Pygformula Documentation

Release 0.0.1

Apr 24, 2024

CONTENTS

The pygformula package implements the non-iterative conditional expectation (NICE) algorithm of the g-formula algorithm[1, 2]. The g-formula can estimate an outcome's counterfactual mean or risk under hypothetical treatment strategies (interventions) when there is sufficient information on time-varying treatments and confounders.

This package can be used for discrete or continuous time-varying treatments and for failure time outcomes or continuous/binary end of follow-up outcomes. The package can handle a random measurement/visit process and a priori knowledge of the data structure, as well as censoring (e.g., by loss to follow-up) and two options for handling competing events for failure time outcomes. Interventions can be flexibly specified, both as interventions on a single treatment or as joint interventions on multiple treatments.

For a quick overview of how to use the pygformula, see a simple example in *Get Started*. For a detailed list of options, see *Specifications*.

[1] Robins JM. A new approach to causal inference in mortality studies with a sustained exposure period: application to the healthy worker survivor effect. Mathematical Modelling. 1986;7:1393–1512. [Errata (1987) in Computers and Mathematics with Applications 14, 917-921. Addendum (1987) in Computers and Mathematics with Applications 14, 923-945. Errata (1987) to addendum in Computers and Mathematics with Applications 18, 477.

[2] Hernán, M.A., and Robins, J. (2020). Causal Inference: What If (Chapman & Hall/CRC).

CHAPTER

ONE

INSTALLATION

1.1 Requirements

The package requires python $\geq$ 3.8 and these necessary dependencies:

- cmprsk
- joblib
- lifelines
- matplotlib
- numpy
- pandas
- prettytable
- pytruncreg
- scipy
- seaborn
- statsmodels
- tqdm

All the dependencies needed by the pygformula are listed in the file "requirements.txt" , users can install them by:

```
pip install -r requirements.txt
```

1.2 Install pygformula

Users can use the following command to install the pygformula package:

```
pip install pygformula
```

CHAPTER

TWO

GET STARTED

2.1 Algorithm outline

The parametric g-formula estimator of the noniterative conditional expectation (NICE) requires the specification of models for the joint density of the confounders, treatments, and outcomes over time. The algorithm has three steps: (1) Parametric estimation, (2) Monte Carlo simulation , and (3) Calculation of risk/mean under each intervention.

- **Parametric estimation**: (a) estimate the conditional densities of each covariate given past covariate history by fitting user-specified regression models, (b) estimate the discrete hazard (for survival outcome) or mean (for binary/continuous end of follow-up) of the outcome conditional on past covariate history by fitting a user-specified regression model, (c) if the event of interest is subject to competing events and competing events are not treated as censoring events, estimate the conditional probability of the competing event conditional on past covariate history by fitting user-specified regression model for the competing event.
- **Monte Carlo simulation**: (a) generate a new dataset which is usually larger than original dataset, for each covariate, generate simulated values at each time step using the estimated covariate models from step (1), (b) for the covariates that are to undergo intervention, their values are assigned according to the user-specified intervention rule, (c) obtain the discrete hazard / mean of the outcome based on the estimated outcome model from step (1), (d) if the event of interest is subject to competing events and competing events are not treated as censoring events, obtain the discrete hazard of the competing event based on the estimated competing model from step (1).
- **Calculation of risk/mean under each intervention**: for binary/continuous end of follow-up, the final estimate is the mean of the estimated outcome of all individuals in the new dataset computed from Step (2). For survival outcome, the final estimate is obtained by calculating the mean of cumulative risks for all individuals using the discrete hazards computed from step (2).

Arguments:

ParametricGformula(obs_data, id, time_name, ...)	G-formula estimation under parametric models.

class pygformula.parametric_gformula.**ParametricGformula**(*obs_data*, *id*, *time_name*, *outcome_name*, *ymodel*, *covnames=None*, *covtypes=None*, *covmodels=None*, *int_descript=None*, *custom_histvars=None*, *custom_histories=None*, *covfits_custom=None*, *covpredict_custom=None*, *nsamples=0*, *compevent_name=None*, *compevent_model=None*, *compevent_cens=False*, *intcomp=None*, *censor_name=None*, *censor_model=None*, *model_fits=False*, *boot_diag=False*, *ipw_cutoff_quantile=None*, *ipw_cutoff_value=None*, *outcome_type=None*, *trunc_params=None*, *time_thresholds=None*, *time_points=None*, *n_simul=None*, *baselags=False*, *visitprocess=None*, *restrictions=None*, *yrestrictions=None*, *compevent_restrictions=None*, *basecovs=None*, *parallel=False*, *ncores=None*, *ref_int=None*, *ci_method=None*, *seed=None*, *save_path=None*, *save_results=False*, ***interventions*)

G-formula estimation under parametric models.

Parameters

- **obs_data** (*DataFrame*) – A data frame containing the observed data.
- **id** (*Str*) – A string specifying the name of the id variable in obs_data.
- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **outcome_name** (*Str*) – A string specifying the name of the outcome variable in obs_data.
- **ymodel** (*Str*) – A string specifying the model statement for the outcome variable.
- **covnames** (*List, default is None*) – A list of strings specifying the names of the time-varying covariates in obs_data.
- **covtypes** (*List, default is None*) – A list of strings specifying the "type" of each time-varying covariate included in covnames. The supported types: "binary", "normal", "categorical", "bounded normal", "zero-inflated normal", "truncated normal", "absorbing", "categorical time", "square time" and "custom". The list must be the same length as covnames and in the same order.
- **covmodels** (*List, default is None*) – A list of strings, where each string is the model statement of the time-varying covariate. The list must be the same length as covnames and in the same order. If a model is not required for a certain covariate, it should be set to 'NA' at that index.
- **int_descript** (*List, default is None*) – A list of strings, each of which describes a user-specified intervention.
- **custom_histvars** (*List, default is None*) – A list of strings, each of which specifies the names of the time-varying covariates with user-specified custom histories.

- **custom_histories** (*List, default is None*) – A list of functions, each function is the user-specified custom history functions for covariates. The list should be the same length as custom_histvars and in the same order.
- **covfits_custom** (*List, default is None*) – A list, each element could be 'NA' or a user-specified fit function. The non-NA value is set for the covariates with custom type. The 'NA' value is set for other covariates. The list must be the same length as covnames and in the same order.
- **covpredict_custom** (*List, default is None*) – A list, each element could be 'NA' or a user-specified predict function. The non-NA value is set for the covariates with custom type. The 'NA' value is set for other covariates. The list must be the same length as covnames and in the same order.
- **nsamples** (*Int, default is 0*) – An integer specifying the number of bootstrap samples to generate.
- **compevent_name** (*Str, default is None*) – A string specifying the name of the competing event variable in obs_data. Only applicable for survival outcomes.
- **compevent_model** (*Str, default is None*) – A string specifying the model statement for the competing event variable. Only applicable for survival outcomes.
- **compevent_cens** (*Bool, default is False*) – A boolean value indicating whether to treat competing events as censoring events.
- **intcomp** (*List, default is None*) – List of two numbers indicating a pair of interventions to be compared by a hazard ratio.
- **censor_name** (*Str, default is None*) – A string specifying the name of the censoring variable in obs_data. Only applicable when using inverse probability weights to estimate the natural course means / risk from the observed data.
- **censor_model** (*Str, default is None*) – A string specifying the model statement for the censoring variable. Only applicable when using inverse probability weights to estimate the natural course means / risk from the observed data.
- **model_fits** (*Bool, default is False*) – A boolean value indicating whether to return the parameter estimates of the models.
- **boot_diag** (*Bool, default is False*) – A boolean value indicating whether to return the parametric g-formula estimates as well as the coefficients, standard errors, and variance-covariance matrices of the parameters of the fitted models in the bootstrap samples.
- **ipw_cutoff_quantile** (*Float, default is None*) – Percentile value for truncation of the inverse probability weights.
- **ipw_cutoff_value** (*Float, default is None*) – Absolute value for truncation of the inverse probability weights.
- **outcome_type** (*Str, default is None*) – A string specifying the "type" of outcome. The possible "types" are: "survival", "continuous_eof", and "binary_eof".
- **trunc_params** (*List, default is None*) – A list, each element could be 'NA' or a two-element list. If not 'NA', the first element specifies the truncated value and the second element specifies the truncated direction ('left' or 'right'). The non-NA value is set for the truncated normal covariates. The 'NA' value is set for other covariates. The list should be the same length as covnames and in the same order.
- **time_thresholds** (*List, default is None*) – A list of integers that splits the time points into different intervals. It is used to create the variable "categorical time".

- **time_points** (*Int, default is K+1*) – An integer indicating the number of time points to simulate. It is set equal to the maximum number of records (K) that obs_data contains for any individual plus 1, if not specified by users.
- **n_simul** (*Int, default is M*) – An integer indicating the number of subjects for whom to simulate data. It is set equal to the number (M) of subjects in obs_data, if not specified by users.
- **baselags** (*Bool, default is False*) – A boolean value specifying the convention used for lagi and lag_cumavgi terms in the model statements when pre-baseline times are not included in obs_data and when the current time index, t, is such that t < i. If this argument is set to False, the value of all lagi and lag_cumavgi terms in this context are set to 0 (for non-categorical covariates) or the reference level (for categorical covariates). If this argument is set to True, the value of lagi and lag_cumavgi terms are set to their values at time 0. The default is False.
- **visitprocess** (*List, default is None*) – List of lists. Each inner list contains its first entry the covariate name of a visit process; its second entry the name of a covariate whose modeling depends on the visit process; and its third entry the maximum number of consecutive visits that can be missed before an individual is censored.
- **restrictions** (*List, default is None*) – List of lists. Each inner list contains its first entry the covariate name of that its deterministic knowledge is known; its second entry is a dictionary whose key is the conditions which should be True when the covariate is modeled, the third entry is the value that is set to the covariate during simulation when the conditions in the second entry are not True.
- **yrestrictions** (*List, default is None*) – List of lists. For each inner list, its first entry is a dictionary whose key is the conditions which should be True when the outcome is modeled, the second entry is the value that is set to the outcome during simulation when the conditions in the first entry are not True.
- **compevent_restrictions** (*List, default is None*) – List of lists. For each inner list, its first entry is a dictionary whose key is the conditions which should be True when the competing event is modeled, the second entry is the value that is set to the competing event during simulation when the conditions in the first entry are not True. Only applicable for survival outcomes.
- **basecovs** (*List, default is None*) – A list of strings specifying the names of baseline covariates in obs_data. These covariates should not be included in covnames.
- **parallel** (*Bool, default is False*) – A boolean value indicating whether to parallelize simulations of different interventions to multiple cores.
- **ncores** (*Int, default is 1*) – An integer indicating the number of cores used in parallelization. It is set to 1 if not specified by users.
- **ref_int** (*Int, default is 0*) – An integer indicating the intervention to be used as the reference for calculating the end-of-follow-up mean/risk ratio and mean/risk difference. 0 denotes the natural course, while subsequent integers denote user-specified interventions in the order that they are named in interventions. It is set to 0 if not specified by users.
- **ci_method** (*Str, default is "percentile"*) – A string specifying the method for calculating the bootstrap 95% confidence intervals, if applicable. The options are "percentile" and "normal". It is set to "percentile" if not specified by users.
- **seed** (*Int, default is 1234*) – An integer indicating the starting seed for simulations and bootstrapping. It is set to 1234 if not specified by users.

- **save_path** (*Path, default is None*) – A path to save all the returned results. A folder will be created automatically in the current working directory if the save_path is not specified by users.
- **save_results** (*Bool, default is False*) – A boolean value indicating whether to save all the returned results to the save_path.
- ****interventions** (*Dict, default is None*) – A dictionary whose key is the treatment name in the intervention with the format Intervention{id}_{treatment_name}, value is a list that contains the intervention function, values required by the function, and a list of time points in which the intervention is applied.

2.2 Example

The observational dataset example_data_basicdata_nocomp consists of 13,170 observations on 2,500 individuals with a maximum of 7 follow-up times. The dataset contains the following variables:

- id: Unique identifier for each individual.
- t0: Time index.
- L1: Binary time-varying covariate.
- L2: Continuous time-varying covariate.
- L3: Categorical baseline covariate.
- A: Binary treatment variable.
- Y: Outcome of interest; time-varying indicator of failure.

We are interested in the risk by the end of follow-up under the static interventions "Never treat" (set treatment to 0 at all times) and "Always treat" (set treatment to 1 at all times).

- First, import the g-formula package:

```
import pygformula
from pygformula import ParametricGformula
```

- Then, load the data (here is an example of loading simulated data in the package, users can also load their own data) as required pandas DataFrame type

```
from pygformula.data import load_basicdata_nocomp
obs_data = load_basicdata_nocomp()
```

- Specify the name of the time variable, and the name of the individual identifier in the input data

```
time_name = 't0'
id = 'id'
```

- Specify covariate names, covariate types, and corresponding model statements

```
covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']
```

If there are baseline covariates (i.e., covariate with same value at all times) in the model statement, specify them in the "basecovs" argument:

```
basecovs = ['L3']
```

- Specify the static interventions of interest:

```
from pygformula.parametric_gformula.interventions import static

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

Intervention1_A = [static, np.zeros(time_points)],
Intervention2_A = [static, np.ones(time_points)],
```

- Specify the outcome name, outcome model statement, and the outcome type

```
outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'
```

- Speficy all the arguments in the "ParametricGformula" class and call its "fit" function:

```
g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    time_points=time_points,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    outcome_name=outcome_name, ymodel=ymodel,
    outcome_type = outcome_type)

g.fit()
```

- Finally, get the output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.50396             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.73346             |    1.45539     |       0.22950       |
|     Always treat    |    NA   |             0.23297             |    0.46228     |       -0.27099      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

  - "Intervention": the name of natural course intervention and user-specified interventions.
  - "NP-risk": the nonparametric estimates of the natural course risk.
  - "g-formula risk": the parametric g-formula estimates of each interventions.
  - "Risk Ratio (RR)": the risk ratio comparing each intervention and reference intervention.
  - "Risk Difference (RD)": the risk difference comparing each intervention and reference intervention.

In the output table, the g-formula risk results under the specified interventions are shown, as well as the natural course. Furthermore, the nonparametric risk under the natural course is provided, which can be used to assess model misspecification of parametric g-formula. The risk ratio and risk difference comparing the specific intervention and the reference intervention (set to natural course by default) are also calculated.

Users can also get the standard errors and 95% confidence intervals of the g-formula estimates by specifying the "nsamples" argument. For example, specifying "nsamples" as 20 with parallel processing using 8 cores:

```
g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type,
    nsamples=20, parallel=True, ncores=8)
g.fit()
```

The package will return following results:

Intervention	NP risk	g-formula risk (NICE-parametric)	Risk Standard Error	Risk 95% lower bound	Risk 95% upper bound
Natural course(ref)	0.50560	0.50396	0.00974	0.48622	0.51881
Never treat	NA	0.73346	0.02645	0.69309	0.78848
Always treat	NA	0.23297	0.01100	0.20990	0.24656

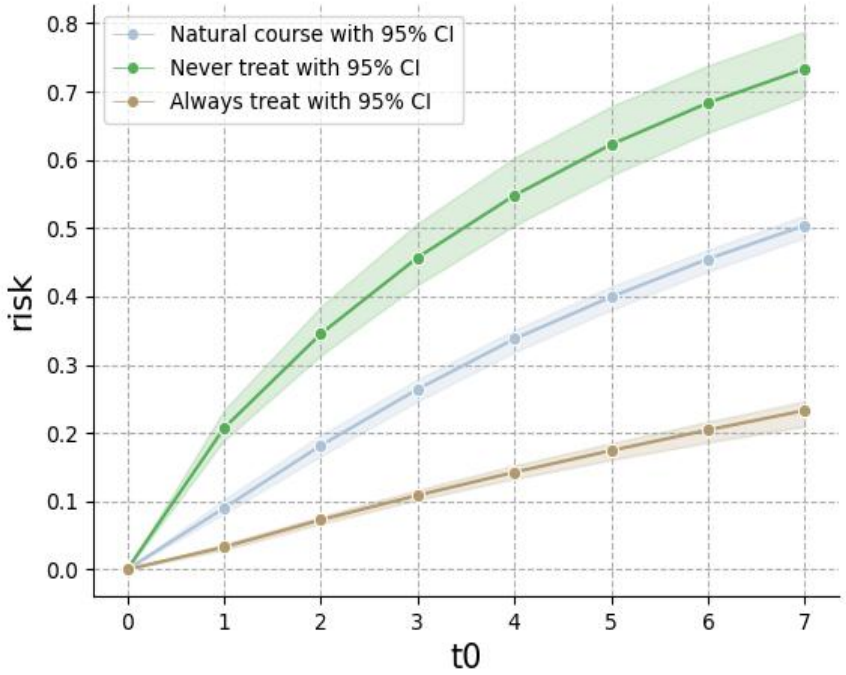
Risk Ratio(RR)	RR SE	RR 95% lower bound	RR 95% upper bound	Risk Difference(RD)	RD SE	RD 95% lower bound	RD 95% upper bound
1.00000	0.00000	1.00000	1.00000	0.00000	0.00000	0.00000	0.00000
1.45539	0.04858	1.36997	1.53877	0.22950	0.02434	0.18822	0.27486
0.46228	0.02169	0.41900	0.48869	-0.27099	0.01293	-0.29532	-0.24997

The result table contains 95% lower bound and upper bound for the risk, risk difference and risk ratio for all interventions.

The pygformula also provides plots for risk curves of interventions, which can be called by:

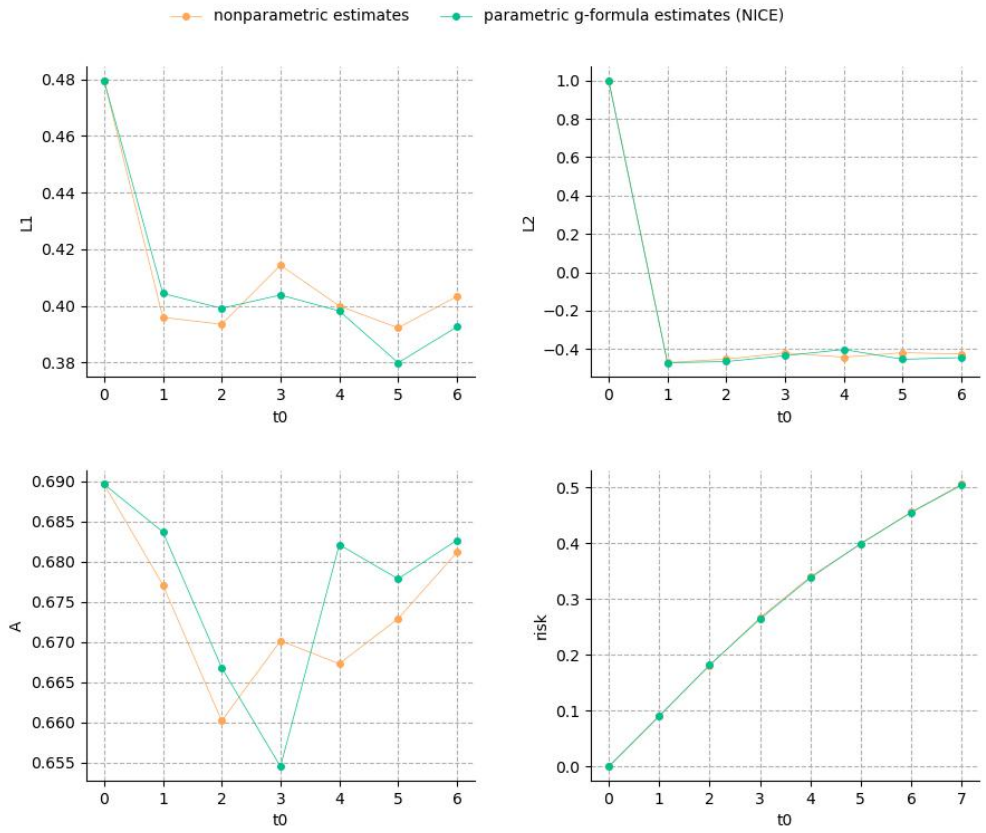
```
g.plot_interventions()
```

It will return the g-formula risk (with 95% confidence intervals if using bootstrap samples) at all follow-up times under each intervention:



User can also get the plots of parametric and nonparametric estimates of the risks and covariate means under natural course by:

```
g.plot_natural_course()
```



Running example [code]:

```
import numpy as np
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0
↪',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
```

(continues on next page)

(continued from previous page)

```
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
             time_points = time_points, int_descript = int_descript,
             covnames=covnames, covtypes=covtypes,
             covmodels=covmodels, basecovs=basecovs,
             outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_
↪type,
             Intervention1_A = [static, np.zeros(time_points)],
             Intervention2_A = [static, np.ones(time_points)],
             nsamples=20, parallel=True, ncores=8)
g.fit()
g.plot_natural_course()
g.plot_interventions()
```

CHAPTER

THREE

SPECIFICATIONS

The "Specifications" section gives detailed instructions about how to specify the required or optional arguments in different modules of pygformula to construct a specific analysis. To use the g-formula method in the package, the first step is to make sure that the input data meets the requirement of *Input data*. Then, users need to specify their parametric covariate model (see *Covariate models*), parametric outcome model (see *Outcome model*) , as well as the intervention of interest (see *Interventions*). Once these required modules are well-defined, the g-formula in pygformula can be called and output the results of the method.

Additionally, if there are censoring events, the package provides the option to obtain inverse probability weighted estimates for comparison with the g-formula estimates, see *Censoring event*. If there are competing events, the package provides two options for handling competing events in the case of survival outcomes, see *Competing event*. The package also provides option for calculating the hazard ratio of any two interventions of interest in *Hazard ratio*. If the data structure contains visit process, users can also perform g-formula analysis for this setting in *Visit process*. If there is deterministic knowledge about the relationship between the variables, it can be incorporated into the estimation of g-formula by applying restrictions, see *Deterministic knowledge*.

Contents:

3.1 Input data

The input dataset is specified by the "obs_data" argument which should contain: "id" specifying the individual identifier, "time_name" specifying the time index, "covnames" specifying the names of time-varying covariates, "outcome_name" specifying the name of the outcome of interest, "compevent_name" indicating the competing event status (if present), "censor_name" indicating the censoring event status (if present).

The related arguments:

Arguments	Description
obs_data	(Required) A data frame containing the observed data.
id	(Required) A string specifying the name of the id variable in obs_data.
time_name	(Required) A string specifying the name of the time variable in obs_data.
outcome_name	(Required) A string specifying the name of the outcome variable in obs_data.
covnames	(Required) A list of strings specifying the names of the time-varying covariates in obs_data.
compevent_name	(Optional) A string specifying the name of the competing event variable in obs_data. Only applicable for survival outcomes.
censor_name	(Optional) A string specifying the name of the censoring variable in obs_data. Only applicable when using inverse probability weights to estimate the natural course means / risk from the observed data.
time_points	(Optional) An integer indicating the number of time points to simulate. It is set equal to the maximum number of records (K) that obs_data contains for any individual plus 1, if not specified by users.

The input data should contain one record for each follow-up time k for each subject (identified by the individual identifier). The time index k for each subject should increment by 1 for each subsequent interval (the starting index is 0 in the following examples, pre-baseline times are also allowed). The record at each line in the data corresponds to an interval k, which contains the covariate measurements at interval k and the outcome measurement at interval k+1.

Here is an example of input data structure for one subject which contains 7 records on the measurements of three time-varying covariates "L1", "L2", "A", one baseline covariate "L3" and the outcome "Y". See "example_data_basicdata_nocomp" for complete example data.

id	t0	L1	L2	L3	A	Y
1	0	0	1.147092	5	1	0
1	1	0	-0.9254	5	1	0
1	2	0	-0.98998	5	0	0
1	3	1	1.005742	5	1	0
1	4	1	-1.19565	5	1	0
1	5	0	-0.96977	5	1	0
1	6	1	-1.0887	5	1	0

Censoring events. When there are censoring events, and users want to compute nature course estimate via inverse probability weighting, there should be a variable in the input data set that is an indicator of censoring in the time between covariate measurements in interval k and interval k+1. 1 indicates the subject is censored (C_k+1 = 1) and 0 indicates the subject is not censored (C_k+1 = 0). Subjects have no more records after they are censored. Note that the censoring indicator is not needed if users don't want to compute the natural course estimate using IPW.

For survival outcome, the outcome Y_k+1 on the line where individual is censored (C_k+1 = 1) can be coded NA or 0. This choice will make no difference to estimates in the algorithm when intervals are made small enough such that there are no failures in intervals where there are censoring events. It depends on whether to count such subjects in the time k risk set or not[1] [,2]. For fixed binary/continuous end of follow-up, the outcome Y_k+1 should be coded NA.

Here is an example of input data structure with a censoring event (identified by "C"). The subject contains 8 records on the measurements of two time-varying covariates "L", "A", the outcome "Y" and is censored at time index k+1=8. See "example_data_censor" for complete example data.

id	t0	L	A	C	Y
1	0	0	0.169346	0	0
1	1	0	0.238415	0	0
1	2	0	0.22646	0	0
1	3	0	0.320099	0	0
1	4	0	0.330357	0	0
1	5	0	0.558025	0	0
1	6	0	0.676486	0	0
1	7	0	0.601818	1	NA

Competing events. When there are competing events in the data, if the user chooses to treat competing events as censoring events, the data should be structured as censoring case above. If competing events are not treated as censoring events, there should be a variable in the input data set that is an indicator of competing event between interval k and k+1 covariate measurements, where 1 indicates there is a competing event for the subject (D_k+1 = 1) and 0 indicates

[1] McGrath S, Lin V, Zhang Z, Petito LC, Logan RW, Hernán MA, Young JG. gfoRmula: An R Package for Estimating the Effects of Sustained Treatment Strategies via the Parametric g-formula. Patterns (N Y). 2020;1(3):100008. gfoRmula.

[2] Roger W. Logan, Jessica G. Young, Sarah Taubman, Yu-Han Chiu, Sara Lodi, Sally Picciotto, Goodarz Danaei, Miguel A. Hernán. GFORMULA SAS.

no competing event (D_k+1 = 0). If D_k+1 = 1 on a record line k for a given subject, that subject will only have k+1 lines in the follow-up data with follow-up time k on the last line, and on that line, Y_k+1 should be coded NA. Note that the competing case is only applicable for survival outcome.

Here is an example of input data structure with a competing event (identified by "D"). The subject contains 7 records on three time-varying covariates "L1", "L2", "A", one baseline covariate "L3" and the outcome "Y". The subject experiences a competing event after measurement of interval k=6 covariates. See "example_data_basicdata" for complete example data.

id	t0	L1	L2	L3	A	D	Y
1	0	0	0.63418	7	1	0	0
1	1	0	-1.5127	7	0	0	0
1	2	1	0.329688	7	0	0	0
1	3	1	0.462713	7	1	0	0
1	4	0	-1.54026	7	0	0	0
1	5	1	0.414092	7	1	0	0
1	6	0	-1.36756	7	1	1	NA

- Note that the "time_points" argument specifies the desired end of follow-up (a follow-up interval k that is no more than the maximum number of records for an individual in the dataset), and is only applicable for survival outcome.

3.2 Interventions

The package supports natural course intervention, static and dynamic interventions, as well as threshold interventions[1] [2] . It provides pre-coded intervention functions and options for users to define other custom interventions that are beyond the interventions provided.

- **Natural course**: no intervention on any treatment variables.
- **Static intervention**: intervention wherein the rule for assigning treatment at each time point does not depend on past covariates.
- **Dynamic intervention**: intervention wherein the rule for assigning treatment depends on past covariates.
- **Threshold intervention**: intervention wherein the rule for assigning treatment at each time point depends on the natural value of treatment at the time point.

The following are the arguments for specifying the intervention of interest. If not specified, the package will return the result without intervention, i.e., natural course result. This section introduces how to specify different types of intervention with these arguments.

Arguments	Description
int_descript	(Required) A list of strings, each of which describes a user-specified intervention.
interventions	(Required) A dictionary whose key is the treatment name in the intervention with the format Intervention{id}_{treatment_name}, value is a list that contains the intervention function, values required by the function, and a list of time points in which the intervention is applied.

[1] Taubman SL, Robins JM, Mittleman MA, Hernán MA. Intervening on risk factors for coronary heart disease: an application of the parametric g-formula. Int J Epidemiol 2009; 38(6):1599-611.

[2] Young JG, Hernán MA, Robins JM. Identification, estimation and approximation of risk under interventions that depend on the natural value of treatment using observational data. Epidemiologic Methods 2014; 3(1):1-19.

The package uses keyword arguments to implement the intervention and allows any number of interventions. When users specify each intervention, they should specify the intervention id (means the id-th intervention in all interventions, and the id should start from 1) and treatment name in the argument name.

For static interventions, the value of the argument name is a list where the first element is the static intervention function, the second element is the intervened values at all time points, the third element is a list specifying the time points to apply the intervention (if not specified, the default is intervening on all time points);

An example of intervening on "A" in the first three time points with the "Never treat" intervention:

```
Intervention1_A = [static, np.zeros(time_points), [0, 1, 2]]
```

For dynamic interventions, the value of the argument name is a list where the first element is the dynamic intervention function, the second element is a list specifying the time points to apply the intervention (if not specified, the default is intervening on all time points).

An example of intervening on "A" in the first three time points with a dynamic intervention:

```
Intervention1_A = [dynamic, [0, 1, 2]]
```

For threshold interventions, the value of the argument name is a list where the first element is the threshold intervention function, the second element is the threshold values, the third element is a list specifying the time points to apply the intervention.

An example of intervening on "A" in the first three time points with threshold intervention:

```
Intervention1_A = [threshold, [0.5, float('inf')], [0, 1, 2]]
```

If users want to specify multiple interventions, they should use different IDs for different interventions.

An example of intervening on "A" with 3 different interventions:

```
Intervention1_A = [static, np.zeros(time_points)]
Intervention2_A = [dynamic]
Intervention3_A = [threshold, [0.5, float('inf')]]
```

If users want to specify joint intervention where there are multiple treatment variables, they should specify different treatment name with the same intervention id.

An example of intervening on "A1" and "A2" within a static intervention:

```
Intervention1_A1 = [static, np.zeros(time_points)]
Intervention1_A2 = [static, np.ones(time_points)]
```

3.2.1 Natural course intervention

If no intervention is specified, the package will return the result of natural course, containing parametric and nonparametric natural course risk/mean outcome of g-formula. Users may assess model misspecification in the parametric g-formula by comparing the two estimates[3].

Running example [code]:

[3] Yu-Han Chiu, Lan Wen, Sean McGrath, Roger Logan, Issa J Dahabreh, Miguel A Hernán, Evaluating Model Specification When Using the Parametric G-Formula in the Presence of Censoring, American Journal of Epidemiology, Volume 192, Issue 11, November 2023, Pages 1887–1895

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
          time_points = time_points, covnames=covnames, covtypes=covtypes,
          covmodels=covmodels, basecovs=basecovs, outcome_name=outcome_name,
          ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.50396             |    1.00000     |       0.00000       |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

3.2.2 Static interventions

The package has pre-coded static intervention.

static(new_df, pool, int_var, int_values, ...)	This is an internal function to perform a static intervention.

pygformula.parametric_gformula.interventions.**static**(*new_df*, *pool*, *int_var*, *int_values*, *time_name*, *t*)

This is an internal function to perform a static intervention.

Parameters

- **new_df** (*DataFrame*) – A DataFrame that contains the observed or simulated data at time t.
- **pool** (*DataFrame*) – A DataFrame that contains the observed or simulated data up to time t.

- **int_var** (*List*) – A list containing strings of treatment names to be intervened in a particular intervention.
- **int_values** (*List*) – A list containing the value needed when performing a particular intervention function.
- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **t** (*Int*) – An integer indicating the current time index to be intervened.

Return type Nothing is returned, the new_df is changed under a particular intervention.

which can be called by:

```
from pygformula.parametric_gformula.interventions import static
```

When specifying the static intervention for one treatment, the treatment value at each time step k will be replaced by the kth value in the list of intervened values.

Running example of static intervention on one treatment variable [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Always treat']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
               time_points = time_points, covnames=covnames, covtypes=covtypes,
               covmodels=covmodels, basecovs=basecovs, int_descript = int_descript,
               Intervention1_A = [static, np.ones(time_points), [0, 1, 4]],
               outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+--------------------+---------+---------------------------------+----------------+---------------------+
|    Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+--------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.50396             |    1.00000     |       0.00000       |
|    Always treat    |    NA   |             0.41519             |    0.82385     |       -0.08877      |
+--------------------+---------+---------------------------------+----------------+---------------------+
```

Running example of a static intervention on multiple treatment variables [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_multiple_treatments_data

obs_data = load_multiple_treatments_data()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A1', 'A2']
covtypes = ['binary', 'bounded normal', 'binary', 'binary']
covmodels = ['L1 ~ lag1_L1',
             'L2 ~ lag1_L1 + lag1_L2 + lag1_A2 + L1',
             'A1 ~ lag1_L1 + lag1_L2',
             'A2 ~ lag1_A1']

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Always treat on A1 & A2']


outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + A1 + A2'

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
              time_points = time_points, covnames=covnames, covtypes=covtypes,
              covmodels=covmodels, int_descript = int_descript,
              Intervention1_A1 = [static, np.ones(time_points)],
              Intervention1_A2 = [static, np.ones(time_points)],
              outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+------------------------+---------+---------------------------------+----------------+---------------------+
|      Intervention      | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+------------------------+---------+---------------------------------+----------------+---------------------+
|   Natural course(ref)  | 0.67200 |             0.65255             |    1.00000     |       0.00000       |
| Always treat on A1 & A2 |    NA   |             0.28703             |    0.43986     |       -0.36551      |
+------------------------+---------+---------------------------------+----------------+---------------------+
```

Running example of multiple static interventions [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp
```

(continues on next page)

(continued from previous page)

```
obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']


g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
              time_points = time_points, int_descript = int_descript,
              covnames=covnames, covtypes=covtypes,
              covmodels=covmodels, basecovs=basecovs,
              outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type,
              Intervention1_A = [static, np.zeros(time_points)],
              Intervention2_A = [static, np.ones(time_points)])
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.50396             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.73346             |    1.45539     |       0.22950       |
|     Always treat    |    NA   |             0.23297             |    0.46228     |       -0.27099      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

3.2.3 Dynamic interventions

For dynamic intervention, users need to define a dynamic function which encodes the dynamic treatment strategy for one treatment variable and then pass it into the g-formula method by the intervention argument.

Example dynamic intervention: treatment is assgined (A = 1) for individuals where the covariate L2 is above a certain threshold 0.75. Otherwise, the treatment is assigned 0.

Sample syntax of a dynamic function example:

```
def dynamic_intervention(new_df, pool, int_var, time_name, t):
    new_df.loc[new_df[time_name] == t, int_var] = 0
    new_df.loc[new_df['L2'] > 0.75, int_var] = 1
```

(continues on next page)

(continued from previous page)

```
int_descript = ['Dynamic intervention']
Intervention1_A = [dynamic_intervention]

g = ParametricGformula(..., int_descript = int_descript,
                       Intervention1_A = [dynamic_intervention], ...)
```

The dynamic intervention function should contain the following input parameters (these parameters do not all need to be specified in the function). The function should modify the data table "new_df" in place, no output is returned.

- new_df: data table of the simulated data at current time t.
- pool: data table of the simulated data up to current time t.
- int_var: name of the treatment variable to be intervened.
- time_name: name of the time variable.
- t: current time index.

Running example [code]:

```
import pygformula
from pygformula import ParametricGformula
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

time_points = np.max(np.unique(obs_data[time_name])) + 1

def dynamic_intervention(new_df, pool, int_var, time_name, t):
    new_df.loc[new_df[time_name] == t, int_var] = 0
    new_df.loc[new_df['L2'] > 0.75, int_var] = 1

int_descript = ['Dynamic intervention']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
             time_points = time_points, covnames=covnames, covtypes=covtypes,
             covmodels=covmodels, basecovs=basecovs, int_descript = int_descript,
             Intervention1_A = [dynamic_intervention],
             outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+----------------------+---------+----------------------------------+----------------+---------------------+
|     Intervention     | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+----------------------+---------+----------------------------------+----------------+---------------------+
| Natural course(ref)  | 0.50560 |             0.50396              |    1.00000     |       0.00000       |
| Dynamic intervention |    NA   |             0.60878              |    1.20799     |       0.10482       |
+----------------------+---------+----------------------------------+----------------+---------------------+
```

The package also provides two pre-coded dynamic interventions with grace period: natural grace period intervention and uniform grace period intervention. When specifying an intervention with a grace period, the list of the intervention argument should contain the grace period intervention function in the first element, a two-element list with the duration of grace period and conditions in the second element. (If users want to intervene on particular time points, the third element should be specified.)

Natural grace period intervention: once a covariate meets a threshold level, the treatment is initiated within a duration of the grace period. During the grace period, the treatment takes its natural value.

natural_grace_period(new_df, pool, int_var, ...)	This is a pre-coded function to perform a natural grace period intervention.

pygformula.parametric_gformula.interventions.**natural_grace_period**(*new_df*, *pool*, *int_var*, *nperiod*, *conditions*, *time_name*, *t*)

This is a pre-coded function to perform a natural grace period intervention. Once a covariate meets a threshold level, the treatment (int_var) is initiated within m (nperiod) time intervals which is the duration of the grace period. During grace period, the treatment takes its natural value.

Parameters

- **new_df** (*DataFrame*) – A DataFrame that contains the observed or simulated data at time t.
- **pool** (*DataFrame*) – A DataFrame that contains the observed or simulated data up to time t.
- **int_var** (*Str*) – A string specifying the treatment variable to be intervened.
- **nperiod** (*Int*) – An integer indicating the duration of the grace period.
- **conditions** (*Dict*) – A dictionary that contains the covariate and its coditions for initiating the treatment.
- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **t** (*Int*) – An integer indicating the current time index to be intervened.

Return type Nothing is returned, the new_df is changed under a particular intervention.

which can be called by:

```
from pygformula.parametric_gformula.interventions import natural_grace_period
```

Sample syntax of an example:

When the covariate "L1" equals 1, start a treatment initiation in a grace period with duration 3. The "natural_grace_period" specifies the type of the grace period intervention, the two-element list specifies the duration of the grace period in the first entry and the condition of the covariate in the second entry.

```
from pygformula.parametric_gformula.interventions import natural_grace_period

int_descript = ['natural grace period intervention']
```

(continues on next page)

(continued from previous page)

```
g = ParametricGformula(..., int_descript = int_descript,
    Intervention1_A = [natural_grace_period, [3, {'L1': lambda x: x == 1}]], ...)
```

An example of grace period intervention where the treatment is initiated when multiple conditions (the covariate "L1" equals 1, and the covariate "L2" is greater than 2) are met:

```
from pygformula.parametric_gformula.interventions import natural_grace_period

int_descript = ['natural grace period intervention']

g = ParametricGformula(..., int_descript = int_descript,
            Intervention1_A = [natural_grace_period, [natural_grace_period, [3, {'L1':␣
↪lambda x: x == 1, 'L2': lambda x: x >= 2}]], ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import natural_grace_period
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

time_points = np.max(np.unique(obs_data[time_name])) + 1

int_descript = ['natural grace period intervention']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
               time_points = time_points,covnames=covnames, covtypes=covtypes,
               covmodels=covmodels, basecovs=basecovs, int_descript = int_descript,
               Intervention1_A = [natural_grace_period, [3, {'L1': lambda x: x == 1}]],
               outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+----------------------------------+---------+---------------------------------+---------------+---------------------+
|           Intervention           | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+----------------------------------+---------+---------------------------------+---------------+---------------------+
|        Natural course(ref)       | 0.50560 |             0.50396             |    1.00000    |       0.00000       |
| natural grace period intervention |    NA   |             0.37865             |    0.75135    |       -0.12531      |
+----------------------------------+---------+---------------------------------+---------------+---------------------+
```

Uniform grace period intervention: once a covariate meets a threshold level, the treatment is initiated within a duration of the grace period. During grace period, treatment initiation is randomly allocated with a uniform probability of starting treatment in each time interval of the grace period.

uniform_grace_period(new_df, pool, int_var, ...)	This is a pre-coded function to perform a uniform grace period intervention.

pygformula.parametric_gformula.interventions.**uniform_grace_period**(*new_df*, *pool*, *int_var*, *nperiod*, *conditions*, *time_name*, *t*)

This is a pre-coded function to perform a uniform grace period intervention. Once a covariate meets a threshold level, the treatment (int_var) is initiated within m (nperiod) time intervals which is the duration of the grace period. During grace period, treatment initiation is randomly allocated with a uniform probability of starting treatment in each time interval of the grace period.

Parameters

- **new_df** (*DataFrame*) – A DataFrame that contains the observed or simulated data at time t.
- **pool** (*DataFrame*) – A DataFrame that contains the observed or simulated data up to time t.
- **int_var** (*Str*) – A string specifying the treatment variable to be intervened.
- **nperiod** (*Int*) – An integer indicating the duration of the grace period.
- **conditions** (*Dict*) – A dictionary that contains the covariate and its coditions for initiating the treatment.
- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **t** (*Int*) – An integer indicating the current time index to be intervened.

Return type Nothing is returned, the new_df is changed under a particular intervention.

which can be called by:

```
from pygformula.parametric_gformula.interventions import uniform_grace_period
```

Sample syntax of an example:

When the covariate "L1" equals 1, start a treatment initiation in a grace period with duration 3. The "uniform_grace_period" specifies the type of the grace period intervention, the two-element list specifies the duration of the grace period in the first entry and the condition of the covariate in the second entry.

```
from pygformula.parametric_gformula.interventions import uniform_grace_period

int_descript = ['uniform grace period intervention']

g = ParametricGformula(..., int_descript = int_descript,
    Intervention1_A = [uniform_grace_period, [3, {'L1': lambda x: x == 1}]], ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import uniform_grace_period
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

time_points = np.max(np.unique(obs_data[time_name])) + 1

int_descript = ['uniform grace period intervention']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
            time_points = time_points, covnames=covnames, covtypes=covtypes,
            covmodels=covmodels, basecovs=basecovs,int_descript = int_descript,
            Intervention1_A = [uniform_grace_period, [3, {'L1': lambda x: x == 1}]],
            outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+---------------------------------+---------+---------------------------------+----------------+---------------------+
|           Intervention          | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------------------+---------+---------------------------------+----------------+---------------------+
|       Natural course(ref)       | 0.50560 |             0.50396             |    1.00000     |       0.00000       |
| uniform grace period intervention |    NA   |             0.45311             |    0.89911     |       -0.05084      |
+---------------------------------+---------+---------------------------------+----------------+---------------------+
```

3.2.4 Threshold interventions

The threshold interventions in the package implement interventions that depend on the natural value of treatment. In a threshold intervention, if a subject's natural value of treatment at time k is below/above a particular threshold, then set treatment to this threshold value. Otherwise, do not intervene on this subject at k. The natural value of treatment at time k is the value of treatment that would have been observed at time k were the intervention discontinued right before k.

The package provides pre-coded threshold function.

threshold(new_df, pool, int_var, ...)	This is an internal function to perform a threshold intervention.

pygformula.parametric_gformula.interventions.**threshold**(*new_df*, *pool*, *int_var*, *threshold_values*, *time_name*, *t*)

This is an internal function to perform a threshold intervention.

Parameters

- **new_df** (*DataFrame*) – A DataFrame that contains the observed or simulated data at time t.
- **pool** (*DataFrame*) – A DataFrame that contains the observed or simulated data up to time t.
- **int_var** (*List*) – A list containing strings of treatment names to be intervened in a particular intervention.
- **threshold_values** (*List*) – A list containing the threshold values needed when performing a threshold intervention function.
- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **t** (*Int*) – An integer indicating the current time index to be intervened.

Return type Nothing is returned, the new_df is changed under a particular intervention.

which can be called by

```
from pygformula.parametric_gformula.interventions import threshold
```

Users should specify a two-element list (containing minimum and maximum values) of threshold values after the threshold function in the argument.

Example threshold intervention: if the subject's natural value of treatment L2 falls outside the interval [0.5, inf], set the treatment the threshold value.

Sample syntax of example threshold intervention:

```
int_descript = ['Threshold intervention']

g = ParametricGformula(..., int_descript = int_descript,
    Intervention1_A = [threshold, [0.5, float('inf')]], ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import threshold
from pygformula.data import load_threshold_data

obs_data = load_threshold_data()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'normal']
covmodels = ['L1 ~ lag1_L1',
             'L2 ~ lag1_L1 + lag1_L2 + L1',
             'A ~ L1 + L2']
```

(continues on next page)

(continued from previous page)

```
time_points = np.max(np.unique(obs_data[time_name])) + 1

int_descript = ['Threshold intervention']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + A'

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
              time_points = time_points, covnames=covnames, covtypes=covtypes,
              covmodels=covmodels, int_descript = int_descript,
              Intervention1_A = [threshold, [0.5, float('inf')]],
              outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+------------------------+---------+---------------------------------+----------------+---------------------+
|      Intervention      | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+------------------------+---------+---------------------------------+----------------+---------------------+
|  Natural course(ref)   | 0.98900 |             0.98359             |    1.00000     |       0.00000       |
| Threshold intervention |    NA   |             0.32806             |    0.33353     |       -0.65553      |
+------------------------+---------+---------------------------------+----------------+---------------------+
```

3.3 Covariate models

To model the joint densities of covariates in g-formula, the conditional densities of each covariate given past covariate history are estimated. Users can specify the covariate histories by the pre-coded functions of histories or custom histories. The package provides options of modeling different covariate distributions , which contains "binary", "normal", "categorical", "bounded normal", "zero-inflated normal", "truncated normal", "absorbing", "categorical time" and "square time". Once the covariate model and covariate distribution are specified, the pygformula will fit a pooled (over time) parametric model for each covariate. The custom covariate type is also allowed if users have their own covariate distribution, which should be set to "custom" in corresponding covariate types.

3.3.1 Functions of covariate histories

Pre-coded histories

The package provides three pre-coded functions ("lag", "cumavg", "lag_cumavg") for users to specify the covariate histories.

- "lag": For any covariate L, specifying lagi_L will add a variable to the input dataset, which contains the i-th lag of L relative to the current follow-up time k. For example, lag1_L means the value of L at the time k-1, lag2_L means the value of L at the time k-2 etc. The value is set to 0 if k < i if there is no pre-baseline times.
- "cumavg": For any covariate L, specifying cumavg_L will add a variable to the input dataset, which contains the cumulative average of L up until the current follow-up time k.
- "lag_cumavg": For any covariate L, specifying lag_cumavgi_L will add a variable to the input dataset, which contains the i-th lag of the cumulative average of L relative to the current follow-up time k. The value is set to 0 if k < i if there is no pre-baseline times.

An example for specifying the covariate model for covariate L1 based on the history functions:

```
L1 ~ lag1_L1 + cumavg_L1 + lag_cumavg1_L1 + L3 + t0
```

Note: for more covariate transformations (e.g., polynomial terms, spline terms), see patsy for specification.

Custom histories

If users wish to use history functions that are not in the three pre-coded history functions, the package provides "custom_histvars" and "custom_histories" for users to specify their own history functions for corresponding covariates.

Arguments	Description
custom_histvars	(Optional) A list of strings, each of which specifies the names of the time-varying covariates with user-specified custom histories.
custom_histories	(Optional) A list of functions, each function is the user-specified custom history functions for covariates. The list should be the same length as custom_histvars and in the same order.

For each custom history function, the input should be the parameters (not necessary to use all):

- pool: A DataFrame that contains the observed or simulated data up to time t.
- histvar: A string that specifies the name of the variable for which this history function is to be applied.
- time_name: A string specifying the name of the time variable in pool.
- t: An integer specifying the current time index.
- id: A string specifying the name of the ID variable in the pool.

The function output is a dataframe "pool" with the new column of the historical term created.

The following is an example of creating historical functions for covariates ''L1", ''L2" and ''A" by the function 'ave_last3'. This function generates the average of the three most recent values of the covariate (when the t=0, it takes the current value at time 0; when t=1, it takes the average of the covariate values at times 0 and 1). The new historical covariates are named as ave_last3_L1, ave_last3_L2, and ave_last3_A.

Sample syntax:

```
def ave_last3(pool, histvar, time_name, t, id):

  def avg_func(df, time_name, t, histvar):
      if t < 3:
          avg_values = np.mean((df[(df[time_name] >= 0) & (df[time_name] <=␣
↪t)][histvar]))
      else:
          avg_values = np.mean((df[(df[time_name] > t - 3) & (df[time_name] <=␣
↪t)][histvar]))
      return avg_values

  valid_pool = pool.groupby(id_name).filter(lambda x: max(x[time_name]) >= t)
  pool.loc[pool[time_name] == t, '_'.join(['ave_last3', str(histvar)])] = list(valid_
↪pool.groupby(id_name).apply(
      avg_func, time_name=time_name, t=t, histvar=histvar))

return pool

custom_histvars = ['L1', 'L2', 'A']
```

(continues on next page)

(continued from previous page)

```
custom_histories = [ave_last3, ave_last3, ave_last3]

g = ParametricGformula(..., custom_histvars = custom_histvars, custom_histories=custom_
↪histories, ...)
```

The ave_last3 function has been included in the package, users can also call this function by:

```
from pygformula.parametric_gformula.histories import ave_last3
```

3.3.2 Different covariate distributions

To specify a parametric model for each covariate in pygformula, users need to specify the following arguments:

Arguments	Description
covnames	(Required) A list of strings specifying the names of the time-varying covariates in obs_data.
covtypes	(Required) A list of strings specifying the "type" of each time-varying covariate included in covnames. The supported types: "binary", "normal", "categorical", "bounded normal", "zero-inflated normal", "truncated normal", "absorbing", "categorical time", "square time" and "custom". The list must be the same length as covnames and in the same order.
covmodels	(Required) A list of strings, where each string is the model statement of the time-varying covariate. The list must be the same length as covnames and in the same order. If a model is not required for a certain covariate, it should be set to 'NA' at that index.
basecovs	(Optional) A list of strings specifying the names of baseline covariates in obs_data. These covariates should not be included in covnames.
trunc_params	(Optional) A list, each element could be 'NA' or a two-element list. If not 'NA', the first element specifies the truncated value and the second element specifies the truncated direction ('left' or 'right'). The non-NA value is set for the truncated normal covariates. The 'NA' value is set for other covariates. The list should be the same length as covnames and in the same order.
time_thresholds	(Optional) A list of integers that splits the time points into different intervals. It is used to create the variable "categorical time".

Users need to specify the names of time-varying covariates in "covnames", the distribution type of each covariate in "covtypes", as well as the model statement for each covariate in "covmodels". In addition, if there are time-fixed baseline covariates, they should be specified in the argument "basecovs". If the covariate type is "truncated normal", the "trunc_params" argument should be also specified which contains the required truncated direction and truncated value. If the covariate type is "categorical time", users should also define the "time_thresholds" argument to create a desired categorization of time. In the following, this section shows examples for different covariate distributions to show how to specify the above arguments in specific examples.

Note: For the "covmodels" argument which specifies the model statement for each covariate, users need to be careful to avoid the loop between covariates, i.e., in each covariate model statement, the independent variable and dependent variable should follow the direction in the DAG (directed acyclic graph). For example, if the covariate A is the dependent variable of the covariate B (A ~ B), then the covariate B should not be the dependent variable of the covariate A (B ~ A).

Binary

When the covariate is binary, in the fitting step, the input data is used to estimate a generalized linear model where the family function is binomial. Then, in the simulation step, the mean of the covariate conditional on history at each time step is estimated via the coefficients of the fitted model, the covariate values are simulated by sampling from a Bernoulli distribution with parameter the conditional probability.

Sample syntax:

An example where the covariate ''L1" is binomial distribution

```
covnames = [ 'L1', 'A']
covtypes = ['binary', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, basecovs = basecovs, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'A']
covtypes = ['binary', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + L3 + t0',
            'A ~ lag1_A + L1 + lag_cumavg1_L1 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + A + lag1_A + lag1_L1 + L3 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
              time_points = time_points, int_descript = int_descript,
              Intervention1_A = [static, np.zeros(time_points)],
              Intervention2_A = [static, np.ones(time_points)],
              covnames=covnames, covtypes=covtypes,
              covmodels=covmodels, basecovs=basecovs,
              outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+---------------------+---------+----------------------------------+----------------+----------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+----------------------------------+----------------+----------------------+
| Natural course(ref) | 0.50560 |             0.50689              |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.72275              |    1.42587     |       0.21587       |
|     Always treat    |    NA   |             0.23587              |    0.46533     |       -0.27102      |
+---------------------+---------+----------------------------------+----------------+----------------------+
```

Note that in this section, all demonstration examples use the same static interventions ("Never treat" and "Always treat"), and are applied in the survival outcome case. Please refer to *Interventions* for more types of interventions, and *Outcome model* for more types of outcomes.

Normal

When the covariate is normal, in the fitting step, the input data is used to estimate a generalized linear model where the family function is gaussian. Then, in the simulation step, the mean of the covariate conditional on history at each time step is estimated via the coefficients of the fitted model, the covariate values are simulated by sampling from a normal distribution with mean this conditional mean and variance the residual mean squared error from the fitted model. Values generated outside the observed range for the covariate are set to the minimum or maximum of this range.

Sample syntax:

```
covnames = ['L2', 'A']
covtypes = ['normal', 'binary']
covmodels = ['L2 ~ lag1_A + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L2 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, basecovs = basecovs, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L2', 'A']
covtypes = ['normal', 'binary']
covmodels = ['L2 ~ lag1_A + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L2 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L2 + A + lag1_A + L3 + t0'
```

(continues on next page)

(continued from previous page)

```
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
          time_points = time_points, int_descript = int_descript,
          Intervention1_A = [static, np.zeros(time_points)],
          Intervention2_A = [static, np.ones(time_points)],
          covnames=covnames, covtypes=covtypes,
          covmodels=covmodels, basecovs=basecovs,
          outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+--------------------+---------+---------------------------------+---------------+---------------------+
|    Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+--------------------+---------+---------------------------------+---------------+---------------------+
| Natural course(ref) | 0.50560 |            0.51103              |    1.00000    |       0.00000       |
|    Never treat     |    NA   |            0.75901              |    1.48525    |       0.24798       |
|    Always treat    |    NA   |            0.22726              |    0.44472    |       -0.28377      |
+--------------------+---------+---------------------------------+---------------+---------------------+
```

Categorical

When the covariate is categorical, in the fitting step, the input data is used to estimate a multinomial logistic regression model. Then, in the simulation step, the probability that a covariate takes a particular level conditional on history is estimated via the coefficients of the fitted model. The covariate values are simulated at each time step by sampling from a multinoulli distribution with parameters these estimated conditional probabilities of the fitted model.

Sample syntax:

```
covnames = [ 'L', 'A']
covtypes = ['categorical', 'binary']
covmodels = [ 'L ~ C(lag1_L) + t0',
              'A ~ C(L) + C(lag1_L) + t0']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels,...)
```

Note that when the covariate model statement contains any categorical variable, such as "lag1_L" or "L", e.g., in the example above, users need to add a "C( )" on the variable indicating it's categorical.

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_categorical

obs_data = load_categorical()
time_name = 't0'
```

(continues on next page)

(continued from previous page)

```
id = 'id'

covnames = [ 'L', 'A']
covtypes = ['categorical', 'binary']
covmodels = [ 'L ~ C(lag1_L) + t0',
              'A ~ C(L) + C(lag1_L) + t0']

outcome_name = 'Y'
ymodel = 'Y ~ C(lag1_L) + A'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
            time_points = time_points,int_descript = int_descript,
            Intervention1_A = [static, np.zeros(time_points)],
            Intervention2_A = [static, np.ones(time_points)],
            covnames=covnames,  covtypes=covtypes,
            covmodels=covmodels, outcome_name=outcome_name,
            ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.43900 |             0.43242             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.38268             |    0.88497     |       -0.04974      |
|     Always treat    |    NA   |             0.45371             |    1.04922     |       0.02128       |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

Bounded normal

When the covariate is bounded normal, the observed covariate values are first standardized to the interval [0, 1], inclusive, by subtracting the minimum value and dividing by the range. In the fitting step, the input data with standardized covariate is used to estimate a generalized linear model where the family function is gaussian. In the simulation step, the mean of the covariate conditional on history at each time step is estimated via the coefficients of the fitted model, the standardized covariate values are simulated by sampling from a normal distribution with mean this conditional mean and variance the residual mean squared error from the fitted model. Finally, the simulated standardized values are then transformed back to the original scale, and values generated outside the observed range for the covariate are set to the minimum or maximum of this range.

Sample syntax:

```
covnames = ['L2', 'A']
covtypes = ['bounded normal', 'binary']
covmodels = ['L2 ~ lag1_A + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L2 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, basecovs = basecovs, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L2', 'A']
covtypes = ['bounded normal', 'binary']
covmodels = ['L2 ~ lag1_A + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L2 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L2 + A + lag1_A + L3 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
          time_points = time_points, int_descript = int_descript, intcomp=[1, 2],
          Intervention1_A = [static, np.zeros(time_points)],
          Intervention2_A = [static, np.ones(time_points)],
          covnames=covnames, covtypes=covtypes,
          covmodels=covmodels, basecovs=basecovs,
          outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.51103             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.75901             |    1.48525     |       0.24798       |
|     Always treat    |    NA   |             0.22726             |    0.44472     |       -0.28377      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

Zero-inflated normal

When the covariate is zero-inflated normal, in the fitting step, the input data will be added an indicator variable by setting the covariate values that are greater than 0 to 1 and keeping the original zeros values. The input data with the added indicator variable is used to first fit a generalized linear model where the family function is binomial. Then, the input data with positive values at the covariate is used to fit a generalized linear model where the family function is gaussian. In the simulation step, the simulated covariate values are created by first generating an indicator of whether the covariate value is zero or non-zero from a Bernoulli distribution with the parameter the probability from the first fitted model. Covariate values are then generated from a normal distribution with the mean of the second fitted model

and multiplied by the generated zero indicator. The simulated non-zero covariate values that fall outside the observed range are set to the minimum or maximum of the range of non-zero observed values of the covariate.

Sample syntax:

```
covnames = ['L', 'A']
covtypes = ['zero-inflated normal', 'binary']
covmodels = ['L ~ lag1_L + lag1_A + t0',
             'A ~ lag1_A + L + t0']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_zero_inflated_normal

obs_data = load_zero_inflated_normal()
time_name = 't0'
id = 'id'

covnames = ['L', 'A']
covtypes = ['zero-inflated normal', 'binary']
covmodels = ['L ~ lag1_L + lag1_A + t0',
             'A ~ lag1_A + L + t0']

outcome_name = 'Y'
ymodel = 'Y ~ L + A + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
             time_points = time_points, int_descript = int_descript,
             Intervention1_A = [static, np.zeros(time_points)],
             Intervention2_A = [static, np.ones(time_points)],
             covnames=covnames, covtypes=covtypes, covmodels=covmodels,
             outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+--------------------+---------+---------------------------------+---------------+--------------------+
|    Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+--------------------+---------+---------------------------------+---------------+--------------------+
| Natural course(ref) | 0.56000 |             0.55942             |    1.00000    |       0.00000       |
|    Never treat     |    NA   |             0.67914             |    1.21401    |       0.11972       |
|    Always treat    |    NA   |             0.20932             |    0.37417    |       -0.35010      |
+--------------------+---------+---------------------------------+---------------+--------------------+
```

Truncated normal

When the covariate is truncated normal, in the fitting step, the input data is used to fit a truncated normal regression model. In the simulation step, the mean of the covariate conditional on history at each time step is estimated via the coefficients of the fitted model, then the simulated covariate values are generated by sampling from a truncated normal distribution with the covariate mean from the fitted model. The generated covariate values that fall outside the observed range are set to the minimum or maximum of the observed range.

Sample syntax:

```
covnames = ['L', 'A']
covtypes = ['truncated normal', 'binary']
covmodels = ['L ~ lag1_A + lag1_L + t0',
             'A ~ lag1_A + lag1_L + L + t0']

trunc_params = [[1, 'right'], 'NA']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, trunc_params=trunc_params, ...)
```

The package supports covariates with one-sided truncation. To specify the covariates, the elements in the "trunc_params" list should follow the same order as "covnames", in the position where its corresponding covariate is truncated normal, it should be a list with two elements, otherwise it should be 'NA'. In the list of two elements, the first one should be the truncated value of the covariate, and the second one should be the truncated direction ('left' or 'right') of the covariate.

Running example [code]:

```
import numpy as np
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_truncated_normal

obs_data = load_truncated_normal()
time_name = 't0'
id = 'id'

covnames = ['L', 'A']
covtypes = ['truncated normal', 'binary']
covmodels = ['L ~ lag1_A + lag1_L + t0',
             'A ~ lag1_A + lag1_L + L + t0']

trunc_params = [[1, 'right'], 'NA']

outcome_name = 'Y'
ymodel = 'Y ~ L + A + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
             time_points = time_points, int_descript = int_descript,
             Intervention1_A = [static, np.zeros(time_points)],
             Intervention2_A = [static, np.ones(time_points)],
```

(continues on next page)

(continued from previous page)

```
            covnames=covnames, covtypes=covtypes, covmodels=covmodels,
            trunc_params=trunc_params, outcome_name=outcome_name,
            ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+--------------------+---------+---------------------------------+----------------+---------------------+
|    Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+--------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.43600 |             0.44522             |    1.00000     |       0.00000       |
|     Never treat    |    NA   |             0.24447             |    0.54910     |       -0.20075      |
|     Always treat   |    NA   |             0.49355             |    1.10856     |       0.04833       |
+--------------------+---------+---------------------------------+----------------+---------------------+
```

Absorbing

Absorbing means that once the covariate value switches to 1 at one time step, it stays 1 at all subsequent times. When the covariate is absorbing, the input data records where the value of the covariate at k-1 is 0 for all time steps k is used to fit a generalized linear model where the family function is binomial. Then in the simulation step, the mean of the covariate conditional on history at each time step is estimated via the coefficients of the fitted model, the covariate values are simulated by sampling from a Bernoulli distribution with parameter the conditional mean. Once a 1 is first generated, the covariate value at that time and all subsequent times is set to 1.

Sample syntax:

```
covnames = ['L', 'A']
covtypes = ['absorbing', 'binary']
covmodels = ['L ~ lag1_L + lag1_A + t0',
             'A ~ lag1_A + L + t0']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covparams =␣
↪covparams,...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_absorbing_data

obs_data = load_absorbing_data()
time_name = 't0'
id = 'id'

covnames = ['L', 'A']
covtypes = ['absorbing', 'binary']
covmodels = ['L ~ lag1_L + lag1_A + t0',
             'A ~ lag1_A + L + t0']

outcome_name = 'Y'
ymodel = 'Y ~ L + A + t0'
outcome_type = 'survival'
```

(continues on next page)

(continued from previous page)

```
time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
             time_points = time_points, int_descript = int_descript,
             covnames=covnames, covtypes=covtypes, covmodels=covmodels,
             Intervention1_A = [static, np.zeros(time_points)],
             Intervention2_A = [static, np.ones(time_points)],
             outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.59400 |             0.59268             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.60833             |    1.02639     |       0.01564       |
|     Always treat    |    NA   |             0.38695             |    0.65288     |       -0.20573      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

Time variable

When users assume that the distributions of time-varying covariates depend on a function of the time index, they need to specify an additional time variable. The package has two pre-coded time variable: "categorical time" and "square time". The "categorical time" is a variable that categorizes the time index to different time categories. The "square time" is the squared time index.

Sample syntax of categorical time:

```
covnames = ['L1', 'L2', 'A', 't0_f']
covtypes = ['binary', 'bounded normal', 'binary', 'categorical time']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + C(t0_f)
↪',
            'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + C(t0_f)',
            'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + C(t0_f)',
            'NA']

time_thresholds = [1, 3, 5]

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, time_thresholds = time_thresholds, ...)
```

The argument "time_thresholds" creates indicators for categorizing the time index, The time index values inside the interval from each adjacent array (right-closed) form a category. for example, setting time_thresholds = [1, 3, 5] in input data with 7 time points means that categorizing the time index to four time categories (category 1: 0 <= time index <=1, category 2 : 1 < time index <=3, category 3 : 3 < time index <=5, category 4 : 5 < time index <= 6).

Users should specify the name of categorical time variable by adding a "_f" after the time name, and specify its type as 'categorical time' in covtypes argument. In the covmodels argument, the corresponding position of the categorical time variable should be padded with NA. Note that when using the new categorical time variable in the model statement, e.g., "t0_f" in the syntax example above, C() should be added.

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A', 't0_f']
covtypes = ['binary', 'bounded normal', 'binary', 'categorical time']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + C(t0_f)
↪',
            'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + C(t0_f)',
            'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + C(t0_f)',
            'NA']

time_thresholds = [1, 3, 5]

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points, time_thresholds = time_thresholds,
    int_descript = int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.50406             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.73377             |    1.45571     |       0.22971       |
|     Always treat    |    NA   |             0.23283             |    0.46190     |       -0.27124      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

Sample syntax of square time:

Note that when the covariate type is "square time", the corresponding covariate name should be set to the merged strings of 'square' and the time name in the data, e.g., 'square_t0'.

```
covnames = ['L1', 'L2', 'A', 'square_t0']
```

(continues on next page)

(continued from previous page)

```
covtypes = ['binary', 'bounded normal', 'binary', 'square time']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + square_
↪t0',
            'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + square_t0',
            'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + square_t0
↪',
            'NA']

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A', 'square_t0']
covtypes = ['binary', 'bounded normal', 'binary', 'square time']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + square_
↪t0',
            'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + square_t0',
            'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0 + square_t0
↪',
            'NA']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0 + square_t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points, int_descript = int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Custom

In addition to the covariate types above, the package also allows users to choose their own covariate distributions for estimation. In this case, the corresponding covtype should be set to "custom", users need to specify the custom fit function and the predict function, which can be specified by the arguments "covfits_custom" and "covpredict_custom".

Arguments	Description
covfits_custom	(Optional) A list, each element could be 'NA' or a user-specified fit function. The non-NA value is set for the covariates with custom type. The 'NA' value is set for other covariates. The list must be the same length as covnames and in the same order.
covpredict_custom	(Optional) A list, each element could be 'NA' or a user-specified predict function. The non-NA value is set for the covariates with custom type. The 'NA' value is set for other covariates. The list must be the same length as covnames and in the same order.

Each custom fit function has input parameters (not necessary to use all):

- covmodel: model statement of the covariate
- covname: the covariate name
- fit_data: data used to fit the covariate model

and return a fitted model which is used to make prediction in the simulation step.

An example using random forest to fit a covariate model:

```
def fit_rf(covmodel, covname, fit_data):
    max_depth = 2
    y_name, x_name = re.split('~', covmodel.replace(' ', ''))
    x_name = re.split('\+', x_name.replace(' ', ''))
    y = fit_data[y_name].to_numpy()
    X = fit_data[x_name].to_numpy()
    fit_rf = RandomForestRegressor(max_depth=max_depth, random_state=0)
    fit_rf.fit(X, y)
    return fit_rf
```

Each custom predict function has parameters (not necessary to use all):

- covmodel: model statement of the covariate
- new_df: simulated data at time t.
- fit: fitted model of the custom function

and return a list of predicted values at time t.

The custom predict function for the random forest model:

```
def predict_rf(covmodel, new_df, fit):
    y_name, x_name = re.split('~', covmodel.replace(' ', ''))
    x_name = re.split('\+', x_name.replace(' ', ''))
    X = new_df[x_name].to_numpy()
    prediction = fit.predict(X)
    return prediction
```

Sample syntax:

```
covfits_custom = ['NA', fit_rf, 'NA']
covpredict_custom = ['NA', predict_rf, 'NA']

g = ParametricGformula(..., covfits_custom = covfits_custom, covfits_custom = covpredict_
↪custom, ...)
```

Running examples [code]:

```
import numpy as np
import re
from sklearn.ensemble import RandomForestRegressor
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()

time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'custom', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag1_L1 + lag_cumavg1_L2 + t0',
             'L2 ~ lag1_A + L1 + lag1_L1 + lag_cumavg1_L2 + t0',
             'A ~ lag1_A + L1 + L2 +lag1_L1 + lag_cumavg1_L2 + t0']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + A'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

def fit_rf(covmodel, covname, fit_data):
    max_depth = 2
    y_name, x_name = re.split('~', covmodel.replace(' ', ''))
    x_name = re.split('\+', x_name.replace(' ', ''))
    y = fit_data[y_name].to_numpy()
    X = fit_data[x_name].to_numpy()
    fit_rf = RandomForestRegressor(max_depth=max_depth, random_state=0)
    fit_rf.fit(X, y)
    return fit_rf

def predict_rf(covmodel, new_df, fit):
    y_name, x_name = re.split('~', covmodel.replace(' ', ''))
    x_name = re.split('\+', x_name.replace(' ', ''))
    X = new_df[x_name].to_numpy()
    prediction = fit.predict(X)
    return prediction

covfits_custom = ['NA', fit_rf, 'NA']
covpredict_custom = ['NA', predict_rf, 'NA']
```

(continues on next page)

(continued from previous page)

```
g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
             time_points = time_points, int_descript = int_descript,
             Intervention1_A = [static, np.zeros(time_points)],
             Intervention2_A = [static, np.ones(time_points)],
             covnames=covnames,  covtypes=covtypes, covmodels=covmodels,
             covfits_custom = covfits_custom, covpredict_custom=covpredict_custom,
             outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.50961             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.73589             |    1.44401     |       0.22628       |
|     Always treat    |    NA   |             0.23615             |    0.46339     |       -0.27346      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

3.4 Outcome model

The package supports g-formula analysis on three types of outcomes: survival outcomes, fixed binary end of follow-up outcomes and continuous end of follow-up outcomes.

For all types of outcomes, users should specify the name of outcome in the argument "outcome_name", and the model statement for outcome variable in the argument "ymodel". If users are interested in the probability of failing of an event by a specified follow-up time k under different interventions, they need to specify the type of outcome as 'survival' in the argument "outcome_type". If users are interested in the outcome mean at a fixed time point, and the outcome distribution is binary, they need to specify the type of outcome as 'binary_eof'. Similarly, they need to specify the type of outcome as 'continuous_eof' when the distribution of the outcome is continuous.

Arguments	Description
outcome_name	(Required) A string specifying the name of the outcome variable in obs_data.
ymodel	(Required) A string specifying the model statement for the outcome variable.
outcome_type	(Required) A string specifying the "type" of outcome. The possible "types" are: "survival", "continuous_eof", and "binary_eof".

3.4.1 Survival outcome

For survival outcomes, the package will output estimates of contrasts in failure risks by a specified follow-up time k under different user-specified interventions.

Sample syntax:

```
outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'
time_points = 5

g = ParametricGformula(..., outcome_name = outcome_name, outcome_type = outcome_type,␣
↪ymodel = ymodel, time_points = time_points, ...)
```

Users can also specify the follow-up time of interest for survival outcome by the argument "time_points".

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
outcome_model = 'Y ~ L1 + L2 + L3 + A + lag1_A + lag1_L1 + lag1_L2 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points, int_descript = int_descript,
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)])
g.fit()
```

Output:

```
+--------------------+---------+---------------------------------+----------------+--------------------+
|    Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+--------------------+---------+---------------------------------+----------------+--------------------+
| Natural course(ref) | 0.50560 |             0.50396             |    1.00000     |       0.00000       |
|     Never treat    |    NA   |             0.73346             |    1.45539     |       0.22950       |
|     Always treat   |    NA   |             0.23297             |    0.46228     |       -0.27099      |
+--------------------+---------+---------------------------------+----------------+--------------------+
```

3.4.2 Binary end of follow-up outcome

For binary end of follow-up outcomes, the package will output estimates of contrasts in the outcome probability under different user-specified treatment strategies.

Sample syntax:

```
outcome_name = 'Y'
ymodel = 'Y ~ L1 + A + lag1_A + lag1_L1 + L3 + t0'
outcome_type = 'binary_eof'

g = ParametricGformula(..., outcome_name = outcome_name, outcome_type = outcome_type,␣
↪ymodel = ymodel, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import threshold
from pygformula.data import load_binary_eof

obs_data = load_binary_eof()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'zero-inflated normal', 'normal']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L1 + L2 + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L1 + A + lag1_A + lag1_L1 + L3 + t0'
outcome_type = 'binary_eof'

int_descript = ['Threshold intervention']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    int_descript = int_descript,
    Intervention1_A = [threshold, [0.5, float('inf')]],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+------------------------+---------+---------------------------------+----------------+--------------------+
|      Intervention      | NP mean | g-formula mean (NICE-parametric) | Mean Ratio(MR) | Mean Difference(MD) |
+------------------------+---------+---------------------------------+----------------+--------------------+
|  Natural course(ref)   | 0.09880 |             0.09731             |    1.00000     |       0.00000      |
| Threshold intervention |    NA   |             0.06183             |    0.63545     |       -0.03547     |
+------------------------+---------+---------------------------------+----------------+--------------------+
```

3.4.3 Continuous end of follow-up outcome

For continuous end of follow-up outcomes, the package will output estimates of contrasts in the outcome mean under different user-specified treatment strategies.

Sample syntax:

```
outcome_name = 'Y'
ymodel = 'Y ~ C(L1) + L2 + A'
outcome_type = 'continuous_eof'

g = ParametricGformula(..., outcome_name = outcome_name, outcome_type = outcome_type,␣
↪ymodel = ymodel, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_continuous_eof

obs_data = load_continuous_eof()
time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['categorical', 'normal', 'binary']
covmodels = ['L1 ~ C(lag1_L1) + lag1_L2 + t0',
             'L2 ~ lag1_L2 + C(lag1_L1) + lag1_A + t0',
              'A ~ C(L1) + L2 + t0']

basecovs = ['L3']

outcome_name = 'Y'
outcome_model = 'Y ~ C(L1) + L2 + A'
outcome_type = 'continuous_eof'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    int_descript=int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+--------------------+----------+---------------------------------+----------------+---------------------+
|     Intervention   | NP mean  | g-formula mean (NICE-parametric) | Mean Ratio(MR) | Mean Difference(MD) |
+--------------------+----------+---------------------------------+----------------+---------------------+
| Natural course(ref) | -4.41454 |            -4.38216             |    1.00000     |       0.00000       |
|     Never treat    |    NA    |            -3.38995             |    0.77358     |       0.99221       |
|     Always treat   |    NA    |            -4.60302             |    1.05040     |       -0.22086      |
+--------------------+----------+---------------------------------+----------------+---------------------+
```

3.5 Censoring event

When there are censoring events, the package provides the option to obtain inverse probability weighted (IPW) estimates for comparison with the g-formula estimates. The comparison of these two estimates can be useful to assess model misspecification of the g-formula[1]. To get the IPW estimate, the name of the censoring variable in the input data should be specified, users also need to specify a censor model to obtain the weights.

Note that the arguments "censor_name" and "censor_model" are only needed when users want to get the IPW estimate. The package will return the nonparametric observed risk in general cases.

The arguments for censoring events:

Arguments	Description
censor_name	(Optional) A string specifying the name of the censoring variable in obs_data. Only applicable when using inverse probability weights to estimate the natural course means / risk from the observed data.
censor_model	(Optional) A string specifying the model statement for the censoring variable. Only applicable when using inverse probability weights to estimate the natural course means / risk from the observed data.
ipw_cutoff_quantile	(Optional) Percentile value for truncation of the inverse probability weights.
ipw_cutoff_value	(Optional) Absolute value for truncation of the inverse probability weights.

Users can also specify a percentile value (in the argument "ipw_cutoff_quantile") or an absolute value (in the argument "ipw_cutoff_value") to truncate inverse probability weight.

Sample syntax:

```
censor_name = 'C'
censor_model = 'C ~ A + L'

g = ParametricGformula(..., censor_name = censor_name, censor_model = censor_model, ...)
```

Note: When there are categorical covariates (which are assigned a 'C' symbol) in the model statement of censoring variable, please name the censoring variable any name except 'C' to avoild name confusion.

Running example [code]:

```
import numpy as np
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_censor_data
```

(continues on next page)

[1] Yu-Han Chiu, Lan Wen, Sean McGrath, Roger Logan, Issa J Dahabreh, and Miguel A Hernán. 2022. Evaluating model specification when using the parametric g-formula in the presence of censoring. American Journal of Epidemiology.

(continued from previous page)

```
obs_data = load_censor_data()
time_name = 't0'
id = 'id'

covnames = ['L', 'A']
covtypes = ['binary', 'normal']

covmodels = ['L ~ lag1_L + t0',
             'A ~ lag1_A + L + t0']

outcome_name = 'Y'
ymodel = 'Y ~ A + L'

censor_name = 'C'
censor_model = 'C ~ A + L'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points,
    int_descript=int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    censor_name= censor_name, censor_model=censor_model,
    covnames = covnames, covtypes = covtypes, covmodels = covmodels,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+---------------------+------------------+---------------------------------+----------------+---------------------+
|     Intervention    | IP weighted risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+------------------+---------------------------------+----------------+---------------------+
| Natural course(ref) |     0.56498      |             0.56570             |    1.00000     |       0.00000       |
|     Never treat     |        NA        |             0.35805             |    0.63294     |       -0.20765      |
|     Always treat    |        NA        |             0.86679             |    1.53223     |       0.30108       |
+---------------------+------------------+---------------------------------+----------------+---------------------+
```

3.6 Competing event

In the presence of competing events, users may choose whether to treat competing events as censoring events. When competing events are treated as censoring events, risks under different interventions are calculated under elimination of competing events, and are obtained by the Kaplan–Meier estimator. When competing events are not treated as censoring events, risks under different interventions are calculated without elimination of competing events, and are obtained by using an estimate of the subdistribution cumulative incidence function[1, 2].

The arguments for competing events:

[1] Young JG, Stensrud MJ, Tchetgen Tchetgen EJ, Hernán MA. A causal framework for classical statistical estimands in failure-time settings with competing events. Statistics in Medicine. 2020;39:1199-236.

[2] Fine JP and Gray RJ. A proportional hazards model for the subdistribution of a competing risk. Journal of the American Statistical Association, 94(446):496–509, 1999.

Arguments	Description
compevent_name	(Optional) A string specifying the name of the competing event variable in obs_data. Only applicable for survival outcomes.
compevent_model	(Optional) A string specifying the model statement for the competing event variable. Only applicable for survival outcomes.
compevent_cens	(Optional) A boolean value indicating whether to treat competing events as censoring events. Default is False.

Sample syntax:

```
compevent_name = 'D'
compevent_model = 'D ~ A + L1 + L2 + L3 + t0'
compevent_cens = False

g = ParametricGformula(..., compevent_name = compevent_name, compevent_model = compevent_
↪model, compevent_cens = compevent_cens, ...)
```

The name of competing event in the input data should be specified in the argument "compevent_name". The model statement for the competing event variable should be specified in the argument "compevent_model". Users should also specify the argument "compevent_cens" as True or False indicating whether they want to treat the competing event as censoring event (the default is False).

Setting "compevent_cens" as default (False):

Running example [code]:

```
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata

obs_data = load_basicdata()

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2  + L3 + t0',
             'A ~ lag1_A + L1 + L2 +lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

ymodel = 'Y ~ A + L1 + L2 + L3 + lag1_A + lag1_L1 + lag1_L2'

time_name = 't0'
id = 'id'
outcome_name = 'Y'
basecovs = ['L3']

compevent_name = 'D'
compevent_model = 'D ~ A + L1 + L2 + L3 + t0'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id, time_points = time_points,
              time_name=time_name, int_descript = int_descript,
```

(continues on next page)

(continued from previous page)

```
                Intervention1_A = [static, np.zeros(time_points)],
                Intervention2_A = [static, np.ones(time_points)],
                basecovs =basecovs, covnames=covnames,
                covtypes=covtypes, covmodels=covmodels,
                compevent_name = compevent_name, compevent_model=compevent_model,
                outcome_name=outcome_name, outcome_type='survival', ymodel=ymodel)
g.fit()
```

Output:

```
+---------------------+---------+----------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+----------------------------------+----------------+---------------------+
| Natural course(ref) | 0.42400 |             0.42306              |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.58927              |    1.39289     |       0.16622       |
|     Always treat    |    NA   |             0.21410              |    0.50609     |       -0.20895      |
+---------------------+---------+----------------------------------+----------------+---------------------+
```

Setting "compevent_cens" as True:

```
compevent_name = 'D'
compevent_model = 'D ~ A + L1 + L2 + L3 + t0'
compevent_cens = True

g = ParametricGformula(..., compevent_name = compevent_name, compevent_model = compevent_
↪model, compevent_cens = compevent_cens, ...)
```

Output:

```
+---------------------+---------+----------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+----------------------------------+----------------+---------------------+
| Natural course(ref) | 0.52759 |             0.53225              |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.77235              |    1.45111     |       0.24010       |
|     Always treat    |    NA   |             0.24955              |    0.46886     |       -0.28270      |
+---------------------+---------+----------------------------------+----------------+---------------------+
```

3.7 Hazard ratio

For survival outcomes, the pygformula provides the option of calculating the hazard ratio comparing any two interventions of interest. In the presence of competing events, it will return the subdistribution hazard ratio[1]. Note that there is an order requirement for the input data structure that it should have the competing event before the outcome event.

Prerequisite: If users want to calculate the hazard ratio with competing event, they needs to install additional "rpy2" package and install the python "cmprsk" package. Please follow the steps below to install:

- Install R to set up R environment
- Install cmprsk R package in R environment:

```
install.packages("cmprsk")
```

- Install rpy2 package in python environment:

[1] Fine JP and Gray RJ. A proportional hazards model for the subdistribution of a competing risk. Journal of the American Statistical Association, 94(446):496–509, 1999.

```
pip install rpy2
```

- Install cmprsk package in python environment:

```
pip install cmprsk
```

Note: If you encounters the problem of not finding the R environment, you can set up the R path in your environment using the following command in the code:

```
import os
os.environ["R_HOME"] = 'R_HOME'
```

where R_HOME is the R home directory path.

The argument for calculating the hazard ratio:

Arguments	Description
intcomp	(Optional) List of two numbers indicating a pair of interventions to be compared by a hazard ratio.

Users can specify the two interventions by:

```
intcomp = [1, 2]
```

The integer i in "intcomp" denotes the i-th intervention in the user-specified interventions. 0 denotes the natural course intervention.

Running example [code]:

```
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()
time_name = 't0'
id = 'id'

covnames = ['L2', 'A']
covtypes = ['bounded normal', 'binary']
covmodels = ['L2 ~ lag1_A + lag_cumavg1_L2 + L3 + t0',
             'A ~ lag1_A + L2 + lag_cumavg1_L2 + L3 + t0']

basecovs = ['L3']

outcome_name = 'Y'
ymodel = 'Y ~ L2 + A + lag1_A + L3 + t0'
outcome_type = 'survival'

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']
```

(continues on next page)

(continued from previous page)

```
g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points,
    int_descript = int_descript, intcomp=[1, 2],
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    outcome_name=outcome_name, ymodel=ymodel, outcome_type=outcome_type)
g.fit()
```

Output:

```
+---------------------+---------+----------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+----------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.51103              |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.75901              |    1.48525     |       0.24798       |
|     Always treat    |    NA   |             0.22726              |    0.44472     |       -0.28377      |
+---------------------+---------+----------------------------------+----------------+---------------------+
Hazardratio value is 0.17407
```

3.8 Visit process

When the data are not recorded at regular intervals but rather are recorded everytime the patient visits the clinic, the times at which the time-varying covariates are measured will vary by subject. In this setting, it is typical to construct the data such that (i) at a time when there is no visit/measurement, the last measured value of a covariate is carried forward, and (ii) a subject is censored after a certain number of consecutive times with no visit/measurement[1, 2].

In pygformula, the deterministic knowledge (i) and (ii) can be incorporated via the argument "visitprocess". Each vector in "visitprocess" contains three parameters that attach a visit process to one covariate. The first parameter is the name of a time-varying indicator in the input data set of whether a covariate was measured in each interval (1 means there is a visit/measurement, 0 means there is no visit/measurement). The second parameter is the name of the covariate. The third parameter is the maximum number s of missed measurements of this covariate allowed since the last measurement before a subject is censored.

For the visit indicator, in the fitting step, the probability of a visit is estimated only using records where the sum of consecutive missed visits through previous k-1 time points is less than the maximum number of consecutive missed visits s. Then in the simulation step, if the sum of consecutive missed visits through previous k-1 time points is less than s, then the visit indicator is simulated from a distribution based on this estimate; otherwise, the visit indicator is set to 1 so as to eliminate subjects with more than s consecutive missed visits. For the covariate, in the fitting step, the conditional mean of the covariate will be estimated only for data records where there is a current visit. If the visit indicator equals 1, then in simulation step, the value of the dependent covariate will be generated from a distribution based on this estimate; otherwise, the last value is carried forward.

The argument for visit process:

[1] Hernán MA, McAdams M, McGrath N, Lanoy E, Costagliola D. Observation plans in longitudinal studies with time-varying treatments. Statistical Methods in Medical Research 2009;18(1):27-52.

[2] Young JG, Cain LE, Robins JM, O'Reilly E, Hernán MA. Comparative effectiveness of dynamic treatment regimes: an application of the parametric g-formula. Statistics in Biosciences 2011; 3:119-143.

Arguments	Description
visitprocess	(Optional) List of lists. Each inner list contains its first entry the covariate name of a visit process; its second entry the name of a covariate whose modeling depends on the visit process; and its third entry the maximum number of consecutive visits that can be missed before an individual is censored.

```
covnames = ['visit_cd4', 'visit_rna', 'cd4_v', 'rna_v', 'everhaart']
covtypes = ['binary', 'binary', 'normal', 'normal', 'binary']
covmodels = ['visit_cd4 ~ lag1_everhaart + lag_cumavg1_cd4_v + sex + race + month',
             'visit_rna ~ lag1_everhaart + lag_cumavg1_rna_v + sex + race + month',
             'cd4_v ~ lag1_everhaart + lag_cumavg1_cd4_v + sex + race + month',
             'rna_v ~ lag1_everhaart + lag_cumavg1_rna_v + sex + race + month',
             'everhaart ~ lag1_everhaart + cd4_v + rna_v + sex + race + month']

visitprocess = [['visit_cd4', 'cd4_v', 3], ['visit_rna', 'rna_v', 3]]

g = ParametricGformula(..., covnames = covnames, covtypes = covtypes, covmodels =␣
↪covmodels, visitprocess = visitprocess, ...)
```

Here is an example in clinical cohorts of HIV-positive patients, "cd4_v" is a time-varying covariate of CD4 cell count measurement, the visit indicator "visit_cd4" indicats whether the CD4 cell count measurements were taken in interval k. 3 means that the data is constructed such that the subjects are censored once they have not had CD4 measured for 3 consecutive intervals. Note that for the visit indicator "visit_cd4", it should come before the dependent covariate "cd4_v" and be assigned the "binary" covariate type in "covtypes".

Running example [code]:

```
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_visit_process

obs_data = load_visit_process()
time_name = 'month'
id = 'id'

covnames = ['visit_cd4', 'visit_rna', 'cd4_v', 'rna_v', 'everhaart']
covtypes = ['binary', 'binary', 'normal', 'normal', 'binary']
covmodels = ['visit_cd4 ~ lag1_everhaart + lag_cumavg1_cd4_v + sex + race + month',
             'visit_rna ~ lag1_everhaart + lag_cumavg1_rna_v + sex + race + month',
             'cd4_v ~ lag1_everhaart + lag_cumavg1_cd4_v + sex + race + month',
             'rna_v ~ lag1_everhaart + lag_cumavg1_rna_v + sex + race + month',
             'everhaart ~ lag1_everhaart + cd4_v + rna_v + sex + race + month']

basecovs = ['sex', 'race', 'age']

visitprocess = [['visit_cd4', 'cd4_v', 3], ['visit_rna', 'rna_v', 3]]

outcome_name = 'event'
ymodel = 'event ~ cd4_v + rna_v + everhaart + sex + race + month'

time_points = np.max(np.unique(obs_data[time_name])) + 1
```

(continues on next page)

(continued from previous page)

```
int_descript = ['Never treat', 'Always treat']

g = ParametricGformula(obs_data = obs_data, id = id,  time_name = time_name,
    visitprocess = visitprocess,
    int_descript = int_descript,
    Intervention1_everhaart = [static, np.zeros(time_points)],
    Intervention2_everhaart = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs = basecovs,
    outcome_name=ou tcome_name, ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.03774 |             0.05620             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.06979             |    1.24180     |       0.01359       |
|     Always treat    |    NA   |             0.05116             |    0.91032     |       -0.00504      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

3.9 Deterministic knowledge

When there are known priori deterministic knowledge, they can be incorporated into the g-formula algorithm to avoid unnecessary extrapolation. The package allows users to apply restrictions of the deterministic knowledge on the covariates, outcome or competing event.

3.9.1 Restrictions on covariates

When incorporating the deterministic knowledge of one time-varying covariate Z, the estimation is changed as follows:

1. In step 1 of the algorithm, restrict the chosen method of estimating the mean of Z given "history" to only records where deterministic knowledge is absent.

2. In step 2 of the algorithm, set Z deterministically to its known value for histories under which this value is known. Otherwise, draw Z according to the model-based estimate conditional distribution of Z.

For example, when there are two time-varying covariates, one indicator of whether an individual has started menopause by a given interval k (menopause), and another indicator of whether she is pregnant in interval k (pregnancy). The deterministic knowledge is that given menopause == 1, the probability that pregnancy == 0 is 1. In the first estimation step, only records with menopause == 0 are used for model estimation of pregnancy. Then in the second simulation step, if the value of menopause in step 1 at time k is 1 then pregnancy is set to 0. Otherwise, the value of pregnancy at time k is drawn from the estimated distribution in step 1.

The package allows deterministic knowledge incorporation for covariates by the argument "restrictions":

Arguments	Description
restrictions	(Optional) List of lists. Each inner list contains its first entry the covariate name of that its deterministic knowledge is known; its second entry is a dictionary whose key is the conditions which should be True when the covariate is modeled, the third entry is the value that is set to the covariate during simulation when the conditions in the second entry are not True.

Note that for each restricted covariate and its conditional covariates, they need to follow the same order in "covnames", i.e., the restricted covariate should be after its conditional covariates.

An example of the restrictions that encodes the relationship between menopause and pregnancy above:

```
restrictions = [['pregnancy', {'menopause': lambda x: x == 0}, 1]]
g = ParametricGformula(..., restrictions = restrictions, ...)
```

Sample syntax:

An example with one deterministic knowledge conditions for one covariate 'L2': if L1 equals 0, L2 is estimated by its parametric model, otherwise, it is set to a known value 0.5.

```
restrictions = [['L2', {'L1': lambda x: x == 0}, 0.5]]
g = ParametricGformula(..., restrictions = restrictions, ...)
```

An example with multiple deterministic knowledge conditions for one covariate 'A': if L1 equals 0 and L2 is greater than 0.5, A is estimated by its parametric model, otherwise, it is set to a known value 1.

```
restrictions = [['A', {'L1': lambda x: x == 0, 'L2': lambda x: x > 0.5}, 1]]
g = ParametricGformula(..., restrictions = restrictions, ...)
```

An example with multiple restrictions, one for covariate L2 and one for covariate A:

```
restrictions = [['L2', {'L1': lambda x: x == 0}, 0.5], ['A', {'L1': lambda x: x == 0, 'L2
↪': lambda x: x > 0.5}, 1]]
g = ParametricGformula(..., restrictions = restrictions, ...)
```

If the assigned value of the covariate is not a static value, but determined by a user-specified function, the "restrictions" allows an input as a function type. In this case, the third entry for a restriction is a function instead of a value.

For each custom restriction function, the input should be the parameters (not necessary to use all):

- new_df: A DataFrame that contains the observed or simulated data at time t.
- pool: A DataFrame that contains the observed or simulated data up to time t.
- time_name: A string specifying the name of the time variable in pool.
- t: An integer specifying the current time index.

The function output should be a list of values that users wish to assign for the restricted covariate at time t. The package will automatically assign these values for records that are not restricted by the conditions.

An example with one deterministic knowledge condition for covariate L2: if L1 equals 0, L2 is estimated by its parametric model, otherwise, its previous value is carried forward.

```
def carry_forward(new_df, pool, time_name, t):
    assigned_values = pool.loc[pool[time_name] == t-1, 'L2']
    return assigned_values

restrictions = [['L2', {'L1': lambda x: x == 0}, carry_forward]]
g = ParametricGformula(..., restrictions = restrictions, ...)
```

Running example [code]:

```
import numpy as np
import pygformula
from pygformula.parametric_gformula.interventions import static
```

(continues on next page)

(continued from previous page)

```
from pygformula import ParametricGformula
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()

time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'normal', 'binary']
covmodels = ['L1 ~ lag1_L1 + lag1_A',
             'L2 ~ L1 + lag1_L2',
              'A ~ L1 + L2']

basecovs = ['L3']
outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + A'

# define interventions
time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

restrictions = [['L2', {'L1': lambda x: x == 0}, 0.5], ['A', {'L1': lambda x: x == 0, 'L2
↪': lambda x: x > 0.5}, 1]]

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
    time_points = time_points,
    int_descript = int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames, covtypes=covtypes,
    covmodels=covmodels, basecovs=basecovs,
    restrictions=restrictions, outcome_name=outcome_name,
    ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+--------------------+---------+---------------------------------+----------------+----------------------+
|    Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+--------------------+---------+---------------------------------+----------------+----------------------+
| Natural course(ref) | 0.50560 |            0.28715              |    1.00000     |       0.00000       |
|     Never treat    |    NA   |            0.74569              |    2.59683     |       0.45853       |
|     Always treat   |    NA   |            0.23208              |    0.80823     |       -0.05507      |
+--------------------+---------+---------------------------------+----------------+----------------------+
```

3.9.2 Restrictions on outcome

When there is deterministic knowledge of the outcome variable Y, the package offers the argument "restrictions" to incorporate the knowledge:

Arguments	Description
yrestrictions	(Optional) List of lists. For each inner list, its first entry is a dictionary whose key is the conditions which should be True when the outcome is modeled, the second entry is the value that is set to the outcome during simulation when the conditions in the first entry are not True.

Sample syntax:

An example with one deterministic knowledge conditions for outcome Y: if L1 equals 0, the probability of outcome Y is estimated by its parametric model, otherwise, it is set to value 1.

```
yrestrictions = [[{'L1': lambda x: x == 0}, 1]]
g = ParametricGformula(..., yrestrictions = yrestrictions, ...)
```

An example with multiple restrcitions for outcome Y: if L1 equals 0, the probability of outcome Y is estimated by its parametric model, otherwise, it is set to a value 0; if L2 is greater than 0.5, the probability of outcome Y is estimated by its parametric model, otherwise, it is set to a value 0.1;

```
yrestrictions = [[{'L1': lambda x: x == 0}, 0], [{'L2': lambda x: x > 0.5}, 0.1]]
g = ParametricGformula(..., yrestrictions = yrestrictions, ...)
```

Running example [code]:

```
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata_nocomp

obs_data = load_basicdata_nocomp()

time_name = 't0'
id = 'id'

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'normal', 'binary']
covmodels = ['L1 ~ lag1_L1 + lag1_A',
             'L2 ~ L1 + lag1_L2',
              'A ~ L1 + L2']

basecovs = ['L3']
outcome_name = 'Y'
ymodel = 'Y ~ L1 + L2 + A'

# define interventions
time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

yrestrictions = [[{'L1': lambda x: x == 0}, 0], [{'L2': lambda x: x > 0.5}, 0.1]]

g = ParametricGformula(obs_data = obs_data, id = id, time_name=time_name,
```

(continues on next page)

(continued from previous page)

```
    time_points = time_points,
    int_descript = int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    covnames=covnames,  covtypes=covtypes, covmodels=covmodels, basecovs=basecovs,
    yrestrictions=yrestrictions, outcome_name=outcome_name,
    ymodel=ymodel, outcome_type='survival')
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.50560 |             0.46879             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.49002             |    1.04528     |       0.02123       |
|     Always treat    |    NA   |             0.44450             |    0.94818     |       -0.02429      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

3.9.3 Restrictions on competing event

When there is a competing event D and there is known deterministic knowledge of the competing event, the package offers the argument "compevent_restrictions" for incorporation:

Arguments	Description
compevent_restrictions	(Optional) List of lists. For each inner list, its first entry is a dictionary whose key is the conditions which should be True when the competing event is modeled, the second entry is the value that is set to the competing event during simulation when the conditions in the first entry are not True. Only applicable for survival outcomes.

Sample syntax:

An example with one deterministic knowledge conditions for competing event D: if L1 equals 0, the probability of competing event is estimated by its parametric model, otherwise, it is set to a value 1.

```
compevent_restrictions = [{'L1': lambda x: x == 0}, 1]
g = ParametricGformula(..., compevent_restrictions = compevent_restrictions, ...)
```

An example with multiple restrictions for competing event D: if L1 equals 0, the probability of competing event is estimated by its parametric model, otherwise, it is set to a value 1; if L2 is greater than 0.5, the probability of competing event is estimated by its parametric model, otherwise, it is set to a value 0.1;

```
compevent_restrictions = [[{'L1': lambda x: x == 0}, 0], [{'L2': lambda x: x > 0.5}, 0.
↪1]]
g = ParametricGformula(..., compevent_restrictions = compevent_restrictions, ...)
```

Running example [code]:

```
import pygformula
from pygformula import ParametricGformula
from pygformula.parametric_gformula.interventions import static
from pygformula.data import load_basicdata
```

(continues on next page)

(continued from previous page)

```
obs_data = load_basicdata()

covnames = ['L1', 'L2', 'A']
covtypes = ['binary', 'bounded normal', 'binary']
covmodels = ['L1 ~ lag1_A + lag2_A + lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0',
             'L2 ~ lag1_A + L1 + lag_cumavg1_L1 + lag_cumavg1_L2  + L3 + t0',
             'A ~ lag1_A + L1 + L2 +lag_cumavg1_L1 + lag_cumavg1_L2 + L3 + t0']

outcome_model = 'Y ~ A + L1 + L2 + L3 + lag1_A + lag1_L1 + lag1_L2'

time_name = 't0'
id = 'id'
outcome_name = 'Y'
basecovs = ['L3']

compevent_name = 'D'
compevent_model = 'D ~ A + L1 + L2 + L3 + t0'
compevent_cens = False

time_points = np.max(np.unique(obs_data[time_name])) + 1
int_descript = ['Never treat', 'Always treat']

compevent_restrictions = [[{'L1': lambda x: x == 0}, 0], [{'L2': lambda x: x > 0.5}, 0.
↪1]]

g = ParametricGformula(obs_data = obs_data, id = id, time_points = time_points,
    time_name=time_name, int_descript = int_descript,
    Intervention1_A = [static, np.zeros(time_points)],
    Intervention2_A = [static, np.ones(time_points)],
    basecovs =basecovs, covnames=covnames,
    covtypes=covtypes, covmodels=covmodels,
    compevent_restrictions = compevent_restrictions,
    compevent_cens= compevent_cens, compevent_name = compevent_name,
    compevent_model=compevent_model, outcome_name=outcome_name,
    outcome_type='survival', ymodel=ymodel)
g.fit()
```

Output:

```
+---------------------+---------+---------------------------------+----------------+---------------------+
|     Intervention    | NP risk | g-formula risk (NICE-parametric) | Risk Ratio(RR) | Risk Difference(RD) |
+---------------------+---------+---------------------------------+----------------+---------------------+
| Natural course(ref) | 0.42400 |             0.39197             |    1.00000     |       0.00000       |
|     Never treat     |    NA   |             0.59360             |    1.51440     |       0.20163       |
|     Always treat    |    NA   |             0.17603             |    0.44910     |       -0.21593      |
+---------------------+---------+---------------------------------+----------------+---------------------+
```

3.10 Output

3.10.1 Numerical results

The package provides the following outputs:

- **Data table of g-formula estimates**: The result table of g-formula estimates is returned by the fit function, containing (1) the nonparametric estimates of the natural course risk/mean outcome, (2) the parametric g-formula estimates of the risk/mean outcome under each user-specified intervention, (3) the risk ratio between each intervention and the reference intervention (natural course by default, can be specified in the argument "ref_int"), (4) the risk difference between each intervention and the reference intervention.
- **Simulated data table for interventions**: The package gives the simulated data table in the simulation step under each specified intervention, which can be obtained by:

  ```
  sim_data = g.summary_dict['sim_data']
  ```

  To get the simulated data under a particular intervention:

  ```
  sim_data = g.summary_dict['sim_data'][intervention_name]
  ```

- **The IP weights**: To get the inverse probability weights when there is censoring event:

  ```
  ip_weights = g.summary_dict['IP_weights']
  ```

- **The model summary**: The package gives the model summary for each covariate, outcome, competing event (if applicable), censoring event (if applicable). First the argument "model_fits" should be set to True, then the model summary can be obtained by:

  ```
  fitted_models = g.summary_dict['model_fits_summary']
  ```

  To get the fitted model for a particular variable:

  ```
  fitted_model = g.summary_dict['model_fits_summary'][variable_name]
  ```

- **The coefficients**: The package gives the parameter estimates of all the models, which can be obtained by:

  ```
  model_coeffs = g.summary_dict['model_coeffs']
  ```

  To get the coefficients of the model for a particular variable, please use:

  ```
  model_coeffs = g.summary_dict['model_coeffs'][variable_name]
  ```

- **The standard errors**: The package gives the standard errors of the parameter estimates of all the models, which can be obtained by:

  ```
  model_stderrs = g.summary_dict['model_stderrs']
  ```

  To get the standard errors of the model for a particular variable, please use:

  ```
  model_stderrs = g.summary_dict['model_stderrs'][variable_name]
  ```

- **The variance-covariance matrices**: The package gives the variance-covariance matrices of the parameter estimates of all the models, which can be obtained by:

```
model_vcovs = g.summary_dict['model_vcovs']
```

To get the variance-covariance matrix of the parameter estimates of the model for a particular variable, please use:

```
model_vcovs = g.summary_dict['model_vcovs'][variable_name]
```

- **The root mean square error**: The package gives the RMSE values of the models, which can be obtained by:

```
rmses = g.summary_dict['rmses']
```

To get the RMSE of the model for a particular variable, please use:

```
rmses = g.summary_dict['rmses'][variable_name]
```

- **Nonparametric estimates at each time point**: The package gives the nonparametric estimates of all covariates and risk at each time point for survival outcomes, which can be obtained by:

```
obs_estimates = g.summary_dict['obs_plot']
```

To get the nonparametric estimates of a particular variable, e.g., risk, please use:

```
obs_estimates = g.summary_dict['obs_plot']['risk']
```

- **Parametric estimates at each time point**: The package gives the parametric estimates of all covariates and risk at each time point for survival outcomes, which can be obtained by:

```
est_estimates = g.summary_dict['est_plot']
```

To get the parametric estimates of a particular variable, e.g., risk, please use:

```
est_estimates = g.summary_dict['est_plot']['risk']
```

- **Hazard ratio**: The package gives hazard ratio value for the two interventions specified, which can be obtained by:

```
hazard_ratio = g.summary_dict['hazard_ratio']
```

The package also implement nonparametric bootstrapping to obtain 95% confidence intervals for risk/mean estimates by repeating the algorithm for many bootstrap samples. Users can choose the argument "nsamples" to specify the number of new generated bootstrap samples. Users may choose the argument "parallel" to parallelize bootstrapping and simulation steps under each intervention to make the algorithm run faster. The argument "ncores" can be used to specify the desired number of CPU cores in parallarization.

The package provides two ways for calculating the confidence intervals in argument "ci_method", "percentile" means using percentile bootstrap method which takes the 2.5th and 97.5th percentiles of the bootstrap estimates to get the 95% confidence interval, "normal" means using the normal bootstrap method which uses the the original estimate and the standard deviation of the bootstrap estimates to get the normal approximation 95% confidence interval.

- **The g-formula estimates of bootstrap samples**: The package gives the parametric g-formula estimates of all bootstrap samples, which can be obtained by:

```
g = ParametricGformula(..., nsamples = 20, parallel=True, n_core=10, ci_
↪method = 'percentile', ...)
g.fit()
bootests = g.summary_dict['bootests']
```

To get the parametric g-formula estimates of a particular bootstrap sample, please use:

```
g.summary_dict['bootests']['sample_{id}_estimates']
```

where id is the sample id which should be an integer between 0 and "nsamples" - 1.

- **The coefficients of bootstrap samples**: The package gives the parameter estimates of all the models for all generated bootstrap samples, which can be obtained by:

```
g = ParametricGformula(..., nsamples = 20, parallel=True, n_core=10, ci_
↪method = 'percentile', boot_diag=True, ...)
g.fit()
bootcoeffs = g.summary_dict['bootcoeffs']
```

Note that the "boot_diag" should be set to true if users want to obtain the coefficients, standard errors or variance-covariance matrices of bootstrap samples.

To get the coefficients of a particular bootstrap sample, please use:

```
g.summary_dict['bootcoeffs']['sample_{id}_coeffs']
```

- **The standard errors of bootstrap samples**: The package gives the standard errors of the parameter estimates of all the models for all generated bootstrap samples, which can be obtained by:

```
g = ParametricGformula(..., nsamples = 20, parallel=True, n_core=10, ci_
↪method = 'percentile', boot_diag=True, ...)
g.fit()
bootstderrs = g.summary_dict['bootstderrs']
```

To get the standard errors of a particular bootstrap sample, please use:

```
g.summary_dict['bootstderrs']['sample_{id}_stderrs']
```

- **The variance-covariance matrices of bootstrap samples**: The package gives the variance-covariance matrices of the parameter estimates of all the models for all generated bootstrap samples, which can be obtained by:

```
g = ParametricGformula(..., nsamples = 20, parallel=True, n_core=10, ci_
↪method = 'percentile', boot_diag=True, ...)
g.fit()
bootvcovs = g.summary_dict['bootvcovs']
```

To get the variance-covariance matrices of a particular bootstrap sample, please use:

```
g.summary_dict['bootvcovs']['sample_{id}_vcovs']
```

Note that to get bootstrap results of coefficients, standard errors, and variance-covariance matrices, the argument "boot_diag" must be set to True.

All the output results above can be saved by the argument "save_results", once it is set to True, results will be saved locally by creating a folder automatically. Users can also specify the folder path by the argument "save_path":

```
g = ParametricGformula(..., save_results = True, save_path = 'user-specified␣
↪path', ...)
g.fit()
```

Arguments:

Arguments	Description
n_simul	(Optional) An integer indicating the number of subjects for whom to simulate data. It is set equal to the number (M) of subjects in obs_data, if not specified by users.
ref_int	(Optional) An integer indicating the intervention to be used as the reference for calculating the end-of-follow-up mean/risk ratio and mean/risk difference. 0 denotes the natural course, while subsequent integers denote user-specified interventions in the order that they are named in interventions. It is set to 0 if not specified by users.
nsamples	(Optional) An integer specifying the number of bootstrap samples to generate.
parallel	(Optional) A boolean value indicating whether to parallelize simulations of different interventions to multiple cores.
ncores	(Optional) An integer indicating the number of cores used in parallelization. It is set to 1 if not specified by users.
model_fits	(Optional) A boolean value indicating whether to return the parameter estimates of the models.
ci_method	(Optional) A string specifying the method for calculating the bootstrap 95% confidence intervals, if applicable. The options are "percentile" and "normal". It is set to "percentile" if not specified by users.
boot_diag	(Optional) A boolean value indicating whether to return the parametric g-formula estimates as well as the coefficients, standard errors, and variance-covariance matrices of the parameters of the fitted models in the bootstrap samples.
save_results	(Optional) A boolean value indicating whether to save all the returned results to the save_path.
save_path	(Optional) A path to save all the returned results. A folder will be created automatically in the current working directory if the save_path is not specified by users.
seed	(Optional) An integer indicating the starting seed for simulations and bootstrapping. It is set to 1234 if not specified by users.

3.10.2 Graphical results

The package also provides two plotting functions: "plot_natural_course" and "plot_interventions". The plot_natural_course function plots the curves of each covariate mean (for all types of outcomes) and risk (for survival outcomes only) under g-formula parametric and non-parametric estimation.

plot_natural_course(time_points, covnames, ...)	This is an internal function that plots the results comparison of covariate means and risks between non-parametric estimates and g-formula parametric estimates.

pygformula.plot.**plot_natural_course**(*time_points*, *covnames*, *covtypes*, *time_name*, *obs_data*, *obs_means*, *est_means*, *censor*, *outcome_type*, *plot_name*, *marker*, *markersize*, *linewidth*, *colors*, *save_path*, *save_figure*, *boot_table*)

This is an internal function that plots the results comparison of covariate means and risks between non-parametric estimates and g-formula parametric estimates.

Parameters

- **time_points** (*Int*) – An integer indicating the number of time points to simulate. It is set equal to the maximum number of records (K) that obs_data contains for any individual plus 1, if not specified by users.
- **covnames** (*List*) – A list of strings specifying the names of the time-varying covariates in obs_data.
- **covtypes** (*List*) – A list of strings specifying the "type" of each time-varying covariate included in covnames. The supported types: "binary", "normal", "categorical", "bounded nor-

mal", "zero-inflated normal", "truncated normal", "absorbing", "categorical time", "square time" and "custom". The list must be the same length as covnames and in the same order.

- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **obs_data** (*DataFrame*) – A data frame containing the observed data.
- **obs_means** (*Dict*) – A dictionary, where the key is the covariate / risk name and the value is its observational mean at all the time points.
- **est_means** (*Dict*) – A dictionary, where the key is the covariate / risk name and the value is its parametric mean at all the time points.
- **censor** (*Bool*) – A boolean value indicating the if there is a censoring event.
- **outcome_type** (*Str*) – A string specifying the "type" of outcome. The possible "types" are: "survival", "continuous_eof", and "binary_eof".
- **plot_name** (*Str*) – A string specifying the name for plotting, which is set to "all", "risk" or one specific covariate name.
- **marker** (*Str*) – A string used to customize the appearance of points in plotting.
- **markersize** (*Int*) – An integer specifies the size of the markers in plotting.
- **linewidth** (*Float*) – A number that specifies the width of the line in plotting.
- **colors** (*List*) – A list that contains two strings, the first specifies the color for plotting nonparametric estimates, the second specifies the color for plotting the parametric estimates.
- **save_path** (*Path*) – A path to save all the figure results. A folder will be created automatically in the current working directory if the save_path is not specified by users.
- **save_figure** (*Bool*) – A boolean value indicating whether to save the figure or not.
- **boot_table** (*DataFrame*) – A DataFrame with nonparametric risk and parametric risks of all interventions.

Return type Nothing is returned, the figure will be shown.

The plot_interventions function plots the curves of risk under interventions of interest (for survival outcomes only).

plot_interventions(time_points, time_name, ...)	An internal function to plot the risk results comparison of all interventions and the natural course.

pygformula.plot.**plot_interventions**(*time_points*, *time_name*, *risk_results*, *int_descript*, *outcome_type*, *colors*, *marker*, *markersize*, *linewidth*, *save_path*, *save_figure*, *boot_table*)

An internal function to plot the risk results comparison of all interventions and the natural course.

Parameters

- **time_points** (*Int*) – An integer indicating the number of time points to simulate. It is set equal to the maximum number of records (K) that obs_data contains for any individual plus 1, if not specified by users.
- **time_name** (*Str*) – A string specifying the name of the time variable in obs_data.
- **risk_results** (*List*) – A list that contains the risk estimates at all the time points of all interventions.
- **int_descript** (*List*) – A list of strings, each describing a user-specified intervention.

- **outcome_type** (*Str*) – A string specifying the "type" of outcome. The possible "types" are: "survival", "continuous_eof", and "binary_eof".
- **colors** (*List*) – A list that contains strings, each of which specifies the color for plotting the risk curve of the intervention.
- **marker** (*Str*) – A string used to customize the appearance of points in plotting.
- **markersize** (*Int*) – An integar specifies the size of the markers in plotting.
- **linewidth** (*Float*) – A number that specifies the width of the line in plotting.
- **save_path** (*Path*) – A path to save all the figure results. A folder will be created automatically in the current working directory if the save_path is not specified by users.
- **save_figure** (*Bool*) – A boolean value indicating whether to save the figure or not.
- **boot_table** (*DataFrame*) – A DataFrame with nonparametric risk and parametric risks of all interventions.

Return type Nothing is returned, the figure will be shown.

Arguments for plotting:

Arguments	Description
plot_name	A string specifying the name for plotting, which is set to "all", "risk" or one specific covariate name. Only applicable for the plot_natural_course function. The default is "all".
colors	For plot_natural_course function, it is a list wth two elements, specifying the non-parametric estimate curve and parametric curve respectively. Users can choose colors from matplotlib colors. For plot_interventions function, it is a list wth m elements with m the number of interventions plus 1, specifying all intervention curves. If not specified, the function will use default colors.
marker	A string used to customize the appearance of points in plotting. Users can also choose markers from matplotlib markers library.
markersize	An integer specifies the size of the markers in plotting.
linewidth	A number that specifies the width of the line in plotting.
save_figure	A boolean value indicating whether to save the figure or not.

Users can call the 'plot_natural_course' function by:

```
g.plot_natural_course()
```

Users can call the 'plot_interventions' function by:

```
g.plot_interventions()
```

Note that the plotting functions can only be applied after calling the 'g.fit' function.

The figures can be saved by the argument "save_figure", once it is set to True, results will be saved locally by creating a folder automatically. If the argument "save_path" is specified, the figure will be saved to the corresponding folder.

Sample syntax:

```
g.plot_natural_course(plot_name='L1', colors=['blue', 'red'], markersize=5, linewidth=1,␣
↪marker='v', save_figure=True)
g.plot_interventions(colors =['green', 'red', 'yellow'], markersize=5, linewidth=1,␣
↪marker='v', save_figure=True)
```

Note: We recommend setting the "save_figure" as True if users want to access the figure when running the package on Linux system.

CHAPTER

FOUR

DATASETS

We provide simulation "datasets" for users to run the different examples in this tutorial. Additionally, code for replicating all test examples can be found in "running examples".

Note: To run the test examples using the provided datasets, please download the pygformula repository from the github pygformula.

CHAPTER

FIVE

CONTACT

The pygformula package was developed in the CAUSALab by:

- Jing Li (jing_li@hsph.harvard.edu)
- Sophia Rein, srein@hsph.harvard.edu
- Sean McGrath, sean_mcgrath@g.harvard.edu
- Roger Logan, rwlogan@hsph.harvard.edu
- Ryan O'Dea, ryanodea@hsph.harvard.edu
- Miguel Hernán, mhernan@hsph.harvard.edu

If you have any questions or suggestions about this package, please contact jing_li@hsph.harvard.edu. As an ongoing open-source project, contributions are highly welcome for any bug reports or feature suggestions.

- Issue reports: if you have any issues, please let us know by opening an issue on github.
- Feature requests: if you want to contribute any new feature implementation, please make a pull request to post the feature requests.

PYTHON MODULE INDEX

p

INDEX