit
- **intercept_regularizer_scale**: str, float or "inherit"; Scale of intercept regularizer (ignored if regularizer_name==None). If 'inherit', inherits **regularizer_scale**. **Default**: inherit
- **ranef_regularizer_name**: str, "inherit" or None; Name of random effects regularizer (e.g. l1_regularizer, l2_regularizer); overrides **regularizer_name**. If 'inherit', inherits **regularizer_name**. If None, no regularization. **Default (CDR)**: inherit; **Default (CDRNN)**: l2_regularizer
- **ranef_regularizer_scale**: str, float or "inherit"; Scale of random effects regularizer (ignored if regularizer_name==None). If 'inherit', inherits **regularizer_scale**. **Default (CDR)**: inherit; **Default (CDRNN)**: 10.0
- **save_freq**: int; Frequency (in iterations) with which to save model checkpoints. **Default (CDR)**: 100; **Default (CDRNN)**: 10
- **log_freq**: int; Frequency (in iterations) with which to log model params to Tensorboard. **Default (CDR)**: 100; **Default (CDRNN)**: 1
- **log_random**: bool; Log random effects to Tensorboard. **Default**: True
- **log_graph**: bool; Log the network graph to Tensorboard **Default**: False
- **indicator_names**: str; Space-delimited list of predictors that are indicators (0 or 1). Used for plotting and effect estimation (value 0 is always used as reference, rather than mean). **Default**:
- **default_reference_type**: "mean" or 0.0; Reference stimulus to use by default for plotting and effect estimation. If 0, zero vector. If *mean*, training set mean by predictor. **Default (CDR)**: 0.0; **Default (CDRNN)**: mean
- **reference_values**: str; Predictor values to use as a reference in plotting and effect estimation. Structured as space-delimited pairs NAME=FLOAT. Any predictor without a specified reference value will use either 0 or the training set mean, depending on **plot_mean_as_reference**. **Default**:
- **plot_step**: str; Size of step by predictor to take above reference in univariate IRF plots. Structured as space-delimited pairs NAME=FLOAT. Any predictor without a specified step size will step 1 SD from training set. **Default**:
- **plot_step_default**: str or float; Default size of step to take above reference in univariate IRF plots, if not specified in **plot_step**. Either a float or the string 'sd', which indicates training sample standard deviation. **Default**: 1.0
- **reference_time**: float; Timepoint at which to plot interactions. **Default**: 0.0

- **plot_n_time_units**: `float`; Number of time units to use for plotting. **Default**: 5
- **plot_n_time_points**: `int`; Resolution of plot axis (for 3D plots, uses `sqrt` of this number for each axis). **Default**: 1024
- **plot_x_inches**: `float`; Width of plot in inches. **Default**: 6.0
- **plot_y_inches**: `float`; Height of plot in inches. **Default**: 4.0
- **plot_legend**: `bool`; Whether to include a legend in plots with multiple components. **Default**: `True`
- **generate_curvature_plots**: `bool`; Whether to plot IRF curvature at time **reference_time**. **Default (CDR)**: `False`; **Default (CDRNN)**: `True`
- **generate_irf_surface_plots**: `bool`; Whether to plot IRF surfaces. **Default (CDR)**: `False`; **Default (CDRNN)**: `True`
- **generate_interaction_surface_plots**: `bool`; Whether to plot IRF interaction surfaces at time **reference_time**. **Default**: `True`
- **generate_err_dist_plots**: `bool`; Whether to plot the average error distribution for real-valued responses. **Default**: `True`
- **generate_nonstationarity_surface_plots**: `bool`; Whether to plot IRF surfaces showing non-stationarity in the response. **Default (CDR)**: `False`; **Default (CDRNN)**: `True`
- **cmap**: `str`; Name of Matplotlib `cmap` specification to use for plotting (determines the color of lines in the plot). **Default**: `gist_rainbow`
- **dpi**: `int`; Dots per inch of saved plot image file. **Default**: 300
- **keep_plot_history**: `bool`; Keep IRF plots from each checkpoint of a run, which can help visualize learning trajectories but can also consume a lot of disk space. If `False`, only the most recent plot of each type is kept. **Default**: `False`

3.3.2 All CDR models

- **irf_regularizer_name**: `str`, `"inherit"` or `None`; Name of IRF parameter regularizer (e.g. `l1_regularizer`, `l2_regularizer`); overrides **regularizer_name**. If `'inherit'`, inherits **regularizer_name**. If `None`, no regularization. **Default**: `inherit`
- **irf_regularizer_scale**: `str`, `float` or `"inherit"`; Scale of IRF parameter regularizer (ignored if `regularizer_name==None`). If `'inherit'`, inherits **regularizer_scale**. **Default**: `inherit`

3.3.3 CDRMLE

3.3.4 CDRBayes

- **coef_prior_sd**: `str`, `float` or `None`; Standard deviation of prior on fixed coefficients. Can be a space-delimited list of `;`-delimited floats (one per distributional parameter per response variable), a `float` (applied to all responses), or `None`, in which case the prior is inferred from **prior_sd_scaling_coefficient** and the empirical variance of the response on the training set. **Default**: `None`
- **irf_param_prior_sd**: `str` or `float`; Standard deviation of prior on convolutional IRF parameters. Can be either a space-delimited list of `;`-delimited floats (one per distributional parameter per response variable) or a `float` (applied to all responses) **Default**: 1.0
- **y_sd_prior_sd**: `float` or `None`; Standard deviation of prior on standard deviation of output model. If `None`, inferred as **y_sd_prior_sd_scaling_coefficient** times the empirical variance of the response on the training set. **Default**: `None`

- **prior_sd_scaling_coefficient:** `float`; Factor by which to multiply priors on intercepts and coefficients if inferred from the empirical variance of the data (i.e. if `intercept_prior_sd` or `coef_prior_sd` is `None`). Ignored for any prior widths that are explicitly specified. **Default:** 1
- **y_sd_prior_sd_scaling_coefficient:** `float`; Factor by which to multiply prior on output model variance if inferred from the empirical variance of the data (i.e. if `y_sd_prior_sd` is `None`). Ignored if prior width is explicitly specified. **Default:** 1
- **ranef_to_fixef_prior_sd_ratio:** `float`; Ratio of widths of random to fixed effects priors. I.e. if less than 1, random effects have tighter priors. **Default:** 0.1
- **posterior_to_prior_sd_ratio:** `float`; Ratio of posterior initialization SD to prior SD. Low values are often beneficial to stability, convergence speed, and quality of final fit by avoiding erratic sampling and divergent behavior early in training. **Default:** 0.01

3.3.5 All CDRNN models

- **center_X_time:** `bool`; Whether to center time values as inputs under the hood. Times are automatically shifted back to the source location for plotting and model criticism. **Default:** `False`
- **center_t_delta:** `bool`; Whether to center time offset values under the hood. Offsets are automatically shifted back to the source location for plotting and model criticism. **Default:** `False`
- **rescale_X_time:** `bool`; Whether to rescale time values as inputs by their training SD under the hood. Times are automatically reconverted back to the source scale for plotting and model criticism. **Default:** `True`
- **rescale_t_delta:** `bool`; Whether to rescale time offset values by their training SD under the hood. Offsets are automatically reconverted back to the source scale for plotting and model criticism. **Default:** `True`
- **nonstationary:** `bool`; Whether to model non-stationarity by feeding impulse timestamps as input. **Default:** `True`
- **n_layers_input_projection:** `int` or `None`; Number of hidden layers in input projection. If `None`, inferred from length of `n_units_input_projection`. **Default:** 2
- **n_units_input_projection:** `int`, `str` or `None`; Number of units per input projection hidden layer. Can be an `int`, which will be used for all layers, or a `str` with `n_layers_rnn` space-delimited integers, one for each layer in order from bottom to top. If 0 or `None`, no hidden layers in input projection. **Default:** 32
- **n_layers_rnn:** `int` or `None`; Number of RNN layers. If `None`, inferred from length of `n_units_rnn`. **Default:** `None`
- **n_units_rnn:** `int`, `str` or `None`; Number of units per RNN layer. Can be an `int`, which will be used for all layers, or a `str` with `n_layers_rnn` space-delimited integers, one for each layer in order from bottom to top. Can also be `'infer'`, which infers the size from the number of predictors, or `'inherit'`, which uses size `n_units_hidden_state`. If 0 or `None`, no RNN encoding (i.e. use a context-independent convolution kernel). **Default:** `None`
- **n_layers_rnn_projection:** `int` or `None`; Number of hidden layers in projection of RNN state (or of timestamp + predictors if no RNN). If `None`, inferred automatically. **Default:** `None`
- **n_units_rnn_projection:** `int`, `str` or `None`; Number of units per hidden layer in projection of RNN state. Can be an `int`, which will be used for all layers, or a `str` with `n_units_rnn_projection` space-delimited integers, one for each layer in order from bottom to top. If 0 or `None`, no hidden layers in RNN projection. **Default:** `None`
- **n_units_hidden_state:** `int` or `str`; Number of units in CDRNN hidden state. Must be an `int`. **Default:** 32
- **n_layers_irf:** `int` or `None`; Number of IRF hidden layers. If `None`, inferred from length of `n_units_irf`. **Default:** 2

- **n_units_irf**: `int`, `str` or `None`; Number of units per hidden layer in IRF. Can be an `int`, which will be used for all layers, or a `str` with **n_units_irf** space-delimited integers, one for each layer in order from bottom to top. If 0 or `None`, no hidden layers. **Default**: 32
- **input_projection_inner_activation**: `str` or `None`; Name of activation function to use for hidden layers in input projection. **Default**: `gelu`
- **input_projection_activation**: `str` or `None`; Name of activation function to use for output of input projection. **Default**: `None`
- **rnn_activation**: `str` or `None`; Name of activation to use in RNN layers. **Default**: `tanh`
- **recurrent_activation**: `str` or `None`; Name of recurrent activation to use in RNN layers. **Default**: `sigmoid`
- **rnn_projection_inner_activation**: `str` or `None`; Name of activation function to use for hidden layers in projection of RNN state. **Default**: `gelu`
- **rnn_projection_activation**: `str` or `None`; Name of activation function to use for final layer in projection of RNN state. **Default**: `None`
- **hidden_state_activation**: `str` or `None`; Name of activation function to use for CDRNN hidden state. **Default**: `gelu`
- **irf_inner_activation**: `str` or `None`; Name of activation function to use for hidden layers in IRF. **Default**: `gelu`
- **irf_activation**: `str` or `None`; Name of activation function to use for final layer in IRF. **Default**: `None`
- **kernel_initializer**: `str` or `None`; Name of initializer to use in encoder kernels. **Default**: `glorot_uniform_initializer`
- **recurrent_initializer**: `str` or `None`; Name of initializer to use in encoder recurrent kernels. **Default**: `orthogonal_initializer`
- **batch_normalization_decay**: `float` or `None`; Decay rate to use for batch normalization in internal layers. If `None`, no batch normalization. **Default**: `None`
- **normalize_input_projection**: `bool`; Whether to apply normalization (if applicable) to hidden layers of the input projection. **Default**: `True`
- **normalize_h**: `bool`; Whether to apply normalization (if applicable) to the hidden state. **Default**: `True`
- **normalize_irf_l1**: `bool`; Whether to apply normalization (if applicable) to the first IRF layer. **Default**: `False`
- **normalize_irf**: `bool`; Whether to apply normalization (if applicable) to non-initial internal IRF layers. **Default**: `False`
- **normalize_after_activation**: `bool`; Whether to apply normalization (if applicable) after the non-linearity (otherwise, applied before). **Default**: `True`
- **normalization_use_gamma**: `bool`; Whether to use trainable scale in batch/layer normalization layers. **Default**: `True`
- **layer_normalization_type**: `str` or `None`; Type of layer normalization, one of [`'z'`, `'length'`, `None`]. If `'z'`, classical z-transform-based normalization. If `'length'`, normalize by the norm of the activation vector. If `None`, no layer normalization. Incompatible with batch normalization. **Default**: `None`
- **nn_regularizer_name**: `str`, `"inherit"` or `None`; Name of weight regularizer (e.g. `l1_regularizer`, `l2_regularizer`); overrides **regularizer_name**. If `'inherit'`, inherits **regularizer_name**. If `None`, no regularization. **Default**: `l2_regularizer`
- **nn_regularizer_scale**: `str`, `float` or `"inherit"`; Scale of weight regularizer (ignored if `regularizer_name==None`). If `'inherit'`, inherits **regularizer_scale**. **Default**: 1.0

- **context_regularizer_name:** `str`, "inherit" or `None`; Name of regularizer on contribution of context (RNN) to hidden state (e.g. `l1_regularizer`, `l2_regularizer`); overrides **regularizer_name**. If 'inherit', inherits **regularizer_name**. If `None`, no regularization. **Default:** `l2_regularizer`
- **context_regularizer_scale:** `float` or "inherit"; Scale of weight regularizer (ignored if `context_regularizer_name==None`). If 'inherit', inherits **regularizer_scale**. **Default:** `1.0`
- **maxnorm:** `float` or `None`; Bound on norm of dense kernel dimensions for max-norm regularization. If `None`, no max-norm regularization. **Default:** `None`
- **input_dropout_rate:** `float` or `None`; Rate at which to drop `input_features`. **Default:** `None`
- **input_projection_dropout_rate:** `float` or `None`; Rate at which to drop neurons of input projection layers. **Default:** `0.2`
- **rnn_h_dropout_rate:** `float` or `None`; Rate at which to drop neurons of RNN hidden state. **Default:** `None`
- **rnn_c_dropout_rate:** `float` or `None`; Rate at which to drop neurons of RNN cell state. **Default:** `None`
- **h_in_dropout_rate:** `float` or `None`; Rate at which to drop neurons of `h_in`. **Default:** `0.2`
- **h_rnn_dropout_rate:** `float` or `None`; Rate at which to drop neurons of `h_rnn`. **Default:** `0.2`
- **h_dropout_rate:** `float` or `None`; Rate at which to drop neurons of `h`. **Default:** `None`
- **rnn_dropout_rate:** `float` or `None`; Rate at which to entirely drop the RNN. **Default:** `None`
- **irf_dropout_rate:** `float` or `None`; Rate at which to drop neurons of IRF layers. **Default:** `0.2`
- **ranef_dropout_rate:** `float` or `None`; Rate at which to drop random effects indicators. **Default:** `0.2`

3.3.6 CDRNNMLE

- **weight_sd_init:** `float` or `str`; Standard deviation of kernel initialization distribution (Normal, mean=0). Can also be 'glorot', which uses the SD of the Glorot normal initializer. **Default:** `glorot`

3.3.7 CDRNNBayes

- **declare_priors_weights:** `bool`; Specify Gaussian priors for all fixed model parameters (if `False`, use implicit improper uniform priors). **Default:** `True`
- **declare_priors_biases:** `bool`; Specify Gaussian priors for model biases (if `False`, use implicit improper uniform priors). **Default:** `False`
- **declare_priors_gamma:** `bool`; Specify Gaussian priors for gamma parameters of any batch normalization layers (if `False`, use implicit improper uniform priors). **Default:** `False`
- **weight_prior_sd:** `str` or `float`; Standard deviation of prior on CDRNN hidden weights. A `float`, 'glorot', or 'he'. **Default:** `glorot`
- **bias_prior_sd:** `str` or `float`; Standard deviation of prior on CDRNN hidden biases. A `float`, 'glorot', or 'he'. **Default:** `1.0`
- **gamma_prior_sd:** `str` or `float`; Standard deviation of prior on batch norm gammas. A `float`, 'glorot', or 'he'. Ignored unless batch normalization is used **Default:** `glorot`
- **y_sd_prior_sd:** `float` or `None`; Standard deviation of prior on standard deviation of output model. If `None`, inferred as **y_sd_prior_sd_scaling_coefficient** times the empirical variance of the response on the training set. **Default:** `None`

- **prior_sd_scaling_coefficient**: `float`; Factor by which to multiply priors on intercepts and coefficients if inferred from the empirical variance of the data (i.e. if `intercept_prior_sd` or `coef_prior_sd` is `None`). Ignored for any prior widths that are explicitly specified. **Default**: 1
- **y_sd_prior_sd_scaling_coefficient**: `float`; Factor by which to multiply prior on output model variance if inferred from the empirical variance of the data (i.e. if `y_sd_prior_sd` is `None`). Ignored if prior width is explicitly specified. **Default**: 1
- **ranef_to_fixef_prior_sd_ratio**: `float`; Ratio of widths of random to fixed effects priors. I.e. if less than 1, random effects have tighter priors. **Default**: 0.1
- **weight_sd_init**: `str, float` or `None`; Initial standard deviation of variational posterior over weights. If `None`, inferred from other hyperparams. **Default**: `None`
- **bias_sd_init**: `str, float` or `None`; Initial standard deviation of variational posterior over biases. If `None`, inferred from other hyperparams. **Default**: `None`
- **gamma_sd_init**: `str, float` or `None`; Initial standard deviation of variational posterior over batch norm gammas. If `None`, inferred from other hyperparams. Ignored unless batch normalization is used. **Default**: `None`
- **posterior_to_prior_sd_ratio**: `float`; Ratio of posterior initialization SD to prior SD. Low values are often beneficial to stability, convergence speed, and quality of final fit by avoiding erratic sampling and divergent behavior early in training. **Default**: 0.01

3.4 Section: `[irf_name_map]`

The optional `[irf_name_map]` section simply permits prettier variable naming in plots. For example, the internal name for a convolution applied to predictor A may be `ShiftedGammaKgt1.s(A)-Terminal.s(A)`, which is not very readable. To address this, the string above can be mapped to a more readable name using an INI key-value pair, as shown:

```
ShiftedGammaKgt1.s(A)-Terminal.s(A) = A
```

The model will then print A in plots rather than `ShiftedGammaKgt1.s(A)-Terminal.s(A)`. Unused entries in the name map are ignored, and model variables that do not have an entry in the name map print with their default internal identifier.

3.5 Sections: `[model_CDR_*]`

Arbitrarily many sections named `[model_CDR_*]` can be provided in the config file, where `*` stands in for a unique identifier. Each such section defines a different CDR model and must contain at least one field — `formula` — whose value is a CDR model formula (see *CDR Model Formulas* for more on CDR formula syntax) The identifier `CDR_*` will be used by the CDR utilities to reference the fitted model and its output files.

For example, to define a CDR model called `readingtimes`, the section header `[model_CDR_readingtimes]` is included in the config file along with an appropriate `formula` specification. To use this specific model once fitted, it can be referenced using the identifier `CDR_readingtimes`. For example, the following call will extract predictions on dev data from a fitted `CDR_readingtimes` defined in config file **config.ini**:

```
python -m cdr.bin.predict config.ini -m CDR_readingtimes -p dev
```

Additional fields from `[cdr_settings]` may be specified for a given model, in which case the locally-specified setting (rather than the globally specified setting or the default value) will be used to train the model. For example, imagine that `[cdr_settings]` contains the field `n_iter = 1000`. All CDR models subsequently specified in

the config file will train for 1000 iterations. However, imagine that model `[model_CDR_longertrain]` should train for 5000 iterations instead. This can be specified within the same config file as:

```
[model_CDR_longertrain]
n_iter = 5000
formula = ...
```

This setup allows a single config file to define a variety of CDR models, as long as they all share the same data. Distinct datasets require distinct config files.

For hypothesis testing, fixed effect ablation can be conveniently automated using the `ablate` model field. For example, the following specification implicitly defines 7 unique models, one for each of the $|\text{powerset}(a, b, c)| - 1 = 7$ non-null ablations of `a`, `b`, and `c`:

```
[model_CDR_example]
n_iter = 5000
ablate = a b c
formula = C(a + b + c, Normal()) + (C(a + b + c, Normal()) | subject)
```

The ablated models are named using `'!'` followed by the ablated impulse name for each ablated impulse. Therefore, the above specification is equivalent to (and much easier to write than) the following:

```
[model_CDR_example]
n_iter = 5000
formula = C(a + b + c, Normal()) + (C(a + b + c, Normal()) | subject)

[model_CDR_example!a]
n_iter = 5000
formula = C(b + c, Normal()) + (C(a + b + c, Normal()) | subject)

[model_CDR_example!b]
n_iter = 5000
formula = C(a + c, Normal()) + (C(a + b + c, Normal()) | subject)

[model_CDR_example!c]
n_iter = 5000
formula = C(a + b, Normal()) + (C(a + b + c, Normal()) | subject)

[model_CDR_example!a!b]
n_iter = 5000
formula = C(c, Normal()) + (C(a + b + c, Normal()) | subject)

[model_CDR_example!a!c]
n_iter = 5000
formula = C(b, Normal()) + (C(a + b + c, Normal()) | subject)

[model_CDR_example!b!c]
n_iter = 5000
formula = C(a, Normal()) + (C(a + b + c, Normal()) | subject)
```


CDR MODEL FORMULAS

4.1 Basic Overview

This package constructs CDR models from **R**-style formula strings defining the model structure. A CDR formula has the following template:

```
RESPONSE ~ FIXED_EFFECTS + RANDOM_EFFECTS
```

The left-hand side (LHS) of the formula contains the name a single (possibly transformed) variable from the input data table, and the right-hand side (RHS) contains fixed and random effects, each of which must consist exclusively of intercept terms and/or convolutional terms. Intercept terms can be added by including `1` in the RHS and removed by including `0` in the RHS. If neither of these appears in the RHS, an intercept is added by default.

WARNING: The compiler for CDR formulas is still early in development and may fail to correctly process certain formula strings, especially ones that contain hierarchical term expansions, categorical variable expansions, and/or interactions. Please double-check the correctness of the model formula reported at the start of training and report problems in the issue tracker on [Github](#).

WARNING: If you are using interaction terms, it is easy to accidentally mis-specify your formula because interactions are trickier in CDR than in linear models. Make sure to read *Interactions in CDR* before fitting models with interactions.

4.2 CDR Model Formulas (not CDRNN)

4.2.1 Linear (DiracDelta) Predictors

A trivial way of “deconvolving” a predictor `A` is to assume a Dirac delta response to `A`: `A` at $t = 0$, `0` otherwise. DiracDelta IRF are thus equivalent to predictors in linear models; only the value of the predictor that coincides with the response measurement is used for prediction. Linear (mixed) models of time series are therefore a special subtype of CDR, namely CDR models that exclusively use Dirac delta response kernels. While you therefore *can* use this package to fit LME models, that doesn’t mean you should! The stochastic optimization used here is likely much less efficient in this simple case than that of specialized LME regression tools.

However, the availability of Dirac delta responses in this package means that CDR can combine linear and convolved substructures into a single model, which is important for many use cases. For example, imagine a bi-variate model with predictors `stimulus` and `condition`, where `stimulus` varies within each time series can might engender a temporally diffuse response while `condition` is held fixed within each time series. In this case, a convolution of `condition` would be meaningless (or at least perfectly confounded with `rate`, the deconvolutional intercept); instead, we’re interested in a *linear* effect of `condition` and a *convolved* effect of `stimulus` on the response. Dirac delta responses can specified manually using the syntax described in the following sections. However, for convenience, bare predictor names in model formulas (outside of convolution calls `C()`) are automatically interpreted

as Dirac delta. This allows you to retain stock formula syntax that might already be familiar from linear (mixed) models in **R** in order to define linear substructures of your CDR model. For example, the following is interpreted as specifying linear effects for each of A, B, and the interaction A:B:

```
y ~ A + B + C:B
```

An important caveat in using Dirac delta responses is that the predictors they apply to should be *response-aligned*, that is, measured at the same moments in time that the responses themselves are. If the stimuli and responses are measured at different points in time, you cannot fit a Dirac delta response to a stimulus-aligned variable unless that variable happens at least sometimes to accidentally be measured concurrently with the response. CDR does not internally distinguish stimulus-aligned and response-aligned predictors, so it's up to users to make sure that models are sensible in this regard. However, the model does internally mask all elements in a Dirac delta predictor that are not simultaneous with the response. Therefore, fitting a Dirac delta response to a stimulus-aligned predictor will not produce an erroneous model; in the worst case, the entire predictor will be masked and therefore ignored by the model.

4.2.2 Defining an Impulse Response Function (IRF)

A convolutional term is defined using the syntax `C(..., IRF_FAMILY())`, where `...` is replaced by names of predictors contained in the input data. For example, to define a Gamma convolution of predictor A, the expression `C(A, Gamma())` is added to the RHS. Separate terms are delimited by `+`. For example, to add a Gaussian convolution of predictor B, the RHS above becomes `C(A, Gamma()) + C(B, Normal())`.

4.2.3 Supported Parametric IRFs

The currently supported parametric IRF families are the following. All IRFs are normalized to integrate to 1 over the positive real line. For simplicity, normalization constants are omitted from the equations below. For details, see Shain & Schuler (2021).

- **DiracDelta**: Stick function (equivalent to a predictor in linear regression, see `linear`)
 - Parameters:
 - * None
 - Definition: 1 at $x = 0$, 0 otherwise
- **Exp**: PDF of exponential distribution
 - Parameters:
 - * $\beta > 0$: beta (rate)
 - Definition: $\beta e^{-\beta x}$
 - Note: Dropping the normalization constant from the PDF helps deconfound IRF shape and magnitude. Normalization is unnecessary since this kernel defines an IRF, not a probability distribution.
- **Gamma**: PDF of gamma distribution
 - Parameters:
 - * $\alpha > 0$: alpha (shape)
 - * $\beta > 0$: beta (rate)
 - Definition: $\frac{\beta^\alpha x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha)}$
- **ShiftedGamma**: PDF of gamma distribution with support starting at $0 + \delta$
 - Parameters:

- * $\alpha > 0$: alpha (shape)
- * $\beta > 0$: beta (rate)
- * $\delta < 0$: delta (shift)
- Definition: $\frac{\beta^\alpha (x-\delta)^{\alpha-1} e^{-\frac{x-\delta}{\beta}}}{\Gamma(\alpha)}$
- GammaShapeGT1: PDF of gamma distribution, $\alpha > 1$ (enforces rising-then-falling shape)
 - Parameters:
 - * $\alpha > 1$: alpha (shape)
 - * $\beta > 1$: beta (rate)
 - Definition: $\frac{\beta^\alpha x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha)}$
- ShiftedGammaShapeGT1: PDF of gamma distribution with support starting at $0 + \delta$, $\alpha > 1$ (enforces rising-then-falling shape)
 - Parameters:
 - * $\alpha > 1$: alpha (shape)
 - * $\beta > 0$: beta (rate)
 - * $\delta < 0$: delta (shift)
 - Definition: $\frac{\beta^\alpha (x-\delta)^{\alpha-1} e^{-\frac{x-\delta}{\beta}}}{\Gamma(\alpha)}$
- Normal: PDF of Gaussian (normal) distribution
 - Parameters:
 - * μ : mu (mean)
 - * $\sigma^2 > 0$: sigma2 (variance)
 - Definition: $\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
 - Note: Dropping the normalization constant from the PDF helps deconfound IRF shape and magnitude. Normalization is unnecessary since this kernel defines an IRF, not a probability distribution.
- SkewNormal: PDF of SkewNormal distribution (normal distribution augmented with left/right skew parameter)
 - Parameters:
 - * μ (mean)
 - * $\sigma > 0$ (standard deviation)
 - * α (skew)
 - Definition: Let ϕ and Φ denote the PDF and CDF (respectively) of the standard normal distribution. Then the SkewNormal distribution is: $\frac{2}{\sigma} \phi\left(\frac{x-\mu}{\sigma}\right) \Phi\left(\alpha \frac{x-\mu}{\sigma}\right)$
- EMG: PDF of exponentially modified gaussian distribution (convolution of a normal with an exponential distribution, can be right-skewed)
 - Parameters:
 - * μ : mu (mean)
 - * $\sigma > 0$: sigma (standard deviation)
 - * $\beta > 0$: beta (rate)

- Definition: $\frac{\beta}{2} e^{\frac{\beta}{2}(2\mu+\beta\sigma^2-2x)} \operatorname{erfc}\left(\frac{m+\beta\sigma^2-x}{\sqrt{2}\sigma}\right)$, where $\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$.
- BetaPrime: PDF of BetaPrime (inverted beta) distribution
 - Parameters:
 - * $\alpha > 0$: alpha (shape)
 - * $\beta > 0$: beta (shape)
 - Definition: $\frac{x^{\alpha-1}(1+x)^{-\alpha-\beta}}{B(\alpha,\beta)}$
- ShiftedBetaPrime: PDF of BetaPrime (inverted beta) distribution with support starting at $0 + \delta$
 - Parameters:
 - * $\alpha > 0$: alpha (shape)
 - * $\beta > 0$: beta (shape)
 - * $\delta < 0$: delta (shift)
 - Definition: $\frac{(x-\delta)^{\alpha-1}(1+(x-\delta))^{-\alpha-\beta}}{B(\alpha,\beta)}$
- HRFSingleGamma: Single-gamma hemodynamic response function (fMRI). Identical to GammaShapeGT1 except in its initial parameter values, which are inherited from the peak response model of the canonical HRF in SPM ($\alpha = 6$ and $\beta = 1$)
 - Parameters:
 - * $\alpha > 0$: alpha (shape)
 - * $\beta > 0$: beta (rate)
 - Definition: $\frac{\beta^\alpha x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha)}$
- HRFDoubleGamma1: 1-parameter double-gamma hemodynamic response function (fMRI). Shape parameters are fixed at SPM's defaults for both the first and second gammas (6 and 16, respectively). Parameter β is tied between both gammas. The coefficient on the second gamma is fixed at SPM's default ($\frac{1}{6}$). This is a "stretchable" canonical HRF.
 - Parameters:
 - * $\beta > 0$: beta (peak and undershoot rate)
 - Definition: $\frac{\beta^6 x^{6-1} e^{-\frac{x}{\beta}}}{\Gamma(6)} - \frac{1}{6} \frac{\beta^{16} x^{15} e^{-\frac{x}{\beta}}}{\Gamma(16)}$
- HRFDoubleGamma2: 2-parameter double-gamma hemodynamic response function (fMRI). Parameter α of the second gamma is fixed to the *alpha* of the first gamma using SPM's default offset (10). Parameter β is tied between both gammas. The coefficient on the second gamma is fixed at SPM's default ($\frac{1}{6}$).
 - Parameters:
 - $\alpha > 1$: alpha (peak shape)
 - $\beta > 0$: beta (peak and undershoot rate)
 - Definition: $\frac{\beta^\alpha x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha)} - \frac{1}{6} \frac{\beta^{\alpha+10} x^{\alpha+9} e^{-\frac{x}{\beta}}}{\Gamma(\alpha+10)}$
 - HRFDoubleGamma3: 3-parameter double-gamma hemodynamic response function (fMRI). Parameter α of the second gamma is fixed to the *alpha* of the first gamma using SPM's default offset (10). Parameter β is tied between both gammas.
 - Parameters:

- $\alpha > 1$: alpha (peak shape)
- $\beta > 0$: beta (peak and undershoot rate)
- c : c (undershoot coefficient)
- Definition: $\frac{\beta^\alpha x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha)} - c \frac{\beta^{\alpha+10} x^{\alpha+9} e^{-\frac{x}{\beta}}}{\Gamma(\alpha+10)}$
- HRFDoubleGamma4: 4-parameter double-gamma hemodynamic response function (fMRI). Parameter β is tied between both gammas.
 - Parameters:
 - * $\alpha_1 > 1$: alpha_main (peak shape)
 - * $\alpha_2 > 1$: alpha_undershoot (undershoot shape)
 - * $\beta > 0$: beta (peak and undershoot rate)
 - * c : c (undershoot coefficient)
 - Definition: $\frac{\beta^{\alpha_1} x^{\alpha_1-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha_1)} - c \frac{\beta^{\alpha_2} x^{\alpha_2-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha_2)}$
- HRFDoubleGamma5: 5-parameter double-gamma hemodynamic response function (fMRI). All parameters are free.
 - Parameters:
 - * $\alpha_1 > 1$: alpha_main (peak shape)
 - * $\alpha_2 > 1$: alpha_undershoot (undershoot shape)
 - * $\beta_1 > 0$: beta_main (peak rate)
 - * $\beta_2 > 0$: beta_undershoot (undershoot rate)
 - * c : c (undershoot coefficient)
 - Definition: $\frac{\beta^{\alpha_1} x^{\alpha_1-1} e^{-\frac{x}{\beta_1}}}{\Gamma(\alpha_1)} - c \frac{\beta^{\alpha_2} x^{\alpha_2-1} e^{-\frac{x}{\beta_2}}}{\Gamma(\alpha_2)}$

4.2.4 Interactions in CDR

In comparison to interactions in linear models, deconvolution introduces the additional complexity of needing to decide and specify whether interactions precede (impulse-level interactions) or follow (response-level interactions) the convolution step. Impulse-level interactions consider interactions as *events* which may trigger a temporally diffuse response (i.e. a response to both A and B happening together at a particular point in time). Response-level interactions capture non-additive effects of multiple (possibly convolved) variables; they do not get their own impulse responses. Response-level interactions correspond to interactions in linear models and are almost always what you want except in the special case of linear (DiracDelta IRF) predictors, where impulse-level interactions should be used (just like in linear models).

CDR formulas use a simple syntax to distinguish these two types of interactions: impulse-level interactions are specified *inside* the first argument of convolution calls $C()$, while response-level interactions are specified outside them. As in **R**, interaction terms are designated with $:$, as in $A:B$. And as in **R**, for convenience, two-way cross-product interactions can be designated with $*$ (e.g. $A*B$ is shorthand for $A + B + A:B$) and multi-way cross-product interactions can be designated with power notation $^{\langle \text{INT} \rangle}$ or $**\langle \text{INT} \rangle$ (e.g. $(A+B+C)^3$ equals $A + B + C + A:B + B:C + A:C + A:B:C$). The following defines an impulse-level interaction between A and B underneath a Normal IRF kernel:

```
C(A:B, Normal())
```

The following defines a response-level interaction between Normal convolutions of A and B:

```
C(A, Normal()):C(B, Normal())
```

In order to fit interactions between convolved variables, the convolutions themselves must exist. Therefore, unlike linear interactions, which can be fit even if their subcomponents are not included in the model, $C(A, Normal()):C(B, Normal())$ requires the existence of model estimates for both $C(A, Normal())$ and $C(B, Normal())$, and these terms are therefore automatically inserted when used by any response-level interactions.

Response-level interactions do not need to be convolved variables. They can also be predictors supplied by the data *as long as the predictors are response-aligned* (i.e. measured concurrently with the responses, rather than the impulses). For example, suppose we have a response-aligned variable C provided by our data. We can interact responses with it, like so:

```
C(A, Normal()):C
```

This will fit a normal response to A , along with an estimate for the modulation of that response by C . Unlike convolved inputs to response-level interactions, estimates for regular variables are not automatically added to the model. In order to fit a separate (linear) effect for C , we could use the multiplication operator instead:

```
C(A, Normal())*C = C(A, Normal()) + C + C(A, Normal()):C
```

For convenience, response-level interactions distribute across the inputs to a convolution call $C()$. Thus, interacting a variable with a convolution of multiple inputs is equivalent to interacting the variable with a convolution of each of the inputs:

```
C(A + B, Gamma()):C = C(A + B, Gamma()) + C(A, Gamma()):C + C(B, Gamma()):C
```

Similarly, interacting multiple convolution calls each containing multiple inputs is equivalent to defining interactions over the Cartesian-product of the responses to the two sets of inputs:

```
C(A + B, Gamma()):C(C + D, EMG()) = C(A + B, Gamma()) + C(C + D, EMG()) + \
                                     C(A, Gamma()):C(C, EMG()) + C(B, Gamma()):C(C, \
↪EMG()) + \
                                     C(A, Gamma()):C(D, EMG()) + C(B, Gamma()):C(D, \
↪EMG())
```

Order of operations between term expansions can be enforced through parentheses:

```
(A*B):E = A:E + B:E + A:B:E
A*(B:E) = A + B:E + A:B:E
```

4.2.5 Automatic Term Expansion

For convenience, the $C()$ function distributes the impulse response family over multiple $+$ -delimited terms in its first argument. Therefore, the following two expressions are equivalent:

```
C(A + B, Gamma())
C(A, Gamma()) + C(B, Gamma())
```

R-style expansions for interactions are also available, as discussed above. IRF distribute across the expansion of interaction terms, such that the following expressions are equivalent:

```
C((A + B + C)**3, Gamma())
C(A, Gamma()) + C(B, Gamma()) + C(C, Gamma()) + C(A:B, Gamma()) + C(B:C, Gamma()) + \
↪C(A:C, Gamma()) + C(A:B:C, Gamma())
```


Categorical variables are automatically discovered and expanded in CDR models. This process imposes a transformation on the model. For example, imagine that predictor B in the following model turns out to be categorical in the data set with categories B_1 , B_2 , and B_3 :

```
C(A + B, EMG())
```

When the CDR model is initialized, the categorical nature of B is detected and the model is expanded out as:

```
C(A + B2 + B3, EMG())
```

However, they can be included simply by adding binary indicator vectors for each of $n - 1$ of the levels of the variable to the input data as a preprocessing step, then defining the model in terms of the binary indicators.

Note that the term expansions described above add *separate* IRF for each term in the expansion. For example, $C(A + B, \text{Gamma}())$ adds two distinct Gamma IRF parameterizations to the model, one for each predictor. It is also possible to tie IRF between predictor variables (details below).

Note also that (unlike **R**) redundant terms are **not** automatically collapsed, so care must be taken to ensure that no duplicate terms are produced via term expansion.

4.2.6 Random Effects

Random effects in CDR are specified using the following syntax:

```
(RANDOM_TERMS | GROUPING_FACTOR)
```

where `RANDOM_TERMS` are terms as they would appear in the RHS of the model described above and `GROUPING_FACTOR` is the name of a categorical variable in the input that is used to define the random effect (e.g. a vector of ID's of human subjects). As in the case of fixed effects, a random intercept is automatically added unless `0` appears among the random terms. Mixed models are constructed simply by adding random effects to fixed effects in the RHS of the formula. For example, to construct a mixed model with a fixed and by-subject random coefficient for a Gaussian IRF for predictor A along with a random intercept by subject, the following RHS would be used:

```
C(A, Normal()) + (C(A, Normal()) | subject)
```

IRF in random effects statements are treated as tied to any corresponding fixed effects unless explicitly distinguished by distinct IRF ID's (see section below on parameter tying).

The above formula uses a single parameterization for the Gaussian IRF and fits by-subject coefficients for it. However it is also possible to fit by-subject IRF parameterizations. This can be accomplished by adding `ran=T` to the IRF call, as shown below:

```
C(A, Normal()) + (C(A, Normal(ran=T)) | subject)
```

This formula will fit separate coefficients *and* IRF shapes for this predictor for each subject.

An important complication in fitting mixed models with CDR is that the relevant grouping factor is determined by the current *regression target*, not the properties of the independent variable observations in the series history. This means that random effects are only guaranteed to be meaningful when fit using grouping factors that are constant for the entire series (e.g. the ID of the human subject completing the experiment). Random effects fit for grouping factors that vary during the experiment should therefore be avoided unless they are intercept terms only, which are not affected by the temporal convolution.

4.2.7 Parameter Initialization

IRF parameters can be initialized for a given convolutional term by specifying their initial values in the IRF call, using the parameter name as the keyword (see supported IRF and their associated parameters above). For example, to initialize a Gamma IRF with $\alpha = 2$ and $\beta = 2$ for predictor A, use the following call:

```
C(A, Gamma(alpha=2, beta=2))
```

These values will serve as initializations in both CDRMLE and CDRBayes, and in CDRBayes they will additionally serve as the mean of the prior distribution for that parameter. If no initialization is specified, defaults will be used. These defaults are not guaranteed to be plausible for your particular application and may have a detrimental impact on training. Therefore, it is generally a good idea to think carefully in advance about what kinds of IRF shapes are *a priori* reasonable and choose initializations in that range.

Note that the initialization values are on the constrained space, so make sure to respect the constraints when choosing them. For example, α of the Gamma distribution is constrained to be > 0 , so an initial α of ≤ 0 will result in incorrect behavior. However, keep in mind that for CDRBayes, prior variances are necessarily on the unconstrained space and get squashed by the constraint function, so choosing initializations that are very close to constraint boundaries can indirectly tighten the prior. For example, choosing an initialization $\alpha = 0.001$ for the Gamma distribution will result in a much tighter prior around small values of α .

Initializations for irrelevant parameters in ill-specified formulas will be ignored and the defaults for the parameters will be used instead. For example, if the model receives the IRF specification `Normal(alpha=1, beta=1)`, it will initialize a Normal IRF at $\mu = 0, \sigma = 1$ (the defaults for this kernel), since α and β are not recognized parameter names for the Normal distribution. Therefore, make sure to match the parameter names above when specifying parameter defaults. The correctness of initializations can be checked in the Tensorboard logs.

4.2.8 Using Constant (Non-trainable) Parameters

By default, CDR trains all the variables that parameterize an IRF kernel (e.g. both μ and σ for a Gaussian IRF kernel). But in some cases it's useful to treat certain IRF parameters as constants and leave them untrained. To do this, specify a list of trainable parameters with the keyword argument `trainable`, using Python list syntax. For example, to specify a ShiftedGamma IRF in which the shift parameter δ is held constant at -1, use the following IRF specification:

```
ShiftedGamma(delta=-1, trainable=[alpha, beta])
```

The model will then only train the α and β parameters of the response. As with parameter initialization, unrecognized parameter names in the `trainable` argument will be ignored, and parameter name mismatches can result in more parameters being held constant than intended. For example, the IRF specification `Normal(trainable=[alpha, beta])`, will result in an (untrainable) Normal IRF with all parameters held fixed at their defaults. It is therefore important to make sure that parameter names match those given above. The correctness of the `trainable` specification can be checked in the Tensorboard logs, as well as by the number of trainable parameters reported to standard error at the start of CDR training. Constant parameters will show 0 trainable parameters.

4.2.9 Parameter Tying

A convolutional term in a CDR model is factored into two components, an IRF component with appropriate parameters and a coefficient governing the overall amplitude of the estimate. Unless otherwise specified, both of these terms are fit separately for every predictor in the model. However, parameter tying is possible by passing keyword arguments to the IRF calls in the model formula. Coefficients can be tied using the `coef_id` argument, and IRF parameters can be tied using the `irf_id` argument. For example, the following RHS fits separate IRF and coefficients for each of A and B:

```
C(A, Normal()) + C(B, Normal())
```

The following fits a single IRF (called “IRF_NAME”) but separate coefficients for A and B:

```
C(A, Normal(irf_id=IRF_NAME)) + C(B, Normal(irf_id=IRF_NAME))
```

The following fits separate IRF but a single coefficient (called “COEF_NAME”) for both A and B:

```
C(A, Normal(coef_id=COEF_NAME)) + C(B, Normal(coef_id=COEF_NAME))
```

And the following fits a single IRF (called “IRF_NAME”) and a single coefficient (called “COEF_NAME”), both of which are shared between A and B:

```
C(A, Normal(irf_id=IRF_NAME, coef_id=COEF_NAME)) + C(B, Normal(irf_id=IRF_NAME, coef_
↪id=COEF_NAME))
```

4.2.10 Transforming Variables

CDR provides limited support for automatic variable transformations based on model formulas. As in **R** formulas, a transformation is applied by wrapping the predictor name in the transformation function. For example, to fit a Gamma IRF to a log transform of predictor A, the following is added to the RHS:

```
C(log(A), Gamma())
```

Transformations may be applied to the predictors and/or the response.

The following are the currently supported transformations:

- `log()`: Applies a natural logarithm transformation to the variable
- `log1p()`: Adds 1 to the variable and applies a natural logarithm transformation (useful if predictor can include 0)
- `exp()`: Exponentiates the variable
- `z()`: Z-transforms the variable (subtracts its mean and divides by its standard deviation)
- `c()`: 0-centers the variable (subtracts its mean)
- `s()`: Scales the variable (divides by its standard deviation)

Other transformations must be applied via data preprocessing.

4.2.11 Continuous predictors

CDR’s discrete convolution is only exact for discrete impulses (e.g. spikes of stimulus). Impulse streams that constitute *samples* from a continuous source signal cannot be convolved exactly because the source is generally not analytically integrable. However, CDR supports discrete approximation of convolution with continuous inputs through linear interpolation of the impulse between samples, performed at a fixed frequency.

To flag a predictor as continuous, use the `cont` keyword argument in the IRF call of the model formula, as shown:

```
C(A, Gamma(cont=T))
```

Be warned that, due to the need for interpolation, continuous predictors tend to impose a heavy computational burden that can dramatically slow training and prediction. Speedups can be obtained at the expense of accuracy by choose a small value for the `n_interp` initialization parameter, decreasing the resolution of the interpolation.

4.2.12 Pseudo Non-Parametric IRFs

CDR also supports pseudo non-parametric IRFs in the form of Gaussian kernel functions (linear combination of Gaussians or LCG). Instead of a parametric IRF kernel, the model implements the IRF as a sum of Gaussian kernel functions whose location, spread, and height can be optimized by the model. The advantage of LCG IRFs is that they do not require precommitment to a particular functional form for the IRF. The disadvantage is that fitting them is slower because they involve more parameters and computation.

The kernels themselves have a number of free parameters which are specified by the name of the kernel in the IRF call of the model formula. The syntax for an LCG IRF kernel is as follows:

```
LCG(b ([0-9]+) ) ?
```

This is a string representation of a function call `LCG` with optional keyword argument `b`.

The keyword argument is defined as follows:

- `b` (bases): `int`, number of bases (control points). **Default:** 10.

4.2.13 IRF Composition

In some cases it may be desirable to decompose the response into multiple convolutions of an impulse. For example, it is possible that the BOLD response in fMRI consists underlyingly of 2 convolutional responses: a **neural response** that convolves the impulse into a timecourse of neural activation, which is then convolved with a **hemodynamic response** into a BOLD signal. In this case, it would be desirable to be able to model the BOLD response as a composition of neural and hemodynamic responses.

Exact parametric composition of IRF is not possible in the general case because many pairs of IRF do not have a tractable analytical convolution. Instead, the CDR package uses a discrete approximation to the continuous integral of composed IRF by (1) computing the value of each IRF for some number of interpolation points, (2) computing their convolution via FFT, and (3) rescaling by the temporal distance between interpolation points. The number of interpolation points is defined by the model's `n_interp` initialization parameter.

To compose IRF in a model, simply insert one IRF call into the first argument position of another IRF call. For example, the following first convolves impulse `A` with a normal IRF and then convolves this convolved response with an exponential IRF:

```
C(A, Exp(Normal()))
```

Because convolution has the associative property, the order of composition does not matter, and the above is equivalent to:

```
C(A, Normal(Exp()))
```

The advantage of IRF composition is that it affords the possibility of discovering the structure of latent responses that are not directly observable in the measured response, as in the example described above. The disadvantage is that it is much more computationally expensive due to the interpolation and FFT steps required.

Care must also be taken when using IRF composition to avoid constructing unidentifiable models. For example, the convolution of two Gaussians $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$ is known to be $N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$. As a result, the following composed IRF has infinitely many solutions, and the resulting model is unidentifiable:

```
C(A, Normal(Normal()))
```

CDR is not able to recognize and flag identifiability problems and it will happily find a solution to such a model, disguising the fact that there are infinitely many other optimal solutions. It is up to the user to think carefully about whether the model structure could introduce such problems. For example, in the BOLD example discussed above,

the neural response is predictor-specific while the hemodynamic response is predictor-independent given the neural response. The two responses can thus be separated via parameter tying of the hemodynamic response portion (see below), requiring all predictors to share a single hemodynamic response and forcing predictor-level variation into the neural response alone.

4.3 CDRNN Model Formulas

CDRNN models compute a single joint multivariate IRF with a highly distributed and interactive structure. As a result, it is not necessary (or even possible) to constrain the IRF to have a particular shape. Instead, all shape features are estimated from data. Therefore, CDRNN model formulas are very simple: they contain a list of predictors and (optionally) a list of random grouping factors. For example, the following defines a CDRNN model with predictors `A` and `B` and random grouping factor `subject`:

```
y ~ A + B + (1 | subject)
```

The `(1 | subject)` term may look like a random intercept from linear mixed models, but this is misleading. It simply allows CDRNN to take `subject` into account when computing any aspect of the response. It thus incorporates by-subject variation in the size, shape, and degree of interaction for all variables in the model. Because of the distributed nature of CDRNN representations, it is not possible to distinguish e.g. random intercepts, slopes, and IRF parameters, as CDR permits. A random term is simply present or not, and its influence can be accommodated as needed throughout the model. For details about how random effects are incorporated into the CDRNN architecture, see Shain (2021).

IRF TREES IN CDR

The network of IRFs for each CDR model are represented by the implementations in this package as *trees* from which properties can be read off for the purposes of model construction. This is an implementation detail that will not be pertinent to most users, especially those primarily training and using CDR models through the bundled executables. However, for users who want to program with CDR classes or directly interface with the associated `Formula` class, which is used to store and manipulate information about the model structure, this reference provides a conceptual overview of tree representations of CDR impulse response structures.

There are several reasons for representing IRFs as nodes on a tree. In particular, tree representations facilitate:

- **Factorization** of separable model components, especially *impulses*, *coefficients*, and *IRFs* (see discussion below)
- **IRF composition**: convolving multiple IRF with each other in a hierarchical fashion
- **Parameter tying**: sharing a single IRF or coefficient across multiple portions of the model

Note that these components cannot be cleanly factorized in CDRNN, so CDRNN “trees” are flat. The remainder of this page applies only to CDR.

5.1 Factorization

CDR models contain *impulses*, *coefficients*, and *IRFs*, each of whose structure can be separately manipulated. Impulses are timestamped data streams that constitute the predictors of the model. Impulses are computed from the input data prior to entering the CDR computation graph.

Coefficients govern the amplitude of the response to a predictor. Specifically, the IRF for a given predictor is multiplied by a coefficient governing the magnitude and sign of the overall response. Because multiplication distributes over convolution, coefficients are only needed at the point at which an IRF is convolved with an impulse. In hierarchical models involving convolution of multiple IRFs together, the coefficient can be seen as the product of all coefficients on all composed IRFs. It is therefore desirable to restrict coefficients only to those IRF that “make contact” with the impulse data.

IRFs govern the shape of the response to a predictor. The response is subsequently scaled using a coefficient. Arbitrary IRF can be convolved together in a hierarchical fashion using a fast Fourier transform-based discrete approximation to the continuous convolution.

Tree structures permit natural separation of these three components. Specifically, the CDR IRF tree consists of *nodes*, each of which represents the application of an IRF. IRF composition is represented in the tree as *dominance*: parent IRF are convolved with their children, which are they convolved with their children, etc. Coefficients are disallowed except at *terminal* IRF nodes, i.e. “leaves” of the tree that have no children. In this implementation, `Terminal` is distinguished as special type of IRF kernel. The leaves of the IRF tree must always and only be `Terminal`. Only `Terminal` IRF nodes are permitted to have coefficients. Attempts to specify coefficients at other portions of the tree are ignored. `Terminal`s always have a 1-to-1 link to an impulse contained in the data.

Thus, impulses, coefficients, and IRFs are kept separate by representing the IRF as nodes in a tree and the impulses and coefficients as decorations on a distinguished `Terminal` node type.

5.2 IRF composition

All IRF trees are rooted at a special `ROOT` node, and subsequent application and composition of IRF is represented by tree structures dominated by `ROOT`. A simple model with no IRF composition will have a tree depth of 3:

```
ROOT --> IRF --> Terminals.
```

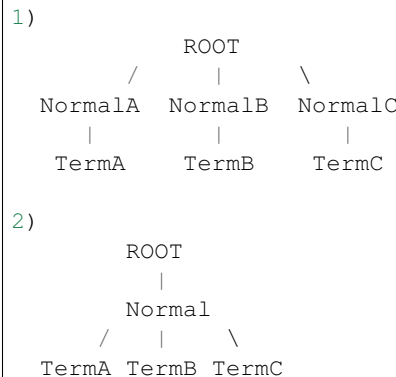
Models involving composition of IRF will have greater depth, as composition of IRF is represented as dominance relations between them. For example, imagine that the IRF for a given predictor is expected to be a Gamma convolved with another Gamma, which is then convolved with a Normal. The path through the tree describing this structure will be:

```
ROOT --> Gamma1 --> Gamma2 --> Terminal
```

At the terminal, the composed IRF is multiplied by its coefficient and then convolved with its associated impulse. Although IRF composition in CDR respects the hierarchical order in which the IRF are specified, in most cases order of composition does not matter because of associativity of convolution.

5.3 Parameter tying

IRF can be tied in CDR, a property which can be conveniently represented via branching structures in the tree. Specifically, if a single IRF is convolved with (1) multiple downstream IRF in a composition hierarchy or (2) multiple impulses, this fact can be represented by attaching multiple child nodes to a node representing the IRF. For example, consider the following two IRF trees involving impulses A, B, and C:



In Tree (1), each terminal has its own IRF, so the branching occurs below `ROOT`, while in Tree (2), a single IRF is shared by all three impulses, so the branch occurs below `IRF`. More complex models could be formed e.g. by replacing one of the terminals in these examples with other IRF treelets, and so on.

Tying of coefficients is also supported but this is not represented in the tree structure. It is enforced simply by requiring multiple terminals to share the same value for the coefficient ID property.

5.4 CDR tree summaries

In the CDR model summary, the IRF tree is printed in string format with indentation representing dominance. The ID of each IRF on the tree is printed on its own line, along with other metadata when relevant (such as trainable params and associated random grouping factors). For example, Tree (1) above would be represented as follows:

```
ROOT
  Normal.A; trainable params: mu, sigma
    Normal.A-Terminal.A
  Normal.B; trainable params: mu, sigma
    Normal.B-Terminal.B
  Normal.C; trainable params: mu, sigma
    Normal.C-Terminal.C
```

while Tree (2) would be represented as follows:

```
ROOT
  Normal; trainable params: mu, sigma
    Normal-Terminal.A
    Normal-Terminal.B
    Normal-Terminal.C
```

CDR PACKAGE API

Complete API for all public classes and methods in this package.

6.1 `cdr.backend` module

`cdr.backend.CDRNNStateTuple`
alias of `cdr.backend.AttentionalLSTMDecoderStateTuple`

6.2 `cdr.base` module

6.3 `cdr.config` module

`class cdr.config.Config(path)`
Bases: `object`

Parses an *.ini file and stores settings needed to define a set of CDR experiments.

Parameters `path` – Path to *.ini file

`build_cdr_settings(settings, add_defaults=True, global_settings=None, is_cdr=True, is_cdrnn=False)`

Given a settings object parsed from a config file, compute CDR parameter dictionary.

Parameters

- **settings** – settings from a `ConfigParser` object.
- **add_defaults** – `bool`; whether to add default settings not explicitly specified in the config.
- **global_settings** – `dict` or `None`; dictionary of global defaults for parameters missing from **settings**.
- **is_cdr** – `bool`; whether this is a CDR(NN) model.
- **is_cdrnn** – `bool`; whether this is a CDRNN model.

Returns `dict`; dictionary of settings key-value pairs.

`set_model(model_name=None)`

Change internal state to that of model named **model_name**. `Config` instances can store settings for multiple models. `set_model()` determines which model's settings are returned by `Config` getter methods.

Parameters `model_name` – str; name of target model

Returns None

class `cdr.config.PlotConfig` (*path=None*)

Bases: `object`

Parses an *.ini file and stores settings needed to define CDR plots

Parameters `path` – Path to *.ini file

build_plot_settings (*settings*)

Given a settings object parsed from a config file, compute plot parameters.

Parameters `settings` – settings from a ConfigParser object.

Returns dict; dictionary of settings key-value pairs.

6.4 cdr.data module

`cdr.data.add_responses` (*names, y*)

Add response variable(s) to a dataframe, applying any preprocessing required by the formula string.

Parameters

- **names** – str or list of str; name(s) of dependent variable(s)
- **y** – pandas DataFrame; response data.

Returns pandas DataFrame; response data with any missing ops applied.

`cdr.data.build_CDR_impulse_data` (*X, first_obs, last_obs, X_in_Y_names=None, X_in_Y=None, impulse_names=None, history_length=128, future_length=0, int_type='int32', float_type='float32'*)

Construct impulse data arrays in the required format for CDR fitting/evaluation for a single response array.

Parameters

- **X** – list of pandas tables; impulse (predictor) data.
- **first_obs** – list of index vectors (list, pandas series, or numpy vector) of first observations; the list contains vectors of row indices, one for each element of **X**, of the first impulse in the time series associated with the response. If None, inferred from **Y**.
- **last_obs** – list of index vectors (list, pandas series, or numpy vector) of last observations; the list contains vectors of row indices, one for each element of **X**, of the last impulse in the time series associated with the response. If None, inferred from **Y**.
- **X_in_Y_names** – list of str; names of predictors contained in **Y** rather than **X**. If None, no such predictors.
- **X_in_Y** – pandas DataFrame or None; table of predictors contained in **Y** rather than **X**. If None, no such predictors.
- **impulse_names** – list of str; names of columns in **X** to be used as impulses by the model. If None, all columns returned.
- **history_length** – int; maximum number of history (backward) observations.
- **future_length** – int; maximum number of future (forward) observations.
- **int_type** – str; name of int type.
- **float_type** – str; name of float type.

Returns triple of `numpy` arrays; let `N`, `T`, `I`, `R` respectively be the number of rows in `Y`, history length, number of impulse dimensions, and number of response dimensions. Outputs are (1) impulses with shape `(N, T, I)`, (2) impulse timestamps with shape `(N, T, I)`, and impulse mask with shape `(N, T, I)`.

```
cdr.data.build_CDR_response_data (responses, Y=None, first_obs=None, last_obs=None,
                                  Y_time=None, Y_gf=None, X_in_Y_names=None,
                                  X_in_Y=None, Y_category_map=None, re-
                                  sponse_to_df_ix=None, gf_names=None, gf_map=None)
```

Construct response data arrays in the required format for CDR fitting/evaluation for one or more response arrays.

Parameters

- **responses** – list of `str`; names of columns in `Y` to be used as responses (dependent variables) by the model.
- **Y** – list of `pandas` tables, or `None`; response data. If `None`, does not return a response array.
- **first_obs** – list of list of index vectors (`list`, `pandas` series, or `numpy` vector) of first observations, or `None`; the list contains one element for each response array. Inner lists contain vectors of row indices, one for each element of `X`, of the first impulse in the time series associated with each response. If `None`, inferred from `Y`.
- **last_obs** – list of list of index vectors (`list`, `pandas` series, or `numpy` vector) of last observations, or `None`; the list contains one element for each response array. Inner lists contain vectors of row indices, one for each element of `X`, of the last impulse in the time series associated with each response. If `None`, inferred from `Y`.
- **Y_time** – list of response timestamp vectors (`list`, `pandas` series, or `numpy` vector), or `None`; vector(s) of response timestamps, one for each response array. Needed to timestamp any response-aligned predictors (ignored if none in model).
- **Y_gf** – list of `pandas` `DataFrame`, or `None`; vector(s) of response timestamps, one for each response array. Data frames containing random grouping factor levels, if applicable.
- **X_in_Y_names** – list of `str`; names of predictors contained in `Y` rather than `X` (must be present in all elements of `Y`). If `None`, no such predictors.
- **X_in_Y** – list of `pandas` `DataFrame` or `None`; tables (one per response array) of predictors contained in `Y` rather than `X` (must be present in all elements of `Y`). If `None`, no such predictors.
- **Y_category_map** – `dict` or `None`; map from category labels to integers for each categorical response.
- **response_to_df_ix** – `dict` or `None`; map from response names to lists of indices of the response files that contain them.
- **gf_names** – list or `None`; list of names of random grouping factor variables. If `None` and `Y_gf` provided, will use all columns of `Y_gf`.
- **gf_map** – list of `dict` or `None`; list maps from random grouping factor levels to their indices, one map per grouping factor variable in `gf_names`.

Returns 7-tuple of `numpy` arrays; let `N`, `R`, `XF`, `YF`, `Z`, and `K` respectively be the number of rows (sum total number of rows in `Y`), number of response dimensions, number of distinct predictor files (`X`), number of distinct response files (`Y`), number of random grouping factor variables, and number of response_aligned predictors. Outputs are (1) responses with shape `(N, R)` or `None` if `Y` is `None`, (2) an `XF`-tuple of first observation vectors indexing start indices for each entry in `X`, (3) a `YF`-tuple of first observation vectors indexing end indices for each entry in `X`, (4) response timestamps with shape `(N,)`, (5) response masks (masking out any missing response variables

per row) with shape (N, R), (6) random grouping factor matrix with shape (N, Z), or None if no random grouping factors provided, and (7) response-aligned predictors with shape (N, K).

`cdr.data.c(df)`

Zero-center pandas series or data frame

Parameters `df` – pandas Series or DataFrame; input data

Returns pandas Series or DataFrame; centered data

`cdr.data.compute_filter(y, field, cond)`

Compute filter given a field and condition

Parameters

- `y` – pandas DataFrame; response data.
- `field` – str; name of column on whose values to filter.
- `cond` – str; string representation of condition to use for filtering.

Returns numpy vector; boolean mask to use for pandas subsetting operations.

`cdr.data.compute_filters(Y, filters=None)`

Compute filters given a filter map.

Parameters

- `Y` – pandas DataFrame; response data.
- `filters` – list; list of key-value pairs mapping column names to filtering criteria for their values.

Returns numpy vector; boolean mask to use for pandas subsetting operations.

`cdr.data.compute_partition(y, modulus, n)`

Given a `splitID` column, use modular arithmetic to partition data into `n` subparts.

Parameters

- `y` – pandas DataFrame; response data.
- `modulus` – int; modulus to use for splitting, must be at least as large as `n`.
- `n` – int; number of subparts in the partition.

Returns list of numpy vectors; one boolean vector per subpart of the partition, selecting only those elements of `y` that belong.

`cdr.data.compute_splitID(y, split_fields)`

Map tuples in columns designated by `split_fields` into integer ID to use for data partitioning.

Parameters

- `y` – pandas DataFrame; response data.
- `split_fields` – list of str; column names to use for computing split ID.

Returns numpy vector; integer vector of split ID's.

`cdr.data.compute_time_mask(X_time, first_obs, last_obs, history_length=128, future_length=0, int_type='int32', float_type='float32')`

Compute mask for expanded impulse data zeroing out non-existent impulses.

Parameters

- `X_time` – pandas Series; timestamps associated with each impulse in `X`.

- **first_obs** – pandas Series; vector of row indices in **X** of the first impulse in the time series associated with each response.
- **last_obs** – pandas Series; vector of row indices in **X** of the last preceding impulse in the time series associated with each response.
- **history_length** – int; maximum number of history (backward) observations.
- **future_length** – int; maximum number of future (forward) observations.
- **int_type** – str; name of int type.
- **float_type** – str; name of float type.

Returns numpy array; boolean impulse mask.

`cdr.data.corr_cdr(X_2d, impulse_names, impulse_names_2d, time, time_mask)`

Compute correlation matrix, including correlations across time where necessitated by 2D predictors.

Parameters

- **x_2d** – numpy array; the impulse data. Must be of shape `(batch_len, history_length+future_length, n_impulses)`, can be computed from sources by `build_CDR_impulse_data()`.
- **impulse_names** – list of str; names of columns in **X_2d** to be used as impulses by the model.
- **impulse_names_2d** – list of str; names of columns in **X_2d** that designate to 2D predictors.
- **time** – 3D numpy array; array of timestamps for each event in **X_2d**.
- **time_mask** – 3D numpy array; array of masks over padding events in **X_2d**.

Returns pandas DataFrame; the correlation matrix.

`cdr.data.expand_impulse_sequence(X, X_time, first_obs, last_obs, window_length, int_type='int32', float_type='float32', fill=0.0)`

Expand out impulse stream in **X** for each response in the target data.

Parameters

- **X** – pandas DataFrame; impulse (predictor) data.
- **X_time** – pandas Series; timestamps associated with each impulse in **X**.
- **first_obs** – pandas Series; vector of row indices in **X** of the first impulse in the time series associated with each response.
- **last_obs** – pandas Series; vector of row indices in **X** of the last preceding impulse in the time series associated with each response.
- **window_length** – int; number of steps in time dimension of output
- **int_type** – str; name of int type.
- **float_type** – str; name of float type.
- **fill** – float; fill value for padding cells.

Returns 3-tuple of numpy arrays; the expanded impulse array, the expanded timestamp array, and a boolean mask zeroing out locations of non-existent impulses.

`cdr.data.filter_invalid_responses(Y, dv, crossval_factor=None, crossval_fold=None)`

Filter out rows with non-finite responses.

Parameters

- **Y** – pandas table or list of pandas tables; response data.
- **dv** – str or list of str; name(s) of column(s) containing the dependent variable(s)
- **crossval_factor** – str or None; name of column containing the selection variable for cross validation. If None, no cross validation filtering.
- **crossval_fold** – list or None; list of valid values for cross-validation selection. Used only if **crossval_factor** is not None.

Returns 2-tuple of pandas DataFrame and pandas Series; valid data and indicator vector used to filter out invalid data.

`cdr.data.get_first_last_obs_lists(y)`

Convenience utility to extract out all `first_obs` and `last_obs` columns in **Y** sorted by file index

Parameters **y** – pandas DataFrame; response data.

Returns pair of list of str; `first_obs` column names and `last_obs` column names

`cdr.data.get_rangf_array(Y, rangf_names, rangf_map)`

Collect random grouping factor indicators as numpy integer arrays that can be read by Tensorflow. Returns vertical concatenation of GF arrays from each element of **Y**.

Parameters

- **Y** – pandas table or list of pandas tables; response data.
- **rangf_names** – list of str; names of columns containing random grouping factor levels (order is preserved, changing the order will change the resulting array).
- **rangf_map** – list of dict; map for each random grouping factor from levels to unique indices.

Returns

`cdr.data.get_time_windows(X, Y, series_ids, forward=False, window_length=128, verbose=True)`

Compute row indices in **X** of initial and final impulses for each element of **y**. Assumes time series are already sorted by `series_ids`.

Parameters

- **X** – pandas DataFrame; impulse (predictor) data.
- **Y** – pandas DataFrame; response data.
- **series_ids** – list of str; column names whose jointly unique values define unique time series.
- **forward** – bool; whether to compute forward windows (future inputs) or backward windows (past inputs, used if **forward** is False).
- **window_length** – int; maximum size of time window to consider. If `np.inf`, no bound on window size.
- **verbose** – bool; whether to report progress to stderr

Returns 2-tuple of numpy vectors; first and last impulse observations (respectively) for each response in **y**

`cdr.data.preprocess_data(X, Y, formula_list, series_ids, filters=None, history_length=128, future_length=0, all_interactions=False, verbose=True, debug=False)`

Preprocess CDR data.

Parameters

- **X** – list of pandas tables; impulse (predictor) data.

- **Y** – list of pandas tables; response data.
- **formula_list** – list of Formula; CDR formula for which to preprocess data.
- **series_ids** – list of str; column names whose jointly unique values define unique time series.
- **filters** – list; list of key-value pairs mapping column names to filtering criteria for their values.
- **history_length** – int; maximum number of history (backward) observations.
- **future_length** – int; maximum number of future (forward) observations.
- **all_interactions** – bool; add powerset of all conformable interactions.
- **verbose** – bool; whether to report progress to stderr
- **debug** – bool; print debugging information

Returns 7-tuple; predictor data, response data, filtering mask, response-aligned predictor names, response-aligned predictors, 2D predictor names, and 2D predictors

`cdr.data.s(df)`

Rescale pandas series or data frame by its standard deviation

Parameters `df` – pandas Series or DataFrame; input data

Returns pandas Series or DataFrame; rescaled data

`cdr.data.split_cdr_outputs(outputs, lengths)`

Takes a dictionary of arbitrary depth containing CDR outputs with their labels as keys and splits each output into a list of outputs with lengths corresponding to **lengths**. Useful for aligning CDR outputs to response files, since multiple response files can be provided, which are underlyingly concatenated by CDR. Recursively modifies the dict in place.

Parameters

- **outputs** – dict of arbitrary depth with numpy arrays at the leaves; the source CDR outputs
- **lengths** – array-like vector of lengths to split the outputs into

Returns dict; same key-val structure as **outputs** but with each leaf split into a list of `len(lengths)` vectors, one for each length value.

`cdr.data.z(df)`

Z-transform pandas series or data frame

Parameters `df` – pandas Series or DataFrame; input data

Returns pandas Series or DataFrame; z-transformed data

6.5 cdr.cdrbase module

6.6 cdr.cdrbayes module

6.7 cdr.cdrmle module

6.8 cdr.cdrnnbase module

6.9 cdr.cdrnnbayes module

6.10 cdr.cdrnnmle module

6.11 cdr.formula module

class `cdr.formula.Formula` (*bform_str*, *standardize=True*)

Bases: `object`

A class for parsing R-style mixed-effects CDR model formula strings and applying them to CDR data matrices.

Parameters `bform_str` – `str`; an R-style mixed-effects CDR model formula string

ablate_impulses (*impulse_ids*)

Remove impulses in `impulse_ids` from fixed effects (retaining in any random effects).

Parameters `impulse_ids` – list of `str`; impulse ID's

Returns `None`

apply_formula (*X*, *Y*, *X_in_Y_names=None*, *all_interactions=False*, *series_ids=None*)

Extract all data and compute all transforms required by the model formula.

Parameters

- `X` – list of `pandas` tables; impulse data.
- `Y` – list of `pandas` tables; response data.
- `X_in_Y_names` – list or `None`; List of column names for response-aligned predictors (predictors measured for every response rather than for every input) if applicable, `None` otherwise.
- `all_interactions` – `bool`; add powerset of all conformable interactions.
- `series_ids` – list of `str` or `None`; list of ids to use as grouping factors for lagged effects. If `None`, lagging will not be attempted.

Returns triple; transformed `X`, transformed `y`, response-aligned predictor names

apply_op (*op*, *arr*)

Apply op `op` to array `arr`.

Parameters

- `op` – `str`; name of op.
- `arr` – `numpy` or `pandas` array; source data.

Returns numpy array; transformed data.

apply_op_2d (*op, arr, time_mask*)

Apply op to 2D predictor (predictor whose value depends on properties of the response).

Parameters

- **op** – str; name of op.
- **arr** – numpy or array; source data.
- **time_mask** – numpy array; mask for padding cells

Returns numpy array; transformed data

apply_ops (*impulse, X*)

Apply all ops defined for an impulse

Parameters

- **impulse** – Impulse object; the impulse.
- **X** – list of pandas tables; table containing the impulse data.

Returns pandas table; table augmented with transformed impulse.

apply_ops_2d (*impulse, X_2d_predictor_names, X_2d_predictors, time_mask*)

Apply all ops defined for a 2D predictor (predictor whose value depends on properties of the response).

Parameters

- **impulse** – Impulse object; the impulse.
- **X_2d_predictor_names** – list of str; names of 2D predictors.
- **X_2d_predictors** – numpy array; source data.
- **time_mask** – numpy array; mask for padding cells

Returns 2-tuple; list of new predictor name, numpy array of predictor values

static bases (*family*)

Get the number of bases of a spline kernel.

Parameters **family** – str; name of IRF family

Returns int or None; number of bases of spline kernel, or None if **family** is not a spline.

build (*bform_str, standardize=True*)

Construct internal data from formula string

Parameters **bform_str** – str; source string.

Returns None

categorical_transform (*X*)

Get transformed formula with categorical predictors in **X** expanded.

Parameters **X** – list of pandas tables; input data.

Returns Formula; transformed Formula object

compute_2d_predictor (*predictor_name, X, first_obs, last_obs, history_length=128, future_length=None, minibatch_size=50000*)

Compute 2D predictor (predictor whose value depends on properties of the most recent impulse).

Parameters

- **predictor_name** – str; name of predictor

- **X** – pandas table; input data
- **first_obs** – pandas Series or 1D numpy array; row indices in X of the start of the series associated with each regression target.
- **last_obs** – pandas Series or 1D numpy array; row indices in X of the most recent observation in the series associated with each regression target.
- **minibatch_size** – int; minibatch size for computing predictor, can help with memory footprint

Returns 2-tuple; new predictor name, numpy array of predictor values

insert_impulses (*impulses*, *irf_str*, *rangf=None*)

Insert impulses in **impulse_ids** into fixed effects and all random terms.

Parameters **impulse_ids** – list of str; impulse ID's

Returns None

static irf_params (*family*)

Return list of parameter names for a given IRF family.

Parameters **family** – str; name of IRF family

Returns list of str; parameter names

static is_LCG (*family*)

Check whether a kernel is LCG.

Parameters **family** – str; name of IRF family

Returns bool; whether the kernel is LCG (linear combination of Gaussians)

pc_transform (*n_pc*, *pointers=None*)

Get transformed formula with impulses replaced by principal components.

Parameters

- **n_pc** – int; number of principal components in transform.
- **pointers** – dict; map from source nodes to transformed nodes.

Returns list of IRFNode; tree forest representing current state of the transform.

process_ast (*t*, *terms=None*, *has_intercept=None*, *ops=None*, *rangf=None*, *impulses_by_name=None*, *interactions_by_name=None*, *under_irf=False*, *under_interaction=False*)

Recursively process a node of the Python abstract syntax tree (AST) representation of the formula string and insert data into internal representation of model formula.

Parameters

- **t** – AST node.
- **terms** – list or None; CDR terms computed so far, or None if no CDR terms computed.
- **has_intercept** – dict; map from random grouping factors to boolean values representing whether that grouping factor has a random intercept. None is used as a key to refer to the population-level intercept.
- **ops** – list; names of ops computed so far, or None if no ops computed.
- **rangf** – str or None; name of rangf for random term currently being processed, or None if currently processing fixed effects portion of model.

Returns None

process_irf (*t*, *input*, *ops=None*, *rangf=None*)

Process data from AST node representing part of an IRF definition and insert data into internal representation of the model.

Parameters

- **t** – AST node.
- **input** – IRFNode object; child IRF of current node
- **ops** – list of str, or None; ops applied to IRF. If None, no ops applied
- **rangf** – str or None; name of rangf for random term currently being processed, or None if currently processing fixed effects portion of model.

Returns IRFNode object; the IRF node

remove_impulses (*impulse_ids*)

Remove impulses in **impulse_ids** from the model (both fixed and random effects).

Parameters **impulse_ids** – list of str; impulse ID's

Returns None

response_names ()

Get list of names modeled response variables.

Returns list of str; names modeled response variables.

responses ()

Get list of modeled response variables.

Returns list of Impulse; modeled response variables.

to_lmer_formula_string (*z=False*, *correlated=True*)

Generate an lme4-style LMER model string representing the structure of the current CDR model. Useful for 2-step analysis in which data are transformed using CDR, then fitted using LME.

Parameters

- **z** – bool; z-transform convolved predictors.
- **correlated** – bool; whether to use correlated random intercepts and slopes.

Returns str; the LMER formula string.

to_string (*t=None*)

Stringify the formula, using **t** as the RHS.

Parameters **t** – IRFNode or None; IRF node to use as RHS. If None, uses root IRF associated with Formula instance.

Returns str; stringified formula.

unablate_impulses (*impulse_ids*)

Insert impulses in **impulse_ids** into fixed effects (leaving random effects structure unchanged).

Parameters **impulse_ids** – list of str; impulse ID's

Returns None

class cdr.formula.IRFNode (*family=None*, *impulse=None*, *p=None*, *irfID=None*, *coefID=None*, *ops=None*, *cont=False*, *fixed=True*, *rangf=None*, *param_init=None*, *trainable=None*)

Bases: object

Data structure representing a node in a CDR IRF tree. For more information on how the CDR IRF structure is encoded as a tree, see the reference on CDR IRF trees.

Parameters

- **family** – `str`; name of IRF kernel family.
- **impulse** – `Impulse` object or `None`; the impulse if terminal, else `None`.
- **p** – `IRFNode` object or `None`; the parent IRF node, or `None` if no parent (parent nodes can be connected after initialization).
- **irfID** – `str` or `None`; string ID of node if applicable. If `None`, automatically-generated ID will describe node’s family and structural position.
- **coefID** – `str` or `None`; string ID of coefficient if applicable. If `None`, automatically-generated ID will describe node’s family and structural position. Only applicable to terminal nodes, so this property will not be used if the node is non-terminal.
- **ops** – list of `str`, or `None`; ops to apply to IRF node. If `None`, no ops.
- **cont** – `bool`; Node connects directly to a continuous predictor. Only applicable to terminal nodes, so this property will not be used if the node is non-terminal.
- **fixed** – `bool`; Whether node exists in the model’s fixed effects structure.
- **rangf** – list of `str`, `str`, or `None`; names of any random grouping factors associated with the node.
- **param_init** – `dict`; map from parameter names to initial values, which will also be used as prior means.
- **trainable** – list of `str`, or `None`; trainable parameters at this node. If `None`, all parameters are trainable.

ablate_impulses (*impulse_ids*)

Remove impulses in **impulse_ids** from fixed effects (retaining in any random effects).

Parameters **impulse_ids** – list of `str`; impulse ID’s

Returns `None`

add_child (*t*)

Add child to this node in the IRF tree

Parameters **t** – `IRFNode`; child node.

Returns `IRFNode`; child node with updated parent.

add_interactions (*response_interactions*)

Add a `ResponseInteraction` object (or list of them) to this node.

Parameters **response_interaction** – `ResponseInteraction` or list of `ResponseInteraction`; response interaction(s) to add

Returns `None`

add_rangf (*rangf*)

Add random grouping factor name to this node.

Parameters **rangf** – `str`; random grouping factor name

Returns `None`

atomic_irf_by_family ()

Get map from IRF kernel family names to list of IDs of `IRFNode` instances belonging to that family.

Returns dict from str to list of str; IRF IDs by family.

atomic_irf_param_init_by_family()

Get map from IRF kernel family names to maps from IRF IDs to maps from IRF parameter names to their initialization values.

Returns dict; parameter initialization maps by family.

atomic_irf_param_trainable_by_family()

Get map from IRF kernel family names to maps from IRF IDs to lists of trainable parameters.

Returns dict; trainable parameter maps by family.

bases()

Get the number of bases of node.

Returns int or None; number of bases of node, or None if node is not a spline.

categorical_transform(X, expansion_map=None)

Generate transformed copy of node with categorical predictors in **X** expanded. Recursive. Returns a tree forest representing the current state of the transform. When run from ROOT, should always return a length-1 list representing a single-tree forest, in which case the transformed tree is accessible as the 0th element.

Parameters

- **x** – list of pandas tables; input data.
- **expansion_map** – dict; Internal variable. Do not use.

Returns list of IRFNode; tree forest representing current state of the transform.

coef2impulse()

Get map from coefficient IDs dominated by node to lists of corresponding impulses.

Returns dict; map from coefficient IDs to lists of corresponding impulses.

coef2terminal()

Get map from coefficient IDs dominated by node to lists of corresponding terminal IRF nodes.

Returns dict; map from coefficient IDs to lists of corresponding terminal IRF nodes.

coef_by_rangf()

Get map from random grouping factor names to associated coefficient IDs dominated by node.

Returns dict; map from random grouping factor names to associated coefficient IDs.

coef_id()

Get coefficient ID for this node.

Returns str or None; coefficient ID, or None if non-terminal.

coef_names()

Get list of names of coefficients dominated by node.

Returns list of str; names of coefficients dominated by node.

depth()

Get depth of node in tree.

Returns int; depth

fixed_coef_names()

Get list of names of fixed coefficients dominated by node.

Returns list of str; names of fixed coefficients dominated by node.

fixed_interaction_names ()

Get list of names of fixed interactions dominated by node.

Returns list of str; names of fixed interactions dominated by node.

formula_terms ()

Return data structure representing formula terms dominated by node, grouped by random grouping factor. Key None represents the fixed portion of the model (no random grouping factor).

Returns dict; map from random grouping factors data structure representing formula terms. Data structure contains 2 fields, 'impulses' containing impulses and 'irf' containing IRF Nodes.

has_coefficient (rangf)

Report whether **rangf** has any coefficients in this subtree

Parameters **rangf** – Random grouping factor

Returns bool: Whether **rangf** has any coefficients in this subtree

has_composed_irf ()

Check whether node dominates any IRF compositions.

Returns bool, whether node dominates any IRF compositions.

has_irf (rangf)

Report whether **rangf** has any IRFs in this subtree

Parameters **rangf** – Random grouping factor

Returns bool: Whether **rangf** has any IRFs in this subtree

impulse2coef ()

Get map from impulses dominated by node to lists of corresponding coefficient IDs.

Returns dict; map from impulses to lists of corresponding coefficient IDs.

impulse2terminal ()

Get map from impulses dominated by node to lists of corresponding terminal IRF nodes.

Returns dict; map from impulses to lists of corresponding terminal IRF nodes.

impulse_names (include_interactions=False)

Get list of names of impulses dominated by node.

Parameters **include_interactions** – bool; whether to return impulses defined by interaction terms.

Returns list of str; names of impulses dominated by node.

impulses (include_interactions=False)

Get list of impulses dominated by node.

Parameters **include_interactions** – bool; whether to return impulses defined by interaction terms.

Returns list of Impulse; impulses dominated by node.

impulses_by_name ()

Get dictionary mapping names of impulses dominated by node to their corresponding impulses.

Returns dict; map from impulse names to impulses

impulses_from_response_interaction ()

Get list of any impulses from response interactions associated with this node.

Returns list of Impulse; impulses dominated by node.

interaction_by_rangf()

Get map from random grouping factor names to associated interaction IDs dominated by node.

Returns dict; map from random grouping factor names to associated interaction IDs.

interaction_names()

Get list of names of interactions dominated by node.

Returns list of str; names of interactions dominated by node.

interactions()

Return list of all response interactions used in this subtree, sorted by name.

Returns list of ResponseInteraction

interactions2inputs()

Get map from IDs of ResponseInteractions dominated by node to lists of IDs of their inputs.

Returns dict; map from IDs of ResponseInteractions nodes to lists of their inputs.

irf_by_rangf()

Get map from random grouping factor names to IDs of associated IRF nodes dominated by node.

Returns dict; map from random grouping factor names to IDs of associated IRF nodes.

irf_id()

Get IRF ID for this node.

Returns str or None; IRF ID, or None if terminal.

irf_to_formula(rangf=None)

Generates a representation of this node's impulse response kernel in formula string syntax

Parameters rangf – random grouping factor for which to generate the stringification (fixed effects if rangf==None).

Returns str; formula string representation of node

is_LCG()

Check the non-parametric type of a node's kernel, or return None if parametric.

Parameters family – str; name of IRF family

Returns str or None; name of kernel type if non-parametric, else None.

local_name()

Get descriptive name for this node, ignoring its position in the IRF tree.

Returns str; name.

name()

Get descriptive name for this node.

Returns str; name.

node_table()

Get map from names to nodes of all nodes dominated by node (including self).

Returns dict; map from names to nodes of all nodes dominated by node.

nonparametric_coef_names()

Get list of names of nonparametric coefficients dominated by node. :return: list of str; names of spline coefficients dominated by node.

pc_transform (*n_pc*, *pointers=None*)

Generate principal-components-transformed copy of node. Recursive. Returns a tree forest representing the current state of the transform. When run from ROOT, should always return a length-1 list representing a single-tree forest, in which case the transformed tree is accessible as the 0th element.

Parameters

- **n_pc** – int; number of principal components in transform.
- **pointers** – dict; map from source nodes to transformed nodes.

Returns list of IRFNode; tree forest representing current state of the transform.

static pointers2namemmaps (*p*)

Get a map from source to transformed IRF node names.

Parameters **p** – dict; map from source to transformed IRF nodes.

Returns dict; map from source to transformed IRF node names.

remove_impulses (*impulse_ids*)

Remove impulses in **impulse_ids** from the model (both fixed and random effects).

Parameters **impulse_ids** – list of str; impulse ID's

Returns None

supports_non_causal ()

Check whether model contains only IRF kernels that lack the causality constraint $t \geq 0$.

Returns bool; whether model contains only IRF kernels that lack the causality constraint $t \geq 0$.

terminal ()

Check whether node is terminal.

Returns bool; whether node is terminal.

terminal2coef ()

Get map from IDs of terminal IRF nodes dominated by node to lists of corresponding coefficient IDs.

Returns dict; map from IDs of terminal IRF nodes to lists of corresponding coefficient IDs.

terminal2impulse ()

Get map from terminal IRF nodes dominated by node to lists of corresponding impulses.

Returns dict; map from terminal IRF nodes to lists of corresponding impulses.

terminal_names ()

Get list of names of terminal IRF nodes dominated by node.

Returns list of str; names of terminal IRF nodes dominated by node.

terminals ()

Get list of terminal IRF nodes dominated by node.

Returns list of IRFNode; terminal IRF nodes dominated by node.

terminals_by_name ()

Get dictionary mapping names of terminal IRF nodes dominated by node to their corresponding nodes.

Returns dict; map from node names to nodes

unablate_impulses (*impulse_ids*)

Insert impulses in **impulse_ids** into fixed effects (leaving random effects structure unchanged).

Parameters **impulse_ids** – list of str; impulse ID's

Returns None

unary_nonparametric_coef_names ()

Get list of names of non-parametric coefficients with no siblings dominated by node. Because unary splines are non-parametric, their coefficients are fixed at 1. Trainable coefficients are therefore perfectly confounded with the spline parameters. Splines dominating multiple coefficients are excepted, since the same kernel shape must be scaled in different ways.

Returns list of str; names of unary spline coefficients dominated by node.

class cdr.formula.**Impulse** (*name*, *ops=None*)

Bases: object

Data structure representing an impulse in a CDR model.

Parameters

- **name** – str; name of impulse
- **ops** – list of str, or None; ops to apply to impulse. If None, no ops.

categorical (*X*)

Checks whether impulse is categorical in a dataset

Parameters **X** – list pandas tables; data to to check.

Returns bool; True if impulse is categorical in **X**, False otherwise.

expand_categorical (*X*)

Expand any categorical predictors in **X** into 1-hot columns.

Parameters **X** – list of pandas tables; input data

Returns 2-tuple of pandas table, list of Impulse; expanded data, list of expanded Impulse objects

name ()

Get name of term.

Returns str; name.

class cdr.formula.**ImpulseInteraction** (*impulses*, *ops=None*)

Bases: object

Data structure representing an interaction of impulse-aligned variables (impulses) in a CDR model.

Parameters

- **impulses** – list of Impulse; impulses to interact.
- **ops** – list of str, or None; ops to apply to interaction. If None, no ops.

expand_categorical (*X*)

Expand any categorical predictors in **X** into 1-hot columns.

Parameters **X** – list of pandas tables; input data.

Returns 3-tuple of pandas table, list of ImpulseInteraction, list of list of Impulse; expanded data, list of expanded ImpulseInteraction objects, list of lists of expanded Impulse objects, one list for each interaction.

impulses ()

Get list of impulses dominated by interaction.

Returns list of Impulse; impulses dominated by interaction.

name ()

Get name of interaction impulse.

Returns `str`; name.

class `cdr.formula.ResponseInteraction` (*responses, rangf=None*)

Bases: `object`

Data structure representing an interaction of response-aligned variables (containing at least one IRF-involved impulse) in a CDR model.

Parameters

- **responses** – list of terminal `IRFNode`, `Impulse`, and/or `ImpulseInteraction` objects; responses to interact.
- **rangf** – `str` or list of `str`; random grouping factors for which to build random effects for this interaction.

add_rangf (*rangf*)

Add random grouping factor name to this interaction.

Parameters **rangf** – `str`; random grouping factor name

Returns `None`

contains_member (*x*)

Check if object is a member of the set of responses belonging to this interaction

Parameters **x** – `IRFNode`, `Impulse`, and/or `ImpulseInteraction` object; object to check.

Returns `bool`; whether `x` is a member of the set of responses

irf_responses ()

Get list of IRFs dominated by interaction.

Returns list of `IRFNode` objects; terminal IRFs dominated by interaction.

name ()

Get name of interaction impulse.

Returns `str`; name.

non_irf_responses ()

Get list of non-IRF response-aligned variables dominated by interaction.

Returns list of `Impulse` and/or `ImpulseInteraction` objects; non-IRF variables dominated by interaction.

replace (*old, new*)

Replace an old input with a new one

Parameters

- **old** – `IRFNode`, `Impulse`, and/or `ImpulseInteraction` object; response to remove.
- **new** – `IRFNode`, `Impulse`, and/or `ImpulseInteraction` object; response to add.

Returns `None`

responses ()

Get list of variables dominated by interaction.

Returns list of IRFNode, Impulse, and/or ImpulseInteraction objects; impulses dominated by interaction.

`cdr.formula.pythonize_string(s)`

Convert string to valid python variable name

Parameters `s` – str; source string

Returns str; pythonized string

`cdr.formula.standardize_formula_string(s)`

Standardize a formula string, removing notational variation. IRF specifications `C(...)` are sorted alphabetically by the IRF call name e.g. `Gamma()`. The order of impulses within an IRF specification is preserved.

Parameters `s` – str; the formula string to be standardized

Returns str; standardization of `s`

6.12 cdr.io module

`cdr.io.read_tabular_data(X_paths, Y_paths, series_ids, categorical_columns=None, sep=' ', verbose=True)`

Read impulse and response data into pandas dataframes and perform basic pre-processing.

Parameters

- **X_paths** – str or list of str; path(s) to impulse (predictor) data (multiple tables are concatenated). Each path may also be a ;-delimited list of paths to files containing predictors with different timestamps, where the predictors in each file all share the same set of timestamps.
- **Y_paths** – str or list of str; path(s) to response data (multiple tables are concatenated). Each path may also be a ;-delimited list of paths to files containing different response variables with different timestamps, where the response variables in each file all share the same set of timestamps.
- **series_ids** – list of str; column names whose jointly unique values define unique time series.
- **categorical_columns** – list of str; column names that should be treated as categorical.
- **sep** – str; string representation of field delimiter in input data.
- **verbose** – bool; whether to log progress to stderr.

Returns 2-tuple of list(pandas DataFrame); (impulse data, response data). X and Y each have one element for each dataset in X_paths/Y_paths, each containing the column-wise concatenation of all column files in the path.

6.13 cdr.kwargs module

```
class cdr.kwargs.Kwarg(key, default_value, dtypes, descr, aliases=None, default_value_cdrnn='same', suppress=False)
```

Bases: `object`

Data structure for storing keyword arguments and their docstrings.

Parameters

- **key** – `str`; Key
- **default_value** – Any; Default value
- **dtypes** – `list` or `class`; List of classes or single class. Members can also be specific required values, either `None` or values of type `str`.
- **descr** – `str`; Description of kwarg
- **default_value_cdrnn** – Any; Default value for CDRNN if distinct from CDR. If 'same', CDRNN uses **default_value**.
- **suppress** – `bool`; Whether to print documentation for this kwarg. Useful for hiding deprecated or little-used kwargs in order to simplify autodoc output.

```
dtypes_str()
```

String representation of dtypes permitted for kwarg.

Returns `str`; dtypes string.

```
get_type_name(x)
```

String representation of name of a dtype

Parameters **x** – dtype; the dtype to name.

Returns `str`; name of dtype.

```
in_settings(settings)
```

Check whether kwarg is specified in a settings object parsed from a config file.

Parameters **settings** – settings from a `ConfigParser` object.

Returns `bool`; whether kwarg is found in **settings**.

```
kwarg_from_config(settings, is_cdrnn=False)
```

Given a settings object parsed from a config file, return value of kwarg cast to appropriate dtype. If missing from settings, return default.

Parameters

- **settings** – settings from a `ConfigParser` object.
- **is_cdrnn** – `bool`; whether this is for a CDRNN model.

Returns value of kwarg

```
static type_comparator(a, b)
```

Types precede strings, which precede `None`

Parameters

- **a** – First element
- **b** – Second element

Returns `-1`, `0`, or `1`, depending on outcome of comparison

`cdr.kwargs.cdr_kwarg_docstring()`
Generate docstring snippet summarizing all CDR kwargs, dtypes, and defaults.

Returns `str`; docstring snippet

`cdr.kwargs.docstring_from_kwarg(kwarg)`
Generate docstring from CDR keyword argument object.

Parameters `kwarg` – Keyword argument object.

Returns `str`; docstring.

`cdr.kwargs.plot_kwarg_docstring()`
Generate docstring snippet summarizing all plotting kwargs, dtypes, and defaults.

Returns `str`; docstring snippet

6.14 cdr.opt module

6.15 cdr.plot module

class `cdr.plot.MidpointNormalize` (*vcenter=0.0, vmin=None, vmax=None, clip=False*)

Bases: `matplotlib.colors.Normalize`

`cdr.plot.plot_heatmap` (*m, row_names, col_names, dir='.', filename='eigenvectors.png', plot_x_inches=7, plot_y_inches=5, cmap='Blues'*)

Plot a heatmap. Used in CDR for visualizing eigenvector matrices in principal components models.

Parameters

- **m** – 2D numpy array; source data for plot.
- **row_names** – list of `str`; row names.
- **col_names** – list of `str`; column names.
- **dir** – `str`; output directory.
- **filename** – `str`; filename.
- **plot_x_inches** – float; width of plot in inches.
- **plot_y_inches** – float; height of plot in inches.
- **cmap** – `str`; name of `matplotlib` cmap object (determines colors of plotted IRF).

Returns `None`

`cdr.plot.plot_irf` (*plot_x, plot_y, irf_names, lq=None, uq=None, density=None, sort_names=True, prop_cycle_length=None, prop_cycle_map=None, dir='.', filename='irf_plot.png', irf_name_map=None, plot_x_inches=6, plot_y_inches=4, ylim=None, cmap='gist_rainbow', legend=True, xlab=None, ylab=None, use_line_markers=False, transparent_background=False, dpi=300, dump_source=False*)

Plot impulse response functions.

Parameters

- **plot_x** – numpy array with shape (T,1); time points for which to plot the response. For example, if the plots contain 1000 points from 0s to 10s, **plot_x** could be generated as `np.linspace(0, 10, 1000)`.

- **plot_y** – numpy array with shape (T, N); response of each IRF at each time point.
- **irf_names** – list of str; CDR ID's of IRFs in the same order as they appear in axis 1 of **plot_y**.
- **lq** – numpy array with shape (T, N), or None; lower bound of credible interval for each time point. If None, no credible interval will be plotted.
- **uq** – numpy array with shape (T, N), or None; upper bound of credible interval for each time point. If None, no credible interval will be plotted.
- **sort_names** – bool; alphabetically sort IRF names.
- **prop_cycle_length** – int or None; Length of plotting properties cycle (defines step size in the color map). If None, inferred from **irf_names**.
- **prop_cycle_map** – list of int, or None; Integer indices to use in the properties cycle for each entry in **irf_names**. If None, indices are automatically assigned.
- **dir** – str; output directory.
- **filename** – str; filename.
- **irf_name_map** – dict of str to str; map from CDR IRF ID's to more readable names to appear in legend. Any plotted IRF whose ID is not found in **irf_name_map** will be represented with the CDR IRF ID.
- **plot_x_inches** – float; width of plot in inches.
- **plot_y_inches** – float; height of plot in inches.
- **ylim** – 2-element tuple or list; (lower_bound, upper_bound) to use for y axis. If None, automatically inferred.
- **cmap** – str; name of matplotlib cmap object (determines colors of plotted IRF).
- **legend** – bool; include a legend.
- **xlab** – str or None; x-axis label. If None, no label.
- **ylab** – str or None; y-axis label. If None, no label.
- **use_line_markers** – bool; add markers to IRF lines.
- **transparent_background** – bool; use a transparent background. If False, uses a white background.
- **dpi** – int; dots per inch.
- **dump_source** – bool; Whether to dump the plot source array to a csv file.

Returns None

```
cdr.plot.plot_qq(theoretical, actual, actual_color='royalblue', expected_color='firebrick', dir='.',  
                filename='qq_plot.png', plot_x_inches=6, plot_y_inches=4, legend=True,  
                xlab='Theoretical', ylab='Empirical', ticks=True, as_lines=False, transparent_  
                _background=False, dpi=300)
```

Generate quantile-quantile plot.

Parameters

- **theoretical** – numpy array with shape (T,); theoretical error quantiles.
- **actual** – numpy array with shape (T,); empirical errors.
- **actual_color** – str; color for actual values.
- **expected_color** – str; color for expected values.

- **dir** – str; output directory.
- **filename** – str; filename.
- **plot_x_inches** – float; width of plot in inches.
- **plot_y_inches** – float; height of plot in inches.
- **legend** – bool; include a legend.
- **xlab** – str or None; x-axis label. If None, no label.
- **ylab** – str or None; y-axis label. If None, no label.
- **as_lines** – bool; render QQ plot using lines. Otherwise, use points.
- **transparent_background** – bool; use a transparent background. If False, uses a white background.
- **dpi** – int; dots per inch.

Returns None

```
cdr.plot.plot_surface(x, y, z, lq=None, uq=None, density=None, bounds_as_surface=False,
                    dir='.', filename='surface.png', irf_name_map=None, plot_x_inches=6,
                    plot_y_inches=4, xlim=None, ylim=None, zlim=None, plot_type='wireframe',
                    cmap='coolwarm', xlab=None, ylab=None, zlab='Response', title=None,
                    transparent_background=False, dpi=300, dump_source=False)
```

Plot an IRF or interaction surface.

Parameters

- **x** – numpy array with shape (M,N); x locations for each plot point, copied N times.
- **y** – numpy array with shape (M,N); y locations for each plot point, copied M times.
- **z** – numpy array with shape (M,N); z locations for each plot point.
- **lq** – numpy array with shape (M,N), or None; lower bound of credible interval for each plot point. If None, no credible interval will be plotted.
- **uq** – numpy array with shape (M,N), or None; upper bound of credible interval for each plot point. If None, no credible interval will be plotted.
- **bounds_as_surface** – bool; whether to plot interval bounds using additional surfaces. If False, bounds are plotted with vertical error bars instead. Ignored if lq, uq are None.
- **dir** – str; output directory.
- **filename** – str; filename.
- **irf_name_map** – dict of str to str; map from CDR IRF ID's to more readable names to appear in legend. Any plotted IRF whose ID is not found in **irf_name_map** will be represented with the CDR IRF ID.
- **plot_x_inches** – float; width of plot in inches.
- **plot_y_inches** – float; height of plot in inches.
- **xlim** – 2-element tuple or list or None; (lower_bound, upper_bound) to use for x axis. If None, automatically inferred.
- **ylim** – 2-element tuple or list or None; (lower_bound, upper_bound) to use for y axis. If None, automatically inferred.
- **zlim** – 2-element tuple or list or None; (lower_bound, upper_bound) to use for z axis. If None, automatically inferred.

- **plot_type** – str; name of plot type to generate. One of ["contour", "surf", "trisurf"].
- **cmap** – str; name of matplotlib cmap object (determines colors of plotted IRF).
- **legend** – bool; include a legend.
- **xlab** – str or None; x-axis label. If None, no label.
- **ylab** – str or None; y-axis label. If None, no label.
- **zlab** – str or None; z-axis label. If None, no label.
- **use_line_markers** – bool; add markers to IRF lines.
- **transparent_background** – bool; use a transparent background. If False, uses a white background.
- **dpi** – int; dots per inch.
- **dump_source** – bool; Whether to dump the plot source array to a csv file.

Returns None

6.16 cdr.signif module

`cdr.signif.correlation_test` (*y*, *x1*, *x2*, *nested=False*, *verbose=True*)

Perform a parametric test of difference in correlation with observations between two prediction vectors, based on Steiger (1980).

Parameters

- **y** – numpy vector; observation vector.
- **x1** – numpy vector; first prediction vector.
- **x2** – numpy vector; second prediction vector.
- **nested** – bool; assume that the second model is nested within the first.
- **verbose** – bool; report progress logs to standard error.

Returns

`cdr.signif.permutation_test` (*a*, *b*, *n_iter=10000*, *n_tails=2*, *mode='loss'*, *nested=False*, *verbose=True*)

Perform a paired permutation test for significance.

Parameters

- **a** – numpy vector; first error/loss/prediction vector.
- **b** – numpy vector; second error/loss/prediction vector.
- **n_iter** – int; number of resampling iterations.
- **n_tails** – int; number of tails.
- **mode** – str; one of ["mse", "loglik"], the type of error used (SE's are averaged while loglik's are summed).
- **nested** – bool; assume that the second model is nested within the first.
- **verbose** – bool; report progress logs to standard error.

Returns

6.17 cdr.synth module

class `cdr.synth.SyntheticModel` (*n_pred*, *irf_name*, *irf_params=None*, *coefs=None*)

Bases: `object`

A data structure representing a synthetic “true” model for empirical validation of CDR fits. Contains a randomly generated set of IRFs that can be used to convolve data, and provides methods for sampling data with particular structure and convolving it with the true IRFs in order to generate a response vector.

Parameters

- **n_pred** – `int`; Number of predictors in the synthetic model.
- **irf_name** – `str`; Name of IRF kernel to use. One of ['Exp', 'Normal', 'Gamma', 'ShiftedGamma'].
- **irf_params** – `dict` or `None`; Dictionary of IRF parameters to use, with parameter names as keys and numeric arrays as values. Values must each have **n_pred** cells. If `None`, parameter values will be randomly sampled.
- **coefs** – `numpy array` or `None`; Vector of coefficients to use, where `len(coefs) == n_pred`. If `None`, coefficients will be randomly sampled.

convolve (*X*, *t_X*, *t_y*, *history_length=None*, *err_sd=None*, *allow_instantaneous=True*, *verbose=True*)

Convolve data using the model’s IRFs.

Parameters

- **X** – `numpy array`; 2-D array of predictors.
- **t_X** – `numpy array`; 1-D vector of predictor timestamps.
- **t_y** – `numpy array`; 1-D vector of response timestamps.
- **history_length** – `int` or `None`; Drop preceding events more than `history_length` steps into the past. If `None`, no history clipping.
- **err_sd** – `float` or `None`; Standard deviation of Gaussian noise to inject into responses. If `None`, use the empirical standard deviation of the response vector.
- **allow_instantaneous** – `bool`; Whether to compute responses when `t==0`.
- **verbose** – `bool`; Verbosity.

Returns (2-D `numpy array`, 1-D `numpy array`); Matrix of convolved predictors, vector of responses

convolve_v2 (*X*, *t_X*, *t_y*, *err_sd=None*, *allow_instantaneous=True*, *verbose=True*)

Convolve data using the model’s IRFs. Alternate memory-intensive implementation that is faster for small arrays but can exhaust resources for large ones.

Parameters

- **X** – `numpy array`; 2-D array of predictors.
- **t_X** – `numpy array`; 1-D vector of predictor timestamps.
- **t_y** – `numpy array`; 1-D vector of response timestamps.
- **err_sd** – `float`; Standard deviation of Gaussian noise to inject into responses.
- **allow_instantaneous** – `bool`; Whether to compute responses when `t==0`.
- **verbose** – `bool`; Verbosity.

Returns (2-D numpy array, 1-D numpy array); Matrix of convolved predictors, vector of responses

get_curves (*n_time_units=None, n_time_points=None*)

Extract response curves as an array.

Parameters

- **n_time_units** – float; Number of units of time over which to extract curves.
- **n_time_points** – int; Number of samples to extract for each curve (resolution of curve)

Returns numpy array; 2-D numpy array with shape $[T, K]$, where T is **n_time_points** and K is the number of predictors in the model.

irf (*x, coefs=False*)

Computes the values of the model's IRFs elementwise over a vector of timepoints.

Parameters

- **x** – numpy array; 1-D array with shape $[N]$ containing timepoints at which to query the IRFs.
- **coefs** – bool; Whether to rescale responses by coefficients

Returns numpy array; 2-D array with shape $[N, K]$ containing values of the model's K IRFs evaluated at the timepoints in **x**.

plot_irf (*n_time_units=None, n_time_points=None, dir='.', filename='synth_irf.png', plot_x_inches=6, plot_y_inches=4, cmap='gist_rainbow', legend=False, xlab=None, ylab=None, use_line_markers=False, transparent_background=False*)

Plot impulse response functions.

Parameters

- **n_time_units** – float; number if time units to use for plotting.
- **n_time_points** – int; number of points to use for plotting.
- **dir** – str; output directory.
- **filename** – str; filename.
- **plot_x_inches** – float; width of plot in inches.
- **plot_y_inches** – float; height of plot in inches.
- **cmap** – str; name of matplotlib cmap object (determines colors of plotted IRF).
- **legend** – bool; include a legend.
- **xlab** – str or None; x-axis label. If None, no label.
- **ylab** – str or None; y-axis label. If None, no label.
- **use_line_markers** – bool; add markers to IRF lines.
- **transparent_background** – bool; use a transparent background. If False, uses a white background.

Returns None

sample_data (*m, n=None, X_interval=None, y_interval=None, rho=None, align_X_y=True*)

Samples synthetic predictors and time vectors

Parameters

- **m** – int; Number of predictors.
- **n** – int; Number of response query points.
- **X_interval** – str, float, list, tuple, or None; Predictor interval model. If None, predictor offsets are randomly sampled from an exponential distribution with parameter 1. If float, predictor offsets are evenly spaced with interval **X_interval**. If list or tuple, the first element is the name of a scipy distribution to use for sampling offsets, and all remaining elements are positional arguments to that distribution.
- **y_interval** – str, float, list, tuple, or None; Response interval model. If None, response offsets are randomly sampled from an exponential distribution with parameter 1. If float, response offsets are evenly spaced with interval **y_interval**. If list or tuple, the first element is the name of a scipy distribution to use for sampling offsets, and all remaining elements are positional arguments to that distribution.
- **rho** – float; Level of pairwise correlation between predictors.
- **align_X_y** – bool; Whether predictors and responses are required to be sampled at the same points in time.

Returns (2-D numpy array, 1-D numpy array, 1-D numpy array); Matrix of predictors, vector of predictor timestamps, vector of response timestamps

6.18 cdr.util module

`cdr.util.filter_models` (*names, filters, cdr_only=False*)

Return models contained in **names** that are permitted by **filters**, preserving order in which filters were matched. Filters can be ordinary strings, regular expression objects, or string representations of regular expressions. For a regex filter to be considered a match, the expression must entirely match the name. If **filters** is zero-length, returns **names**.

Parameters

- **names** – list of str; pool of model names to filter.
- **filters** – list of {str, SRE_Pattern}; filters to apply in order
- **cdr_only** – bool; if True, only returns CDR models. If False, returns all models admitted by **filters**.

Returns list of str; names in **names** that pass at least one filter, or all of **names** if no filters are applied.

`cdr.util.filter_names` (*names, filters*)

Return elements of **names** permitted by **filters**, preserving order in which filters were matched. Filters can be ordinary strings, regular expression objects, or string representations of regular expressions. For a regex filter to be considered a match, the expression must entirely match the name.

Parameters

- **names** – list of str; pool of names to filter.
- **filters** – list of {str, SRE_Pattern}; filters to apply in order

Returns list of str; names in **names** that pass at least one filter

`cdr.util.get_random_permutation` (*n*)

Draw a random permutation of integers 0 to **n**. Used to shuffle arrays of length **n**. For example, a permutation and its inverse can be generated by calling `p, p_inv = get_random_permutation(n)`. To randomly shuffle an **n**-dimensional vector **x**, call `x[p]`. To un-shuffle **x** after it has already been shuffled, call `x[p_inv]`.

Parameters *n* – maximum value

Returns 2-tuple of numpy arrays; the permutation and its inverse

`cdr.util.load_cdr(dir_path)`

Convenience method for reconstructing a saved CDR object. First loads in metadata from `m.obj`, then uses that metadata to construct the computation graph. Then, if saved weights are found, these are loaded into the graph.

Parameters *dir_path* – Path to directory containing the CDR checkpoint files.

Returns The loaded CDR instance.

`cdr.util.mae(true, preds)`

Compute mean absolute error (MAE).

Parameters

- **true** – True values
- **preds** – Predicted values

Returns float; MAE

`cdr.util.mse(true, preds)`

Compute mean squared error (MSE).

Parameters

- **true** – True values
- **preds** – Predicted values

Returns float; MSE

`cdr.util.names2ix(names, l, dtype=<class 'numpy.int32'>)`

Generate 1D numpy array of indices in *l* corresponding to names in *names*

Parameters

- **names** – list of str; names to look up in *l*
- **l** – list of str; list of names from which to extract indices
- **dtype** – numpy dtype object; return dtype

Returns numpy array; indices of *names* in *l*

`cdr.util.nested(model_name_1, model_name_2)`

Check whether two CDR models are nested with 1 degree of freedom

Parameters

- **model_name_1** – str; name of first model
- **model_name_2** – str; name of second model

Returns bool; True if models are nested with 1 degree of freedom, False otherwise

`cdr.util.pca(X, n_dim=None, dtype=<class 'numpy.float32'>)`

Perform principal components analysis on a data table.

Parameters

- **X** – numpy or pandas array; the input data
- **n_dim** – int or None; maximum number of principal components. If None, all components are retained.

- **dtype** – numpy dtype; return dtype

Returns 5-tuple of numpy arrays; transformed data, eigenvectors, eigenvalues, input means, and input standard deviations

`cdr.util.percent_variance_explained(true, preds)`

Compute percent variance explained.

Parameters

- **true** – True values
- **preds** – Predicted values

Returns float; percent variance explained

`cdr.util.reg_name(string)`

Standardize a variable name for regularization

Parameters **string** – str; input string

Returns str; transformed string

`cdr.util.sn(string)`

Compute a Tensorboard-compatible version of a string.

Parameters **string** – str; input string

Returns str; transformed string

INDICES

- [genindex](#)

PYTHON MODULE INDEX

C

cdr.backend, 39
cdr.config, 39
cdr.data, 40
cdr.formula, 46
cdr.io, 57
cdr.kwargs, 58
cdr.opt, 59
cdr.plot, 59
cdr.signif, 62
cdr.synth, 63
cdr.util, 65

A

`ablate_impulses()` (*cdr.formula.Formula method*),
 46
`ablate_impulses()` (*cdr.formula.IRFNode
 method*), 50
`add_child()` (*cdr.formula.IRFNode method*), 50
`add_interactions()` (*cdr.formula.IRFNode
 method*), 50
`add_rangf()` (*cdr.formula.IRFNode method*), 50
`add_rangf()` (*cdr.formula.ResponseInteraction
 method*), 56
`add_responses()` (*in module cdr.data*), 40
`apply_formula()` (*cdr.formula.Formula method*), 46
`apply_op()` (*cdr.formula.Formula method*), 46
`apply_op_2d()` (*cdr.formula.Formula method*), 47
`apply_ops()` (*cdr.formula.Formula method*), 47
`apply_ops_2d()` (*cdr.formula.Formula method*), 47
`atomic_irf_by_family()` (*cdr.formula.IRFNode
 method*), 50
`atomic_irf_param_init_by_family()`
 (*cdr.formula.IRFNode method*), 51
`atomic_irf_param_trainable_by_family()`
 (*cdr.formula.IRFNode method*), 51

B

`bases()` (*cdr.formula.Formula static method*), 47
`bases()` (*cdr.formula.IRFNode method*), 51
`build()` (*cdr.formula.Formula method*), 47
`build_CDR_impulse_data()` (*in module cdr.data*),
 40
`build_CDR_response_data()` (*in module
 cdr.data*), 41
`build_cdr_settings()` (*cdr.config.Config
 method*), 39
`build_plot_settings()` (*cdr.config.PlotConfig
 method*), 40

C

`c()` (*in module cdr.data*), 42
`categorical()` (*cdr.formula.Impulse method*), 55
`categorical_transform()` (*cdr.formula.Formula
 method*), 47

`categorical_transform()` (*cdr.formula.IRFNode
 method*), 51
`cdr.backend` (*module*), 39
`cdr.config` (*module*), 39
`cdr.data` (*module*), 40
`cdr.formula` (*module*), 46
`cdr.io` (*module*), 57
`cdr.kwargs` (*module*), 58
`cdr.opt` (*module*), 59
`cdr.plot` (*module*), 59
`cdr.signif` (*module*), 62
`cdr.synth` (*module*), 63
`cdr.util` (*module*), 65
`cdr_kwarg_docstring()` (*in module cdr.kwargs*),
 58
`CDRNNStateTuple` (*in module cdr.backend*), 39
`coef2impulse()` (*cdr.formula.IRFNode method*), 51
`coef2terminal()` (*cdr.formula.IRFNode method*),
 51
`coef_by_rangf()` (*cdr.formula.IRFNode method*),
 51
`coef_id()` (*cdr.formula.IRFNode method*), 51
`coef_names()` (*cdr.formula.IRFNode method*), 51
`compute_2d_predictor()` (*cdr.formula.Formula
 method*), 47
`compute_filter()` (*in module cdr.data*), 42
`compute_filters()` (*in module cdr.data*), 42
`compute_partition()` (*in module cdr.data*), 42
`compute_splitID()` (*in module cdr.data*), 42
`compute_time_mask()` (*in module cdr.data*), 42
`Config` (*class in cdr.config*), 39
`contains_member()`
 (*cdr.formula.ResponseInteraction method*),
 56
`convolve()` (*cdr.synth.SyntheticModel method*), 63
`convolve_v2()` (*cdr.synth.SyntheticModel method*),
 63
`corr_cdr()` (*in module cdr.data*), 43
`correlation_test()` (*in module cdr.signif*), 62

D

`depth()` (*cdr.formula.IRFNode method*), 51

- `docstring_from_kwarg()` (in module `cdr.kwargs`), 59
- `dtypes_str()` (`cdr.kwargs.Kwarg` method), 58
- ## E
- `expand_categorical()` (`cdr.formula.Impulse` method), 55
- `expand_categorical()` (`cdr.formula.ImpulseInteraction` method), 55
- `expand_impulse_sequence()` (in module `cdr.data`), 43
- ## F
- `filter_invalid_responses()` (in module `cdr.data`), 43
- `filter_models()` (in module `cdr.util`), 65
- `filter_names()` (in module `cdr.util`), 65
- `fixed_coef_names()` (`cdr.formula.IRFNode` method), 51
- `fixed_interaction_names()` (`cdr.formula.IRFNode` method), 51
- `Formula` (class in `cdr.formula`), 46
- `formula_terms()` (`cdr.formula.IRFNode` method), 52
- ## G
- `get_curves()` (`cdr.synth.SyntheticModel` method), 64
- `get_first_last_obs_lists()` (in module `cdr.data`), 44
- `get_random_permutation()` (in module `cdr.util`), 65
- `get_rangf_array()` (in module `cdr.data`), 44
- `get_time_windows()` (in module `cdr.data`), 44
- `get_type_name()` (`cdr.kwargs.Kwarg` method), 58
- ## H
- `has_coefficient()` (`cdr.formula.IRFNode` method), 52
- `has_composed_irf()` (`cdr.formula.IRFNode` method), 52
- `has_irf()` (`cdr.formula.IRFNode` method), 52
- ## I
- `Impulse` (class in `cdr.formula`), 55
- `impulse2coef()` (`cdr.formula.IRFNode` method), 52
- `impulse2terminal()` (`cdr.formula.IRFNode` method), 52
- `impulse_names()` (`cdr.formula.IRFNode` method), 52
- `ImpulseInteraction` (class in `cdr.formula`), 55
- `impulses()` (`cdr.formula.ImpulseInteraction` method), 55
- `impulses()` (`cdr.formula.IRFNode` method), 52
- `impulses_by_name()` (`cdr.formula.IRFNode` method), 52
- `impulses_from_response_interaction()` (`cdr.formula.IRFNode` method), 52
- `in_settings()` (`cdr.kwargs.Kwarg` method), 58
- `insert_impulses()` (`cdr.formula.Formula` method), 48
- `interaction_by_rangf()` (`cdr.formula.IRFNode` method), 53
- `interaction_names()` (`cdr.formula.IRFNode` method), 53
- `interactions()` (`cdr.formula.IRFNode` method), 53
- `interactions2inputs()` (`cdr.formula.IRFNode` method), 53
- `irf()` (`cdr.synth.SyntheticModel` method), 64
- `irf_by_rangf()` (`cdr.formula.IRFNode` method), 53
- `irf_id()` (`cdr.formula.IRFNode` method), 53
- `irf_params()` (`cdr.formula.Formula` static method), 48
- `irf_responses()` (`cdr.formula.ResponseInteraction` method), 56
- `irf_to_formula()` (`cdr.formula.IRFNode` method), 53
- `IRFNode` (class in `cdr.formula`), 49
- `is_LCG()` (`cdr.formula.Formula` static method), 48
- `is_LCG()` (`cdr.formula.IRFNode` method), 53
- ## K
- `Kwarg` (class in `cdr.kwargs`), 58
- `kwarg_from_config()` (`cdr.kwargs.Kwarg` method), 58
- ## L
- `load_cdr()` (in module `cdr.util`), 66
- `local_name()` (`cdr.formula.IRFNode` method), 53
- ## M
- `mae()` (in module `cdr.util`), 66
- `MidpointNormalize` (class in `cdr.plot`), 59
- `mse()` (in module `cdr.util`), 66
- ## N
- `name()` (`cdr.formula.Impulse` method), 55
- `name()` (`cdr.formula.ImpulseInteraction` method), 55
- `name()` (`cdr.formula.IRFNode` method), 53
- `name()` (`cdr.formula.ResponseInteraction` method), 56
- `names2ix()` (in module `cdr.util`), 66
- `nested()` (in module `cdr.util`), 66
- `node_table()` (`cdr.formula.IRFNode` method), 53
- `non_irf_responses()` (`cdr.formula.ResponseInteraction` method), 56

`nonparametric_coef_names()`
(*cdr.formula.IRFNode* method), 53

P

`pc_transform()` (*cdr.formula.Formula* method), 48
`pc_transform()` (*cdr.formula.IRFNode* method), 53
`pca()` (in module *cdr.util*), 66
`percent_variance_explained()` (in module *cdr.util*), 67
`permutation_test()` (in module *cdr.signif*), 62
`plot_heatmap()` (in module *cdr.plot*), 59
`plot_irf()` (*cdr.synth.SyntheticModel* method), 64
`plot_irf()` (in module *cdr.plot*), 59
`plot_kwarg_docstring()` (in module *cdr.kwargs*), 59
`plot_qq()` (in module *cdr.plot*), 60
`plot_surface()` (in module *cdr.plot*), 61
`PlotConfig` (class in *cdr.config*), 40
`pointers2namemmaps()` (*cdr.formula.IRFNode* static method), 54
`preprocess_data()` (in module *cdr.data*), 44
`process_ast()` (*cdr.formula.Formula* method), 48
`process_irf()` (*cdr.formula.Formula* method), 49
`pythonize_string()` (in module *cdr.formula*), 57

R

`read_tabular_data()` (in module *cdr.io*), 57
`reg_name()` (in module *cdr.util*), 67
`remove_impulses()` (*cdr.formula.Formula* method), 49
`remove_impulses()` (*cdr.formula.IRFNode* method), 54
`replace()` (*cdr.formula.ResponseInteraction* method), 56
`response_names()` (*cdr.formula.Formula* method), 49
`ResponseInteraction` (class in *cdr.formula*), 56
`responses()` (*cdr.formula.Formula* method), 49
`responses()` (*cdr.formula.ResponseInteraction* method), 56

S

`s()` (in module *cdr.data*), 45
`sample_data()` (*cdr.synth.SyntheticModel* method), 64
`set_model()` (*cdr.config.Config* method), 39
`sn()` (in module *cdr.util*), 67
`split_cdr_outputs()` (in module *cdr.data*), 45
`standardize_formula_string()` (in module *cdr.formula*), 57
`supports_non_causal()` (*cdr.formula.IRFNode* method), 54
`SyntheticModel` (class in *cdr.synth*), 63

T

`terminal()` (*cdr.formula.IRFNode* method), 54
`terminal2coef()` (*cdr.formula.IRFNode* method), 54
`terminal2impulse()` (*cdr.formula.IRFNode* method), 54
`terminal_names()` (*cdr.formula.IRFNode* method), 54
`terminals()` (*cdr.formula.IRFNode* method), 54
`terminals_by_name()` (*cdr.formula.IRFNode* method), 54
`to_lmer_formula_string()` (*cdr.formula.Formula* method), 49
`to_string()` (*cdr.formula.Formula* method), 49
`type_comparator()` (*cdr.kwargs.Kwarg* static method), 58

U

`unablate_impulses()` (*cdr.formula.Formula* method), 49
`unablate_impulses()` (*cdr.formula.IRFNode* method), 54
`unary_nonparametric_coef_names()` (*cdr.formula.IRFNode* method), 55

Z

`z()` (in module *cdr.data*), 45