zEpid

Oct 23, 2022

Contents

zEpid is an epidemiology analysis toolkit for Python 3.6 to 3.10. The purpose of this library is to make epidemiology e-z to do in Python. A variety of calculations, estimators, and plots can be implemented. Current features include:

- Basic epidemiology calculations on pandas Dataframes
- Risk ratio, risk difference, number needed to treat, incidence rate ratio, etc.
- Interaction contrasts and interaction contrast ratios
- Semi-bayes
- Summary measure calculations from summary data
- Risk ratio, risk difference, number needed to treat, incidence rate ratio, etc.
- Interaction contrasts and interaction contrast ratios
- Semi-bayes
- Graphics
- Functional form plots
- Forest plots (effect measure plots)
- P-value plots
- Causal inference
- Parametric g-formula
- Inverse probability of treatment weights
- Augmented inverse probability of treatment weights
- Targeted maximum likelihood estimator
- Monte-Carlo g-formula
- Iterative conditional g-formula
- Generalizability / Transportability
- Inverse probability of sampling weights
- G-transport formula
- Doubly-robust transport formula
- Sensitivity analysis tools
- Monte Carlo bias analysis

The website contains pages with example analyses to help demonstrate the usage of this library. Additionally, examples of graphics are displayed. The Reference page contains the full reference documentation for each function currently implemented. For further guided tutorials of the full range of features available in *zEpid*, check out the following Python for Epidemiologists tutorials. Additionally, if you are starting to learn Python, I recommend looking at those tutorials for the basics and some other useful resources.

CHAPTER 1

Contents:

1.1 Causal Graphs

This page demonstrates analysis for causal diagrams (graphs). These diagrams are meant to help identify the sufficient adjustment set to identify the causal effect. Currently only directed acyclic graphs supported by single-world intervention graphs will be added.

Note that this branch requires installation of `NetworkX` since that library is used to analyse the graph objects

1.1.1 Directed Acyclic Graphs

Directed acyclic graphs (DAGs) provide an easy graphical tool to determine sufficient adjustment sets to control for all confounding and identify the causal effect of an exposure on an outcome. DAGs rely on the assumption of d-separation of the exposure and outcome. Currently the `DirectedAcyclicGraph` class only allows for assessing the d-separation of the exposure and outcome. Additional support for checking d-separation between missingness, censoring, mediators, and time-varying exposures will be added in future versions.

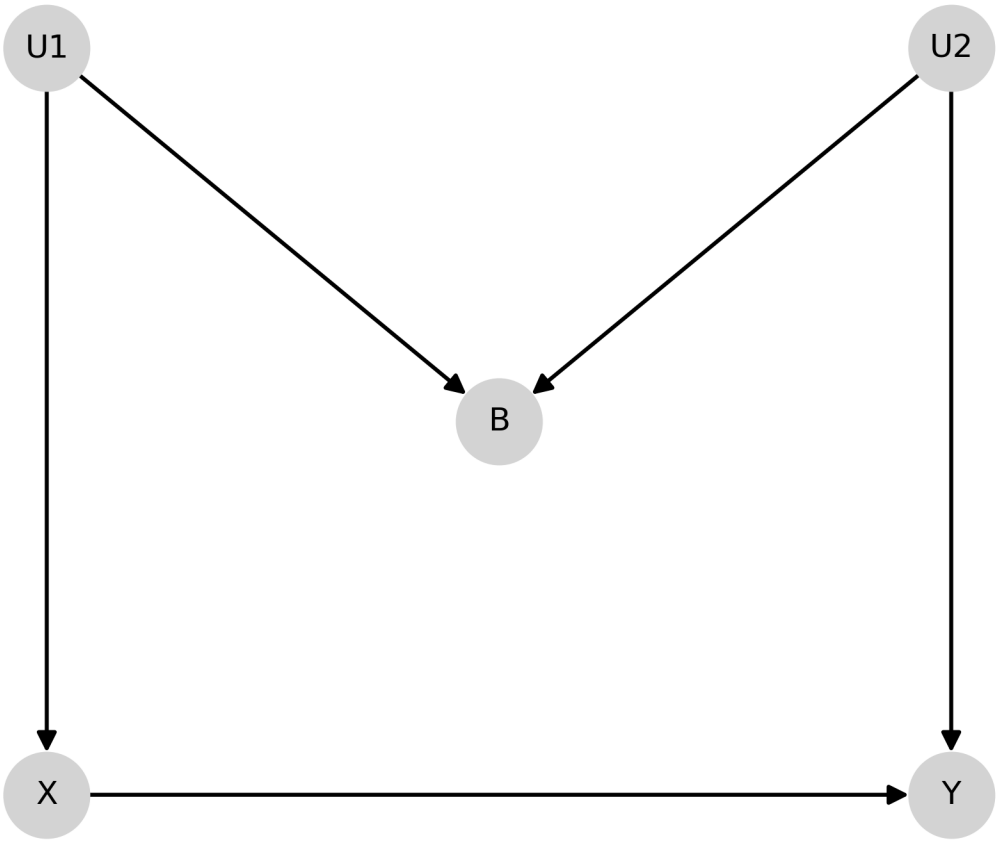
Remember that DAGs should be constructed prior to data collection preferably. Also the major assumptions that a DAG makes is the *lack* of arrows and *lack* of nodes. The assumptions are the items not present within the diagram.

Let's look at some classical examples of DAGs.

M-Bias

First we will create the "M-bias" DAG. This DAG is named after its distinct shape

```
from zepid.causal.causalgraph import DirectedAcyclicGraph
import matplotlib.pyplot as plt

dag = DirectedAcyclicGraph(exposure='X', outcome="Y")
dag.add_arrows((('X', 'Y'),
                ('U1', 'X'), ('U1', 'B'),
                ('U2', 'B'), ('U2', 'Y')
               ))
pos = {"X": [0, 0], "Y": [1, 0], "B": [0.5, 0.5],
       "U1": [0, 1], "U2": [1, 1]}

dag.draw_dag(positions=pos)
plt.tight_layout()
plt.show()
```



After creating the DAG, we can determine the sufficient adjustment set

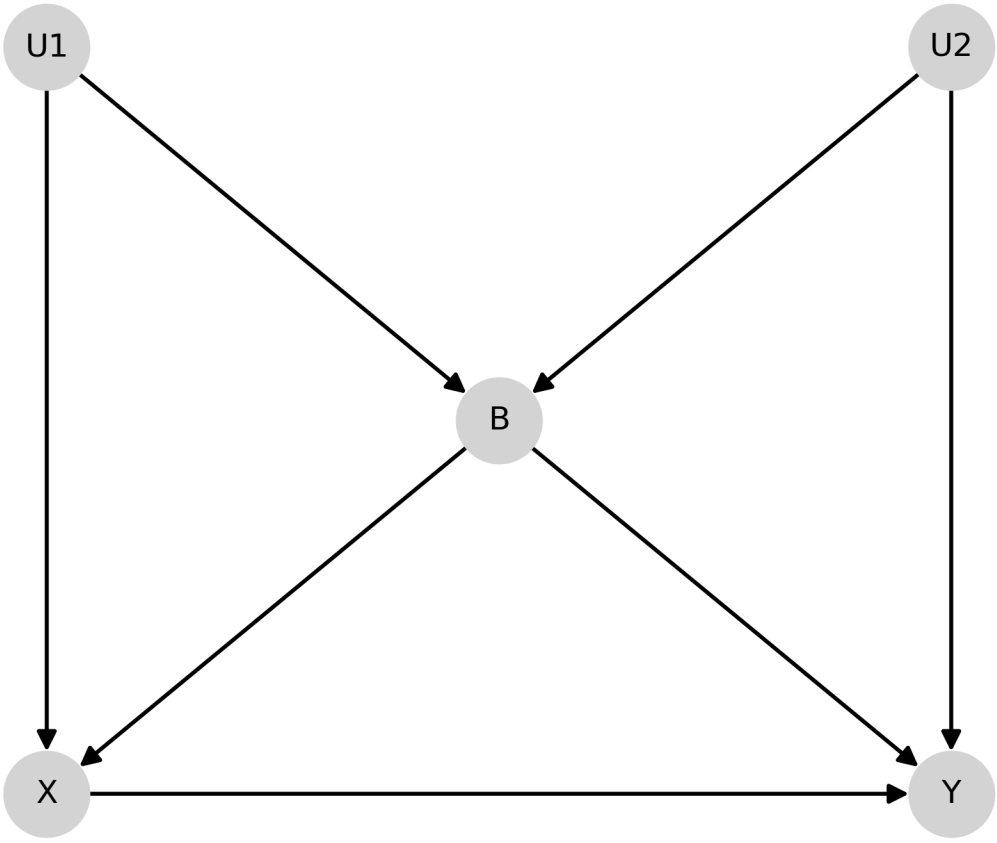
```
dag.calculate_adjustment_sets()
print(dag.adjustment_sets)
```

Since B is a collider, the minimally sufficient adjustment set is the empty set

Butterfly-Bias

Butterfly-bias is an extension of the previous M-bias DAG where we need to adjust for B but B also opens a backdoor path (specifically the path it is a collider on).

```
dag.add_arrows((('X', 'Y'),
                ('U1', 'X'), ('U1', 'B'),
                ('U2', 'B'), ('U2', 'Y'),
                ('B', 'X'), ('B', 'Y')
                ))

dag.draw_dag(positions=pos)
plt.tight_layout()
plt.show()
```



In the case of Butterfly bias, there are 3 possible adjustment sets

```
dag.calculate_adjustment_sets()
print(dag.adjustment_sets)
```

Remember that DAGs should be constructed prior to data collection preferablly. Also the major assumptions that a DAG makes is the *lack* of arrows and *lack* of nodes. The assumptions are the items not present within the diagram

1.2 Time-Fixed Exposure

In this section, we will go through some methods to estimate the average causal effect of a time-fixed treatment / exposure on a specific outcome. We will review binary outcomes, continuous outcomes, and time-to-event data. To follow along with the tutorial, run the following code to set up the data

```
import numpy as np
import pandas as pd
from lifelines import KaplanMeierFitter

from zepid import load_sample_data, spline, RiskDifference
from zepid.causal.gformula import TimeFixedGFormula, SurvivalGFormula
from zepid.causal.ipw import IPTW, IPMW
from zepid.causal.snm import GEstimationSNM
from zepid.causal.doublyrobust import AIPTW, TMLE

df = load_sample_data(timevary=False)
df = df.drop(columns=['cd4_wk45'])
df[['cd4_rs1', 'cd4_rs2']] = spline(df, 'cd40', n_knots=3, term=2, restricted=True)
df[['age_rs1', 'age_rs2']] = spline(df, 'age0', n_knots=3, term=2, restricted=True)
```

1.2.1 Which estimator should I use?

What estimator to use is an important question. Unfortunately, my answer is that it depends. Review the following list of estimators to help you decide. Afterwards, I would recommend the following process.

First, what are you trying to estimate? Depending on what you want to estimate (the estimand), some estimators don't make sense to use. For example, if you wanted to estimate the marginal causal effect comparing all treated versus all untreated, then you wouldn't want to use g-estimation of structural nested models. G-estimation, as detailed below, targets something slightly different than the target estimand. However, if you were interested in average causal effect within defined strata, then g-estimation would be a good choice. Your causal question can (*and should*) narrow down the list of potential estimators

Second, does your question of interest require something not available for all methods? This can also narrow down estimators, at least ones currently available. For example, only *TimeFixedGFormula*, *StochasticIPTW*, and *StochasticTMLE* allow for stochastic treatments. See the tutorials on Python for Epidemiologists for further details on what each estimator can do.

Lastly, if there are multiple estimators to use, then use them all. Each has different advantages/disadvantages that don't necessarily make one unilaterally better than the other. If all the estimators provide similar answers, that can generally be taken as a good sign. It builds some additional confidence in your results. If there are distinctly different results across the estimators, that means that at least one assumption is being substantively broken somewhere. In these situations, I would recommend the doubly robust estimators because they make less restrictive modeling assumptions. Alternatively, machine learning promises to make less restrictive assumptions regarding functional forms. However, the lack of agreement between estimators should be noted.

1.2.2 Binary Outcome

To begin, we are interested in the average causal effect of anti-retroviral therapy (ART) on 45-week risk of death.

$$ACE = \Pr(Y^{a=1}) - \Pr(Y^{a=1})$$

where Y^a indicates the potential outcomes under treatment a. Unfortunately, we cannot observe these potential outcomes (or counterfactuals after they occur). We stuck with our observational data, so we need to make some additional assumptions to go from

$$\Pr(Y|A=1) - \Pr(Y|A=0)$$

to

$$\Pr(Y^{a=1}) - \Pr(Y^{a=1})$$

We will assume conditional mean exchangeability, causal consistency, and positivity. These assumptions allow us to go from our observed data to potential outcomes. See Hernan and Robins for further details on these assumptions and these methods in general. We will assume conditional exchangeability by age (continuous), gender (male / female), baseline CD4 T-cell count (continuous), and baseline detectable viral load (yes / no) throughout. The data set we will use is a simulated data set that comes with *zEpid*

Our set of confounders for conditional exchangeability is quite large and includes some continuous variables. Therefore, we will use parametric models (for the most part). As a result, we assume that our models are correctly specified, in addition to the above assumptions.

Unadjusted Risk Difference

The first option is the unadjusted risk difference. We can calculate this by

```
rd = RiskDifference()
rd.fit(df, exposure='art', outcome='dead')
rd.summary()
```

By using this measure as our average causal effect, we are assuming that there are no confounding variables. However, this is an unreasonable assumption for our observational data. However, the *RiskDifference* gives us some useful information. In the summary, we find *LowerBound* and *UpperBound*. These bounds are the Frechet probability bounds. The true causal effect must be contained within these bounds, without requiring exchangeability. This is a good check. All methods below should produce values that are within these bounds.

Therefore, the Frechet bounds allow for partial identification of the causal effect. We narrowed the range of possible values from two unit width (-1 to 1) to unit width (-0.87 to 0.13). However, we don't have point identification. The following methods allow for point identification under the assumption of conditional exchangeability.

Our unadjusted estimate is -0.05 (-0.13, 0.04), which we could interpret as: ART is associated with a 4.5% point reduction (95% CL: -0.13, 0.04) in the probability of death at 45-weeks. However, this interpretation implies that ART is given randomly (which is unlikely to occur in the data).

Parametric g-formula

The parametric g-formula allows us to estimate the average causal effect of ART on death by specifying an outcome model. From our outcome model, we predict individuals' counterfactual outcomes under our treatment plans and marginalize over these predicted counterfactuals. This allows us to estimate the marginal risk under our treatment plan of interest.

To estimate the parametric g-formula, we can use the following code

```
g = TimeFixedGFormula(df, exposure='art', outcome='dead')
g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')

# Estimating marginal effect under treat-all plan
g.fit(treatment='all')
r_all = g.marginal_outcome

# Estimating marginal effect under treat-none plan
g.fit(treatment='none')
r_none = g.marginal_outcome

riskd = r_all - r_none
print('RD:', riskd)
```

which gives us an estimated risk difference of -0.076. To calculate confidence intervals, we need to use a bootstrapping procedure. Below is an example that uses bootstrapped confidence limits.

```
rd_results = []
for i in range(1000):
    s = dfs.sample(n=df.shape[0],replace=True)
    g = TimeFixedGFormula(s,exposure='art',outcome='dead')
    g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0',
                    print_results=False)
    g.fit(treatment='all')
    r_all = g.marginal_outcome
    g.fit(treatment='none')
    r_none = g.marginal_outcome
    rd_results.append(r_all - r_none)

se = np.std(rd_results)
print('95% LCL', riskd - 1.96*se)
print('95% UCL', riskd + 1.96*se)
```

In my run (your results may differ), the estimate's 95% confidence limits were -0.15, 0.00. We could interpret our results as; the 45-week risk of death when everyone was treated with ART at enrollment was 7.6% points (95% CL: -0.15, -0.00) lower than if no one had been treated with ART at enrollment. For further details and examples of other usage of this estimator see this tutorial

Inverse probability of treatment weights

For the g-formula, we specified the outcome model. Another option is to specify a treatment / exposure model. Specifically, this model predicts the probability of treatment, sometimes called propensity scores. From these propensity scores, we can calculate inverse probability of treatment weights.

Below is some code to calculate our stabilized inverse probability of treatment weights for ART.

```
iptw = IPTW(df, treatment='art')
iptw.treatment_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0',
                     print_results=False)
```

A variety of diagnostics is available to check the calculated weights. See the below referenced tutorial for further details and examples. For our analysis, we use the following marginal structural model

$$\Pr(Y|A) = \alpha_0 + \alpha_1 A$$

While this model looks like a crude regression model, we are fitting it with the weighted data. The weights make it such that there is no confounding in our pseudo-population. As of v0.8.0, *IPTW* now estimates the marginal structural model for you. GEE is used to estimate the standard error. Robust standard errors are required since weighting our population builds in some correlation between our observations. We need to account for this. While GEE does account for this, our confidence intervals will be somewhat conservative. Below is code to estimate the marginal structural model and print the results

```
iptw.marginal_structural_model('art')
iptw.fit()
iptw.summary()
```

My results were fairly similar to the g-formula (RD = -0.08; 95% CL: -0.16, -0.01). We would interpret this in a similar way: the 45-week risk of death when everyone was treated with ART at enrollment was 8.2% points (95% CL: -0.16, -0.01) lower than if no one had been treated with ART at enrollment.

To account for data that is missing at random, inverse probability of missing weights can be stacked together with IPTW. As of v0.8.0, this is built into the *IPTW* class. Below is an example with accounting for informative censoring (missing outcome data)

When accounting for censoring by the above variables, a similar is obtained (RD = -0.08, 95% CL: -0.16, -0.01). For further details and examples of other usage of this estimator see this tutorial

Augmented inverse probability weights

As you read through the previous estimators, you may have thought "is there a way to combine these approaches?" The answer is yes! Augmented inverse probability of treatment weights require you to specify both a treatment model (pi-model) and an outcome model (Q-model). But why would you want to specify two models? Well, by specifying both and merging them, AIPTW becomes doubly robust. This means that as long as one model is correct, our estimate will be unbiased on average. Essentially, we get two attempts to get our models correct.

We can calculate the AIPTW estimator through the following code

```
aipw = AIPTW(df, exposure='art', outcome='dead')

# Treatment model
aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')

# Outcome model
aipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2␣
↪+ dvl0')

# Calculating estimate
aipw.fit()

# Printing summary results
aipw.summary()
```

In the printed results, we have an estimated risk difference of -0.08 (95% CL: -0.15, -0.02). Confidence intervals come from the efficient influence curve. You can also bootstrap confidence intervals. For the risk ratio, you will need to bootstrap the confidence intervals currently. Our results can be interpreted as: the 45-week risk of death when everyone was treated with ART at enrollment was 8.4% points (95% CL: -0.15, -0.02) lower than if no one had been treated with ART at enrollment.

Similarly, we can also account for missing outcome data using inverse probability weights. Below is an example

```
aipw = AIPTW(df, exposure='art', outcome='dead')
aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
aipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2␣
↪+ dvl0')
aipw.missing_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2␣
↪+ dvl0')
aipw.fit()
aipw.summary()
```

AIPTW can also be paired with machine learning algorithms, particularly super-learner. The use of machine learning with AIPTW means we are making less restrictive parametric assumptions than all the model described above. For further details, using super-learner / sklearn with AIPTW, and examples see this tutorial

Targeted maximum likelihood estimation

For AIPTW, we merged IPW and the g-formula. The targeted maximum likelihood estimator (TMLE) is another variation on this procedure. TMLE uses a targeting step to update the estimate of the average causal effect. This approach is doubly robust but keeps some of the nice properties of plug-in estimators (like the g-formula). In general, TMLE will likely have narrower confidence intervals than AIPTW.

Below is code to generate the average causal effect of ART on death using TMLE. Additionally, we will specify a missing outcome data model (like *AIPTW* and *IPTW*).

```
tmle = TMLE(df, exposure='art', outcome='dead')
tmle.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
tmle.missing_model('art + male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
tmle.outcome_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
tmle.fit()
tmle.summary()
```

Using TMLE, we estimate a risk difference of -0.08 (95% CL: -0.15, -0.01). We can interpret this as: the 45-week risk of death when everyone was treated with ART at enrollment was 8.3% points (95% CL: -0.15, -0.01) lower than if no one had been treated with ART at enrollment.

TMLE can also be paired with machine learning algorithms, particularly super-learner. The use of machine learning with TMLE means we are making less restrictive parametric assumptions than all the model described above. For further details, using super-learner / sklearn with TMLE, and examples see this tutorial

Single Cross-fit TMLE

While both AIPTW and TMLE are able to incorporate the use of *some* machine learning algorithms, there are limits. More specifically, both require that the machine learning algorithms are Donsker. Unfortunately, many flexible algorithms we may want to use may not be Donsker. In this scenario, confidence interval coverage may be below what is expected (i.e. the confidence interval are overly narrow due to over-fitting by the machine learning algorithms).

Recently, cross-fitting procedures have been proposed as a way to weaken this condition. Cross-fitting allows for non-Donsker algorithms. For more extensive details on the cross-fitting procedure and why it is necessary, please see my paper and the references within.

zEpid supports both single and double cross-fitting for AIPTW and TMLE. The following are simple examples that use *SuperLearner* with a single cross-fitting procedure for TMLE. The 10-fold super-learner consists of a GLM, a step-wise GLM with all first-order interactions, and a Random Forest.

```
from sklearn.ensemble import RandomForestClassifier
from zepid.superlearner import GLMSL, StepwiseSL, SuperLearner
from zepid.causal.doublyrobust import SingleCrossfitAIPTW, SingleCrossfitTMLE

# SuperLearner setup
labels = ["LogR", "Step.int", "RandFor"]
candidates = [GLMSL(sm.families.family.Binomial()),
              StepwiseSL(sm.families.family.Binomial(), selection="forward", order_
↪interaction=0),
              RandomForestClassifier()]

# Single cross-fit TMLE
sctmle = SingleCrossfitTMLE(df, exposure='art', outcome='dead')
sctmle.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0',
                      SuperLearner(candidates, labels, folds=10, loss_function=
↪"nloglik"),
                      bound=0.01)
sctmle.outcome_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0',
                     SuperLearner(candidates, labels, folds=10, loss_function="nloglik
↪"))
sctmle.fit()
sctmle.summary()
```

Using *SingleCrossfitTMLE*, we estimate a risk difference of -0.08 (95% CL: -0.17, 0.00). We can interpret this as: the 45-week risk of death when everyone was treated with ART at enrollment was 8.3% points (95% CL: -0.17, 0.00) lower than if no one had been treated with ART at enrollment. When comparing SingleCrossfitTMLE to the previous TMLE, you can see the confidence intervals are wider. This is a result of weakening the parametric modeling restrictions (by including the random forest as a possible option in super learner).

As these are new procedures, guidelines on their use are still developing. In my experience, I would recommend at least 100 different partitions to be used. Additionally, the data set must be fairly large (more than 500 observations) to take advantage of the flexibility of the cross-fit estimators with machine learning. If the data set is not that large, I recommend using a higher number of folds with SuperLearner (if using), using single cross-fitting, and using the minimal number of required splits.

G-estimation of SNM

The final method I will review is g-estimation of structural nested mean models (SNM). G-estimation of SNM is distinct from all of the above estimation procedures. The g-formula, IPTW, AIPTW, and TMLE all estimated the average causal effect of ART on mortality comparing everyone treated to everyone untreated. G-estimation of SNM estimate the average causal effect within levels of the confounders, *not* the average causal effect in the population. Therefore, if no product terms are included in the SNM if there is effect measure modification, then the SNM will be biased due to model misspecification. SNM are useful for learning about effect modification.

To first demonstrate g-estimation, we will assume there is no effect measure modification. For g-estimation, we specify two models; the treatment model and the structural nested model. The treatment model is the same format as the treatment model for IPTW / AIPTW / TMLE. The structural nested model states the interaction effects we are interested in. Since we are assuming no interaction, we only put the treatment variable into the model.

```
snm = GEstimationSNM(df, exposure='art', outcome='dead')

# Specify treatment model
snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
```

(continues on next page)

(continued from previous page)

```
# Specify structural nested model
snm.structural_nested_model('art')

# G-estimation
snm.fit()
snm.summary()

psi = snm.psi
print('Psi:', psi)
```

Similarly, we need to bootstrap our confidence intervals

```
psi_results = []
for i in range(500):
    dfs = df.sample(n=df.shape[0],replace=True)
    snm = GEstimationSNM(dfs, exposure='art', outcome='dead')
    snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0', print_results=False)
    snm.structural_nested_model('art')
    snm.fit()
    psi_results.append(snm.psi)


se = np.std(psi_results)
print('95% LCL', psi - 1.96*se)
print('95% UCL', psi + 1.96*se)
```

Overall, the SNM results are similar to the other models (RD = -0.09; 95% CL: -0.17, -0.00). Instead, we interpret this estimate as: the 45-week risk of death when everyone was treated with ART at enrollment was 8.8% points (95% CL: -0.17, -0.00) lower than if no one had been treated with ART at enrollment across all strata.

SNM can be expanded to include additional terms. Below is code to do that. For this SNM, we will assess if there is modification by gender

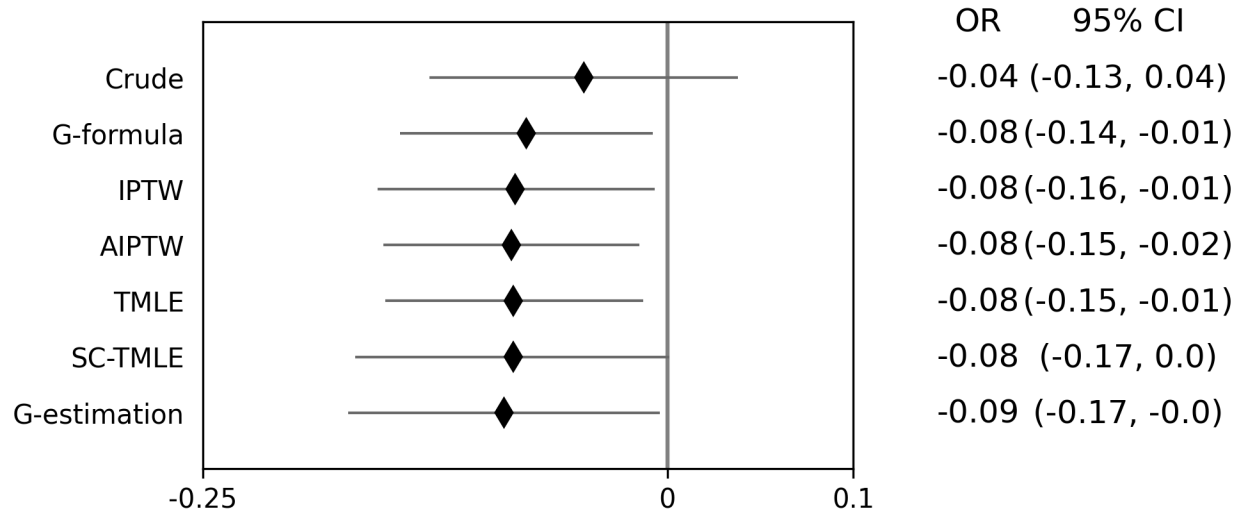
```
snm = GEstimationSNM(df, exposure='art', outcome='dead')
snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
snm.structural_nested_model('art + art:male')
snm.fit()
snm.summary()
```

The 45-week risk of death when everyone was treated with ART at enrollment was 17.6% points lower than if no one had been treated with ART at enrollment, *among women*. Among men, risk of death with ART treatment at enrollment was 6.8% points lower compared to no treatment.

Remember, g-estimation of SNM is distinct from these other methods and targets a different estimand. It is a great method to consider when you are interested in effect measure modification.

Summary

Below is a figure summarizing the results across methods.



As we can see, all the methods provided fairly similar answers, even the misspecified structural nested model. This will not always be the case. Differences in model results may indicate parametric model misspecification. In those scenarios, it may be preferable to use a doubly-robust estimator with machine learning and cross-fitting (when possible).

Additionally, for simplicity we dropped all missing outcome data. We made the assumption that outcome data was missing completely at random, a strong assumption. We could relax this assumption using built-in methods (e.g. *missing_model()* functions)

1.2.3 Continuous Outcome

In the previous example we focused on a binary outcome, death. In this example, we will repeat the above procedure but focus on the 45-week CD4 T-cell count. This can be expressed as

$$E[Y^{a=1}] - E[Y^{a=0}]$$

For illustrative purposes, we will ignore the implications of competing risks (those dying before week 45 cannot have a CD4 T-cell count). We will start by restricting our data to only those who are not missing a week 45 T-cell count. In an actual analysis, you wouldn't want to do this

```
df = load_sample_data(timevary=False)
dfs = df.drop(columns=['dead']).dropna()
```

With our data loaded and restricted, let's compare the estimators. Overall, the estimators are pretty much the same as the binary case. However, we are interested in estimating the average treatment effect instead. Most of the methods auto-detect binary or continuous data in the background. Additionally, we will assume that CD4 T-cell count is appropriately fit by a normal-distribution. Poisson is also available.

Parametric g-formula

The parametric g-formula allows us to estimate the average causal effect of ART on death by specifying an outcome model. From our outcome model, we predict individuals' counterfactual outcomes under our treatment plans and marginalize over these predicted counterfactuals. This allows us to estimate the marginal risk under our treatment plan of interest.

To estimate the parametric g-formula, we can use the following code

```
g = TimeFixedGFormula(df, exposure='art', outcome='cd4_wk45', outcome_type='normal')
g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
g.fit(treatment='all')
r_all = g.marginal_outcome

g.fit(treatment='none')
r_none = g.marginal_outcome
ate = r_all - r_none

print('ATE:', ate)
```

To calculate confidence intervals, we need to use a bootstrapping procedure. Below is an example that uses bootstrapped confidence limits.

```
ate_results = []
for i in range(1000):
    s = df.sample(n=df.shape[0],replace=True)
    g = TimeFixedGFormula(s,exposure='art',outcome='cd4_wk45', outcome_type='normal')
    g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0',
                    print_results=False)
    g.fit(treatment='all')
    r_all = g.marginal_outcome
    g.fit(treatment='none')
    r_none = g.marginal_outcome
    ate_results.append(r_all - r_none)

se = np.std(ate_results)
print('95% LCL', ate - 1.96*se)
print('95% UCL', ate + 1.96*se)
```

In my run (your results may differ), the estimate's 95% confidence limits were 158.70, 370.54. We can interpret this estimate as: the mean 45-week CD4 T-cell count if everyone had been given ART at enrollment was 264.62 (95% CL: 158.70, 370.54) higher than the mean if everyone has not been given ART at baseline.

Inverse probability of treatment weights

Since inverse probability of treatment weights rely on specification of the treatment-model, there is no difference between the weight calculation and the binary outcome. This is also because we assume the same sufficient adjustment set. We will estimate new weights since there is a different missing data pattern. Below is code to estimate our weights

```
ipw = IPTW(df, treatment='art')
ipw.treatment_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
ipw.marginal_structural_model('art')
ipw.fit()
ipw.summary()
```

Our marginal structural model estimates 222.56 (95% CL: 114.67, 330.46). We can interpret this estimate as: the mean 45-week CD4 T-cell count if everyone had been given ART at enrollment was 222.56 (95% CL: 114.67, 330.46) higher than the mean if everyone has not been given ART at baseline.

Augmented inverse probability weights

Similarly to the binary outcome case, AIPTW follows the same recipe to merge IPTW and g-formula estimates. We can calculate the AIPTW estimator through the following code

```
aipw = AIPTW(df, exposure='art', outcome='cd4_wk45')
aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
aipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2␣
↪+ dvl0')
aipw.fit()
aipw.summary()
```

AIPTW produces a similar estimate to the marginal structural model (ATE = 228.22; 95% CL: 115.33, 341.11). We can interpret this estimate as: the mean 45-week CD4 T-cell count if everyone had been given ART at enrollment was 228.22 (95% CL: 115.33, 341.11) higher than the mean if everyone has not been given ART at baseline.

Targeted maximum likelihood estimation

TMLE also supports continuous outcomes and is similarly doubly robust. Below is code to estimate TMLE for a continuous outcome.

```
tmle = TMLE(df, exposure='art', outcome='cd4_wk45')
tmle.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
tmle.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2␣
↪+ dvl0')
tmle.fit()
tmle.summary()
```

Our results are fairly similar to the other models. The mean 45-week CD4 T-cell count if everyone had been given ART at enrollment was 228.35 (95% CL: 118.97, 337.72) higher than the mean if everyone has not been given ART at baseline.

Single Cross-fit TMLE

Similarly, we can pair TMLE with a cross-fitting procedure and machine learning. In this example, we use Super-Learner with a GLM, a stepwise selection, and a random forest.

```
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor

# SuperLearner set-up
labels = ["LogR", "Step.int", "RandFor"]
b_candidates = [GLMSL(sm.families.family.Binomial()),
                StepwiseSL(sm.families.family.Binomial(), selection="forward", order_
↪interaction=0),
                RandomForestClassifier(random_state=809512)]
c_candidates = [GLMSL(sm.families.family.Gaussian()),
                StepwiseSL(sm.families.family.Gaussian(), selection="forward", order_
↪interaction=0),
                RandomForestRegressor(random_state=809512)]

# Single cross-fit TMLE
sctmle = SingleCrossfitTMLE(df, exposure='art', outcome='cd4_wk45')
sctmle.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0',
```

(continues on next page)

(continued from previous page)

```
                        SuperLearner(b_candidates, labels, folds=10, loss_function=
↪"nloglik"),
                        bound=0.01)
sctmle.outcome_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0',
                     SuperLearner(c_candidates, labels, folds=10))
sctmle.fit(n_partitions=3, random_state=201820)
sctmle.summary()
```

The mean 45-week CD4 T-cell count if everyone had been given ART at enrollment was 176.9 (95% CL: -37.7, 391.5) higher than the mean if everyone has not been given ART at baseline.

The point estimate is similar to other approaches, but the confidence intervals are substantially wider. This is likely a result of the random forest dominating super-learner and being somewhat dependent on the particular split. This is the penalty of weaker modeling assumptions (or rather it showcases the undue confidence that results from assuming that our particular parametric model is sufficient in other estimators).

G-estimation of SNM

Recall that g-estimation of SNM estimates the average causal effect within levels of the confounders, *not* the average causal effect in the population. Therefore, if no product terms are included in the SNM if there is effect measure modification, then the SNM will be biased due to model misspecification.

For illustrative purposes, I will specify a one-parameter SNM. Below is code to estimate the model

```
snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
snm.structural_nested_model('art')
snm.fit()
snm.summary()
```

Overall, the SNM results are similar to the other models (ATE = 227.2). Instead, we interpret this estimate as: the mean 45-week CD T-cell count when everyone was treated with ART at enrollment was 227.2 higher (95% CL: 134.2, 320.2) than if no one had been treated with ART at enrollment across all strata.

SNM can be expanded to include additional terms. Below is code to do that. For this SNM, we will assess if there is modification by gender

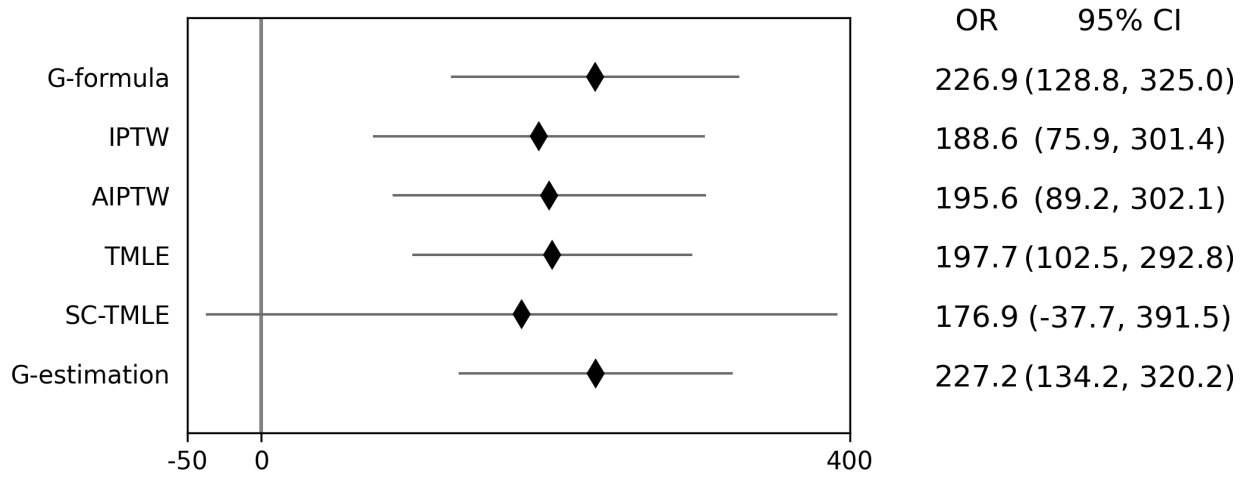
```
snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0
↪')
snm.structural_nested_model('art + art:male')
snm.fit()
snm.summary()
```

The mean 45-week CD4 T-cell count when everyone was treated with ART at enrollment was 277.1 higher than if no one had been treated with ART at enrollment, *among women*. Among men, CD4 T-cell count with ART treatment at enrollment was 213.8 higher compared to no treatment.

Remember, g-estimation of SNM is distinct from these other methods and targets a different estimand. It is a great method to consider when you are interested in effect measure modification.

Summary

Below is a figure summarizing the results across methods.



There was some difference in results between outcome models and treatment models. Specifically, the g-formula and IPTW differ. AIPTW and TMLE are similar to IPTW. This may indicate substantive misspecification of the outcome model. This highlights why you may consider using multiple models.

Additionally, for simplicity we dropped all missing outcome data. We made the assumption that outcome data was missing complete at random, a strong assumption. We could relax this assumption by pairing the above methods with inverse-probability-of-missing-weights or using built-in methods (like *TMLE*'s *missing_model*)

1.2.4 Causal Survival Analysis

Previously, we focused on the risk of death at 45-weeks. However, we may be interested in conducting a time-to-event analysis. For the following methods, we will focus on treatment at baseline. Specifically, we will not allow the treatment to vary over time. For methods that allow for time-varying treatment, see the tutorial for time-varying exposures.

For the following analysis, we are interested in the average causal effect of ART treatment at baseline compared to no treatment. We will compare the parametric g-formula and IPTW. The parametric g-formula is further described in Hernan's "The hazards of hazard ratio" paper. For the analysis in this section, we will get a little help from the *lifelines* library. It is a great library with a variety of survival models and procedures. We will use the *KaplanMeierFitter* function to estimate risk function

Parametric g-formula

We can use a similar g-formula procedure to estimate average causal effects with time-to-event data. To do this, we use a pooled logistic model. We then use the pooled logistic regression model to predict outcomes at each time under the treatment strategy of interest. For the pooled logistic model, it is fit to data in a long format, where each row corresponds to one unit of time per participant. There will be multiple rows per participant.

For *SurvivalGFormula*, we need to convert the data set into a long format. We can do that with the following code

```
df = load_sample_data(False).drop(columns=['cd4_wk45'])
df['t'] = np.round(df['t']).astype(int)
df = pd.DataFrame(np.repeat(df.values, df['t'], axis=0), columns=df.columns)
df['t'] = df.groupby('id')['t'].cumcount() + 1
df.loc[((df['dead'] == 1) & (df['id'] != df['id'].shift(-1))), 'd'] = 1
df['d'] = df['d'].fillna(0)
```

(continues on next page)

(continued from previous page)

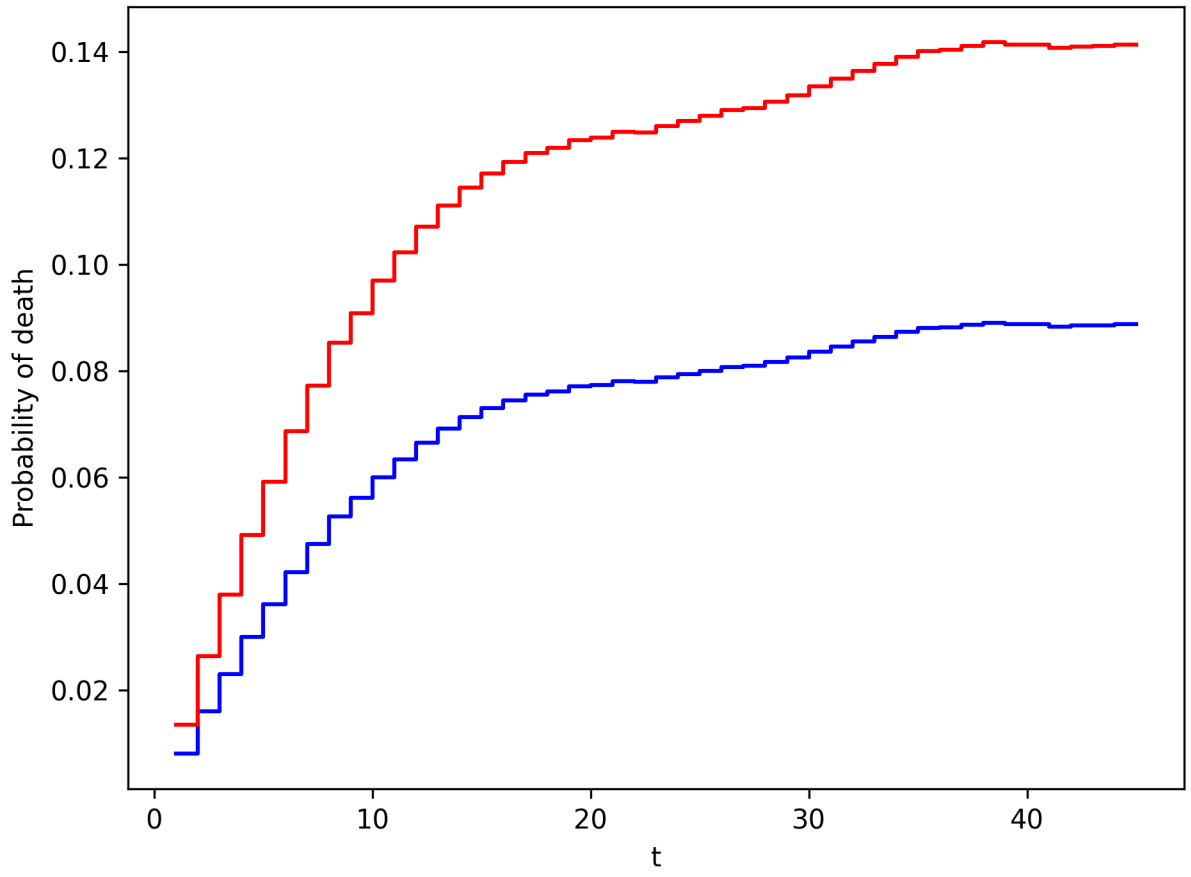
```
# Spline terms
df[['t_rs1', 't_rs2', 't_rs3']] = spline(df, 't', n_knots=4, term=2, restricted=True)
df[['cd4_rs1', 'cd4_rs2']] = spline(df, 'cd40', n_knots=3, term=2, restricted=True)
df[['age_rs1', 'age_rs2']] = spline(df, 'age0', n_knots=3, term=2, restricted=True)
```

If you look at this data, you will notice there are multiple rows per participant. Each row for a participant corresponds to one unit of time (weeks in this example) up to the event time or 45-weeks. All variables (aside from time and outcomes) take the same value over follow-up. This is because we are interested in the baseline exposure. We then adjust for all baseline confounders. Nothing should be time-varying in this model (aside from the outcome and time).

We can estimate the average causal effect comparing a treat-all plan versus a treat-none. Below is code to estimate the time-to-event g-formula

```
sgf = SurvivalGFormula(df.drop(columns=['dead']), idvar='id', exposure='art', outcome=
↪'d', time='t')
sgf.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + cd40 + '
                        'cd4_rs1 + cd4_rs2 + dvl0 + t + t_rs1 + t_rs2 + t_rs3')
sgf.fit(treatment='all')
sgf.plot(c='b')

sgf.fit(treatment='none')
sgf.plot(c='r')
plt.ylabel('Probability of death')
plt.show()
```

The plot functionality will return the following plot of the cumulative incidence function



We see that ART reduces mortality throughout follow-up

Inverse probability of treatment weights

A new estimator, *SurvivalIPTW* will soon be implemented and available to estimate IPTW-adjusted survival curves.

Summary

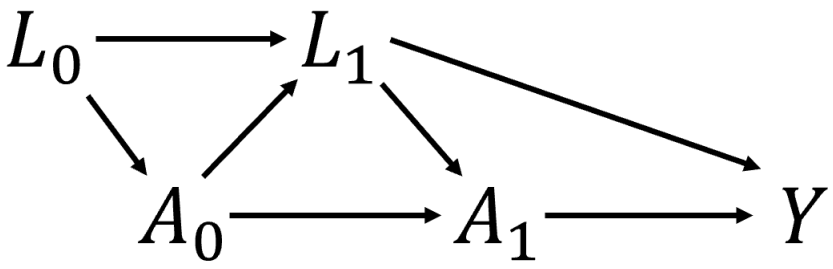
Currently, only these two options are available. I plan on adding further functionalities in future updates

The difference in these results highlight the differences between the approaches. The g-formula makes some strong parametric assumptions, but smooths over sparse data. IPTW uses the observed data, so it is more sensitive to sparse data. IPTW particularly highlights why we might consider using methods to handle time-varying treatments. Particularly, if few participants are treated at baseline, then we may have trouble estimating the average causal effect. Please refer to the *Time-Varying Treatment* tutorial for further discussion.

1.3 Time-Varying Exposure

In this section, we will go through some methods to estimate the average causal effect of a time-varying treatment / exposure on a specific outcome. The key problem we must overcome is time-varying confounding. Time-varying confounders are both a mediator and a confounder, depending on the causal path. As such, conditional models will not correctly estimate the causal effects with time-varying confounders. We need to use special methods to account for time-varying confounding. We will focus on time-to-event and longitudinal data separately.

To help solidify understanding, consider the following causal diagram, where subscripts are used to indicate time



If we are interested in the effect of A (*not only* A_0), then we need to account for confounding variables. L_0 is easy, since we know we can condition on this variable safely. The problem comes with L_1. On the A_0 causal path to Y, L_1 is a mediator. However, on the A_1 to Y causal path, L_1 is a confounder. Therefore, we need to simultaneously condition on L_1, while also not condition on it. We can't do that, so we are damned if we do and damned if we don't.

Not all hope is lost. James Robins developed his "g-methods" (g-formula, IPTW, g-estimation) for this exact problem. These methods allows us to account for confounding by L_1, but do not require conditioning on any variables. Instead the g-methods provide marginal estimates rather than conditional. I introduced g-methods in the baseline exposure setting, but time-varying exposure is where these methods really shine.

We will assume conditional mean exchangeability, causal consistency, and positivity throughout. These assumptions allow us to go from our observed data to potential outcomes. See Hernan and Robins for further details on these assumptions and the g-methods in general.

This section is divided into two scenarios; time-to-event and longitudinal data. For time-to-event, I mean that we have data collected on the exact time of the event. For the g-methods, we will coarsen this data to discrete time, but this is only necessary since we have finite data. As for longitudinal, I mean that our input data is already coarsened. The data comes from follow-ups at constant intervals. The event at the follow-up visit happened some time between the previous visit and the current visit. I draw this distinction, since some approaches for estimation work better in one scenario over the other.

1.3.1 Time-to-Event Data

We will start with estimating the average causal effect of ART on mortality, assuming that once someone is treated with ART, they remain on treatment (I will refer to as the intent-to-treat assumption in this tutorial). We will set up the environment with the following code

```
import numpy as np
import pandas as pd
from lifelines import KaplanMeierFitter

from zepid import load_sample_data, spline
```

(continues on next page)

(continued from previous page)

```
from zepid.causal.gformula import MonteCarloGFormula
from zepid.causal.ipw import IPTW, IPCW

df = load_sample_data(timevary=True)

# Background variable preparations
df['lag_art'] = df['art'].shift(1)
df['lag_art'] = np.where(df.groupby('id').cumcount() == 0, 0, df['lag_art'])
df['lag_cd4'] = df['cd4'].shift(1)
df['lag_cd4'] = np.where(df.groupby('id').cumcount() == 0, df['cd40'], df['lag_cd4'])
df['lag_dvl'] = df['dvl'].shift(1)
df['lag_dvl'] = np.where(df.groupby('id').cumcount() == 0, df['dvl0'], df['lag_dvl'])
df[['age_rs0', 'age_rs1', 'age_rs2']] = spline(df, 'age0', n_knots=4, term=2,␣
↪restricted=True)  # age spline
df['cd40_sq'] = df['cd40'] ** 2  # cd4 baseline cubic
df['cd40_cu'] = df['cd40'] ** 3
df['cd4_sq'] = df['cd4'] ** 2  # cd4 current cubic
df['cd4_cu'] = df['cd4'] ** 3
df['enter_sq'] = df['enter'] ** 2  # entry time cubic
df['enter_cu'] = df['enter'] ** 3
```

We will assume conditional exchangeability by age (continuous), gender (male / female), baseline CD4 T-cell count (continuous), and baseline detectable viral load (yes / no), CD4 T-cell count (continuous), detectable viral load (yes / no), and previous ART treatment. CD4 T-cell count and detectable viral load are time-varying confounders in this example.

Our set of confounders for conditional exchangeability is quite large and includes some continuous variables. Therefore, we will use parametric models (for the most part). As a result, we assume that our models are correctly specified, in addition to the above assumptions.

Monte-Carlo g-formula

The first option is to use the Monte-Carlo g-formula. This approach works by estimating pooled logistic regression models for each time-varying variable (treatment, outcome, time-varying confounding). We then sample the population from baseline and predict individuals' time-varying variables going forward in time. We use Monte Carlo re-sampling to reduce simulation error of the outcomes.

To begin, we initialize the Monte-Carlo g-formula with

```
mcgf = MonteCarloGFormula(df,  # Data set
                          idvar='id',  # ID variable
                          exposure='art',  # Exposure
                          outcome='dead',  # Outcome
                          time_in='enter',  # Start of study period
                          time_out='out')  # End of time per study period
```

We then specify models for each of our time-varying variables (ART, death, CD4 T-cell count, detectable viral load). Additionally, we will specify a model for censoring. Below is example code for this procedure

```
# Pooled Logistic Model: Treatment
exp_m = ('male + age0 + age_rs0 + age_rs1 + age_rs2 + cd40 + cd40_sq + cd40_cu + dvl0␣
↪+ '
         'cd4 + cd4_sq + cd4_cu + dvl + enter + enter_sq + enter_cu')
mcgf.exposure_model(exp_m,
                    restriction="g['lag_art']==0")  # Restricts to only untreated␣
↪(for ITT assumption)
```

(continues on next page)

(continued from previous page)

```
# Pooled Logistic Model: Outcome
out_m = ('art + male + age0 + age_rs0 + age_rs1 + age_rs2 + cd40 + cd40_sq + cd40_cu␣
↪+ dvl0 + '
         'cd4 + cd4_sq + cd4_cu + dvl + enter + enter_sq + enter_cu')
mcgf.outcome_model(out_m,
                   restriction="g['drop']==0")  # Restricting to only uncensored␣
↪individuals

# Pooled Logistic Model: Detectable viral load
dvl_m = ('male + age0 + age_rs0 + age_rs1 + age_rs2 + cd40 + cd40_sq + cd40_cu + dvl0␣
↪+ '
         'lag_cd4 + lag_dvl + lag_art + enter + enter_sq + enter_cu')
mcgf.add_covariate_model(label=1,  # Order to fit time-varying models in
                         covariate='dvl',  # Time-varying confounder
                         model=dvl_m,
                         var_type='binary')  # Variable type

# Pooled Logistic Model: CD4 T-cell count
cd4_m = ('male + age0 + age_rs0 + age_rs1 + age_rs2 +  cd40 + cd40_sq + cd40_cu +␣
↪dvl0 + lag_cd4 + '
         'lag_dvl + lag_art + enter + enter_sq + enter_cu')
cd4_recode_scheme = ("g['cd4'] = np.maximum(g['cd4'], 1);"
                     "g['cd4_sq'] = g['cd4']**2;"
                     "g['cd4_cu'] = g['cd4']**3")
mcgf.add_covariate_model(label=2,  # Order to fit time-varying models in
                         covariate='cd4',  # Time-varying confounder
                         model=cd4_m,
                         recode=cd4_recode_scheme,  # Recoding process to use for␣
↪each iteraction of MCMC
                         var_type='continuous')  # Variable type

# Pooled Logistic Model: Censoring
cens_m = ("male + age0 + age_rs0 + age_rs1 + age_rs2 +  cd40 + cd40_sq + cd40_cu +␣
↪dvl0 + lag_cd4 + " +
          "lag_dvl + lag_art + enter + enter_sq + enter_cu")
mcgf.censoring_model(cens_m)
```

After our models are specified, we can now predict the outcome plans under our treatment plan. To start, we will compare to the natural course. The natural course is the world observed as it is. Since we are relying on the ITT assumption, we will use the custom treatment option to fit the natural course. Below is code to estimate the natural course under the ITT assumption

```
mcgf.fit(treatment="((g['art']==1) | (g['lag_art']==1))",  # Treatment plan
         lags={'art': 'lag_art',  # Lagged variables to create each loop
               'cd4': 'lag_cd4',
               'dvl': 'lag_dvl'},
         in_recode=("g['enter_sq'] = g['enter']**2;"  # Recode statement to execute␣
↪at the start
                    "g['enter_cu'] = g['enter']**3"),
         sample=20000)  # Number of resamples from population (should be large number)
```

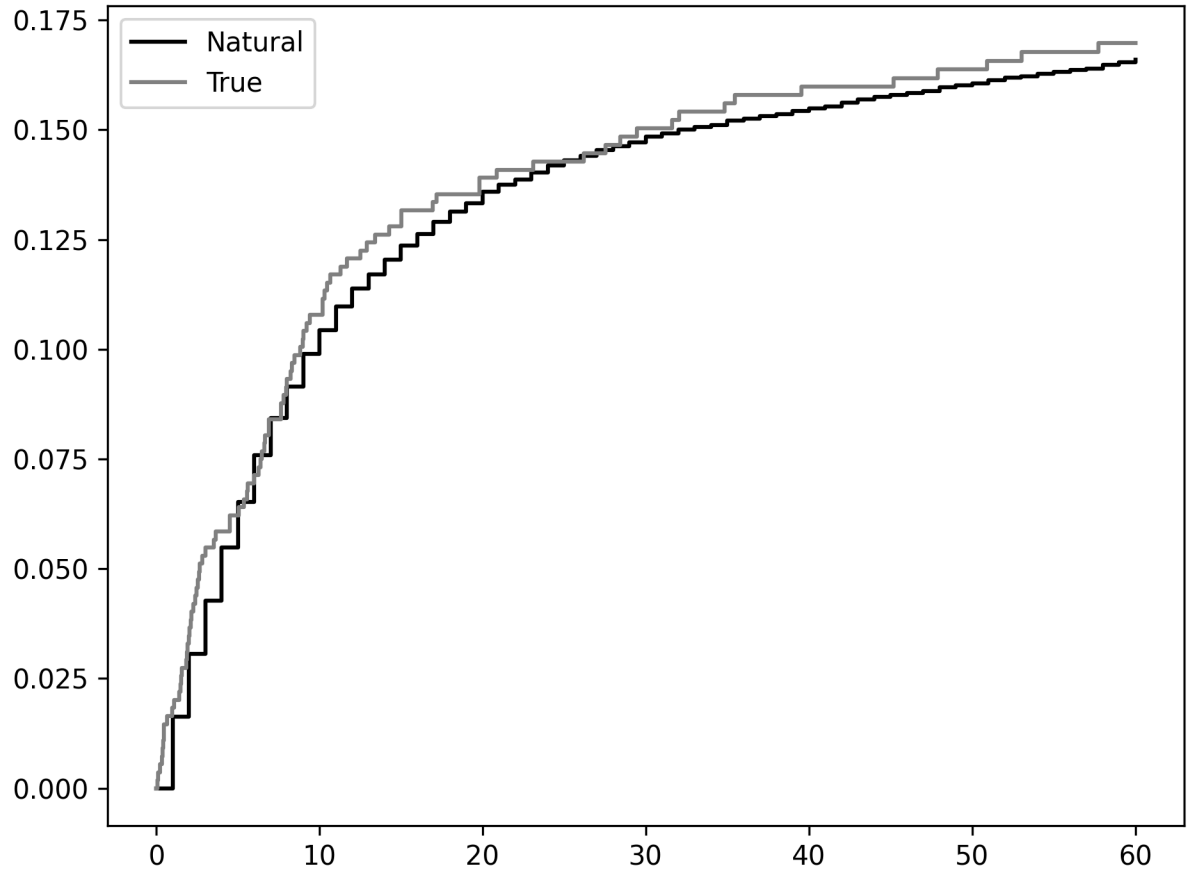
Afterwards, we can generate a plot of the risk curves.

```
# Accessing predicted outcome values
gf = mcgf.predicted_outcomes
```

(continues on next page)

(continued from previous page)

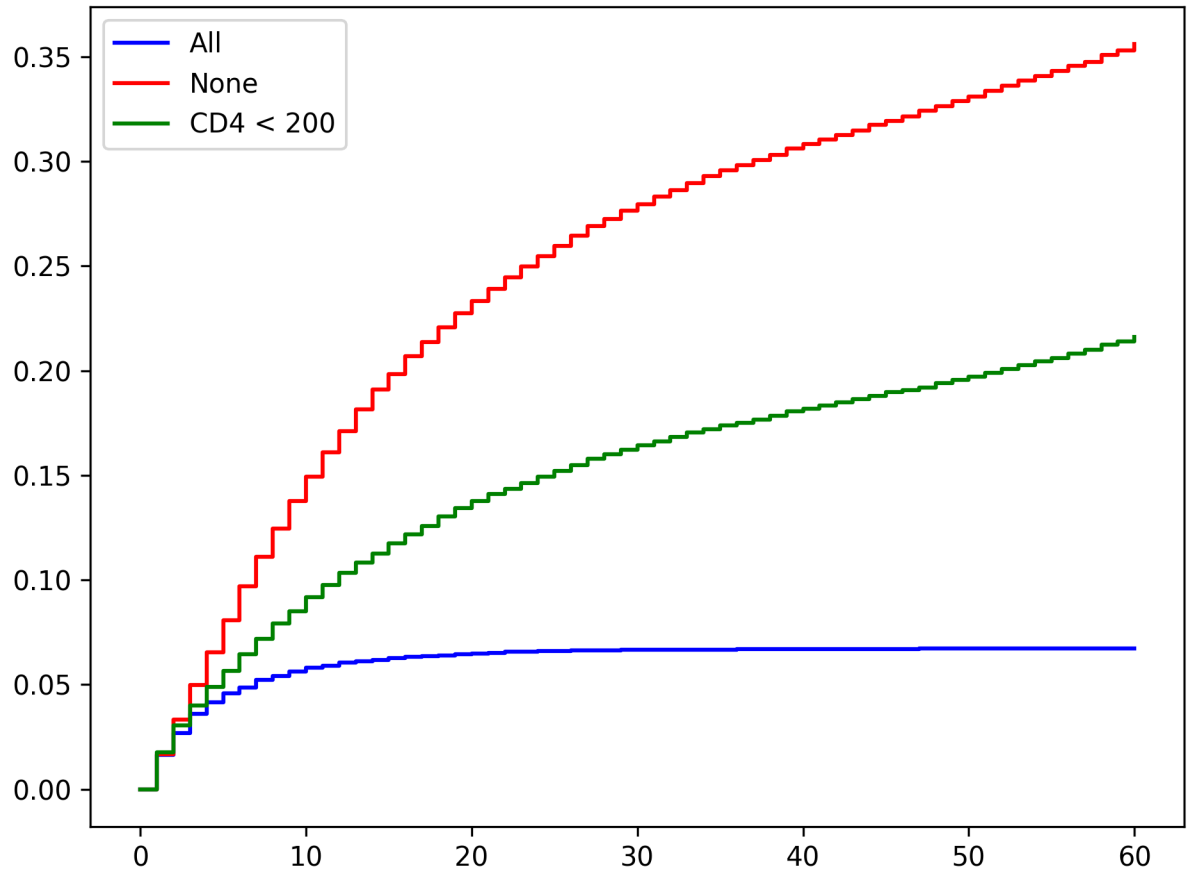
```
# Fitting Kaplan Meier to Natural Course
kmn = KaplanMeierFitter()
kmn.fit(durations=gfs['out'], event_observed=gfs['dead'])

# Fitting Kaplan Meier to Observed Data
kmo = KaplanMeierFitter()
kmo.fit(durations=df['out'], event_observed=df['dead'], entry=df['enter'])

# Plotting risk functions
plt.step(kmn.event_table.index, 1 - kmn.survival_function_, c='k', where='post',␣
↪label='Natural')
plt.step(kmo.event_table.index, 1 - kmo.survival_function_, c='gray', where='post',␣
↪label='True')
plt.legend()
plt.show()
```



From this we can see that out natural course predictions (black) follow the observed data pretty well (gray). Note: this does not mean that our models are correctly specified. *Rather it only means they may not be incorrectly specified.* Sadly, there is no way to know that all our models are correctly specified... We may take some comfort that our curves largely overlap, but do not take this for granted.

We can now estimate the counterfactual outcomes under various treatment plans. In the following code, we will estimate the outcomes under treat-all plan, treat-none plan, and treat only once CD4 T-cell count drops below 200.

```
# Treat-all plan
mcgf.fit(treatment="all",
         lags={'art': 'lag_art',
               'cd4': 'lag_cd4',
               'dvl': 'lag_dvl'},
         in_recode=("g['enter_sq'] = g['enter']**2;"
                    "g['enter_cu'] = g['enter']**3"),
         sample=20000)
g_all = mcgf.predicted_outcomes

# Treat-none plan
mcgf.fit(treatment="none",
         lags={'art': 'lag_art',
               'cd4': 'lag_cd4',
               'dvl': 'lag_dvl'},
         in_recode=("g['enter_sq'] = g['enter']**2;"
                    "g['enter_cu'] = g['enter']**3"),
         sample=20000)
g_none = mcgf.predicted_outcomes

# Custom treatment plan
mcgf.fit(treatment="g['cd4'] <= 200",
         lags={'art': 'lag_art',
               'cd4': 'lag_cd4',
               'dvl': 'lag_dvl'},
         in_recode=("g['enter_sq'] = g['enter']**2;"
                    "g['enter_cu'] = g['enter']**3"),
         sample=20000,
         t_max=None)
g_cd4 = mcgf.predicted_outcomes

# Risk curve under treat-all
gfs = g_all.loc[g_all['uid_g_zepid'] != g_all['uid_g_zepid'].shift(-1)].copy()
kma = KaplanMeierFitter()
kma.fit(durations=gfs['out'], event_observed=gfs['dead'])

# Risk curve under treat-all
gfs = g_none.loc[g_none['uid_g_zepid'] != g_none['uid_g_zepid'].shift(-1)].copy()
kmn = KaplanMeierFitter()
kmn.fit(durations=gfs['out'], event_observed=gfs['dead'])

# Risk curve under treat-all
gfs = g_cd4.loc[g_cd4['uid_g_zepid'] != g_cd4['uid_g_zepid'].shift(-1)].copy()
kmc = KaplanMeierFitter()
kmc.fit(durations=gfs['out'], event_observed=gfs['dead'])

# Plotting risk functions
plt.step(kma.event_table.index, 1 - kma.survival_function_, c='blue', where='post',␣
↪label='All')
plt.step(kmn.event_table.index, 1 - kmn.survival_function_, c='red', where='post',␣
↪label='None')
plt.step(kmc.event_table.index, 1 - kmc.survival_function_, c='green', where='post',␣
↪label='CD4 < 200')
plt.legend()
plt.show()
```



From these results, we can see that the treat-all plan reduces the probability of death across all time points. Importantly, the treat-all plan outperforms the custom treatment plan. Based on this result, we would recommend that all HIV-infected individuals receive ART treatment as soon as they are diagnosed.

To obtain confidence intervals, nonparametric bootstrapping should be used. Take note that this will take awhile to finish (especially if a high number of resamples are used). As it stands, *MonteCarloGFormula* is slow, and future work is to try to optimize the Monte Carlo procedure (specifically with some large matrix multiplications)

Marginal Structural Model

We can also use inverse probability of treatment weights to estimate a marginal structural model for time-varying treatments. Similar to the Monte-Carlo g-formula, we will rely on the same ITT assumption previous described. To calculate the corresponding IPTW, we will use `IPTW` again. Since we will need to do further manipulation of the predicted probabilities, we will have `IPTW` return the predicted probabilities of the denominator and numerator, respectively. We do this through the following code

```
# Specifying models
modeln = 'enter + enter_q + enter_c'
modeld = ('enter + enter_q + enter_c + male + age0 + age0_q + age0_c + dvl0 + cd40 + '
          'cd40_q + cd40_c + dvl + cd4 + cd4_q + cd4_c')

# Restricting to only the previously untreated data
dfs = df.loc[df['lagart']==0].copy()
```

(continues on next page)

(continued from previous page)

```
# Calculating probabilities for IPTW
ipt = IPTW(dfs, treatment='art')
ipt.regression_models(model_denominator=modeld, model_numerator=modeln)
ipt.fit()

# Extracting probabilities for later manipulation
df['p_denom'] = ipt.ProbabilityDenominator
df['p_numer'] = ipt.ProbabilityNumerator
```

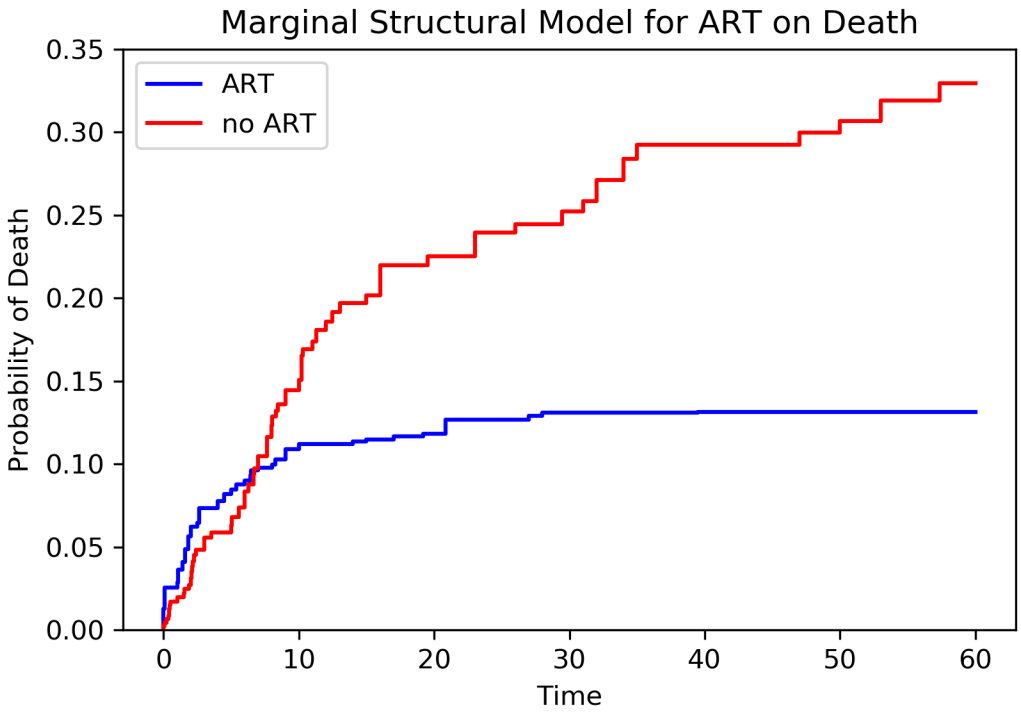
Note: you should only use stabilized weights for time-varying treatments. Unstabilized weights can have poor performance

We now need to do some further manipulation of the weights

```
#Condition 1: First record weight is 1
cond1 = (df.groupby('id').cumcount() == 0)
df['p_denom'] = np.where(cond1, 1, df['p_denom']) #Setting first visit to Pr(...) = 1
df['p_numer'] = np.where(cond1, 1, df['p_numer'])
df['ip_denom'] = np.where(cond1, 1, (1-df['p_denom']))
df['ip_numer'] = np.where(cond1, 1, (1-df['p_numer']))
df['den'] = np.where(cond1, df['p_denom'], np.nan)
df['num'] = np.where(cond1, df['p_numer'], np.nan)

#Condition 2: Records before ART initiation
cond2 = ((df['lagart']==0) & (df['art']==0) & (df.groupby('id').cumcount() != 0))
df['num'] = np.where(cond2, df.groupby('id')['ip_numer'].cumprod(), df['num'])
df['den'] = np.where(cond2, df.groupby('id')['ip_denom'].cumprod(), df['den'])

#Condition 3: Records at ART initiation
cond3 = ((df['lagart']==0) & (df['art']==1) & (df.groupby('id').cumcount() != 0))
df['num'] = np.where(cond3, df['num'].shift(1)*(df['p_numer']), df['num'])
df['den'] = np.where(cond3, df['den'].shift(1)*(df['p_denom']), df['den'])

#Condition 4: Records after ART initiation
df['num'] = df['num'].ffill()
df['den'] = df['den'].ffill()

#Calculating weights
df['w'] = df['num'] / df['den']
```

After calculating our weights, we can estimate the risk functions via a weighted Kaplan Meier. Note that `lifelines` version will need to be `0.14.5` or greater. The following code will generate our risk function plot

```
kme = KaplanMeierFitter()
dfe = df.loc[df['art']==1].copy()
kme.fit(dfe['out'],event_observed=dfe['dead'],entry=dfe['enter'],weights=dfe['w'])

kmu = KaplanMeierFitter()
dfu = df.loc[df['art']==0].copy()
kmu.fit(dfu['out'],event_observed=dfu['dead'],entry=dfu['enter'],weights=dfu['w'])

plt.step(kme.event_table.index,1 - kme.survival_function_,c='b',label='ART')
plt.step(kmu.event_table.index,1 - kmu.survival_function_,c='r',label='no ART')
plt.show()
```



Similarly, we see the treat-all plan is better than the never-treat plan. We see a discrepancy between the two approaches during the early times (weeks less than 5). Note that we did not account for informative censoring. To account for informative censoring, we could use inverse probability of censoring weights. See the Missing Data tutorial for further details.

1.3.2 Longitudinal Data

We will use a different simulated data set within *zEpid* for this section. This data is longitudinal data simulated for demonstrative purposes. This data set is in a wide-format, such that each row is a single person and columns are variables measured at specific time points. *Note:* this format is distinct from the time-to-event data, which was in a long format. Below is code to load this data set

```
from zepid import load_longitudinal_data
df = load_longitudinal_data()
```

In this data, we have outcomes measured at three time points. Additionally, we have treatments (A), time-varying confounder (L), and a baseline confounder (W) measured in our data. We will assume exchangeability (sometimes also referred to as sequential ignorability) for the effect of A on Y by L and W.

Iterative Conditional g-formula

The iterative conditional g-formula is an alternative to the Monte-Carlo estimation procedure, as detailed in the previous sections. While the Monte-Carlo g-formula requires that we specify a parametric regression model for *all* time-varying variables, the iterative conditional approach only requires that we specify an outcome regression model. This drastically cuts down on the potential for model misspecification. However, we no longer use a pooled logistic regression model, so the iterative conditional g-formula does not estimate nicely in sparse survival data (in my experience).

The iterative conditional procedure works like the following. Starting at the last observed time, we fit our specified outcome model. From this model, we predict the probability of the outcome under observed treatment ($\hat{Q}$) and under the counterfactual treatment of interest (Q^*). Next, we move to the previous time point. For those who were observed at the last time point, we use their $\hat{Q}$ as their outcome. If they were not observed at the furtherest time point, we use their observed Y instead. We repeat the process of model fitting. We then repeat this whole procedure (hence "iterative" conditionals) until we end up at the origin. Now our predicted Q^* value is the counterfactual mean under the specified treatment plan

The following is code to use the iterative conditional process. We will start with estimating the counterfactual mean under a treat-all strategy for t=3.

```
icgf = IterativeCondGFormula(df, exposures=['A1', 'A2', 'A3'], outcomes=['Y1', 'Y2',
↪'Y3'])

# Specifying regression models for each treatment-outcome pair
icgf.outcome_model(models=['A1 + L1',
                           'A2 + A1 + L2',
                           'A3 + A2 + L3'],
                   print_results=False)

# Estimating marginal 'Y3' under treat-all at every time
icgf.fit(treatments=[1, 1, 1])
r_all = icgf.marginal_outcome
```

r_all is the overall risk of Y at time 3 under a treat-all at all time points strategy. This value was 0.433. We can estimate the overall risk of Y at time 3 under a treat-none strategy by running

```
icgf.fit(treatments=[0, 0, 0])
r_non = icgf.marginal_outcome

print('RD =', r_all - r_non)
```

We can interpret our estimated risk difference as: the risk of Y at time 3 under a treat-all strategy was 19.5% points lower than under a treat-none strategy. We can make further comparisons between treatment plans by changing the *treatments* argument. Below is an example where treatment is only given a baseline

```
icgf.fit(treatments=[1, 0, 0])
```

The estimated risk under this treatment strategy is 0.547. To estimate Y at t=2, we use a similar process as above but limit our data to *Y2*. Below is an example of estimating Y at t=2 for a treat-all strategy

```
icgf = IterativeCondGFormula(df, exposures=['A1', 'A2'], outcomes=['Y1', 'Y2'])
icgf.outcome_model(models=['A1 + L1',
                           'A2 + A1 + L2'],
                   print_results=False)
icgf.fit(treatments=[1, 1])
```

The estimate risk of Y at t=2 under a treat-all strategy was 0.350. The above process can be repeated for all observation times in a wide data set. For calculation of confidence intervals, a non-parametric bootstrapping procedure should be used.

Marginal Structural Model

We can also use inverse probability weights to estimate a marginal structural model. Easier implementation of this estimation will be added later.

Longitudinal TMLE

In a future update, the longitudinal targeted maximum likelihood estimator will be added.

G-estimation

Currently, g-estimation of structural nested models for time-varying exposures is not implemented. I plan to add AFT estimation procedures in a future update

1.3.3 Summary

G-methods allow us to answer more complex questions than standard methods. With these tools, we can start to ask questions about ideal treatment strategies. See further tutorials at this GitHub repo for further examples

1.4 Generalizability

This section details generalizability and transportability. Throughout this section, our data comes from a randomized trial. However, these methods can be extended to observational studies. Additionally, we have a random sample from our target population. Our study sample has information on treatment, outcome, and modifiers. Our target population sample only has information on modifiers.

zEpid comes with a simulated data set for determining generalizability and transportability. Variables included in this data set are A (treatment), Y (outcome), S (indicator of being in study sample), and L W (potential effect measure modifiers).

```
import numpy as np
import pandas as pd

from zepid import load_generalize_data
from zepid.causal.generalize import IPSW, GTransportFormula, AIPSW

df = load_generalize_data(False)
```

You will notice that the data set is essentially a stacked data set of the study sample (S=*1*) and the target population sample (S=*0*). A and Y are only observed when S=*1*

1.4.1 Generalizability

Generalizability is the concept that our study sample is not a random sample from the population we want to make inferences about (target population). The concept of generalizability is often referred to as external validity.

For demonstration, consider our simulated trial data to assess the effect of A on Y. While our trial results are internally valid (correct estimation for our study sample), we are concerned that they are no longer reflective of our target population. Specifically, we are concerned that the individuals who enrolled in our trial are not a random sample

of our target population. We believe that our study sample and target population are exchangeable (or a conditional random sample) by observed variables *L* and *W*.

In addition to our trial data, we also collected basic information on the target population (assessed via non-enrollees in our trial). With this information and assumptions, we will now look at three approaches to estimate the effect in our target population; inverse probability of sampling weights, g-transport formula, and augmented inverse probability of sampling weights (doubly robust).

IPSW

Inverse probability of sampling weights work by re-weighting our study sample to be reflective of our target population. To estimate the risk difference and risk ratio in the target population, we can use the following code

```
ipsw = IPSW(df, exposure='A', outcome='Y', selection='S', generalize=True)
ipsw.regression_models('L + W + L:W', print_results=False)
ipsw.fit()
ipsw.summary()
```

Based on the summary output, the target population estimates were RD=0.10, RR=1.38. We would interpret this as; the probability of Y given everyone in the target population had *A=1* would have been 10% points higher than if everyone in the target population had *A=0*. For confidence intervals, we would need to use a non-parametric bootstrapping procedure. However, we need to modify our bootstrapping procedure. Specifically, we need to account for random variability in our study sample and the random variability in our target population selection.

For confidence intervals, we (1) divided our stacked data set, (2) sample with replacement in each of the data sets, (3) re-stack the data sets, and (4) recalculate IPSW and the corresponding measures. Below is example code to do that procedure with 200 resamples

```
rd = ipsw.risk_difference
rd_bs = []

# Step 1: divide data
dfss = df.loc[df['S'] == 1].copy()
dftp = df.loc[df['S'] == 0].copy()

for i in range(200):
    # Step 2: Resample data
    dfs = dfss.sample(n=dfss.shape[0], replace=True)
    dft = dftp.sample(n=dftp.shape[0], replace=True)

    # Step 3: restack the data
    dfb = pd.concat([dfs, dft])

    # Step 4: Estimate IPSW
    ipsw = IPSW(dfb, exposure='A', outcome='Y', selection='S', generalize=True)
    ipsw.regression_models('L + W + L:W', print_results=False)
    ipsw.fit()

    rd_bs.append(ipsw.risk_difference)

se = np.std(rd_bs, ddof=1)

print('95% LCL:', np.round(rd - 1.96*se, 3))
print('95% UCL:', np.round(rd + 1.96*se, 3))
```

In my run of the bootstrap procedure, I ended up with an estimated 95% confidence interval of (0.01, 0.19).

To account for confounding, this approach can be paired with inverse probability of treatment weights. For confidence intervals, we would need to add a step to estimate IPTW between steps 2 and 4.

G-transport formula

The g-transport formula is an extension of the g-formula for generalizability and transportability. Similar to the standard parametric g-formula, we fit a parametric regression model predicting the outcome as a function of treatment (and baseline covariates). From our estimated parametric model, we then predict the potential outcomes under the treatment strategies for the entire population (study sample *and* target population).

The g-transport formula differs from the g-formula, in that we need to specify all modifiers within the model (and corresponding interaction terms). If we were only interested in internal validity, our g-formula for our trial data would only include treatment in the regression model. For the g-transport formula, we now need to include terms in the model for all effect measure modifiers. Below is example code for the procedure

```
gtf = GTransportFormula(df, exposure='A', outcome='Y', selection='S', generalize=True)
gtf.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
gtf.fit()
gtf.summary()
```

Based on the summary output, the target population estimates were RD=0.07, RR=1.22. We would interpret this as; the probability of Y given everyone in the target population had *A=1* would have been 7% points higher than if everyone in the target population had *A=0*. For confidence intervals, we would need to use a similar non-parametric bootstrapping procedure to IPSW. Below is example code with 200 bootstraps

```
rd = gtf.risk_difference
rd_bs = []

# Step 1: divide data
dfss = df.loc[df['S'] == 1].copy()
dftp = df.loc[df['S'] == 0].copy()

for i in range(200):
    # Step 2: Resample data
    dfs = dfss.sample(n=dfss.shape[0], replace=True)
    dft = dftp.sample(n=dftp.shape[0], replace=True)

    # Step 3: restack the data
    dfb = pd.concat([dfs, dft])

    # Step 4: Estimate IPSW
    gtf = GTransportFormula(dfb, exposure='A', outcome='Y', selection='S',␣
↪generalize=True)
    gtf.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
    gtf.fit()

    rd_bs.append(gtf.risk_difference)

se = np.std(rd_bs, ddof=1)
print('95% LCL:', np.round(rd - 1.96 * se, 3))
print('95% UCL:', np.round(rd + 1.96 * se, 3))
```

The 95% confidence intervals for the risk difference were; -0.03, 0.16.

For observational data, the g-transport formula more naturally extends to account for confounding. To correct for confounding, the confounding terms are included in the parametric regression model (we don't need any outside weights or calculations). Remember that if there is an effect of treatment, then there must be modification by the

confounder on at least one scale (additive / multiplicative). This suggests you want to include as many interaction terms in the g-transport formula as possible.

AIPSW

At this point, you may be wondering which approach is better. Similar to other causal inference methods, there exists a recipe to combine IPSW and the g-transport formula into a single estimate. This approach is doubly robust, such that if either the g-transport formula or the IPSW is correctly specified, then our estimate will be unbiased. While I am unaware of a formal name for this approach, I refer to it as augmented-IPSW.

Similar to AIPTW, AIPSW requires that we specify the g-transport formula and the IPSW models. Below is code for this procedure

```
aipw = AIPSW(df, exposure='A', outcome='Y', selection='S', generalize=True)
aipw.weight_model('L + W_sq', print_results=False)
aipw.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
aipw.fit()
aipw.summary()
```

Our results are similar to the g-transport formula (RD=0.07 RR=1.23). For confidence intervals, we repeat the same bootstrapping procedure as before

```
rd = aipw.risk_difference
rd_bs = []

# Step 1: divide data
dfss = df.loc[df['S'] == 1].copy()
dftp = df.loc[df['S'] == 0].copy()

for i in range(200):
    # Step 2: Resample data
    dfs = dfss.sample(n=dfss.shape[0], replace=True)
    dft = dftp.sample(n=dftp.shape[0], replace=True)

    # Step 3: restack the data
    dfb = pd.concat([dfs, dft])

    # Step 4: Estimate IPSW
    aipw = AIPSW(dfb, exposure='A', outcome='Y', selection='S', generalize=True)
    aipw.weight_model('L + W + L:W', print_results=False)
    aipw.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
    aipw.fit()

    rd_bs.append(aipw.risk_difference)

se = np.std(rd_bs, ddof=1)
print('95% LCL:', np.round(rd - 1.96 * se, 3))
print('95% UCL:', np.round(rd + 1.96 * se, 3))
```

The 95% CL were -0.02, 0.15 for the risk difference.

To extend AIPSW to observational data, we use both the IPSW approach for observation data and the g-transport formula approach. For observational data, we need to calculate IPTW for both IPSW and AIPSW approaches.

1.4.2 Transportability

Transportability is a related concept. Rather than our study sample not being a random sample from our target population, our study sample is not part of our target population. As an example, our study on the effect of drug X on death may have been conducted in the United States, but we want to estimate the effect of drug X on death in Canada. Since our study sample is not part of the target population, some authors draw a distinction between the two problems.

What this changes for our estimators is who we are marginalizing over. For generalizability, our estimates are marginalized over the study sample and the random sample of the target population. For transportability, we only marginalize over the random sample of the target population. Depending on the distribution of effect measure modifiers, the generalizability and transportability estimates may differ.

Within *zEpid*, the same functions are used, but we set *generalize=False* to use transportability instead. Below are examples

IPSW

IPSW takes a slightly different form for transportability compared to generalizability. Notably, IPSW becomes inverse *odds* of sampling weights for the transportability problem. Implementation-wise, there is no large difference between *IPSW* for generalizability and transportability. Below is how to estimate the average causal effect in the target population

```
ipsw = IPSW(df, exposure='A', outcome='Y', selection='S', generalize=False)
ipsw.regression_models('L + W + L:W', print_results=False)
ipsw.fit()
ipsw.summary()
```

The estimates in our target population were RD=0.10 and RR=1.36 (remember the target population is where *S=0*). We can calculate confidence intervals using the same non-parametric bootstrapping procedure.

G-transport formula

The g-transport formula for transportability follows the same procedure as the generalizability approach. However, instead of marginalizing over the study sample and the target sample, we only marginalize over the target sample. Code-wise, we only have to change *generalize=False*. Below is example code

```
gtf = GTransportFormula(df, exposure='A', outcome='Y', selection='S',␣
↪generalize=False)
gtf.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
gtf.fit()
gtf.summary()
```

The estimated RD=0.061 and RR=1.20 for our target population (*S=0*). Similarly, we can calculate confidence intervals via non-parametric bootstrapping.

AIPSW

Again, AIPSW is the doubly robust procedure to merge our IPSW and g-transport formula into a singular estimate. It follows the same approach as IPSW and g-transport formula for the transportability problem. Below is code to estimate AIPSW

```
aipw = AIPSW(df, exposure='A', outcome='Y', selection='S', generalize=False)
aipw.weight_model('L + W + L:W', print_results=False)
aipw.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
```

(continues on next page)

(continued from previous page)

```
aipw.fit()
aipw.summary()
```

Our estimates for AIPSW were similar to the g-transport formula (RD=0.06, RR=1.20). Confidence intervals can be calculated using the same non-parametric bootstrap procedure.

1.4.3 Summary

Similar to other causal inference methods, each estimator requires different assumptions. Notably, the g-transport formula requires we specify a more complex model. AIPSW, our doubly robust method, allows us to have 'two chances' to specify our models correctly. While framed in terms of randomized study sample data, these methods extend to observational data.

For observational data, you may be stuck with using IPSW. G-transport formula and AIPSW both require that confounders are measured in both the study sample and the target population. The random sample from the target population (if you did not collect it) may *not* have information on these variables. Since this information is necessary for the g-transport formula, neither the g-transport formula nor AIPSW can be estimated.

1.5 Missing Data

Missing data is a common occurrence in research and is unfortunately often ignored. Most software drops the missing data to be helpful. However, by dropping that data we assume that data is missing completely at random. This is often an unreasonable assumption and often unlikely to be true. While missing data may have a negligible effect when only a few observations are missing, this is not always the case if there is substantial missing data.

We will describe inverse probability weighting approaches to account for missing data. We will detail inverse probability of missing weights for different patterns of missing data, and inverse probability of censoring weights (a special case of IPMW). Note: I am neglecting to mention multiple imputation, which is another approach to handling data.

1.5.1 IPMW

Inverse probability of missing weights are one way to account for missing data. IPMW works by reweighting the observed sample to reflect the full data. IPMW can be calculated for any missing variable in the data. To help frame the discussion of missing data, consider the following data sets

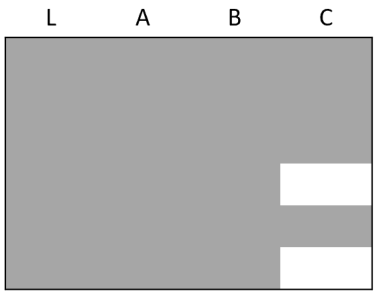


Figure 1.A

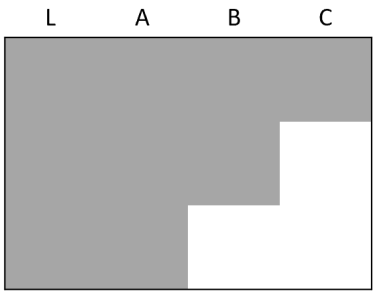


Figure 1.B

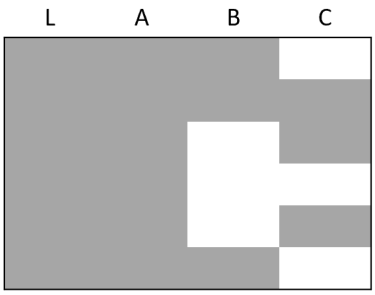


Figure 1.C

Figure 1.A summarizes missing data for a single variable. Single variables only require a single IPMW estimation step. Figure 1.B is an example of monotonic missing data. For monotonic missing data, if one variable is missing (*B*), then the the next missing variable must also be missing (*C*). In this scenario, we use an iterative process of calculating IPMW. Lastly, there is non-monotonic missing data. Non-monotonic missing data does not follow the pattern of monotonic missing data. A variable missing for one column may or may not be missing for another. This is more complex to solve and likely more common in practice.

Single Variable

First, we will focus on the case shown in Figure 1.C, missing data for a single variable. We will load the sample simulation data. Loading the simulated data

```
from zepid import load_sample_data
from zepid.causal.ipw import IPMW
df = load_sample_data(timevary=False)
```

The missing variable is this data set we will focus on is *dead*. Since *dead* is our outcome in later analyses, these weights could also be referred to as inverse probability of censoring weights. However, we will use *IPMW* to calculate weights for outcomes measured at a single time.

In this example, we will assume data is missing completely at random conditional on age, ART, and gender. Additionally, we will stabilize the weights and include ART in both the denominator and numerator. This weight formation is useful for later analyses our the average causal effect of ART on death. Please see the tutorials on Time-Fixed Exposures for further information (I leave it to the reader to merge IPMW with the methods described in Time-Fixed Exposures)

```
ipm = IPMW(df, missing='dead', stabilized=True)
ipm.regression_models(model_denominator='age0 + art + male',
                      model_numerator='art')
ipm.fit()
```

After calculating our weights, we can save the calculated weights for later usage

```
df['ipmw'] = ipm.Weight
```

Additionally, we don't necessarily need to use monotonic IPMW if we have data as shown in Figure 1.B. We may be willing to assume that *C*'s missingness does not depend on *B*. In that scenario, we could calculate two sets of IPMW following the above procedure. Then we would multiply the two sets of weights to obtain our final set of IPMW. If we are not willing to assume that *C* missing does not depend on *B*, we will need to use the IPMW formulation described in the following section. That concludes IPMW for a single missing variable.

Monotone Missingness

For this next tutorial, we will load another simulated data. In this data set, there are four variables. Two of the variables are missing (*B* and *C*) and follow the pattern shown in Figure 1.B

```
from zepid import load_monotone_missing_data
from zepid.causal.ipw import IPMW
df = load_monotone_missing_data()
```

For monotonic missing data, we use a similar process. However, we provide a list of missing variables instead of a single string. Additionally, we specify a list of regression models. Specifically, we assume that *B* is missing completely at random given *L* and *A*. We assume *C* is missing completely at random given *L* and *B*. Since *C* depends on *B* and *B* is missing, we need to use this iterative process to calculate IPMW.

```
ipm = IPMW(df, missing_variable=['B', 'C'], monotone=True)
ipm.regression_models(model_denominator=['L + A', 'L + B'])
ipm.fit()
```

Behind the scenes, the model for *B* is fit, *C* is fit, then the calculated weights are multiplied together to obtain our full IPMW set. Again, we can set the calculated weights as a variable in our data for later use

```
df['ipmmw'] = ipm.Weight
```

There is also a special case of monotonic data missing data. If variable *C* was always missing when *B* was missing in Figure 1, then the monotonic IPMW becomes the same as single-variable IPMW. Behind the scenes, *IPMW* checks for this special case and uses the single-variable process if it detects it. You can manually do this by only specifying one of the missing variables

Non-Monotone Missingness

Non-monotonic missing data is not currently supported. Future plans are to include IPMW for non-monotonic data

1.5.2 AIPMW

Augmented-IPMW is a doubly robust procedure to account for missing data. This is not currently implemented but is planned for the future. This expands to the same scenarios that IPMW does

1.5.3 IPCW

As mentioned in the introduction, inverse probability of censoring weights can be viewed as a special case of missing data. Specifically, censoring is missing data on the outcome. Additionally, censored data will generally follow a monotone missing pattern (once a participant is censored, they are censored for all future time points).

IPCW is built to accounting for censoring in time-to-event data. For missing outcome data at a single follow-up time, *IPMW* should be used instead. For the *IPCW* tutorial, we will use the time-varying simulated sample data. To motivate this example, we are interested in estimating the overall risk of mortality over time. However, we are concerned about censoring being dependent on gender and age.

We will load the data via

```
from zepid import load_sample_data, spline
from zepid.causal.ipw import IPCW
```

(continues on next page)

(continued from previous page)

```
df = load_sample_data(True)
df[['age_rs1', 'age_rs2']] = spline(df, 'age0', n_knots=3, term=2, restricted=True)
df[['enter_rs1', 'enter_rs2']] = spline(df, 'enter', n_knots=3, term=2,␣
↪restricted=True)
```

After loading our data, we can calculate IPCW with the following code. For IPCW, it is recommended to use stabilized weights. We will stabilize our weights by time (*enter*), which is common practice

```
ipcw = IPCW(df, idvar='id', time='enter', event='dead')
ipcw.regression_models('enter + enter_rs1 + enter_rs2 + male + age0 + age_rs1 + age_
↪rs2',
                       model_numerator='enter + enter_rs1 + enter_rs2',
                       print_results=False)
ipcw.fit()
```

Finally, we can add these weights to our data set.

```
df['cw'] = ipcw.Weight
```

Now, we can estimate a weighted Kaplan-Meier to obtain the risk curve, allowing for non-informative censoring conditional on age and gender

1.5.4 Summary

This concludes the discussion of approaches to account for missing data with *zEpid*. Please see the online tutorials at this GitHub repo for further descriptions and examples

1.6 Graphics

This page demonstrates some of the different graphics possible to generate with *zEpid*. These plots are all generated using *matplotlib*. The functions themselves return *matplotlib.axes* objects, so users can further edit and customize their plots. Let's look at some of the plots

1.6.1 Functional Form Assessment

zEpid makes graphical (qualitative) and statistical (quantitative) functional form assessment easy to implement. Functional form assessments are available for either discrete or continuous variables. The distinction only matters for the calculation of the LOESS curve generated in the plots.

Plots and regression model results come from generalized linear models fit with *statsmodels*.

Let's look at some examples. We will look at baseline age (discrete variable). We will compare linear, quadratic, and splines for the functional form. First, we set up the data

```
import zepid as ze
from zepid.graphics import functional_form_plot

df = ze.load_sample_data(timevary=False)
df['age0_sq'] = df['age0']**2
df[['rqs0', 'rqs1']] = ze.spline(df, var='age0', term=2, n_knots=3, knots=[30, 40,␣
↪55], restricted=True)
```

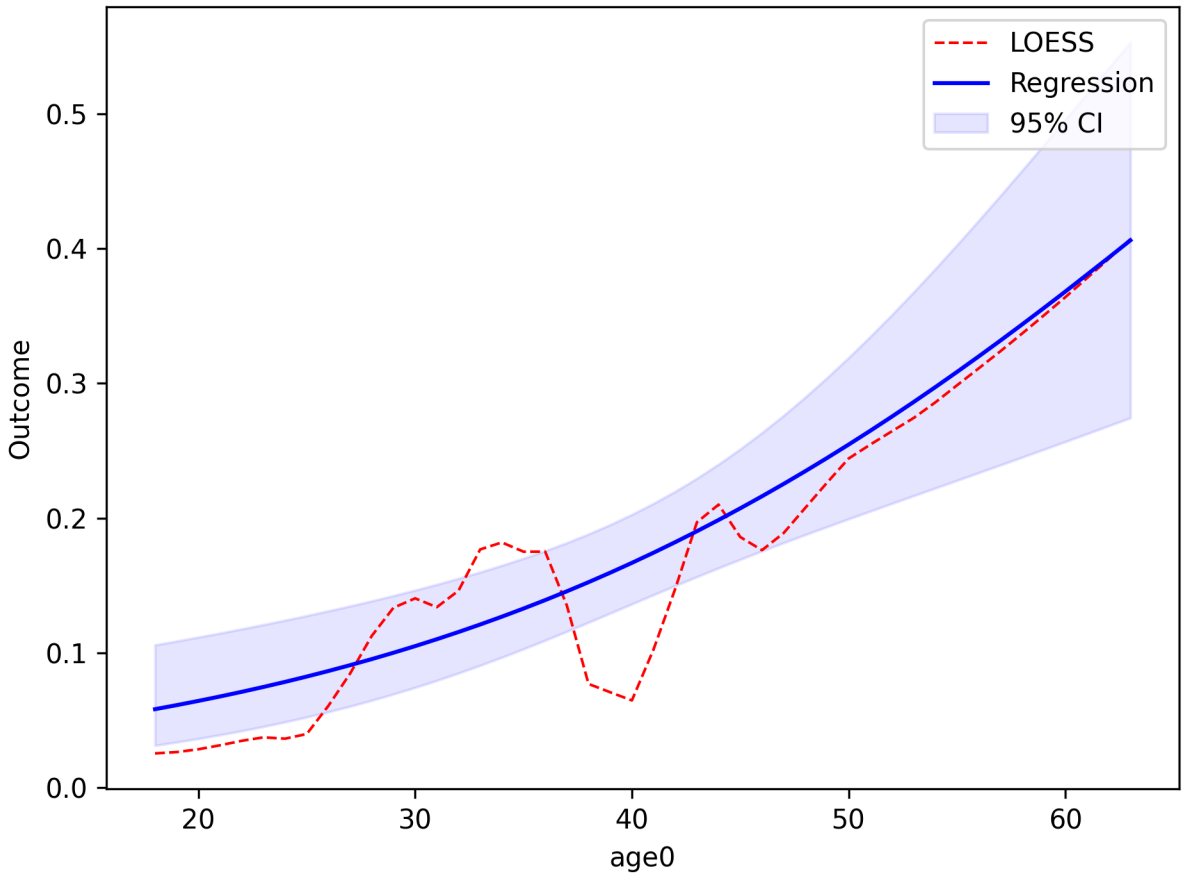
Linear

Now that our variables are all prepared, we will look at a basic linear term for `age0`.

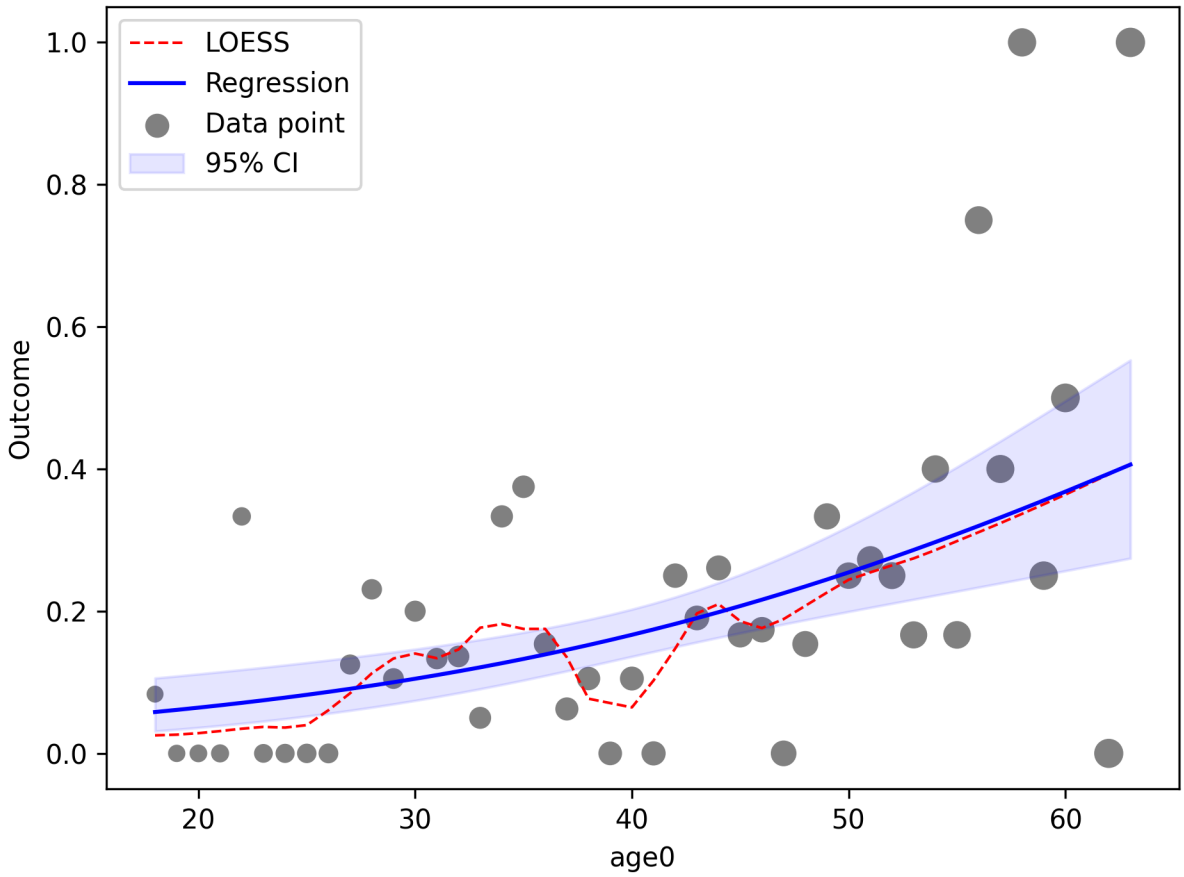
```
functional_form_plot(df, outcome='dead', var='age0', discrete=True)
plt.show()
```

In the console, the following results will be printed

```
Warning: missing observations of model variables are dropped
0  observations were dropped from the functional form assessment
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                    dead   No. Observations:                  547
Model:                             GLM   Df Residuals:                      545
Model Family:                Binomial   Df Model:                            1
Link Function:                   logit   Scale:                          1.0000
Method:                           IRLS   Log-Likelihood:                -239.25
Date:                 Tue, 26 Jun 2018   Deviance:                       478.51
Time:                         08:25:47   Pearson chi2:                     553.
No. Iterations:                      5   Covariance Type:             nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -3.6271      0.537     -6.760      0.000      -4.679      -2.575
age0           0.0507      0.013      4.012      0.000       0.026       0.075
==============================================================================
AIC:  482.50783872152573
BIC:  -2957.4167585984537
```



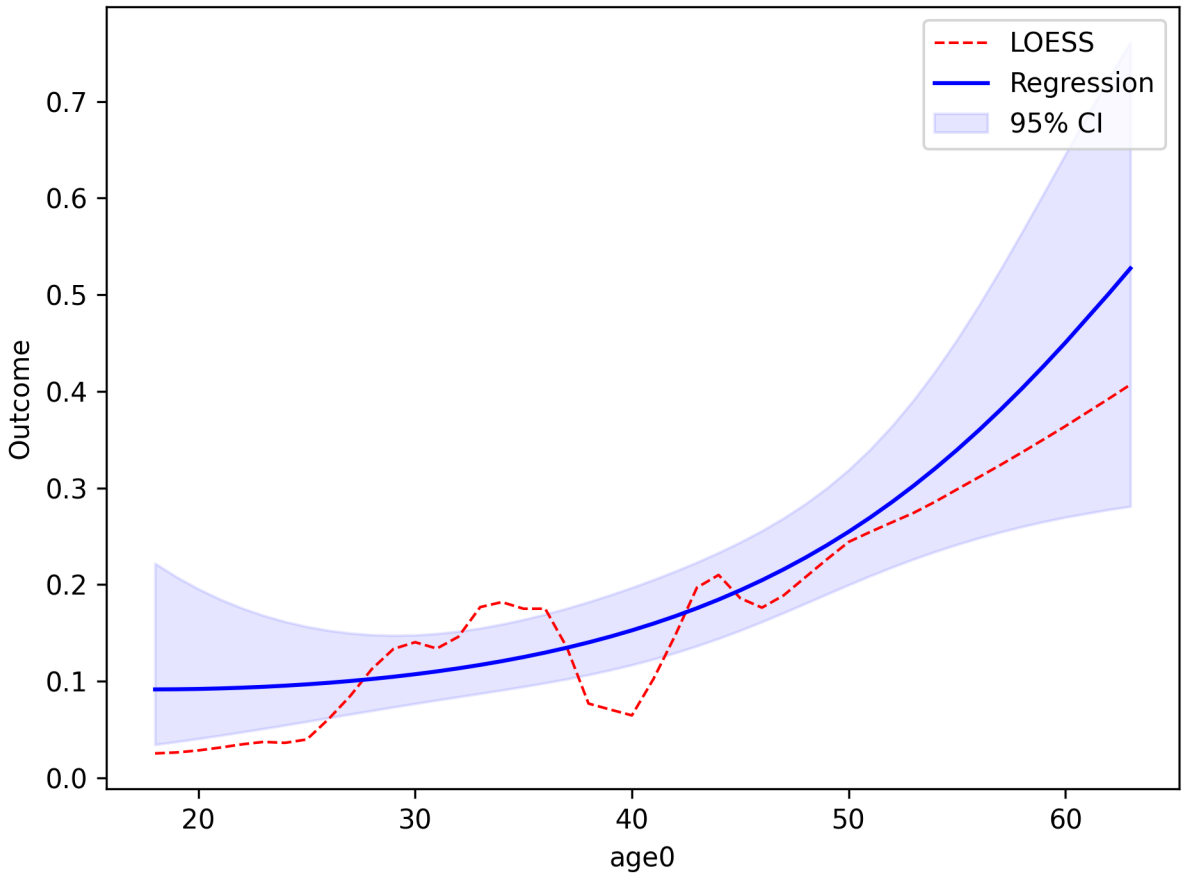
In the image, the blue line corresponds to the regression line and the shaded blue region is the 95% confidence intervals. The red-dashed line is the `statsmodels` generated LOESS curve. We can also have the data points that the LOESS curve is fit to plot as well



Quadratic

To implement other functional forms besides linear terms, the optional `f_form` argument must be supplied. Note that any terms specified in the `f_form` argument must be part of the data set. We can assess a quadratic functional form like the following

```
functional_form_plot(df, outcome='dead', var='age0', f_form='age0 + age0_sq',␣
↪discrete=True)
plt.show()
```

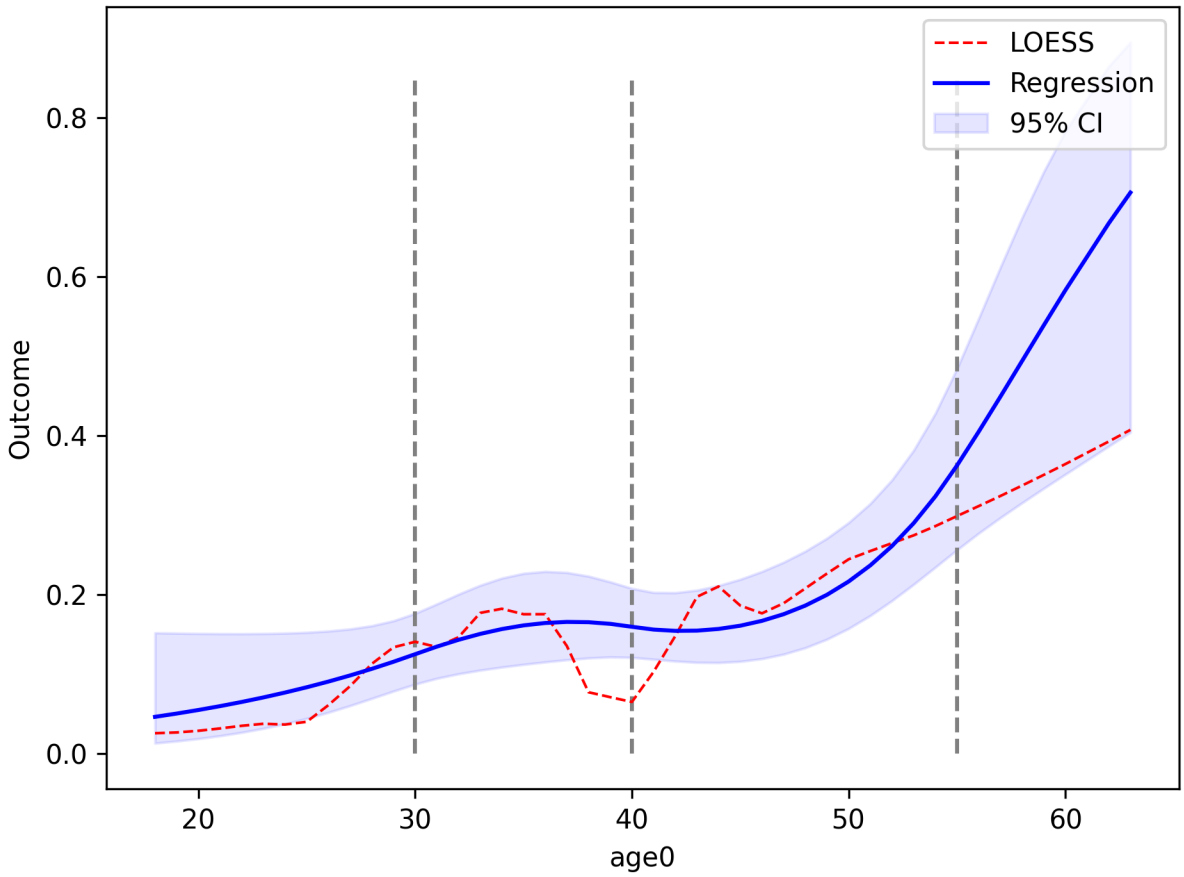


The `f_form` argument is used to specify any functional form variables that are coded by the user

Spline

We will now compare using restricted quadratic splines for the functional form of age. To show how users can further edit the plot, we will add dashed lines to designate where the spline knots are located

```
functional_form_plot(df, outcome='dead', var='age0', f_form='age0 + rqs0 + rqs1',␣
↪discrete=True)
plt.vlines(30, 0, 0.85, colors='gray', linestyles='--')
plt.vlines(40, 0, 0.85, colors='gray', linestyles='--')
plt.vlines(55, 0, 0.85, colors='gray', linestyles='--')
plt.show()
```



Continuous Variables

For non-discrete variables (indicated by `discrete=False`, the default), data is binned into categories automatically. The number of categories is determined via the maximum value minus the minimum divided by 5.

$$\frac{(max(X) - min(X))}{5}$$

To adjust the number of categories, the continuous variable can be multiplied by some constant. If more categories are desired, then the continuous variable can be multiplied by some constant greater than 1. Conversely, if less categories are desired, then the continuous variable can be multiplied by some constant between 0 and 1. In this example we will look at `cd40` which corresponds to baseline viral load.

```
functional_form_plot(df, outcome='dead', var='cd40')
plt.show()
```

If we use the current values, the number of categories is indicated in the console output as

```
A total of 99 categories were created. If you would like to influence the number of␣
↪categories
the spline is fit to, do the following
    Increase: multiply by a constant >1
    Decrease: multiply by a constant <1 and >0
```

We can see that `statsmodels` has an overflow issue in some exponential. We can decrease the number of categories within `cd40` to see if that fixes this. We will decrease the number of categories by multiplying by `0.25`.

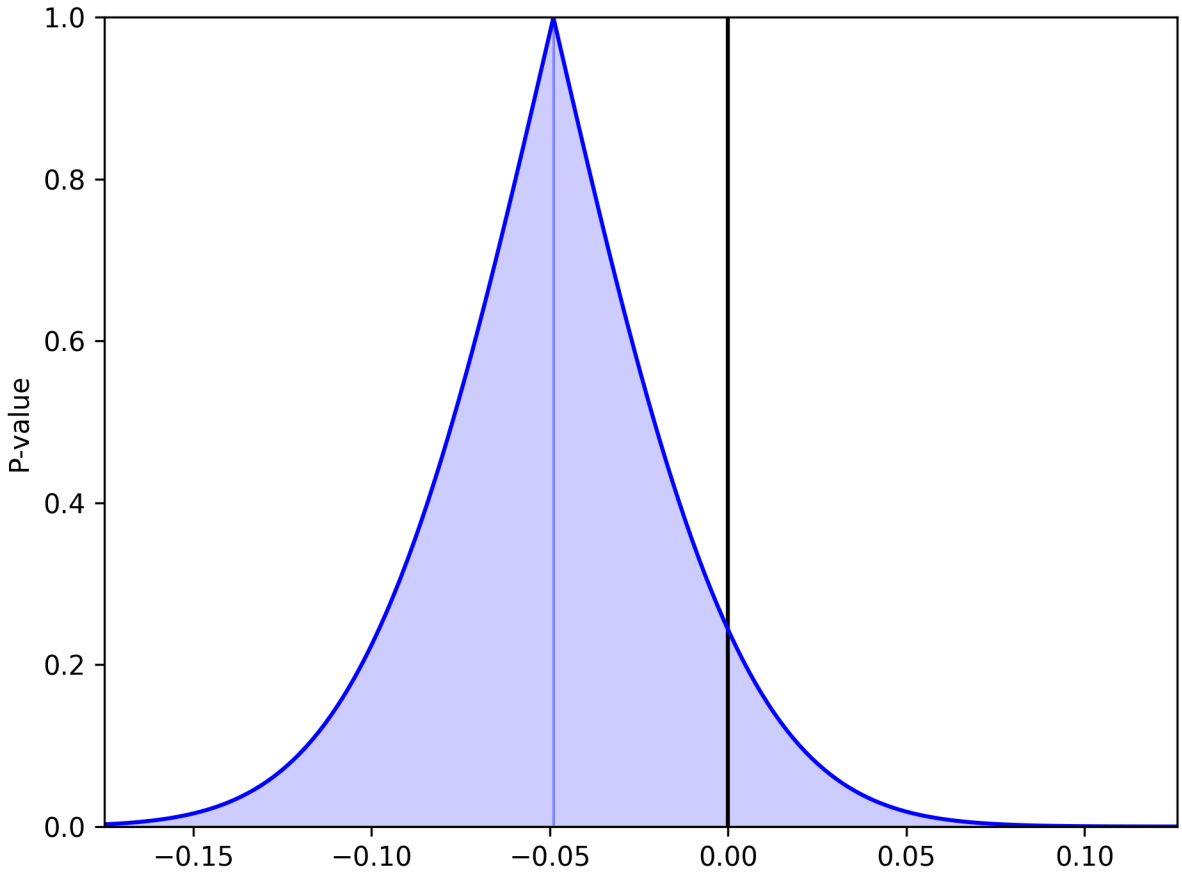
```
df['cd4_red'] = df['cd40']*0.25
functional_form_plot(df, outcome='dead', var='cd4_red')
plt.show()
```

Now only 24 categories are created and it removes the overflow issue.

1.6.2 P-value Plot

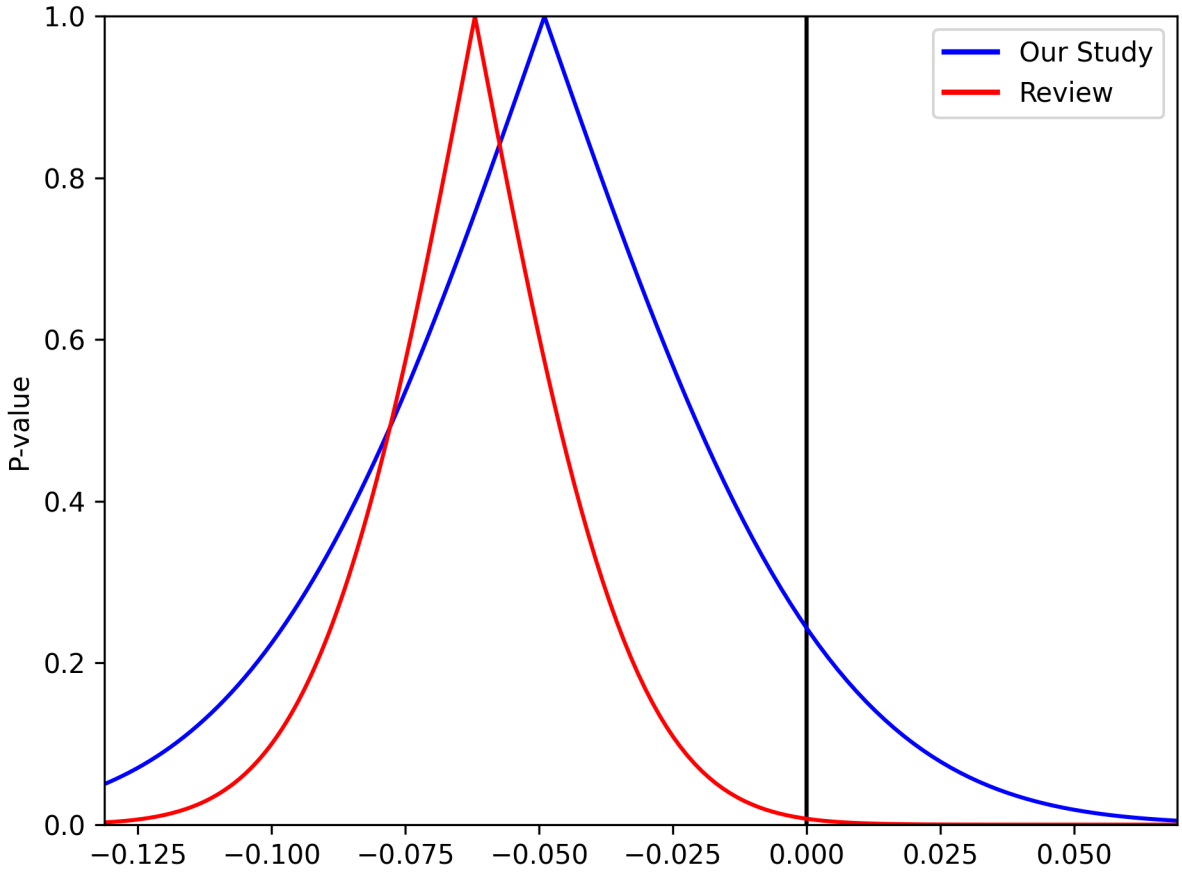
As described and shown in *Epidemiology* 2nd Edition by K. Rothman, this function is meant to plot the p-value distribution for a variable. From this distribution, p-values and confidence intervals can be visualized to compare or contrast results. Note that this functionality only works for linear variables (i.e. Risk Difference and log(Risk Ratio)). Returning to our results from the Measures section, we will look at plots of the Risk Difference. For a risk difference of -0.049 (SE: 0.042), the plot is

```
from zepid.graphics import pvalue_plot

pvalue_plot(point=-0.049, sd=0.042)
plt.show()
```



We can stack multiple p-value plots together. Imagine a systematic review was conducted prior to our study and resulted in a summary risk difference of -0.062 (SE: 0.0231). We can use the p-value plots to visually display the results of our data and the systematic review

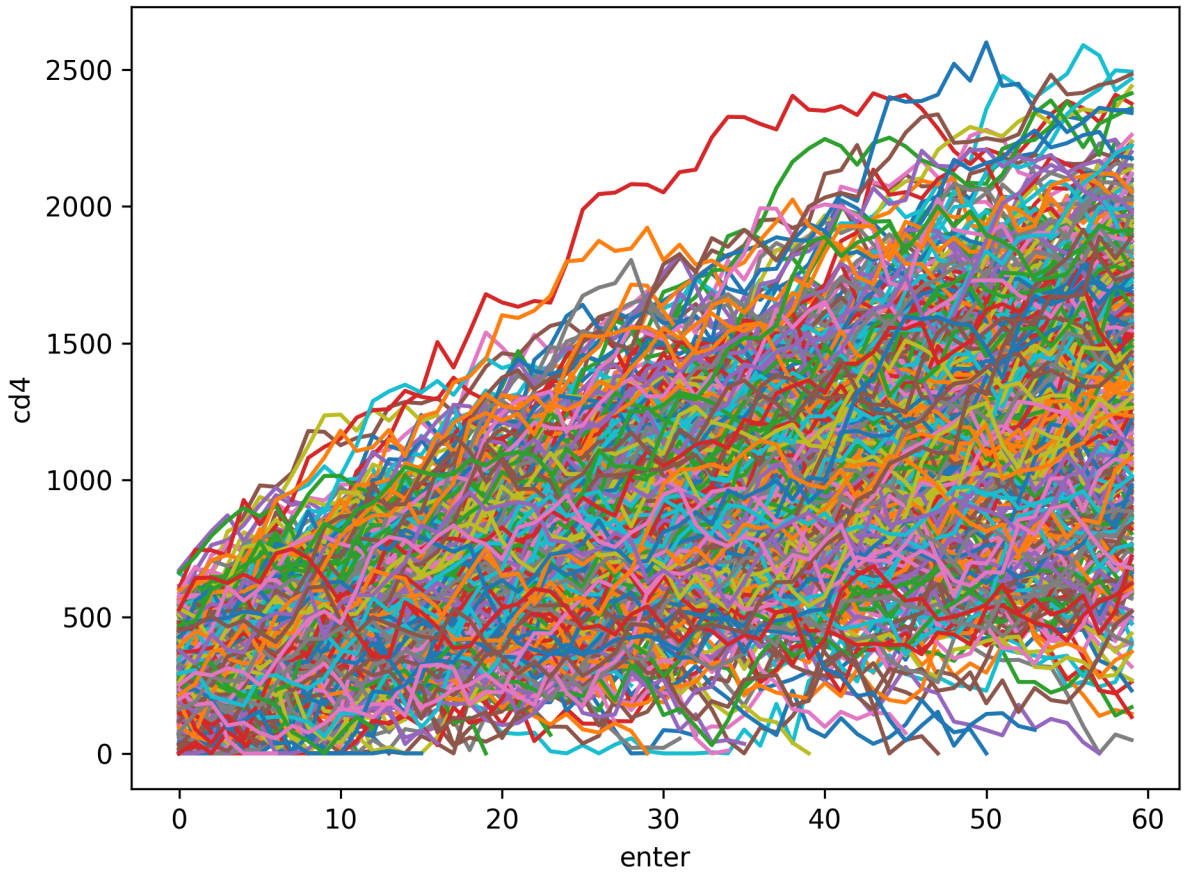
```
from matplotlib.lines import Line2D

pvalue_plot(point=-0.049, sd=0.042, color='b', fill=False)
pvalue_plot(point=-0.062, sd=0.0231, color='r', fill=False)
plt.legend([Line2D([0], [0], color='b', lw=2),
            Line2D([0], [0], color='r', lw=2)],
           ['Our Study', 'Review'])
plt.show()
```



1.6.3 Spaghetti Plot

Spaghetti plots are a fun (sometimes useful) way to look for outliers/patterns in longitudinal data. The following is an example spaghetti plot using the longitudinal data from *zEpid* and looking at CD4 T cell count over time.

```
df = ze.load_sample_data(timevary=True)
ze.graphics.spaghetti_plot(df,idvar='id',variable='cd4',time='enter')
plt.show()
```



NOTE If your data set is particularly large, a spaghetti plot may take a long time to generate and may not be useful as a visualization. They are generally easiest to observe with a smaller number of participants. However, they can be useful for finding extreme outliers in large data sets.

1.6.4 Effect Measure Plots

Effect measure plots are also referred to as forest plots. Forest plots generally summarize the of various studies and collapse the studies into a single summary measure. Effect measure plots are similar but do not use the same summary measure. For an example, I am going to replicate Figure 2 from my 2017 paper "Influenza vaccination status and outcomes among influenza-associated hospitalizations in Columbus, Ohio (2012-2015)" published in *Epidemiology and Infection*

The first step to creating the effect measure plot is to create lists containing; labels, point estimates, lower confidence limits, and upper confidence limits

```
import numpy as np
from zepid.graphics import EffectMeasurePlot

labs = ['Overall', 'Adjusted', '',
        '2012-2013', 'Adjusted', '',
        '2013-2014', 'Adjusted', '',
        '2014-2015', 'Adjusted']
measure = [np.nan, 0.94, np.nan, np.nan, 1.22, np.nan, np.nan, 0.59, np.nan, np.nan,␣
↪1.09]
```

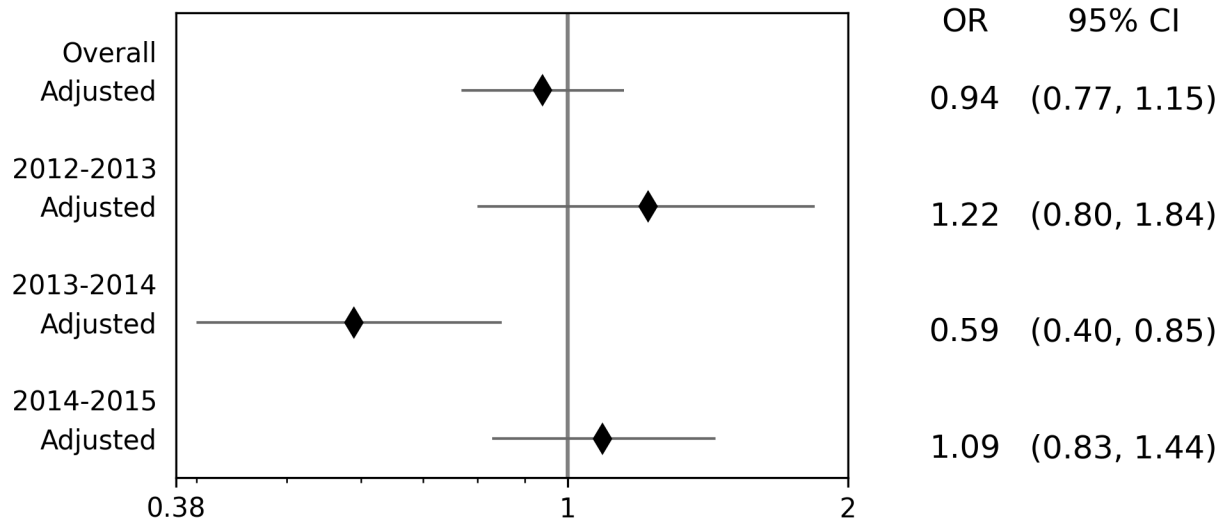
(continues on next page)

(continued from previous page)

```
lower = [np.nan, 0.77, np.nan, np.nan, '0.80', np.nan, np.nan, '0.40', np.nan, np.nan,
↪ 0.83]
upper = [np.nan, 1.15, np.nan, np.nan, 1.84, np.nan, np.nan, 0.85, np.nan, np.nan, 1.
↪44]
```

Some general notes about the above code: (1) for blank y-axis labels, a blank string is indicated, (2) for blank measure/confidence intervals, `np.nan` is specified, (3) for floats ending with a zero, they must be input as characters. If floats that end in `0` (such as `0.80`) are put into a list as a string and not a float, the floating `0` will be truncated from the table. Now that our data is all prepared, we can now generate our plot

```
p = EffectMeasurePlot(label=labs, effect_measure=measure, lcl=lower, ucl=upper)
p.labels(scale='log')
p.plot(figsize=(6.5, 3), t_adjuster=0.02, max_value=2, min_value=0.38)
plt.tight_layout()
plt.show()
```



There are other optional arguments to adjust the plot (colors of points/point shape/etc.). Take a look through the Reference page for available options

NOTE There is one part of the effect measure plot that is not particularly pretty. In the `plot()` function there is an optional argument `t_adjuster`. This argument changes the alignment of the table so that the table aligns properly with the plot values. I have NOT figured out a way to do this automatically. Currently, `t_adjuster` must be changed by the user manually to find a good table alignment. I recommend using changes of `0.01` in `t_adjuster` until a good alignment is found. Additionally, sometimes the plot will be squished. To fix this, the plot size can be changed by the `figsize` argument

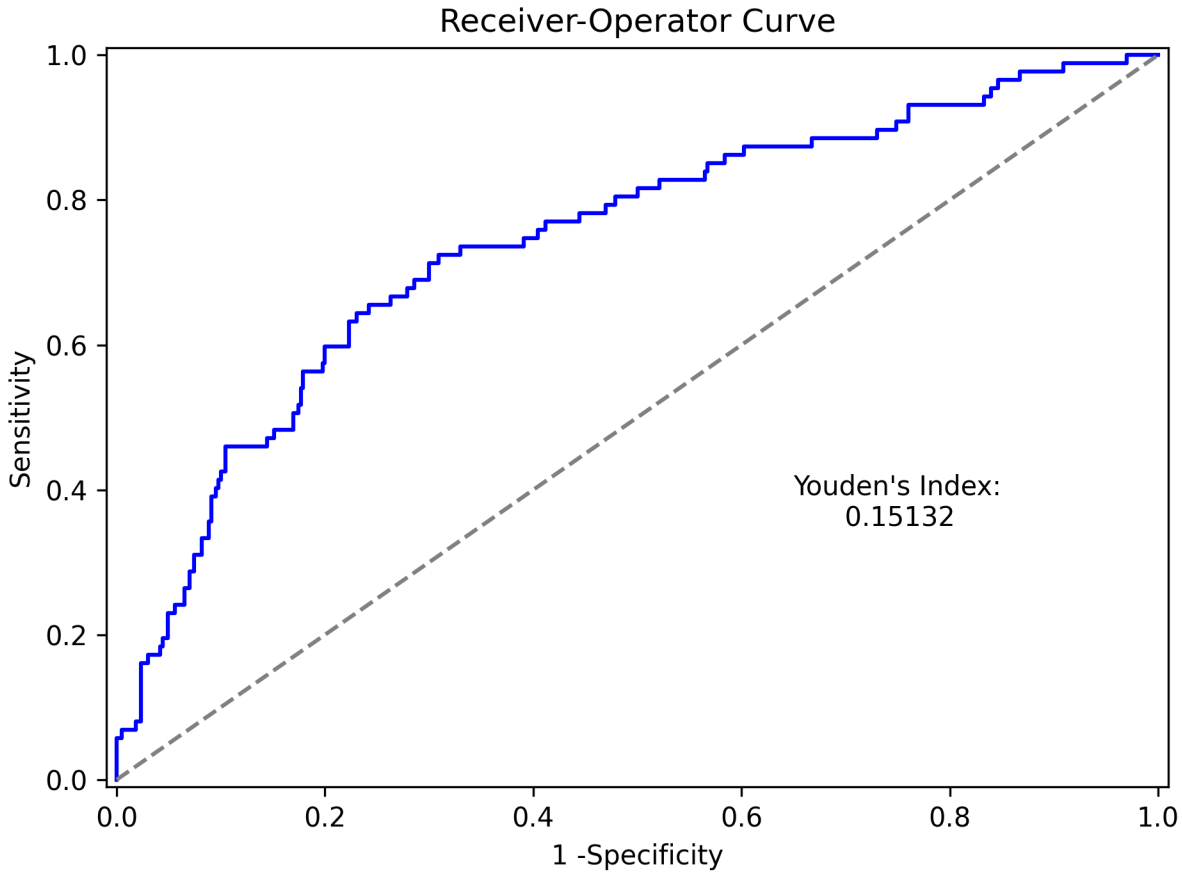
1.6.5 Receiver-Operator Curves

Receiver-Operator Curves (ROC) are a fundamental tool for diagnosing the sensitivity and specificity of a test over a variety of thresholds. ROC curves can be generated for predicted probabilities from a model or different diagnostics thresholds (ex. ALT levels to predict infections). In this example, we will predict the probability of death among the sample data set. First, we will need to get some predicted probabilities. We will use `statsmodels` to build a simple predictive model and obtain predicted probabilities.

```
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.genmod.families import family,links
from zepid.graphics import roc

df = ze.load_sample_data(timevary=False)
f = sm.families.family.Binomial(sm.families.links.logit)
df['age0_sq'] = df['age0']**2
df['cd40sq'] = df['cd40']**2
model = 'dead ~ art + age0 + age0_sq + cd40 + cd40sq + dvl0 + male'
m = smf.glm(model, df, family=f).fit()
df['predicted'] = m.predict(df)
```

Now with predicted probabilities, we can generate a ROC plot

```
roc(df.dropna(), true='dead', threshold='predicted')
plt.tight_layout()
plt.title('Receiver-Operator Curve')
plt.show()
```



Which generates the following plot. For this plot the Youden's Index is also calculated by default. The following output is printed to the console

```
------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Youden's Index:  0.15328818469754796
Predictive values at Youden's Index
   Sensitivity:  0.6739130434782609
       Specificity:  0.6857142857142857
------------------------------------------------------------------------
```

Youden's index is the solution to the following

$$Sensitivity + Specificity - 1$$

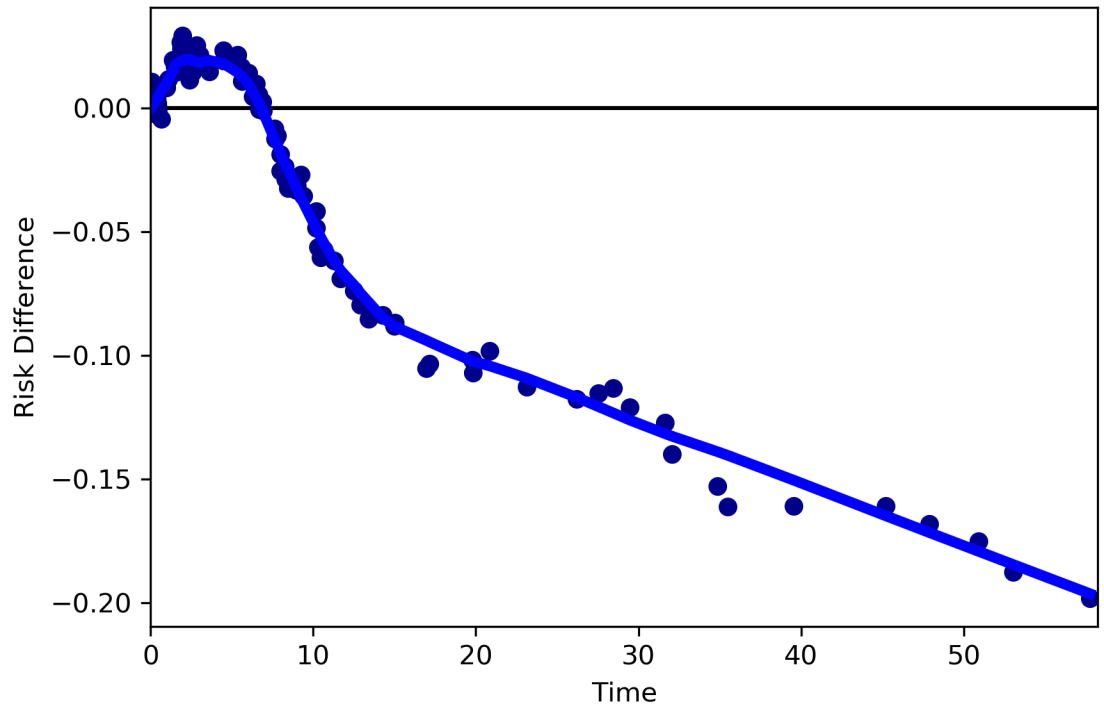
where Youden's index is the value that maximizes the above. Basically, it maximizes both sensitivity and specificity. You can learn more from HERE

1.6.6 Dynamic Risk Plots

Dynamic risk plots allow the visualization of how the risk difference/ratio changes over time. For a published example, see HERE and discussed further HERE

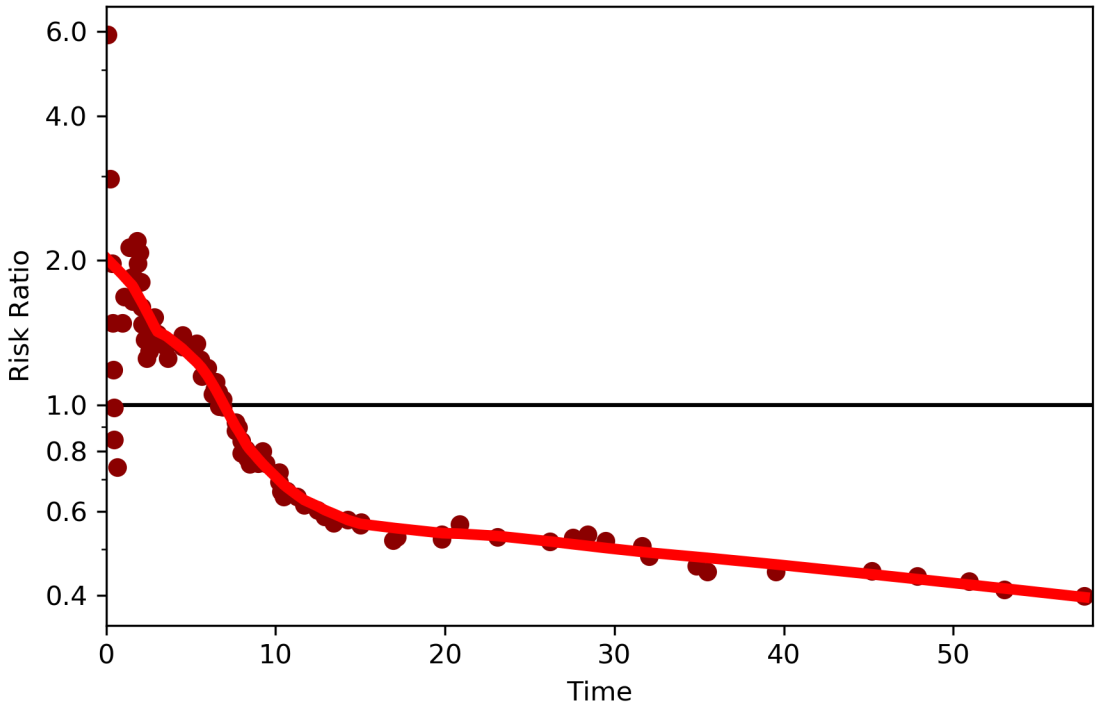
For this example, we will borrow our results from our IPTW marginal structural model. We will use the fitted survival functions to obtain the risk estimates for our exposed and unexposed groups. These were generated from the `lifelines` Kaplan Meier curves (estimated via `KaplanMeierFitter`).

```
a = 1 - kme.survival_function_
b = 1 - kmu.survival_function_
dynamic_risk_plot(a, b)
plt.show()
```



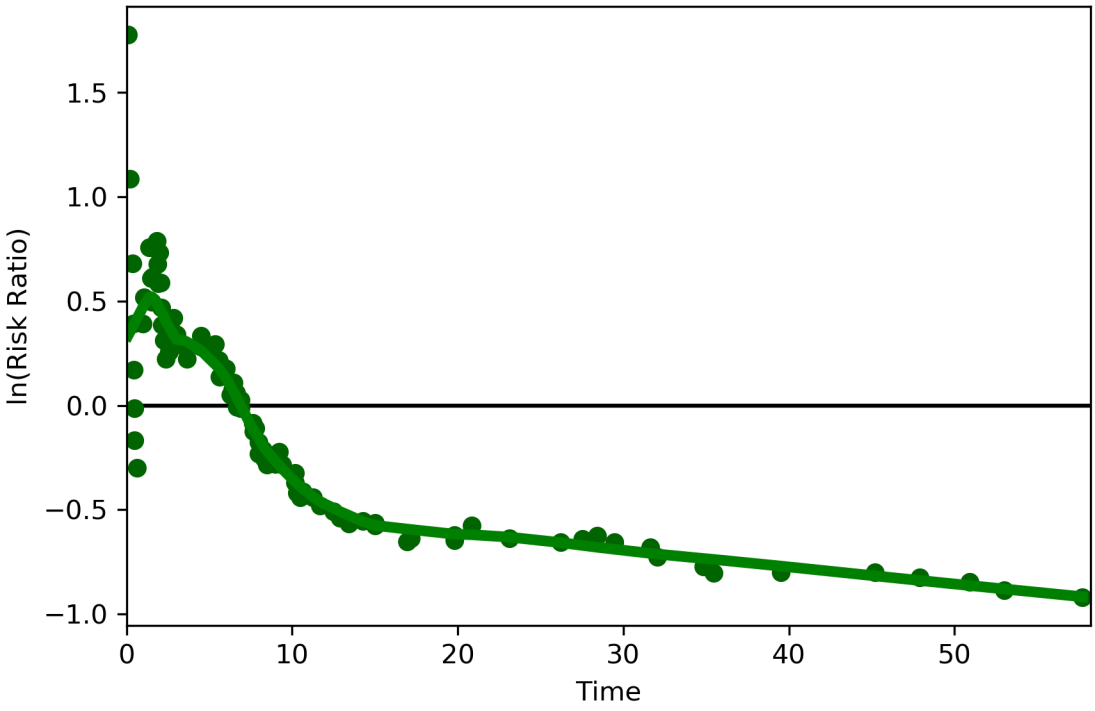
By default, the function returns the risk difference plot. You can also request a risk ratio plot (and with different colors).

```
dynamic_risk_plot(a, b, measure='RR', point_color='darkred', line_color='r', scale=
↪'log')
plt.yticks([0.4, 0.6, 0.8, 1, 2, 4, 6])
plt.show()
```



The log-transformed risk ratio is also available

```
dynamic_risk_plot(a, b, measure='RR', point_color='darkgreen', line_color='g', scale=
↪'log-transform')
plt.show()
```

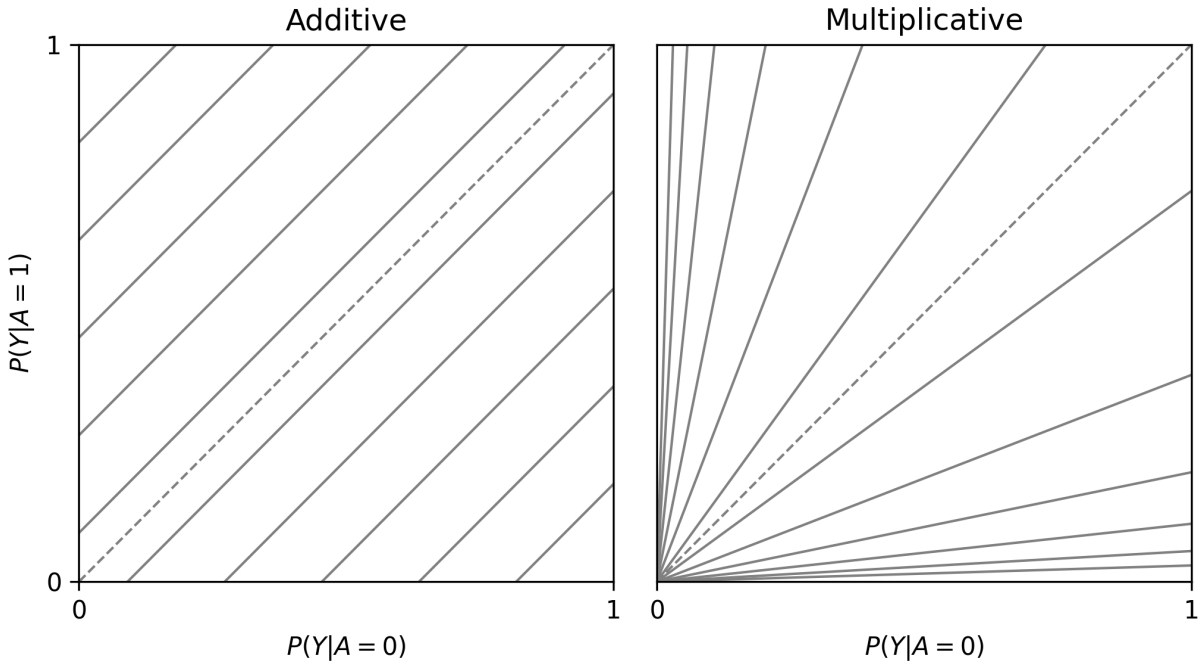


1.6.7 L'Abbe Plots

L'Abbe plots have generally been use to display meta-analysis results. However, I also find them to be a useful tool to explain effect/association measure modification on the additive or the multiplicative scales. Furthermore, it visually demonstrates that when there is a non-null average causal effect, then there must be modification on at least one scale.

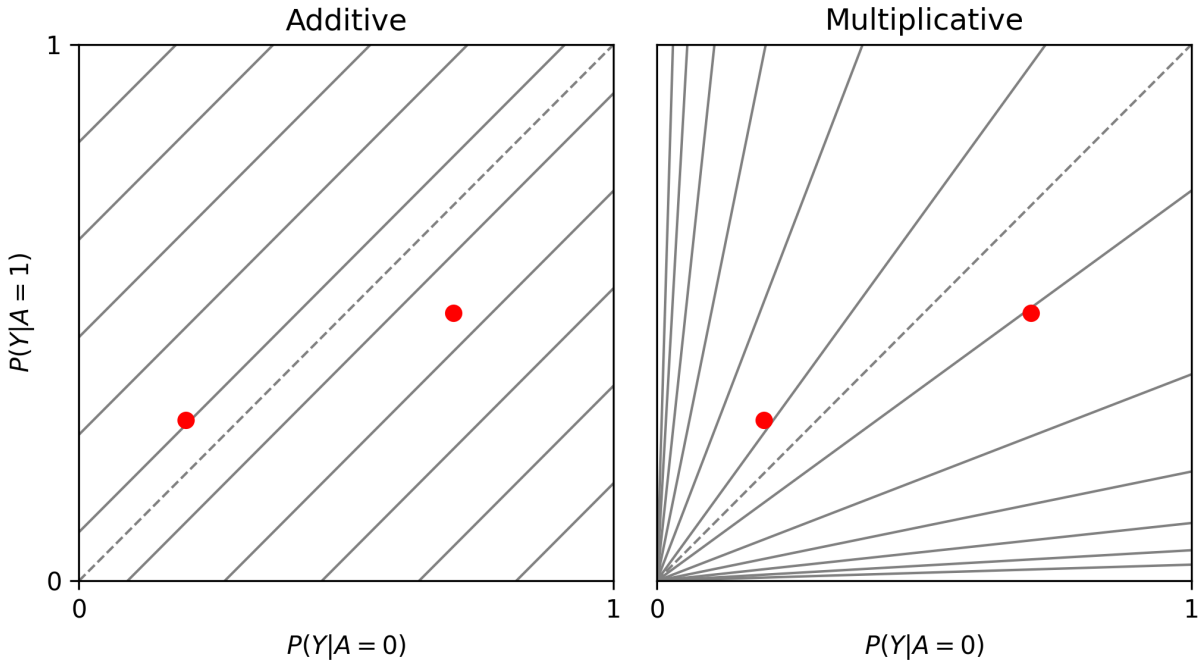
To generate a L'Abbe plot, you can use the `labbe_plot()` function. Below is example code to generate an empty L'Abbe plot.

```
from zepid.graphics import labbe_plot

labbe_plot()
plt.show()
```



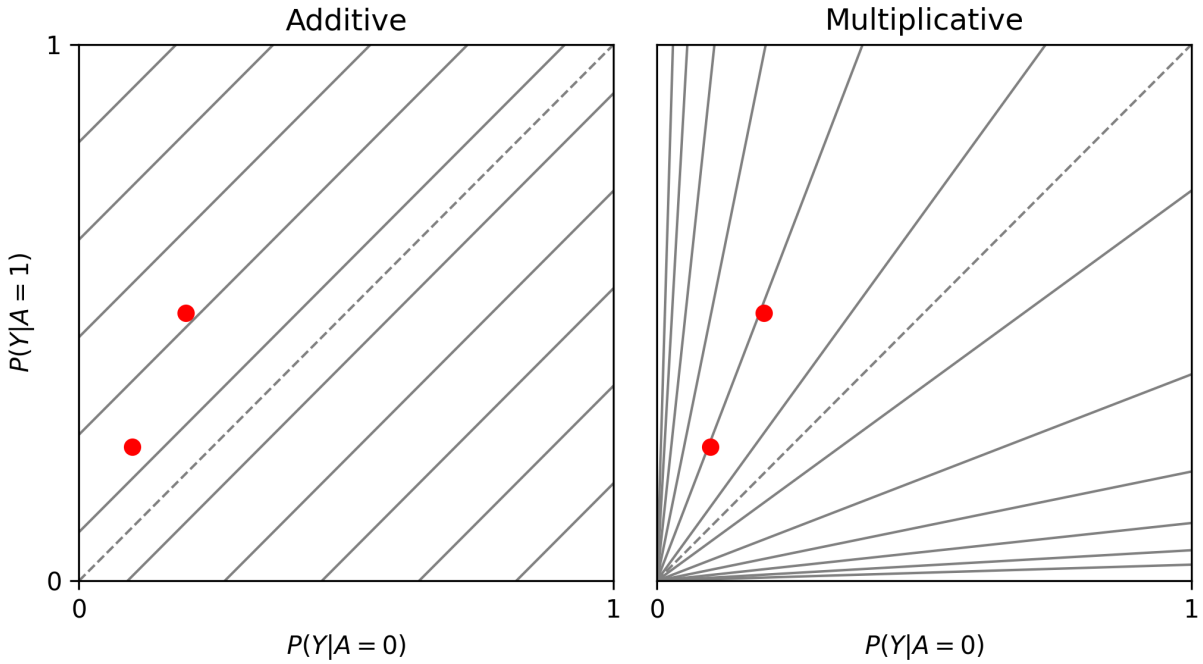
In this plot, you are presented lines that indicate where stratified measures would need to lie on for there to be no additive / multiplicative interaction. By default, both the additive and multiplicative plots are presented. Let's look at an example with some data

```
from zepid.graphics import labbe_plot

labbe_plot(r1=[0.3, 0.5], r0=[0.2, 0.7], color='red')
plt.show()
```



As seen in the plot, there is both additive and multiplicative interaction. As would be described by Hernan, Robins, and others, there is qualitative modification (estimates are on opposite sides of the null, the dashed-line). Let's look at one more example,

```
from zepid.graphics import labbe_plot

labbe_plot(r1=[0.25, 0.5], r0=[0.1, 0.2], color='red')
plt.show()
```

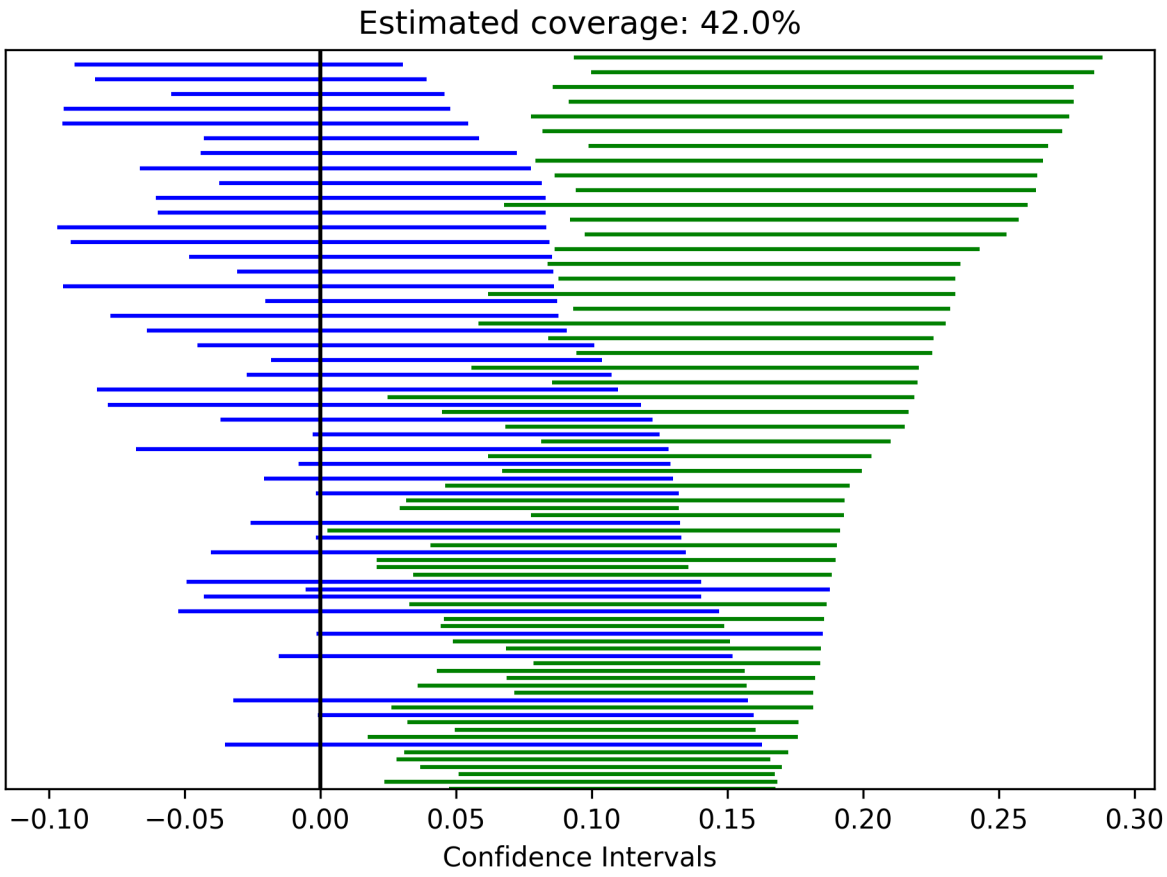


In this example, there is additive modification, but *no multiplicative modification*. These plots also can have the number of reference lines displayed changed, and support the keyword arguments of *plt.plot()* function. See the function documentation for further details.

1.6.8 Zipper Plot

Zipper plots provide an easy way to visualize the performance of confidence intervals in simulations. Confidence intervals across simulations are displayed in a single plot, with the option to color the confidence limits by whether they include the true value. Below is an example of a zipper plot. For ease, I generated the confidence intervals using some random numbers (you would pull the confidence intervals from the estimators in practice).

```
from zepid.graphics import zipper_plot
lower = np.random.uniform(-0.1, 0.1, size=100)
upper = lower + np.random.uniform(0.1, 0.2, size=100)

zipper_plot(truth=0,
            lcl=lower,
            ucl=upper,
            colors=('blue', 'green'))
plt.show()
```



In this example, confidence interval coverage would be considered rather poor (if we are expecting the usual 95% coverage).

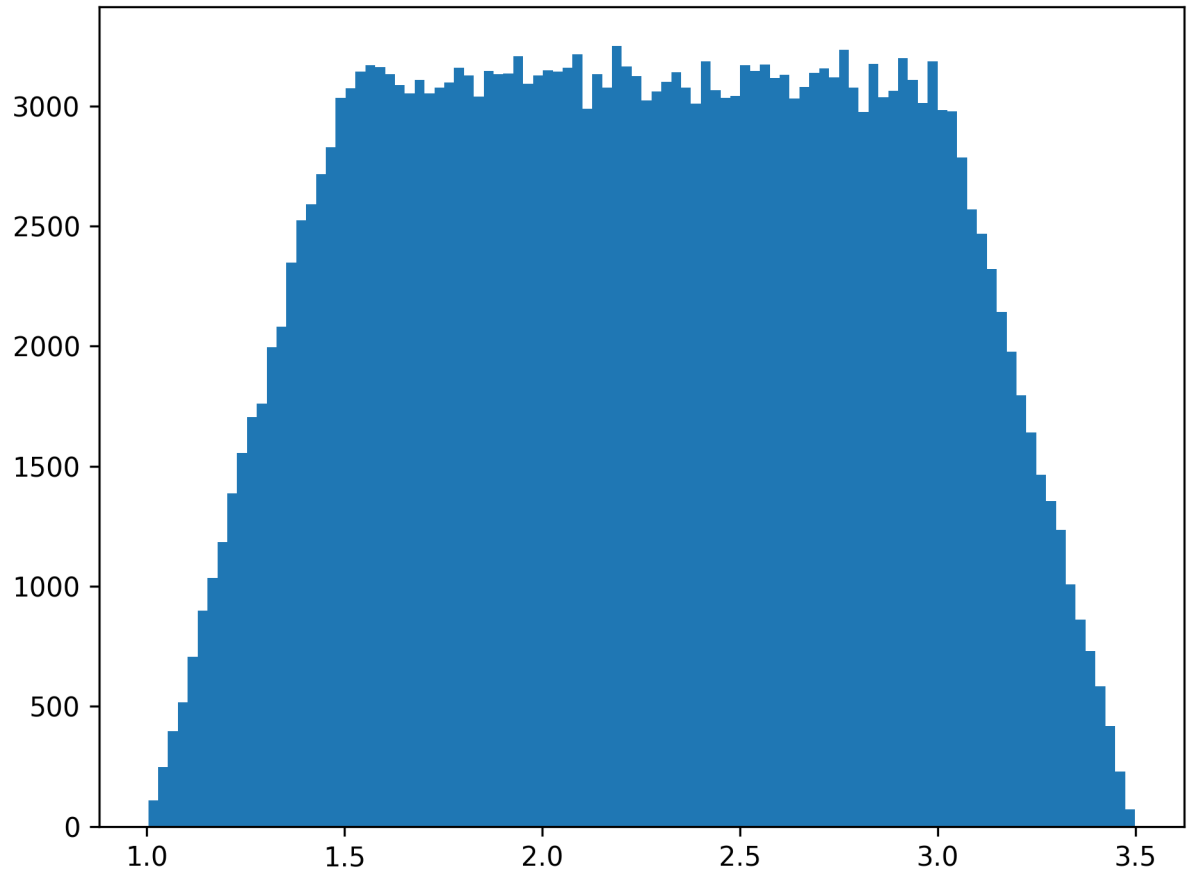
1.7 Sensitivity Analyses

Sensitivity analyses are a way to determine the robustness of findings against certain assumptions or unmeasured factors. Currently, *zEpid* supports Monte Carlo bias analysis

1.7.1 Trapezoidal Distribution

NumPy doesn't have a trapezoidal distribution, so this is an implementation. The trapezoid distribution is contains a central "zone of indifference" where values are from a uniform distribution. The tails of this distribution reflect the uncertainty around the edges of the distribution. I think a visual will explain it more clearly, so let's generate one

```
from zepid.sensitivity_analysis import trapezoidal
import matplotlib.pyplot as plt

plt.hist(trapezoidal(mini=1, mode1=1.5, mode2=3, maxi=3.5, size=250000), bins=100)
plt.show()
```



As can be seen in the histogram, `mini` refers to the smallest value of the distribution, `maxi` refers to the maximum value of the distribution, and `mode1` and `mode2` refer to the start and end of the uniform distribution respectively. `size` is how many samples to draw from the distribution. When `size` is not specified, a single draw from the distribution is generated.

```
trapezoidal(mini=1, mode1=1.5, mode2=3, maxi=3.5)
```

1.7.2 Monte Carlo Risk Ratio

As described in Lash TL, Fink AK 2003 and Fox et al. 2005 , a probability distribution is defined for unmeasured confounder - outcome risk ratio, proportion of individuals in exposed group with unmeasured confounder, and proportion of individuals in unexposed group with unmeasured confounder. This version only supports binary exposures, binary outcomes, and binary unmeasured confounders.

```
import matplotlib.pyplot as plt()
from zepid.sensitivity_analysis import MonteCarloRR, trapezoidal
```

Below is code to complete the Monte Carlo bias analysis procedure

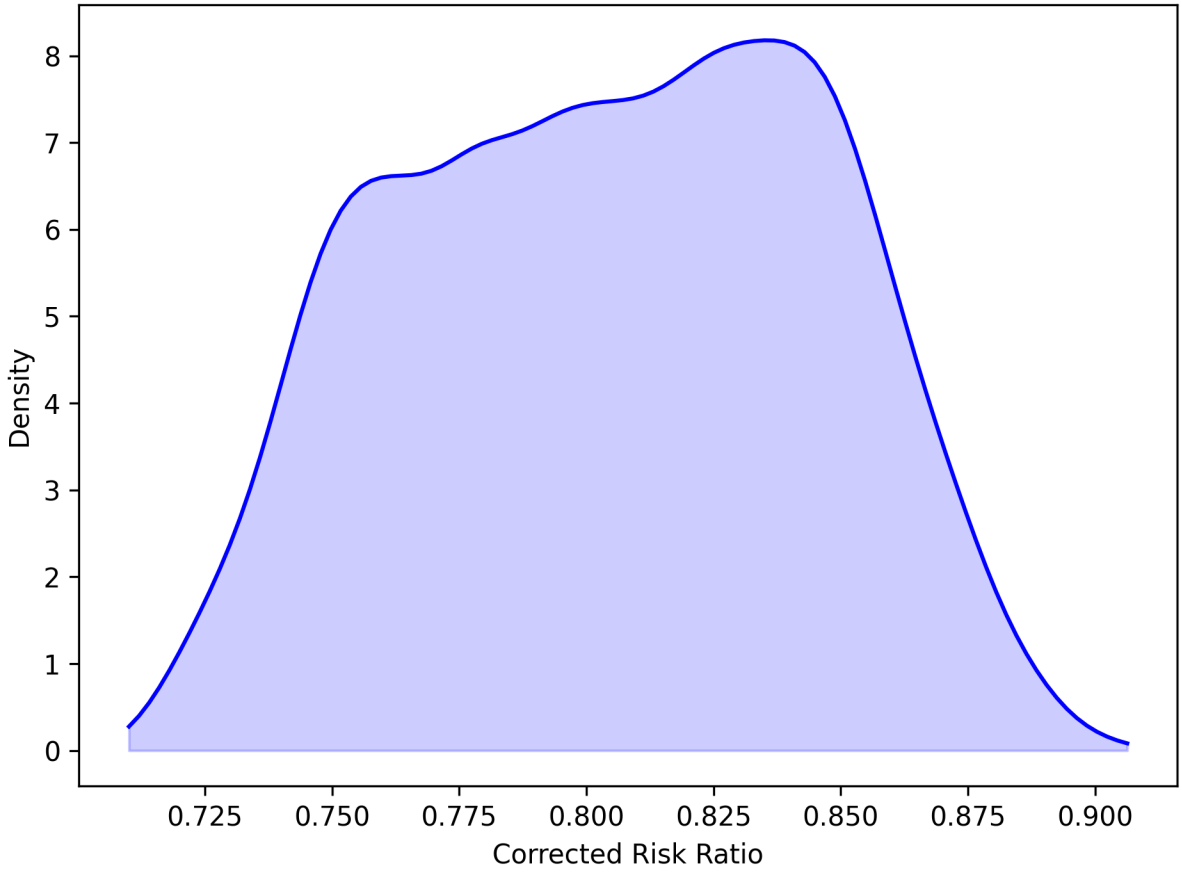
```
mcrr = MonteCarloRR(observed_RR=0.73322, sample=10000)
mcrr.confounder_RR_distribution(trapezoidal(mini=0.9, mode1=1.1, mode2=1.7, maxi=1.8,␣
↪size=10000))
mcrr.prop_confounder_exposed(trapezoidal(mini=0.25, mode1=0.28, mode2=0.32, maxi=0.35,
↪ size=10000))
mcrr.prop_confounder_unexposed(trapezoidal(mini=0.55, mode1=0.58, mode2=0.62, maxi=0.
↪65, size=10000))
mcrr.fit()
```

We can view basic summary information about the distribution of the corrected Risk Ratios

```
mcrr.summary()
```

Alternatively, we can easily get a kernel density plot of the distribution of corrected RR

```
mcrr.plot()
plt.show()
```



1.8 Reference

This page links to documentation for each class of functions. The specific function, the parameters, and an example are provided for each. *Calculations* contains information on the summary measure calculators, *Graphics* details the graphic generators, *Causal* details the implemented causal inference methods, *Sensitivity* details the sensitivity analysis

tools, and *Data* details the data sets included with zEpid. For a more narrative-driven description of the tools, please see the side-bar for each corresponding section.

1.8.1 Measures

Below is documentation for each of the implemented calculation functionalities available for a pandas DataFrame

Measures

RiskRatio([reference, alpha])	Estimate of Risk Ratio with a (1-alpha)*100% Confidence interval from a pandas DataFrame.
RiskDifference([reference, alpha])	Estimate of Risk Difference with a (1-alpha)*100% Confidence interval from a pandas DataFrame.
NNT([reference, alpha])	Estimates of Number Needed to Treat.
OddsRatio([reference, alpha])	Estimates of Odds Ratio with a (1-alpha)*100% Confidence interval.
IncidenceRateRatio([reference, alpha])	Estimates of Incidence Rate Ratio with a (1-alpha)*100% Confidence interval.
IncidenceRateDifference([reference, alpha])	Estimates of Incidence Rate Difference with a (1-alpha)*100% Confidence interval.
interaction_contrast(df, exposure, outcome, ...)	Calculate the Interaction Contrast (IC) using a pandas dataframe and statsmodels to fit a linear binomial regression.
interaction_contrast_ratio(df, exposure, ...)	Calculate the Interaction Contrast Ratio (ICR) using a pandas dataframe, and conducts either log binomial or logistic regression through statsmodels.

zepid.base.RiskRatio

class zepid.base.**RiskRatio**(*reference=0*, *alpha=0.05*)

Estimate of Risk Ratio with a (1-alpha)*100% Confidence interval from a pandas DataFrame. Missing data is ignored. Exposure categories should be mutually exclusive

Risk ratio is calculated from

$$RR = \frac{\Pr(Y|A=1)}{\Pr(Y|A=0)}$$

Risk ratio standard error is

$$SE = \left(\frac{1}{a} - \frac{1}{a+b} + \frac{1}{c} - \frac{1}{c+d}\right)^{\frac{1}{2}}$$

Note: Outcome must be coded as (1: yes, 0:no). Only works supports binary outcomes

Parameters

- **reference** (*integer, optional*) – Reference category for comparisons. Default reference category is 0
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the risk ratio in a data set

```
>>> from zepid import RiskRatio, load_sample_data
>>> df = load_sample_data(False)
>>> rr = RiskRatio()
>>> rr.fit(df, exposure='art', outcome='dead')
>>> rr.summary()
```

Calculate the risk ratio with exposure of '1' as the reference category

```
>>> rr = RiskRatio(reference=1)
>>> rr.fit(df, exposure='art', outcome='dead')
>>> rr.summary()
```

Generate a plot of the calculated risk ratio(s)

```
>>> import matplotlib.pyplot as plt
>>> rr = RiskRatio()
>>> rr.fit(df, exposure='art', outcome='dead')
>>> rr.plot()
>>> plt.show()
```

__init__(*reference=0*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([reference, alpha])	Initialize self.
fit(df, exposure, outcome)	Calculates the Risk Ratio given a data set
plot([measure, scale, center])	Plot the risk ratios or the risks along with their corresponding confidence intervals.
summary([decimal])	Prints the summary results

zepid.base.RiskDifference

class zepid.base.**RiskDifference**(*reference=0*, *alpha=0.05*)
Estimate of Risk Difference with a (1-alpha)*100% Confidence interval from a pandas DataFrame. Missing data is ignored. Exposure categories should be mutually exclusive

Risk difference is calculated as

$$RD = \Pr(Y|A=1) - \Pr(Y|A=0)$$

Risk difference standard error is calculated as

$$SE = \left(\frac{R_1 \times (1-R_1)}{a+b} + \frac{R_0 \times (1-R_0)}{c+d} \right)^{\frac{1}{2}}$$

In addition to confidence intervals, the Frechet bounds are calculated as well. These probability bounds are useful for a comparison. Within these bounds, the true causal risk difference in the sample must live. The only assumptions these bounds require are no measurement error, causal consistency, no selection bias, and any missing data is MCAR. These bounds are always unit width (width of one), but they do not require any

assumptions regarding confounding / conditional exchangeability. They are calculated via the following formula

$$Lower = \Pr(Y|A=a)\Pr(A=a) - \Pr(Y|A\neq a)\Pr(A\neq a) - \Pr(A=a)$$
$$Upper = \Pr(Y|A=a)\Pr(A=a) + \Pr(A\neq a) - \Pr(Y|A\neq a)\Pr(A\neq a)$$

For further details on these bounds, see the references

Note: Outcome must be coded as (1: yes, 0:no). Only supports binary outcomes

---

Parameters

- **reference** (*integer, optional*) – -reference category for comparisons. Default reference category is 0
- **alpha** (*float, optional*) – -Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

References

Cole SR et al. (2019) Nonparametric Bounds for the Risk Function. American Journal of Epidemiology. 188(4), 632-636

Examples

Calculate the risk difference in a data set

```
>>> from zepid import RiskDifference, load_sample_data
>>> df = load_sample_data(False)
>>> rd = RiskDifference()
>>> rd.fit(df, exposure='art', outcome='dead')
>>> rd.summary()
```

Calculate the risk difference with exposure of '1' as the reference category

```
>>> rd = RiskDifference(reference=1)
>>> rd.fit(df, exposure='art', outcome='dead')
>>> rd.summary()
```

Generate a plot of the calculated risk difference(s)

```
>>> import matplotlib.pyplot as plt
>>> rd = RiskDifference()
>>> rd.fit(df, exposure='art', outcome='dead')
>>> rd.plot()
>>> plt.show()
```

__init__(*reference=0*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([reference, alpha])	Initialize self.
`fit`(df, exposure, outcome)	Calculates the Risk Difference
`plot`([measure, center])	Plot the risk differences or the risks along with their corresponding confidence intervals.
`summary`([decimal])	Prints the summary results

zepid.base.NNT

class zepid.base.**NNT**(*reference=0*, *alpha=0.05*)

Estimates of Number Needed to Treat. NNT (1-alpha)*100% confidence interval presentation is based on Altman, DG (BMJ 1998). Missing data is ignored

Number needed to treat is calculated as

$$NNT = \frac{1}{RD}$$

Risk difference the corresponding confidence intervals come from

$$RD = \Pr(Y|A=1) - \Pr(Y|A=0)$$

Risk difference standard error is calculated as

$$SE = \left(\frac{R_1 \times (1-R_1)}{a+b} + \frac{R_0 \times (1-R_0)}{c+d} \right)^{\frac{1}{2}}$$

Note: Outcome must be coded as (1: yes, 0:no). Only works for binary outcomes

Parameters

- **reference** (*integer, optional*) – Reference category for comparisons. Default reference category is 0
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the number needed to treat in a data set

```
>>> from zepid import NNT, load_sample_data
>>> df = load_sample_data(False)
>>> nnt = NNT()
>>> nnt.fit(df, exposure='art', outcome='dead')
>>> nnt.summary()
```

Calculate the number needed to treat with '1' as the reference category

```
>>> nnt = NNT(reference=1)
>>> nnt.fit(df, exposure='art', outcome='dead')
>>> nnt.summary()
```

__init__(*reference=0*, *alpha=0.05*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([reference, alpha])	Initialize self.
`fit`(df, exposure, outcome)	Calculates the NNT
`summary`([decimal])	Prints the summary results

zepid.base.OddsRatio

class zepid.base.**OddsRatio**(*reference=0*, *alpha=0.05*)

Estimates of Odds Ratio with a (1-alpha)*100% Confidence interval. Missing data is ignored

Odds ratio is calculated from

$$OR = \frac{\Pr(Y|A=1)}{1-\Pr(Y|A=1)} \Big/ \frac{\Pr(Y|A=0)}{1-\Pr(Y|A=0)}$$

Odds ratio standard error is

$$SE = \left(\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d}\right)^{\frac{1}{2}}$$

Note: Outcome must be coded as (1: yes, 0:no). Only works for binary outcomes

Parameters

- **reference** (*integer, optional*) – Reference category for comparisons. Default reference category is 0
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the odds ratio in a data set

```
>>> from zepid import OddsRatio, load_sample_data
>>> df = load_sample_data(False)
>>> ort = OddsRatio()
>>> ort.fit(df, exposure='art', outcome='dead')
>>> ort.summary()
```

Calculate the odds ratio with exposure of '1' as the reference category

```
>>> ort = OddsRatio(reference=1)
>>> ort.fit(df, exposure='art', outcome='dead')
>>> ort.summary()
```

Generate a plot of the calculated odds ratio(s)

```
>>> import matplotlib.pyplot as plt
>>> ort = OddsRatio()
>>> ort.fit(df, exposure='art', outcome='dead')
>>> ort.plot()
>>> plt.show()
```

__init__(*reference=0*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([reference, alpha])	Initialize self.
fit(df, exposure, outcome)	Calculates the Odds Ratio
plot([scale, center])	Plot the odds ratios along with their corresponding confidence intervals.
summary([decimal])	Prints the summary results

zepid.base.IncidenceRateRatio

class zepid.base.**IncidenceRateRatio**(*reference=0*, *alpha=0.05*)
Estimates of Incidence Rate Ratio with a (1-alpha)*100% Confidence interval. Missing data is ignored

Incidence rate ratio is calculated from

$$IR = \frac{a}{t_1} / \frac{c}{t_0}$$

Incidence rate ratio standard error is

$$SE = \left(\frac{1}{a} + \frac{1}{c} \right)^{\frac{1}{2}}$$

Note: Outcome must be coded as (1: yes, 0:no). Only works for binary outcomes

Parameters

- **reference** (*integer, optional*) – Reference category for comparisons. Default reference category is 0
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the incidence rate ratio in a data set

```
>>> from zepid import IncidenceRateRatio, load_sample_data
>>> df = load_sample_data(False)
>>> irr = IncidenceRateRatio()
>>> irr.fit(df, exposure='art', outcome='dead', time='t')
>>> irr.summary()
```

Calculate the incidence rate ratio with exposure of '1' as the reference category

```
>>> irr = IncidenceRateRatio(reference=1)
>>> irr.fit(df, exposure='art', outcome='dead', time='t')
>>> irr.summary()
```

Generate a plot of the calculated incidence rate ratio(s)

```
>>> import matplotlib.pyplot as plt
>>> irr = IncidenceRateRatio()
>>> irr.fit(df, exposure='art', outcome='dead', time='t')
>>> irr.plot()
>>> plt.show()
```

__init__(*reference=0*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([reference, alpha])	Initialize self.
fit(df, exposure, outcome, time)	Calculate the Incidence Rate Ratio
plot([measure, scale, center])	Plot the risk ratios or the risks along with their corresponding confidence intervals.
summary([decimal])	Prints the summary results

zepid.base.IncidenceRateDifference

class zepid.base.**IncidenceRateDifference**(*reference=0*, *alpha=0.05*)
Estimates of Incidence Rate Difference with a (1-alpha)*100% Confidence interval. Missing data is ignored.

Incidence rate difference is calculated from

$$ID = \frac{a}{t_1} - \frac{c}{t_0}$$

Incidence rate difference standard error is

$$SE = \left(\frac{a}{t_1^2} + \frac{c}{t_0^2}\right)^{\frac{1}{2}}$$

Note: Outcome must be coded as (1: yes, 0:no). Only works for binary outcomes

Parameters

- **reference** (*integer, optional*) – Reference category for comparisons. Default reference category is 0
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the incidence rate difference in a data set

```
>>> from zepid import IncidenceRateDifference, load_sample_data
>>> df = load_sample_data(False)
>>> ird = IncidenceRateDifference()
>>> ird.fit(df, exposure='art', outcome='dead', time='t')
>>> ird.summary()
```

Calculate the incidence rate difference with exposure of '1' as the reference category

```
>>> ird = IncidenceRateDifference(reference=1)
>>> ird.fit(df, exposure='art', outcome='dead', time='t')
>>> ird.summary()
```

Generate a plot of the calculated incidence rate difference(s)

```
>>> import matplotlib.pyplot as plt
>>> ird = IncidenceRateDifference()
>>> ird.fit(df, exposure='art', outcome='dead', time='t')
>>> ird.plot()
>>> plt.show()
```

__init__(*reference=0*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([reference, alpha])	Initialize self.
fit(df, exposure, outcome, time)	Calculates the Incidence Rate Difference
plot([measure, center])	Plot the incidence rate differences or the incidence rates along with their corresponding confidence intervals.
summary([decimal])	Prints the summary results

zepid.base.interaction_contrast

zepid.base.**interaction_contrast**(*df*, *exposure*, *outcome*, *modifier*, *adjust=None*, *decimal=3*, *print_results=True*)

Calculate the Interaction Contrast (IC) using a pandas dataframe and statsmodels to fit a linear binomial regression. Can ONLY be used for a 0,1 coded exposure and modifier (exposure = {0,1}, modifier = {0,1}, outcome = {0,1}). Can handle adjustment for other confounders in the regression model. Prints the fit of the linear binomial regression, the IC, and the corresponding IC 95% confidence interval.

Interaction Contrast is defined as the following

$$IC = RD_{11} - RD_{10} - RD_{01}$$

Note: statsmodels may produce a domain error in some versions.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing variables of interest
- **exposure** (*string*) – Column name of exposure variable. Must be coded as (0,1) where 1 is exposure
- **outcome** (*string*) – Column name of outcome variable. Must be coded as (0,1) where 1 is outcome of interest
- **modifier** (*string*) – Column name of modifier variable. Must be coded as (0,1) where 1 is modifier

- **adjust** (*string, optional*) – String of other variables to adjust for, in correct statsmodels format. Default is None. Variables can *NOT* be named {E1M0,E0M1,E1M1} since this function creates variables with those names. Answers will be incorrect Example of accepted input is 'C1 + C2 + C3 + Z'
- **decimal** (*integer, optional*) – Decimal places to display in result. Default is 3
- **print_results** (*bool, optional*) – Whether to print results from interaction contrast assessment

Examples

Setting up environment

```
>>> from zepid import interaction_contrast, load_sample_data
>>> df = load_sample_data(False)
```

Calculating interaction contrast for ART and gender

```
>>> interaction_contrast(df, exposure='art', outcome='dead', modifier='male')
```

zepid.base.interaction_contrast_ratio

zepid.base.**interaction_contrast_ratio**(*df*, *exposure*, *outcome*, *modifier*, *adjust=None*, *regression='logit'*, *ci='delta'*, *b_sample=200*, *alpha=0.05*, *decimal=5*, *print_results=True*)

Calculate the Interaction Contrast Ratio (ICR) using a pandas dataframe, and conducts either log binomial or logistic regression through statsmodels. Can ONLY be used for a 0,1 coded exposure and modifier (exposure = {0,1}, modifier = {0,1}, outcome = {0,1}). Can handle missing data and adjustment for other confounders in the regression model. Prints the fit of the binomial regression, the ICR, and the corresponding ICR confidence interval

Interaction contrast ratio is defined as

$$ICR = RR_{11} - RR_{10} - RR_{01} + 1$$

Confidence intervals can be generated either through a bootstrap procedure or using the delta method

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing variables of interest
- **exposure** (*string*) – Column name of exposure variable. Must be coded as (0,1) where 1 is exposure
- **outcome** (*string*) – Column name of outcome variable. Must be coded as (0,1) where 1 is outcome of interest
- **modifier** (*string*) – Column name of modifier variable. Must be coded as (0,1) where 1 is modifier
- **adjust** (*string, optional*) – String of other variables to adjust for, in correct statsmodels format. Default is None. Variables can *NOT* be named {E1M0,E0M1,E1M1} since this function creates variables with those names. Answers will be incorrect Example of accepted input is 'C1 + C2 + C3 + Z'

- **regression** (*string, optional*) – Type of regression model to fit. Default is 'log' which fits the log-binomial model. Options include: * 'log' Log-binomial model. Estimates the Risk Ratio * 'logit' Logistic model. Estimates Odds Ratio. Only valid when odds ratio approximates the risk ratio
- **ci** (*string, optional*) – Type of confidence interval to return. Default is the delta method. Options include: * 'delta': Delta method as described by Hosmer and Lemeshow (1992) * 'bootstrap': bootstrap method (Assmann et al. 1996). The delta method is more time efficient than bootstrap
- **b_sample** (*integer, optional*) – Number of times to resample to generate bootstrap confidence intervals. Only used if bootstrap confidence intervals are requested. Default is 1000
- **alpha** (*float, optional*) – Alpha level for confidence interval. Default is 0.05, which returns 95% confidence intervals
- **decimal** (*integer, optional*) – Decimal places to display in result. Default is 3
- **print_results** (*bool, optional*) – Whether to print results from interaction contrast assessment

Note: statsmodels may produce a domain error for log binomial models in some versions

Examples

Setting up environment

```
>>> from zepid import interaction_contrast_ratio, load_sample_data
>>> df = load_sample_data(False)
```

Calculating interaction contrast ratio for ART and gender

```
>>> interaction_contrast_ratio(df, exposure='art', outcome='dead', modifier='male
↪')
```

Calculating interaction contrast ratio for ART and gender, confidence intervals from bootstrap

```
>>> interaction_contrast_ratio(df, exposure='art', outcome='dead', modifier='male
↪', ci='bootstrap')
```

Diagnostics

Sensitivity([alpha])	Generates the sensitivity and (1-alpha)% confidence interval, comparing test results to disease status from pandas dataframe
Specificity([alpha])	Generates the sensitivity and (1-alpha)% confidence interval, comparing test results to disease status from pandas dataframe
Diagnostics([alpha])	Generates the Sensitivity, Specificity, and the corresponding (1-alpha)% confidence intervals, comparing test results to disease status from pandas DataFrame

zepid.base.Sensitivity

class zepid.base.**Sensitivity**(*alpha=0.05*)

Generates the sensitivity and (1-alpha)% confidence interval, comparing test results to disease status from pandas dataframe

Sensitivity is calculated from

$$Sensitivity = \frac{TP}{P}$$

Wald standard error is

$$SE_{Wald} = \left(\frac{1}{TP} - \frac{1}{P}\right)^{\frac{1}{2}}$$

Note: Disease & Test must be coded as (1: yes, 0:no)

Parameters **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the sensitivity in a data set

```
>>> from zepid import Sensitivity, load_sample_data
>>> df = load_sample_data(False)
>>> sens = Sensitivity()
>>> sens.fit(df, test='art', disease='dead')  # Note this example is not great...␣
↪ART is a treatment not test
>>> sens.summary()
```

__init__(*alpha=0.05*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([alpha])	Initialize self.
fit(df, test, disease)	Calculates the Sensitivity
summary([decimal])	Prints the summary results

zepid.base.Specificity

class zepid.base.**Specificity**(*alpha=0.05*)

Generates the sensitivity and (1-alpha)% confidence interval, comparing test results to disease status from pandas dataframe

Specificity is calculated from

$$Sp = \frac{FN}{N}$$

Wald standard error is

$$SE_{Wald} = (\frac{1}{FN} - \frac{1}{N})^{\frac{1}{2}}$$

Note: Disease & Test must be coded as (1: yes, 0:no)

Parameters **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the specificity in a data set >>> from zepid import Specificity, load_sample_data >>> df = load_sample_data(False) >>> spec = Specificity() >>> spec.fit(df, test='art', disease='dead') # Note this example is not great... ART is a treatment not test >>> spec.summary()

__init__(*alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([alpha])	Initialize self.
fit(df, test, disease)	Calculates specificity
summary([decimal])	Prints the summary results

zepid.base.Diagnostics

class zepid.base.**Diagnostics**(*alpha=0.05*)
Generates the Sensitivity, Specificity, and the corresponding (1-alpha)% confidence intervals, comparing test results to disease status from pandas DataFrame

Sensitivity is calculated from

$$Se = \frac{TP}{P}$$

Wald standard error is

$$SE_{Wald} = \left(\frac{1}{TP} - \frac{1}{P}\right)^{\frac{1}{2}}$$

Specificity is calculated from

$$Sp = \frac{FN}{N}$$

Wald standard error is

$$SE_{Wald} = \left(\frac{1}{FN} - \frac{1}{N}\right)^{\frac{1}{2}}$$

Note: Disease & Test must be coded as (1: yes, 0:no)

Parameters **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Examples

Calculate the sensitivity and specificity in a data set

```
>>> from zepid import Diagnostics, load_sample_data
>>> df = load_sample_data(False)
>>> diag = Diagnostics()
>>> diag.fit(df, test='art', disease='dead')  # Note this example is not great...␣
↪ART is a treatment not test
>>> diag.summary()
```

__init__(*alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([alpha])	Initialize self.
fit(df, test, disease)	Calculates sensitivity and specificity
summary([decimal])	Prints the results

Others

spline(df, var[, n_knots, knots, term, ...])	Creates spline dummy variables based on either user specified knot locations or automatically determines knot locations based on percentiles.
create_spline_transform(array[, n_knots, ...])	Creates spline dummy variables based on either user specified knot locations or automatically determines knot locations based on percentiles.
table1_generator(df, cols, variable_type[, ...])	Code to automatically generate a descriptive table of your study population (often referred to as a Table 1).

zepid.base.spline

zepid.base.**spline**(*df*, *var*, *n_knots=3*, *knots=None*, *term=1*, *restricted=False*)
Creates spline dummy variables based on either user specified knot locations or automatically determines knot locations based on percentiles. Options are available to set the number of knots, location of knots (value), term (linear, quadratic, etc.), and restricted/unrestricted.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **var** (*string*) – Continuous variable to generate spline for
- **n_knots** (*integer, optional*) – Number of knots requested. Options for knots include any positive integer if the location of knots are specified. If knot locations are not specified, n_knots must be an integer between 1 to 7. Default is 3 knots
- **knots** (*list, optional*) – Location of specified knots in a list. To specify the location of knots, put desired numbers for knots into a list. Be sure that the length of the list

is the same as the specified number of knots. Default is None, so that the function will automatically determine knot locations without user specification

- **term** (*integer, float, optional*) – High order term for the spline terms. To calculate a quadratic spline change to 2, cubic spline change to 3, etc. Default is 1, i.e. a linear spline
- **restricted** (*bool, optional*) – Whether to return a restricted spline. Note that the restricted spline returns one less column than the number of knots. An unrestricted spline returns the same number of columns as the number of knots. Default is False, providing an unrestricted spline

Returns Returns a pandas dataframe containing the spline variables (labeled 0 to n_knots)

Return type pd.DataFrame

Notes

Example of output

```
       rspline0     rspline1   rspline2
0   9839.409066  1234.154601   2.785600
1    446.391437     0.000000   0.000000
2   7107.550298   409.780251   0.000000
3   4465.272901     7.614501   0.000000
4  10972.041543  1655.208555  52.167821
..          ...          ...        ...
```

Examples

Calculate unrestricted linear spline with 3 automatic knots

```
>>> from zepid import spline, load_sample_data
>>> df = load_sample_data(False)
>>> spline(df, var='cd40', n_knots=3)
```

Calculate unrestricted quadratic spline with 3 automatic knots

```
>>> spline(df, var='cd40', n_knots=3, term=2)
```

Calculate restricted linear spline with 3 automatic knots

```
>>> spline(df, var='cd40', n_knot=3, restricted=True)
```

Calculate unrestricted linear spline with 3 specified knots

```
>>> spline(df, var='cd40', n_knots=3, knots=[200, 250, 750])
```

Calculate restricted cubic spline with 5 automatic knots

```
>>> spline(df, var='cd40', n_knots=5, term=3, restricted=True)
```

zepid.base.create_spline_transform

zepid.base.**create_spline_transform**(*array*, *n_knots=3*, *knots=None*, *term=1*, *restricted=False*)

Creates spline dummy variables based on either user specified knot locations or automatically determines knot locations based on percentiles. Options are available to set the number of knots, location of knots (value), term (linear, quadratic, etc.), and restricted/unrestricted.

Parameters

- **array** –
- **n_knots** (*integer, optional*) – Number of knots requested. Options for knots include any positive integer if the location of knots are specified. If knot locations are not specified, n_knots must be an integer between 1 to 7. Default is 3 knots
- **knots** (*list, optional*) – Location of specified knots in a list. To specify the location of knots, put desired numbers for knots into a list. Be sure that the length of the list is the same as the specified number of knots. Default is None, so that the function will automatically determine knot locations without user specification
- **term** (*integer, float, optional*) – High order term for the spline terms. To calculate a quadratic spline change to 2, cubic spline change to 3, etc. Default is 1, i.e. a linear spline
- **restricted** (*bool, optional*) – Whether to return a restricted spline. Note that the restricted spline returns one less column than the number of knots. An unrestricted spline returns the same number of columns as the number of knots. Default is False, providing an unrestricted spline

Returns

- **function** (*a lambda function that accepts a numpy array or*)
- **list** (*a list of the knots*)

zepid.base.table1_generator

zepid.base.**table1_generator**(*df*, *cols*, *variable_type*, *continuous_measure='median'*, *strat_by=None*, *decimal=3*)

Code to automatically generate a descriptive table of your study population (often referred to as a Table 1). Personally, I hate copying SAS/R/Python output from the interpreter to an Excel or other spreadsheet software. This code will generate a pandas dataframe object. This object will be a formatted table which can be exported as a CSV, opened in Excel, then final formatting changes/renaming can be done. Variables with np.nan values are counted as missing

Categorical variables will be divided into the unique numbers and have a percent calculated. Additionally, missing data will be counted (but is not included in the percent). Additionally, a single categorical variable can be used to present the results

Continuous variables either have median/IQR or mean/SE calculated depending on what is requested. Missing are counted as a separate category

Parameters

- **df** (*DataFrame*) – Pandas dataframe object containing all variables of interest
- **cols** (*list*) – List of columns of variable names to include in the table. Ex) ['X',var1','var2']

- **variable_type** (*list*) – List of strings indicating the variable types. For example, ['category','continuous','continuous']. Variable types accepted are * 'category' variable with categories only * 'continuous' continuous variable
- **continuous_measure** (*string, optional*) – Whether to use the medians or the means. Default is median. Options are * 'median' returns medians and IQR for continuous variables * 'mean' returns means and SE for continuous variables
- **strat_by** (*string, optional*) – Categorical variable to stratify by. Default is None (no stratification)
- **decimal** (*integer, optional*) – Decimal places to display in the table. Default is 3

Returns Returns a pandas dataframe object containing a formatted Table 1. It is not recommended that this table is used in any part of later analysis, since is id difficult to parse through the table. This function is only meant to reduce the amount of copying from output needed.

Return type pd.DataFrame

1.8.2 Calculations

Below is documentation for each of the implemented calculation functionalities for summary data.

Measures

risk_ci(events, total[, alpha, confint])	Calculate two-sided risk confidence intervals
incidence_rate_ci(events, time[, alpha])	Calculate two-sided incidence rate confidence intervals.
risk_ratio(a, b, c, d[, alpha])	Calculates the risk ratio and confidence intervals from count data.
risk_difference(a, b, c, d[, alpha])	Calculates the risk difference and confidence intervals from count data.
number_needed_to_treat(a, b, c, d[, alpha])	Calculates the number needed to treat and confidence intervals from count data.
odds_ratio(a, b, c, d[, alpha])	Calculates the odds ratio and confidence interval from count data
incidence_rate_ratio(a, c, t1, t2[, alpha])	Calculates the incidence rate ratio and confidence intervals from count data
incidence_rate_difference(a, c, t1, t2[, alpha])	Calculates the incidence rate difference and confidence intervals from count data
attributable_community_risk(a, b, c, d)	Calculates the estimated attributable community risk (ACR) from count data.
population_attributable_fraction(a, b, c, d)	Calculates the population attributable fraction (PAF) from count data

zepid.calc.utils.risk_ci

zepid.calc.utils.**risk_ci**(*events*, *total*, *alpha=0.05*, *confint='wald'*)

Calculate two-sided risk confidence intervals

Risk is calculated from

$$R = \frac{a}{a+b}$$

Wald standard error is

$$SE_{Wald} = \left(\frac{1}{a} - \frac{1}{b}\right)^{\frac{1}{2}}$$

Hypergeometric standard error is

$$SE_{HypGeo} = \left(\frac{ab}{(a+b)^2(a+b-1)}\right)^{\frac{1}{2}}$$

Parameters

- **events** (*integer, float*) – Number of events/outcomes that occurred
- **total** (*integer, float*) – Total number of subjects that could have experienced the event
- **alpha** (*float, optional*) – Alpha level. Default is 0.05
- **confint** (*string, optional*) – Type of confidence interval to generate. Current options include Wald or Hypergeometric confidence intervals

Returns Tuple containing risk, lower CL, upper CL, SE

Return type tuple

Note: Confidence intervals rely on the central limit theorem, so there must be at least 5 events and 5 nonevents

---

Examples

Estimate the risk, standard error, and confidence intervals

```
>>> from zepid.calc import risk_ci
>>> r = risk_ci(45, 100)
```

Extracting the estimated risk

```
>>> r.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> r.lower_bound
>>> r.upper_bound
```

Extracting the standard error

```
>>> r.standard_error
```

zepid.calc.utils.incidence_rate_ci

zepid.calc.utils.**incidence_rate_ci**(*events*, *time*, *alpha=0.05*)

Calculate two-sided incidence rate confidence intervals. Only Wald-type confidence intervals are currently implemented.

Incidence rate is calculated from

$$I = \frac{a}{t}$$

Incidence rate standard error is

$$SE = \left(\frac{a}{t^2}\right)^{\frac{1}{2}})$$

Parameters

- **events** (*integer, float*) – Number of events/outcomes that occurred
- **time** (*integer, float*) – Total person-time contributed in this group
- **alpha** (*float, optional*) – Alpha level. Default is 0.05

Returns Tuple containing incidence rate, lower CL, upper CL, SE

Return type tuple

Examples

Estimate the incidence rate, standard error, and confidence intervals

```
>>> from zepid.calc import incidence_rate_ci
>>> i = incidence_rate_ci(56, 503)
```

Extracting the estimated incidence rate

```
>>> i.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> i.lower_bound
>>> i.upper_bound
```

Extracting the standard error

```
>>> i.standard_error
```

zepid.calc.utils.risk_ratio

zepid.calc.utils.**risk_ratio**(*a*, *b*, *c*, *d*, *alpha=0.05*)

Calculates the risk ratio and confidence intervals from count data.

Risk ratio is calculated from

$$RR = \frac{a}{a+b} / \frac{c}{c+d}$$

Risk ratio standard error is

$$SE = \left(\frac{1}{a} - \frac{1}{a+b} + \frac{1}{c} - \frac{1}{c+d}\right)^{\frac{1}{2}}$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **b** (*integer, float*) – Count of unexposed individuals with outcome
- **c** (*integer, float*) – Count of exposed individuals without outcome
- **d** (*integer, float*) – Count of unexposed individuals without outcome

- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Returns Tuple of risk ratio, lower CL, upper CL, SE

Return type tuple

Examples

Estimate the risk ratio, standard error, and confidence intervals

```
>>> from zepid.calc import risk_ratio
>>> rr = risk_ratio(45, 55, 21, 79)
```

Extracting the estimated risk ratio

```
>>> rr.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> rr.lower_bound
>>> rr.upper_bound
```

Extracting the standard error

```
>>> rr.standard_error
```

zepid.calc.utils.risk_difference

zepid.calc.utils.**risk_difference**(*a*, *b*, *c*, *d*, *alpha=0.05*)

Calculates the risk difference and confidence intervals from count data.

Risk difference is calculated as

$$RD = \frac{a}{a+b} - \frac{c}{c+d}$$

Risk difference standard error is calculated as

$$SE = \left(\frac{R_1 \times (1-R_1)}{a+b} + \frac{R_0 \times (1-R_0)}{c+d} \right)^{\frac{1}{2}}$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **b** (*integer, float*) – Count of unexposed individuals with outcome
- **c** (*integer, float*) – Count of exposed individuals without outcome
- **d** (*integer, float*) – Count of unexposed individuals without outcome
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Returns Tuple of risk difference, lower CL, upper CL, SE

Return type tuple

Examples

Estimate the risk difference, standard error, and confidence intervals

```
>>> from zepid.calc import risk_difference
>>> rd = risk_difference(45, 55, 21, 79)
```

Extracting the estimated risk difference

```
>>> rd.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> rd.lower_bound
>>> rd.upper_bound
```

Extracting the standard error

```
>>> rd.standard_error
```

zepid.calc.utils.number_needed_to_treat

zepid.calc.utils.**number_needed_to_treat**(*a*, *b*, *c*, *d*, *alpha=0.05*)

Calculates the number needed to treat and confidence intervals from count data.

Number needed to treat is calculated as

$$NNT = \frac{1}{RD} = \frac{1}{\frac{a}{a+b} - \frac{c}{c+d}}$$

Confidence intervals are calculated by taking the inverse of the lower and upper confidence limits of the risk difference. The formula for the risk difference standard error is

$$SE = \left(\frac{R_1 \times (1 - R_1)}{a + b} + \frac{R_0 \times (1 - R_0)}{c + d} \right)^{\frac{1}{2}}$$

Note: If the risk difference confidence limits cover the null (RD=0), that means the interpretation will switch from NNT to NNH (number needed to harm). See Altman 1998 for further details on interpretation is this scenario

---

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **b** (*integer, float*) – Count of unexposed individuals with outcome
- **c** (*integer, float*) – Count of exposed individuals without outcome
- **d** (*integer, float*) – Count of unexposed individuals without outcome
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Returns Tuple of NNT, lower CL, upper CL, SE

Return type tuple

Examples

Estimate the number needed to treat, standard error, and confidence intervals

```
>>> from zepid.calc import number_needed_to_treat
>>> nnt = number_needed_to_treat(45, 55, 21, 79)
```

Extracting the estimated number needed to treat

```
>>> nnt.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> nnt.lower_bound
>>> nnt.upper_bound
```

Extracting the standard error

```
>>> nnt.standard_error
```

zepid.calc.utils.odds_ratio

zepid.calc.utils.**odds_ratio**(*a*, *b*, *c*, *d*, *alpha=0.05*)

Calculates the odds ratio and confidence interval from count data

Odds ratio is calculated from

$$OR = \frac{a}{b} / \frac{c}{d}$$

Odds ratio standard error is

$$SE = \left(\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d}\right)^{\frac{1}{2}}$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **b** (*integer, float*) – Count of unexposed individuals with outcome
- **c** (*integer, float*) – Count of exposed individuals without outcome
- **d** (*integer, float*) – Count of unexposed individuals without outcome
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Returns Tuple of OR, lower CL, upper CL, SE

Return type tuple

Examples

Estimate the odds ratio, standard error, and confidence intervals

```
>>> from zepid.calc import odds_ratio
>>> odr = odds_ratio(45, 55, 21, 79)
```

Extracting the estimated odds ratio

```
>>> odr.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> odr.lower_bound
>>> odr.upper_bound
```

Extracting the standard error

```
>>> odr.standard_error
```

zepid.calc.utils.incidence_rate_ratio

zepid.calc.utils.**incidence_rate_ratio**(*a*, *c*, *t1*, *t2*, *alpha=0.05*)

Calculates the incidence rate ratio and confidence intervals from count data

Incidence rate ratio is calculated from

$$IR = \frac{a}{t_1} / \frac{c}{t_2}$$

Incidence rate ratio standard error is

$$SE = \left(\frac{1}{a} + \frac{1}{c}\right)^{\frac{1}{2}}$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **c** (*integer, float*) – Count of unexposed individuals with outcome
- **t1** (*integer, float*) – Person-time contributed by those who were exposed
- **t2** (*integer, float*) – Person-time contributed by those who were unexposed
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Returns Tuple of IR, lower CL, upper CL, SE

Return type tuple

Examples

Estimate the incidence rate ratio, standard error, and confidence intervals

```
>>> from zepid.calc import incidence_rate_ratio
>>> ir = incidence_rate_ratio(45, 21, 109, 158)
```

Extracting the estimated incidence rate ratio

```
>>> ir.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> ir.lower_bound
>>> ir.upper_bound
```

Extracting the standard error

```
>>> ir.standard_error
```

zepid.calc.utils.incidence_rate_difference

zepid.calc.utils.**incidence_rate_difference**(*a*, *c*, *t1*, *t2*, *alpha=0.05*)

Calculates the incidence rate difference and confidence intervals from count data

Incidence rate difference is calculated from

$$ID = \frac{a}{t_1} - \frac{c}{t_2}$$

Incidence rate difference standard error is

$$SE = \left(\frac{a}{t_1^2} + \frac{c}{t_2^2}\right)^{\frac{1}{2}}$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **c** (*integer, float*) – Count of unexposed individuals with outcome
- **t1** (*integer, float*) – Person-time contributed by those who were exposed
- **t2** (*integer, float*) – Person-time contributed by those who were unexposed
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval

Returns Tuple of ID, lower CL, upper CL, SE)

Return type tuple

Examples

Estimate the incidence rate ratio, standard error, and confidence intervals

```
>>> from zepid.calc import incidence_rate_difference
>>> ird = incidence_rate_difference(45, 21, 109, 158)
```

Extracting the estimated incidence rate ratio

```
>>> ird.point_estimate
```

Extracting the lower and upper confidence intervals, respectively

```
>>> ird.lower_bound
>>> ird.upper_bound
```

Extracting the standard error

```
>>> ird.standard_error
```

zepid.calc.utils.attributable_community_risk

zepid.calc.utils.**attributable_community_risk**(*a*, *b*, *c*, *d*)

Calculates the estimated attributable community risk (ACR) from count data. ACR is also known as Population Attributable Risk. Since this is commonly confused with the population attributable fraction, the name ACR is used to clarify differences in the formulas

Attributable community risk is calculated as

$$ACR = \frac{a+c}{a+b+c+d} - \frac{c}{c+d} = R - R_0$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **b** (*integer, float*) – Count of unexposed individuals with outcome
- **c** (*integer, float*) – Count of exposed individuals without outcome
- **d** (*integer, float*) – Count of unexposed individuals without outcome

Returns Attributable community risk

Return type float

Examples

Return the attributable community risk

```
>>> from zepid.calc import attributable_community_risk
>>> attributable_community_risk(45, 55, 21, 79)
```

zepid.calc.utils.population_attributable_fraction

zepid.calc.utils.**population_attributable_fraction**(*a*, *b*, *c*, *d*)

Calculates the population attributable fraction (PAF) from count data

Population attributable fraction is calculated as

$$PAF = \left(\frac{a+c}{a+b+c+d} - \frac{c}{c+d}\right) / \frac{a+c}{a+b+c+d} = (R - R_0)/R$$

Parameters

- **a** (*integer, float*) – Count of exposed individuals with outcome
- **b** (*integer, float*) – Count of unexposed individuals with outcome
- **c** (*integer, float*) – Count of exposed individuals without outcome
- **d** (*integer, float*) – Count of unexposed individuals without outcome

Returns Population attributable fraction

Return type float

Examples

Return the population attributable fraction

```
>>> from zepid.calc import population_attributable_fraction
>>> population_attributable_fraction(45, 55, 21, 79)
```

Diagnostics

sensitivity(detected, cases[, alpha, confint])	Calculate the sensitivity from number of detected cases and the number of total true cases.
specificity(detected, noncases[, alpha, confint])	Calculate the specificity from number of false detections and the number of total non-cases.
ppv_converter(sensitivity, specificity, ...)	Generates the positive predictive value from designated sensitivity, specificity, and prevalence.
npv_converter(sensitivity, specificity, ...)	Generates the negative predictive value from designated sensitivity, specificity, and prevalence.
screening_cost_analyzer(cost_miss_case, ...)	Compares the cost of sensivitiy/specificity of screening criteria to treating the entire population as test-negative and test-positive.

zepid.calc.utils.sensitivity

zepid.calc.utils.**sensitivity**(*detected*, *cases*, *alpha=0.05*, *confint='wald'*)

Calculate the sensitivity from number of detected cases and the number of total true cases.

Parameters

- **detected** (*integer, float*) – Number of true cases detected via testing criteria
- **cases** (*integer, float*) – Total number of true/actual cases
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval
- **confint** (*string, optional*) – Type of confidence interval to generate. Current options include Wald or Hypergeometric confidence intervals

Returns Tuple of sensitivity, lower CL, upper CL, SE

Return type tuple

Examples

Calculating sensitivity

```
>>> from zepid.calc import sensitivity
>>> se = sensitivity(90, 100)
```

Extract sensitivity

```
>>> se[0]
```

Extract confidence intervals for sensitivity

```
>>> se[1:3]
```

Extract standard error

```
>>> se[3]
```

zepid.calc.utils.specificity

zepid.calc.utils.**specificity**(*detected*, *noncases*, *alpha=0.05*, *confint='wald'*)

Calculate the specificity from number of false detections and the number of total non-cases.

Parameters

- **detected** (*integer, float*) – Number of false cases detected via testing criteria
- **noncases** (*integer, float*) – Total number of non-cases
- **alpha** (*float, optional*) – Alpha value to calculate two-sided Wald confidence intervals. Default is 95% confidence interval
- **confint** (*string, optional*) – Type of confidence interval to generate. Current options include Wald or Hypergeometric confidence intervals

Returns Tuple of specificity, lower CL, upper CL, SE

Return type tuple

Examples

Calculating specificity

```
>>> from zepid.calc import specificity
>>> sp = specificity(88, 100)
```

Extract specificity

```
>>> sp[0]
```

Extract confidence intervals for specificity

```
>>> sp[1:3]
```

Extract standard error

```
>>> sp[3]
```

zepid.calc.utils.ppv_converter

zepid.calc.utils.**ppv_converter**(*sensitivity*, *specificity*, *prevalence*)

Generates the positive predictive value from designated sensitivity, specificity, and prevalence.

Positive predictive value is calculated using

$$PPV = \frac{Se \times P}{Se \times P + (1 - Sp)(1 - P)}$$

Parameters

- **sensitivity** (*float*) – Sensitivity of the testing criteria
- **specificity** (*float*) – Specificity of the testing criteria
- **prevalence** (*float*) – Prevalence of the outcome in the population

Returns Positive predictive value

Return type float

Examples

Calculate the positivity predictive value

```
>>> from zepid.calc import ppv_converter
>>> ppv_converter(0.9, 0.88, 0.15)
```

zepid.calc.utils.npv_converter

zepid.calc.utils.**npv_converter**(*sensitivity*, *specificity*, *prevalence*)

Generates the negative predictive value from designated sensitivity, specificity, and prevalence.

$$NPV = \frac{Sp \times (1 - P)}{(1 - Se) \times P + Sp \times (1 - P)}$$

Parameters

- **sensitivity** (*float*) – Sensitivity of the testing criteria
- **specificity** (*float*) – Specificity of the testing criteria
- **prevalence** (*float*) – Prevalence of the outcome in the population

Returns Negative predictive value

Return type float

Examples

Calculate the positivity predictive value

```
>>> from zepid.calc import npv_converter
>>> npv_converter(0.9, 0.88, 0.15)
```

zepid.calc.utils.screening_cost_analyzer

zepid.calc.utils.**screening_cost_analyzer**(*cost_miss_case*, *cost_false_pos*, *prevalence*, *sensitivity*, *specificity*, *population=10000*, *decimal=3*)

Compares the cost of sensivitiy/specificity of screening criteria to treating the entire population as test-negative and test-positive. The lowest per capita cost is considered the ideal choice. Note that this function only provides relative costs

Parameters

- **cost_miss_case** (*float*) – The relative cost of missing a case, compared to false positives. In general, set this to 1 then change the value under 'cost_false_pos' to reflect the relative cost
- **cost_false_pos** (*float*) – The relative cost of a false positive case, compared to a missed case
- **prevalence** (*float*) – The prevalence of the disease in the population. Must be a float
- **sensitivity** (*float*) – The sensitivity level of the screening test. Must be a float
- **specificity** (*float*) – The specificity level of the screening test. Must be a float
- **population** (*float*) – The population size to set. Choose a larger value since this is only necessary for total calculations. Default is 10,000
- **decimal** (*integer*) – Amount of decimal points to display. Default value is 3

Returns Prints results to console

Return type None

Note: When calculating costs, be sure to consult experts in health policy or related fields. Costs should encompass more than just monetary costs, like relative costs (regret, disappointment, stigma, disutility, etc.). Careful consideration of relative costs between false positive and false negatives needs to be considered.

Examples

Calculate the (relative) cost for the proposed screening strategy

```
>>> from zepid.calc import screening_cost_analyzer
>>> screening_cost_analyzer(cost_miss_case=1, cost_false_pos=3, prevalence=0.15,␣
↪sensitivity=0.9, specificity=0.88)
```

Others

probability_to_odds(prob)	Converts given probability (proportion) to odds
odds_to_probability(odds)	Converts given odds to probability (proportion)
counternull_pvalue(estimate, lcl, ucl[, ...])	Calculates the counternull p-value.
semibayes(prior_mean, prior_lcl, prior_ucl, ...)	A simple Bayesian Analysis.
rubins_rules(point_estimates, std_error)	Function to merge multiple imputed data sets into a single summary estimate and variance.
s_value(pvalue)	Function to calculate the S-value.

zepid.calc.utils.probability_to_odds

zepid.calc.utils.**probability_to_odds**(*prob*)

Converts given probability (proportion) to odds

Probability is converted to odds using

$$O = \frac{\Pr}{1 - \Pr}$$

Parameters **prob** (*float, NumPy array*) – Probability or array of probabilities to transform

into odds

Returns Float or array of odds

Return type odds

Examples

Convert a single probability to an odds

```
>>> from zepid.calc import probability_to_odds
>>> probability_to_odds(0.3)
```

Convert an array of probabilities to odds

```
>>> import numpy as np
>>> probs = np.array([0.3, 0.1, 0.2, 0.5, 0.01])
>>> probability_to_odds(probs)
```

zepid.calc.utils.odds_to_probability

zepid.calc.utils.**odds_to_probability**(*odds*)

Converts given odds to probability (proportion)

Probability is converted to odds using

$$\Pr = \frac{O}{1+O}$$

Parameters **odds** (*float, NumPy array*) – Odds or array of odds to transform into probabilities

Returns Float or array of probabilities

Return type prob

Examples

Convert a single odds to a probability

```
>>> from zepid.calc import odds_to_probability
>>> odds_to_probability(0.45)
```

Convert an array of odds to probabilities

```
>>> import numpy as np
>>> odds = np.array([0.3, 0.5, 1, 3.1, 1.1])
>>> odds_to_probability(odds)
```

zepid.calc.utils.counternull_pvalue

zepid.calc.utils.**counternull_pvalue**(*estimate*, *lcl*, *ucl*, *sided='two'*, *alpha=0.05*, *decimal=3*)

Calculates the counternull p-value. It is useful to prevent over-interpretation of results

Parameters

- **estimate** (*float*) – Point estimate for result
- **lcl** (*float*) – Lower confidence limit
- **ucl** (*float*) – Upper confidence limit
- **sided** (*string, optional*) – Whether to compute the upper one-sided, lower one-sided, or two-sided counternull p-value. Default is the two-sided
  - 'upper' Upper one-sided p-value
  - 'lower' Lower one-sided p-value
  - 'two' Two-sided p-value
- **alpha** (*float, optional*) – Alpha level for p-value. Default is 0.05. Verify that this is the same alpha used to generate confidence intervals
- **decimal** (*integer, optional*) – Number of decimal places to display. Default is three

Returns Function does not return an object. It prints results to the console

Return type None

Notes

Make sure that the confidence interval points put into the equation match the alpha level calculation

Examples

Calculate the counternull p-value for a single estimate and confidence interval

```
>>> from zepid.calc import counternull_pvalue
>>> counternull_pvalue(-0.1, -0.3, 0.1)
```

References

Rosenthal R, Rubin DB. (1994). The counternull value of an effect size: A new statistic. Psychological Science, 5(6), 329-334.

zepid.calc.utils.semibayes

zepid.calc.utils.**semibayes**(*prior_mean*, *prior_lcl*, *prior_ucl*, *mean*, *lcl*, *ucl*, *ln_transform=False*, *alpha=0.05*, *decimal=3*, *print_results=True*)

A simple Bayesian Analysis. Note that this analysis assumes a normal distribution for the continuous measure. See chapter 18 of Modern Epidemiology 3rd Edition (specifically pages 334, 340 for this procedure)

The posterior estimate is calculated as

$$E_{posterior} = \frac{\left(E_{prior} \times \frac{1}{Var_{prior}}\right) + (E \times \frac{1}{Var})}{E_{prior} \times \frac{1}{Var_{prior}}}$$

and the posterior variance is

$$Var_{posterior} = \frac{1}{\frac{1}{Var_{prior}} + \frac{1}{Var}}$$

Parameters

- **prior_mean** (*float*) – Prior designated point estimate
- **prior_lcl** (*float*) – Prior designated lower confidence limit
- **prior_ucl** (*float*) – Prior designated upper confidence limit
- **mean** (*float*) – Point estimate result obtained from analysis
- **lcl** (*float*) – Lower confidence limit estimate obtained from analysis
- **ucl** (*float*) – Upper confidence limit estimate obtained from analysis
- **ln_transform** (*bool, optional*) – Whether to natural log transform results before conducting analysis. Should be used for RR, OR, or or other Ratio measure. Default is False (use for RD and other absolute measures)
- **alpha** (*float, optional*) – Alpha level for confidence intervals. Default is 0.05
- **decimal** (*float, optional*) – Number of decimal places to display. Default is three
- **print_results** (*bool, optional*) – Whether to print the results of the semi-Bayesian calculations. Default is True

Returns Tuple of posterior mean, posterior lower CL, posterior upper CL

Return type tuple

Note: Make sure that the alpha used to generate the confidence intervals matches the alpha used in this calculation. Additionally, this calculation can only handle normally distributed priors and observed

---

Examples

Posterior Risk Difference

```
>>> from zepid.calc import semibayes
>>> semibayes(prior_mean=-0.15, prior_lcl=-0.5, prior_ucl=0.2, mean=-0.1, lcl=-0.
↪3, ucl=0.1, print_results=False)
```

Posterior Risk Ratio

```
>>> semibayes(prior_mean=0.9, prior_lcl=0.75, prior_ucl=1.2, mean=0.85, lcl=0.77,␣
↪ucl=0.91, ln_transform=True)
```

References

Rothman KJ, Greenland S, Lash TL. (2008). Modern epidemiology (Vol. 3). Philadelphia: Wolters Kluwer Health/Lippincott Williams & Wilkins.

zepid.calc.utils.rubins_rules

zepid.calc.utils.**rubins_rules**(*point_estimates*, *std_error*)

Function to merge multiple imputed data sets into a single summary estimate and variance. Results are based

on Rubin's Rules for merging estimates. The summary point estimate is calculated via

$$\bar{\beta} = m^{-1} \sum_{k=1}^{m} \hat{\beta}_k$$

where m is the number of imputed data sets. The variance is calculated via

$$Var(\hat{\beta}) = m_{-1} \sum_{k=1}^{m} Var(\hat{\beta}) + (1 + m^{-1})(m-1)^{-1} \sum_{k=1}^{m} (\hat{\beta}_k - \bar{\beta})^2$$

The variance is constructed from the within-sample variance and the between sample variance

Notes

If your point estimates correspond to ratios, be sure to provide the natural-log transformed point estimates and the variance of the natural-log estimate

Parameters

- **point_estimates** (*list*) – Container object of the point estimates
- **std_error** (*list*) – Container object of the estimate standard errors

Returns Tuple of summary beta, summary standard error

Return type tuple

Examples

```
>>> from zepid.calc import rubins_rules
>>> rr_est = []
>>> rr_std = []
```

Calculating summary estimate

```
>>> b = rubins_rules(rr_est, rr_std)
```

Printing the summary risk ratio

```
>>> print("RR = ", np.exp(b[0]))
>>> print("95% LCL:", np.exp(b[0] - 1.96*b[1]))
>>> print("95% UCL:", np.exp(b[0] + 1.96*b[1]))
```

References

Rubin DB. (2004). Multiple imputation for nonresponse in surveys (Vol. 81). John Wiley & Sons.

zepid.calc.utils.s_value

zepid.calc.utils.**s_value**(*pvalue*)

Function to calculate the S-value. The 'S' stands for Shannon information or surprisal values. The name comes from Claude Shannon for this work to information theory. S-values are calculated from p-values using the following transformation

$$s = -\log(p)$$

The S-value transformation allows a more intuitive explanation of what p-values tell us about the null hypothesis and alternative hypothesis compatibility. The S-value tells us how many 'bits' of information exist against the null hypothesis. For an example, a S-value of 5.1 is no more surprising than seeing heads for 5 fair coin tosses. The S-value should be rounded down in the interpretation

---

Note: S-values do NOT have a significant cut-point. Rather this transformation is to help build intuition what information a p-values is providing and the corresponding 'surprisal' of a result

---

Parameters **pvalue** (*float, container*) – P-value (or array of p-values) to convert into a S-value(s)

Returns NumPy array of calculated S-values

Return type array

Examples

```
>>> from zepid.calc import s_value
>>> s_value(pvalue=0.05)
```

References

Greenland S. (2019). Valid P-values behave exactly as they should: Some misleading criticisms of P-values and their resolution with S-values. The American Statistician, 73(sup1), 106-114.

Amrhein V, Trafimow D, & Greenland S. (2018). Inferential Statistics as Descriptive Statistics: There is No Replication Crisis if We Don't Expect Replication. The American Statistician.

1.8.3 Graphics

Below is documentation for each of the implemented graphic generators.

Data Diagnostics

functional_form_plot(df, outcome, var[, ...])	Creates a functional form plot to aid in functional form assessment for continuous/discrete variables.
spaghetti_plot(df, idvar, variable, time)	Create a spaghetti plot by an ID variable.
roc(df, true, threshold[, youden_index])	Generate a Receiver Operator Curve from true values and predicted probabilities.

Displaying Results

EffectMeasurePlot(label, effect_measure, ...)	Used to generate effect measure (AKA forest) plots.
pvalue_plot(point, sd[, color, fill, null, ...])	Creates a plot of the p-value distribution based on a point estimate and standard deviation.
dynamic_risk_plot(risk_exposed, risk_unexposed)	Creates a plot of how the risk difference or risk ratio changes over time with survival data.

Continued on next page

Table 17 – continued from previous page

labbe_plot([r1, r0, scale, additive_tuner, ...])	L'Abbe plots are useful for summarizing measure modification on the difference or ratio scale.
zipper_plot(truth, lcl, ucl[, colors])	Zipper plots are a way to present simulation data, particularly confidence intervals and their width.

class zepid.graphics.graphics.**EffectMeasurePlot**(*label*, *effect_measure*, *lcl*, *ucl*)

Used to generate effect measure (AKA forest) plots. Estimates and confidence intervals are plotted in a diagram on the left and a table of the corresponding estimates is provided in the same plot. See the Graphics page on ReadTheDocs examples of the plots

Parameters

- **label** (*list*) – List of labels to use for y-axis
- **effect_measure** (*list*) – List of numbers for point estimates to plot. If point estimate has trailing zeroes, input as a character object rather than a float
- **lcl** (*list*) – List of numbers for upper confidence limits to plot. If point estimate has trailing zeroes, input as a character object rather than a float
- **ucl** (*list*) – List of numbers for upper confidence limits to plot. If point estimate has trailing zeroes, input as a character object rather than a float

Examples

Setting up the data to plot

```
>>> from matplotlib.pyplot as plt
>>> from zepid.graphics import EffectMeasurePlot
>>> lab = ['One','Two']
>>> emm = [1.01,1.31]
>>> lcl = ['0.90',1.01]  # String allows for trailing zeroes in the table
>>> ucl = [1.11,1.53]
```

Setting up the plot, measure labels, and point colors

```
>>> x = EffectMeasurePlot(lab, emm, lcl, ucl)
>>> x.labels(effectmeasure='RR')  # Changing label of measure
>>> x.colors(pointcolor='r')  # Changing color of the points
```

Generating matplotlib axes object of forest plot

```
>>> x.plot(t_adjuster=0.13)
>>> plt.show()
```

colors(***kwargs*)

Function to change colors and shapes.

Parameters

- **errorbarcolor** (*string, optional*) – Changes the error bar colors
- **linecolor** (*string, optional*) – Changes the color of the reference line
- **pointcolor** (*string, optional*) – Changes the color of the points
- **pointshape** (*string, optional*) – Changes the shape of points

labels(***kwargs*)

Function to change the labels of the outputted table. Additionally, the scale and reference value can be changed.

Parameters

- **effectmeasure** (*string, optional*) – Changes the effect measure label
- **conf_int** (*string, optional*) – Changes the confidence interval label
- **scale** (*string, optional*) – Changes the scale to either log or linear
- **center** (*float, integer, optional*) – Changes the reference line for the center

plot(*figsize=(3, 3)*, *t_adjuster=0.01*, *decimal=3*, *size=3*, *max_value=None*, *min_value=None*, *text_size=12*)

Generates the matplotlib effect measure plot with the default or specified attributes. The following variables can be used to further fine-tune the effect measure plot

Parameters

- **figsize** (*tuple, optional*) – Adjust the size of the figure. Syntax is same as matplotlib *figsize*
- **t_adjuster** (*float, optional*) – Used to refine alignment of the table with the line graphs. When generate plots, trial and error for this value are usually necessary. I haven't come up with an algorithm to determine this yet...
- **decimal** (*integer, optional*) – Number of decimal places to display in the table
- **size** (*integer,*) – Option to adjust the size of the lines and points in the plot
- **max_value** (*float, optional*) – Maximum value of x-axis scale. Default is None, which automatically determines max value
- **min_value** (*float, optional*) – Minimum value of x-axis scale. Default is None, which automatically determines min value
- **text_size** (*int, float, optional*) – Text size for the table. Default is 12.

Returns

Return type matplotlib axes

zepid.graphics.graphics.**dynamic_risk_plot**(*risk_exposed*, *risk_unexposed*, *measure='RD'*, *loess=True*, *loess_value=0.25*, *point_color='darkblue'*, *line_color='b'*, *scale='linear'*)

Creates a plot of how the risk difference or risk ratio changes over time with survival data. See the references for an example of this plot. Input data should be pandas Series indexed by 'timeline' where 'timeline' is the time corresponding to the risk estimate

Parameters

- **risk_exposed** (*Series*) – Pandas Series with the probability of the outcome among the exposed group. Index by 'timeline' where 'timeline' is the time. If you directly output the `1 - survival_function_` from lifelines.KaplanMeierFitter(), this should create a valid input
- **risk_unexposed** (*Series*) – Pandas Series with the probability of the outcome among the exposed group. Index by 'timeline' where 'timeline' is the time
- **measure** (*str, optional*) – Whether to generate the risk difference (RD) or risk ratio (RR). Default is 'RD'

- **loess** (*bool, optional*) – Whether to generate LOESS curve fit to the calculated points. Default is True
- **loess_value** (*float, optional*) – Fraction of values to fit LOESS curve to. Default is 0.25
- **point_color** (*str, optional*) – Color of the points
- **line_color** (*str, optional*) – Color of the LOESS line generated and plotted
- **scale** (*str, optional*) – Change the y-axis scale. Options are 'linear' (default), 'log', 'log-transform'. 'log' and 'log-transform' is only a valid option for Risk Ratio plots

Returns

Return type matplotlib axes

Examples

See graphics documentation or causal documentation for a detailed example.

```
>>> import matplotlib.pyplot as plt
>>> from zepid.graphics import dynamic_risk_plot
```

```
>>> dynamic_risk_plot(a, b, loess=True)
>>> plt.show()
```

References

Cole SR, et al. (2014). Estimation of the standardized risk difference and ratio in a competing risks framework: application to injection drug use and progression to AIDS after initiation of antiretroviral therapy. AJE, 181(4), 238-245.

zepid.graphics.graphics.**functional_form_plot**(*df*, *outcome*, *var*, *f_form=None*, *outcome_type='binary'*, *discrete=False*, *link_dist=None*, *loess=True*, *loess_value=0.25*, *legend=True*, *model_results=True*, *points=False*)

Creates a functional form plot to aid in functional form assessment for continuous/discrete variables. Plots can be created for binary and continuous outcomes. Default options are set to create a functional form plot for a binary outcome. To convert to a continuous outcome, *outcome_type* needs to be changed, in addition to the *link_dist*

Parameters

- **df** (*DataFrame*) – Pandas dataframe that contains the variables of interest
- **outcome** (*string*) – Column name of the outcome variable of interest
- **var** (*string*) – Column name of the variable of interest for the functional form assessment
- **f_form** (*string, optional*) – Regression equation of the functional form to assess. Default is None, which will produce a linear functional form. Input the regression equation following the *patsy* syntax. For example, 'var + var_sq'
- **outcome_type** (*string, optional*) – Variable type of the outcome variable. Currently, only binary and continuous variables are supported. Default is 'binary'

- **link_dist** (*optional*) – Link and distribution for the GLM regression equation. Change this to any valid link and distributions supported by *statsmodels*. Default is *None*, which defaults to logistic regression
- **loess_value** (*float, optional*) – Fraction of observations to use to fit the LOESS curve. This may need to be changed iteratively to determine which percent works best for the data. Default is 0.25
- **legend** (*bool, optional*) – Turn the legend on or off. Default is True, displaying the legend in the graph
- **model_results** (*bool, optional*) – Whether to produce the model results. Default is True, which provides model results
- **loess** (*bool, optional*) – Whether to plot the LOESS curve along with the functional form. Default is True
- **points** (*bool, optional*) – Whether to plot the data points, where size is relative to the number of observations. Default is False
- **discrete** (*bool, optional*) – If your data is truly continuous, leave setting to auto bin the dat. Will automatically bin observations into categories for fitting a model with a disjoint indicator. If data is discrete, you can set this to *True* to use the actual values for the disjoint indicator. If you get a perfect *SeparationError* from *statsmodels*, it means you might have to reshift your categories.

Returns Returns a matplotlib graph with a LOESS line (dashed red-line), regression line (sold blue-line), and confidence interval (shaded blue)

Return type matplotlib axes

Examples

Setting up the environment

```
>>> from zepid import load_sample_data
>>> from zepid.graphics import functional_form_plot
>>> import matplotlib.pyplot as plt
>>> df = load_sample_data(timevary=False)
>>> df['cd4_sq'] = df['cd4']**2
```

Creating a functional form plot for a linear functional form

```
>>> functional_form_plot(df, outcome='dead', var='cd4')
>>> plt.show()
```

Functional form assessment for a quadractic functional form

```
>>> functional_form_plot(df, outcome='dead', var='cd4', f_form='cd4 + cd4_sq')
>>> plt.show()
```

Varying the LOESS value (increased LOESS value to smooth LOESS curve further)

```
>>> functional_form_plot(df, outcome='dead', var='cd4', loess_value=0.5)
>>> plt.show()
```

Removing the LOESS curve and the legend from the plot

```
>>> functional_form_plot(df, outcome='dead', var='cd4', loess=False, legend=False)
>>> plt.show()
```

Adding summary points to the plot. Points are grouped together and their size reflects their relative n

```
>>> functional_form_plot(df, outcome='dead', var='cd4', loess=False, legend=False,
↪ points=True)
>>> plt.show()
```

Functional form assessment for a discrete variable (age)

```
>>> functional_form_plot(df, outcome='dead', var='age0', discrete=True)
>>> plt.show()
```

zepid.graphics.graphics.**labbe_plot**(*r1=None*, *r0=None*, *scale='both'*, *additive_tuner=12*, *multiplicative_tuner=12*, *figsize=(7, 4)*, ***plot_kwargs*)

L'Abbe plots are useful for summarizing measure modification on the difference or ratio scale. Primarily invented for meta-analysis usage, these plots display risk differences (or ratios) by their individual risks by an exposure. I find them most useful for a visualization of why if there is an association and there is no modfication on one scale (additive or multiplicative), then there must be modification on the other scale.

Parameters

- **r1** (*float, list, optional*) – Single probability or a list of probabilities when exposure is 1. Default is None, which does not display points
- **r0** (*float, list, optional*) – Single probability or a list of probabilities when exposure is 0. Default is None, which does not display points
- **scale** (*str, optional*) – Which scale to plot. The default is 'both', which generates side-by-side plots of additive scale and multiplicative scale. Other options are; 'additive' to display the additive plot, and 'multiplicative' to display the multiplicative plot
- **additive_tuner** (*int, optional*) – Optional parameter to change the number of lines displayed in the additive L'Abbe plot. Higher integer produces more reference lines
- **multiplicative_tuner** (*int, optional*) – Optional parameter to change the number of lines displayed in the multiplicative L'Abbe plot. Higher integer produces more reference lines
- **figsize** (*set, optional*) – Optional parameter to change the L'Abbe plot size. Only changes the plot size when scale='both'
- ****plot_kwargs** (*optional*) – Optional keyword arguments for matplotlib. kwargs will pass matplotlib.pyploy.plot kwargs are accepted. See matplotlib 'plot()' function documentation for further details

Returns

Return type matplotlib axes

Examples

Setting up environment

```
>>> import matplotlib.pyplot as plt
>>> from zepid.graphics import labbe_plot
```

Creating a blank plot

```
>>> labbe_plot()
>>> plt.show()
```

Adding customized points to the plot

```
>>> labbe_plot(r1=[0.3, 0.5], r0=[0.2, 0.7], scale='additive', color='r', marker=
↪'D', markersize=10, linestyle='')
>>> plt.show()
```

Only returning the additive plot

```
>>> labbe_plot(r1=[0.3, 0.5], r0=[0.2, 0.7], scale='additive', markersize=10)
>>> plt.show()
```

Only returning the multiplicative plot

```
>>> labbe_plot(r1=[0.3, 0.5], r0=[0.2, 0.7], scale='multiplicative',␣
↪markersize=10)
>>> plt.show()
```

zepid.graphics.graphics.**pvalue_plot**(*point*, *sd*, *color='b'*, *fill=True*, *null=0*, *alpha=None*)

Creates a plot of the p-value distribution based on a point estimate and standard deviation. I find this plot to be useful to explain p-values and how much evidence weight you have in a specific value. I think it is useful to explain what exactly a p-value tells you. Note that this plot only works for measures on a linear scale (i.e. it will plot exp(log(RR)) incorrectly). It also helps to understand what exactly confidence intervals are telling you. These plots are based on Rothman's Epidemiology 2nd Edition pg 152-153 and explained more fully within.

Parameters

- **point** (*float*) – Point estimate. Must be on a linear scale (RD / log(RR))
- **sd** (*float*) – Standard error of the estimate. Must for linear scale (SE(RD) / SE(log(RR)))
- **color** (*str, optional*) – Change color of p-value plot
- **fill** (*bool, optional*) – Hhether to fill the curve under the p-value distribution. Setting to False prevents fill
- **null** (*float, integer, optional*) – The main value to compare to. The default is zero
- **alpha** (*float, optional*) – Whether to draw a line designating significance level area. Default is None, which does not draw this line. Generally, would be set to 0.05 to correspond to the widely used alpha of 0.05

Returns

Return type matplotlib axes

Examples

Setting up the environment

```
>>> from zepid.graphics import pvalue_plot
>>> import matplotlib.pyplot as plt
```

Basic P-value plot

```
>>> pvalue_plot(point=-0.1, sd=0.061, color='r')
>>> plt.show()
```

P-value plot with significance line drawn at 'alpha'

```
>>> pvalue_plot(point=-0.1, sd=0.061, color='r', alpha=0.025)
>>> plt.show()
```

P-value plot with different comparison value

```
>>> pvalue_plot(point=-0.1, sd=0.061, color='r', null=0.1)
>>> plt.show()
```

References

Rothman KJ. (2012). Epidemiology: an introduction. Oxford university press.

zepid.graphics.graphics.**roc**(*df*, *true*, *threshold*, *youden_index=True*)

Generate a Receiver Operator Curve from true values and predicted probabilities. Youden's Index can also be calculated. Youden's index is calculated as

$$P_{Yi} = max(Se_i + Sp_i - 1)$$

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing variables of interest
- **true** (*str*) – True designation of the outcome (1, 0)
- **threshold** (*str*) – Predicted probabilities for the outcome
- **youden_index** (*bool, optional*) – Whether to calculate Youden's index. Youden's index maximizes both sensitivity and specificity. The formula finds the maximum of (sensitivity + specificity - 1)

Returns

Return type matplotlib axes

Examples

Creating a dataframe with true disease status (*'d'*) and predicted probability of the outcome (*'p'*)

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from zepid.graphics import roc
>>> df = pd.DataFrame()
>>> df['d'] = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
>>> df['p'] = [0.1, 0.15, 0.1, 0.7, 0.5, 0.9, 0.95, 0.5, 0.4, 0.8, 0.99, 0.99, 0.
↪89, 0.95]
```

Creating ROC curve

```
>>> roc(df, true='d', threshold='p', youden_index=False)
>>> plt.show()
```

zepid.graphics.graphics.**spaghetti_plot**(*df*, *idvar*, *variable*, *time*)

Create a spaghetti plot by an ID variable. A spaghetti plot can be useful for visualizing trends or looking at longitudinal data patterns for individuals all at once.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing variables of interest
- **idvar** (*str*) – ID variable for observations. This should indicate the group or individual followed over the time variable
- **variable** (*str*) – Variable of interest to see how it varies over time
- **time** (*str*) – Time or other variable in which the variable variation occurs

Returns

Return type matplotlib axes

Examples

Setting up the environment

```
>>> from zepid import load_sample_data
>>> from zepid.graphics import spaghetti_plot
>>> df = load_sample_data(timevary=True)
```

Generating spaghetti plot for changing CD4 count

```
>>> spaghetti_plot(df, idvar='id', variable='cd4', time='enter')
>>> plt.show()
```

zepid.graphics.graphics.**zipper_plot**(*truth*, *lcl*, *ucl*, *colors=('blue', 'red')*)

Zipper plots are a way to present simulation data, particularly confidence intervals and their width. They are also useful for showing the confidence interval coverage of the true parameter.

Parameters

- **truth** (*float*) – The true value with which to compare the confidence interval coverage to
- **lcl** (*list, array, Series, container*) – Container of lower confidence limits
- **ucl** (*list, array, Series, container*) – Container of upper confidence limits
- **colors** (*set, list, container*) – List of colors for confidence intervals. The first color is used to designate confidence intervals that cover the true value, and the second indicates confidence intervals

Returns

Return type matplotlib axes

Examples

Setting up environment

```
>>> import matplotlib.pyplot as plt
>>> from zepid.graphics import zipper_plot
```

Adding customized points to the plot

```
>>> labbe_plot(r1=[0.3, 0.5], r0=[0.2, 0.7], scale='additive', color='r', marker=
↪'D', markersize=10, linestyle='')
>>> plt.show()
```

1.8.4 Causal

Documentation for each of the causal inference methods implemented in zEpid

Causal Diagrams

DirectedAcyclicGraph(exposure, outcome)

zepid.causal.causalgraph.dag.DirectedAcyclicGraph

class zepid.causal.causalgraph.dag.**DirectedAcyclicGraph**(*exposure*, *outcome*)

__init__(*exposure*, *outcome*)

Constructs a Directed Acyclic Graph (DAG) for determination of adjustment sets

Parameters

- **exposure** (*str*) – Exposure of interest in the causal diagram
- **outcome** (*str*) – Outcome of interest in the causal diagram
- **TODO add other implementations in the future... have as self.mediator, self.censor, self.missing** (*#*) –

Examples

Setting up environment

```
>>> from zepid.causal.causalgraph import DirectedAcyclicGraph
```

Creating directed acyclic graph

```
>>> dag = DirectedAcyclicGraph(exposure="X", outcome="Y")
>>> dag.add_arrow(source="X", endpoint="Y")
>>> dag.add_arrow(source="V", endpoint="Y")
>>> dag.add_arrows(pairs=(("W", "X"), ("W", "Y")))
```

Determining adjustment sets

```
>>> dag.calculate_adjustment_sets()
>>> dag.adjustment_sets
>>> dag.minimal_adjustment_sets
```

Plot diagram

```
>>> dag.draw_dag()
>>> plt.show()
```

Assess arrow misdirections that result in the chosen adjustment set being invalid

```
>>> dag.assess_misdirections(chosen_adjustment_set="W")
```

References

Shrier I, & Platt RW. (2008). Reducing bias through directed acyclic graphs. BMC medical research methodology, 8(1), 70.

Methods

__init__(exposure, outcome)	Constructs a Directed Acyclic Graph (DAG) for determination of adjustment sets
add_arrow(source, endpoint)	Add a single arrow to the current causal DAG
add_arrows(pairs)	Add a set of arrows to the current causal DAG
add_from_networkx(network)	
assess_misdirections(chosen_adjustment_set)	Arrow direction can potentially be misspecified.
calculate_adjustment_sets()	Determines all sufficient adjustment sets for the causal diagram using the algorithm described in Shrier & Platt "Reducing bias through directed acyclic graphs" BMC Medical Research Methodology 2008.
draw_dag([positions, invert, fig_size, ...])	Draws the current input causal DAG

Inverse Probability Weights

IPTW(df, treatment, outcome[, weights, ...])	Calculates inverse probability of treatment weights.
StochasticIPTW(df, treatment, outcome[, weights])	Calculates the IPTW estimate for stochastic treatment plans.

zepid.causal.ipw.IPTW.IPTW

class zepid.causal.ipw.IPTW.**IPTW**(*df*, *treatment*, *outcome*, *weights=None*, *standardize='population'*)

Calculates inverse probability of treatment weights. Both stabilized or unstabilized weights are implemented. By default, stabilized weights are stabilized by the prevalence of the treatment in the population. *IPTW* will also now fit the marginal structural model and estimate inverse probability of censoring weights if requested. Confidence intervals are calculated using robust standard errors.

The formula for stabilized IPTW is

$$\pi_i = \frac{\Pr(A=a)}{\Pr(A=a|L=l)}$$

For unstabilized IPTW

$$\pi_i = \frac{1}{\Pr(A=a|L=l)}$$

SMR unstabilized weights for weighting to exposed (A=1)

$$\begin{aligned}\pi_i &= 1 \quad \text{if} \quad A=1 \\ &= \frac{\Pr(A=1|L=l)}{\Pr(A=0|L=l)} \quad \text{if} \quad A=0\end{aligned}$$

For SMR weighted to the unexposed (A=0) the equation becomes

$$\pi_i = \frac{\Pr(A=0|L=l)}{\Pr(A=1|L=l)} \text{ if } A=1$$
$$= 1 \text{ if } A=0$$

Diagnostics are also available for generated IPTW. For a full list of diagnostics, see specific function documentation below. Additionally, review the references listed for an in-depth explanation

Parameters

- **df** (*DataFrame*) – Pandas dataframe object containing all variables of interest
- **treatment** (*str*) – Variable name of treatment of interest. Must be coded as binary
- **outcome** (*str*) – Variable name of outcome of interest. Can be either binary or continuous
- **standardize** (*str, optional*) – Who to standardize the estimate to. Options are the entire population, the exposed, or the unexposed. See Sato & Matsuyama Epidemiology (2003) for details on weighting to exposed/unexposed. Weighting to the exposed or unexposed is also referred to as SMR weighting. Options for standardization are: * 'population' : weight to entire population * 'exposed' : weight to exposed individuals * 'unexposed' : weight to unexposed individuals
- **weights** (*str, optional*) – Optional column for weights. If specified, a weighted regression model is instead used to estimate the inverse probability of treatment weights. This optional is useful in the following scenario; some confounder information is missing and IPMW was used to correct for missing data. IPTW should be estimated with the IPMW to standardize to the correct pseudo-population.

Examples

Setting up environment

```
>>> import matplotlib.pyplot as plt
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.ipw import IPTW
>>> df = load_sample_data(timevary=False).drop(columns=['cd4_wk45'])
>>> df[['cd4_rs1','cd4_rs2']] = spline(df, 'cd40', n_knots=3, term=2,␣
↪restricted=True)
>>> df[['age_rs1','age_rs2']] = spline(df, 'age0', n_knots=3, term=2,␣
↪restricted=True)
```

Calculate stabilized IPTW

```
>>> ipt = IPTW(df, treatment='art', outcome='dead')
>>> ipt.treatment_model('male + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0')
>>> ipt.marginal_structural_model('art')
>>> ipt.fit()
>>> ipt.summary()
```

Diagnostics:

```
>>> ipt.run_diagnostics()
```

Calculate unstabilized IPTW weights

```
>>> ipt = IPTW(df, treatment='art', outcome='dead')
>>> ipt.treatment_model('male + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0', stabilized=False)
>>> ipt.marginal_structural_model('art')
>>> ipt.fit()
>>> ipt.summary()
```

Calculate SMR weight to the exposed population

```
>>> ipt = IPTW(df, treatment='art', outcome='dead', standardize='exposed')
>>> ipt.treatment_model('male + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0')
>>> ipt.marginal_structural_model('art')
>>> ipt.fit()
>>> ipt.summary()
```

Stabilized IPTW with IPCW

```
>>> ipt = IPTW(df, treatment='art', outcome='dead')
>>> ipt.treatment_model('male + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0')
>>> ipt.missing_model('art + male + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2␣
↪+ dvl0')
>>> ipt.marginal_structural_model('art')
>>> ipt.fit()
>>> ipt.summary()
```

Stabilized IPTW with effect measure modifier

```
>>> ipt = IPTW(df, treatment='art', outcome='dead')
>>> ipt.treatment_model('male + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0', model_numerator='male')
>>> ipt.marginal_structural_model('art + male + art:male')
>>> ipt.fit()
>>> ipt.summary()
```

References

Robins JM, Hernan MA, Brumback B. (2000). Marginal structural models and causal inference in epidemiology.

Hernán MÁ, Brumback B, Robins JM. (2000). Marginal structural models to estimate the causal effect of zidovudine on the survival of HIV-positive men. Epidemiology, 561-570.

Bodnar LM, Davidian M, Siega-Riz AM, Tsiatis AA. (2004). Marginal structural models for analyzing causal effects of time-dependent treatments: an application in perinatal epidemiology. American Journal of Epidemiology, 159(10), 926-934.

Cole SR, Hernán MA. (2008). Constructing inverse probability weights for marginal structural models. American journal of epidemiology, 168(6), 656-664.

Austin PC, Stuart EA. (2015). Moving towards best practice when using inverse probability of treatment weighting (IPTW) using the propensity score to estimate causal treatment effects in observational studies. Statistics in medicine, 34(28), 3661-3679.

Sato T, Matsuyama Y. (2003). Marginal structural models as a tool for standardization. Epidemiology, 14(6), 680-686.

Love T. (2004). Graphical Display of Covariate Balance. Presentation, See http://chrp.org/love/JSM2004RoundTableHandout. pdf, 1364.

__init__(*df*, *treatment*, *outcome*, *weights=None*, *standardize='population'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, treatment, outcome[, weights, ...])	Initialize self.
fit([continuous_distribution])	Fit the specified marginal structural model using the calculated inverse probability of treatment weights.
marginal_structural_model(model)	Specify the marginal structural model to estimate using the inverse probability of treatment weights.
missing_model(model_denominator[, ...])	Estimation of Pr(M=0|A=a,L), which is the missing data mechanism for the outcome.
plot_boxplot([measure])	Generates a stratified boxplot that can be used to visually check whether positivity may be violated, qualitatively.
plot_kde([measure, bw_method, fill, ...])	Generates a density plot that can be used to check whether positivity may be violated qualitatively.
plot_love([color_unweighted, ...])	Generates a Love-plot to detail covariate balance based on the IPTW weights.
positivity([decimal, iptw_only])	Use this to assess whether positivity is a valid assumption.
run_diagnostics([iptw_only])	Run all currently implemented diagnostics for inverse probability of treatment weights available.
standardized_mean_differences([iptw_only])	Calculates the standardized mean differences for all variables.
summary([decimal])	Prints a summary of the results for the IPTW estimator.
treatment_model(model_denominator[, ...])	Logistic regression model(s) for propensity score models.

zepid.causal.ipw.IPTW.StochasticIPTW

class zepid.causal.ipw.IPTW.**StochasticIPTW**(*df*, *treatment*, *outcome*, *weights=None*)
Calculates the IPTW estimate for stochastic treatment plans. *StochasticIPTW* will returns the estimated marginal outcome for that treatment plan. This is distinct from *IPTW*, which returns an array of weights. For confidence intervals, a bootstrapping procedure needs to be used.

The formula for IPTW for a stochastic treatment is

$$\pi_i = \frac{\overline{\Pr}(A = a|L)}{\Pr(A = a|L)}$$

where $\bar{\Pr}$ is the new probability of treatment under the proposed stochastic treatment. This probability can be unconditional (everyone treated at some constant percent) or it can be conditional on observed covariates. The denominator is the same estimated probability of treatment in the standard IPTW formula. Basically, we are manipulating how many the treated individuals represent in a new pseudo-population

Note: *StochasticIPTW* estimates the marginal outcome at a specified treatment distribution. Unlike IPTW, it does not immediately result in a comparison between two treatment levels (i.e. we are not estimating a marginal structural model in this case). For a comparison, two different version would need to be specified.

StochasticIPTW does not contain the diagnostics that are contained within *IPTW*. This IPTW estimation approach makes weaker assumptions regarding positivity and causal consistency.

Parameters

- **df** (*DataFrame*) – Pandas dataframe object containing all variables of interest
- **treatment** (*str*) – Variable name of treatment variable of interest. Must be coded as binary. 1 should indicate treatment, while 0 indicates no treatment
- **outcome** (*str*) – Variable name of outcome variable of interest. May be binary or continuous.
- **weights** (*str, optional*) – Optional column for weights. If specified, a weighted regression model is instead used to estimate the inverse probability of treatment weights. This optional is useful in the following scenario; some confounder information is missing and IPMW was used to correct for missing data. IPTW should be estimated with the IPMW to standardize to the correct pseudo-population.

Examples

Loading data

```
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.ipw import StochasticIPTW
>>> df = load_sample_data(timevary=False).drop(columns=['cd4_wk45'])
>>> df[['cd4_rs1','cd4_rs2']] = spline(df,'cd40',n_knots=3,term=2,restricted=True)
>>> df[['age_rs1','age_rs2']] = spline(df,'age0',n_knots=3,term=2,restricted=True)
```

Estimating marginal outcome under treatment plan where 80% are randomly treated

```
>>> ipw = StochasticIPTW(df, treatment='art', outcome='dead')
>>> ipw.treatment_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> ipw.fit(p=0.8)
>>> ipw.summary()
```

Estimating marginal outcome under treatment plan where 10% are randomly treated

```
>>> ipw.fit(p=0.1)
>>> ipw.summary()
```

Estimating marginal outcome under treatment plan where 75% of men are treated and 90% of women

```
>>> ipw.fit(p=[0.75, 0.90], conditional=["df['male']==1", "df['male']==0"])
>>> ipw.summary()
```

References

Muñoz ID & van der Laan M. (2012). Population intervention causal effects based on stochastic interventions. Biometrics, 68(2), 541-549.

__init__ (*df*, *treatment*, *outcome*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, treatment, outcome[, weights])	Initialize self.
fit(p[, conditional])	Estimates the mean outcome under the specified stochastic treatment plan.As currently implemented, *p* percent of the population is randomly treated.
summary([decimal])	Prints the summary information for the marginal outcome under the treatment plan of interest.
treatment_model(model[, print_results])	Specify the parametric regression model for the observed treatment conditional on the sufficient adjustment set.

IPMW(df, missing_variable[, stabilized, ...])	Calculates inverse probability of missing weights.

zepid.causal.ipw.IPMW.IPMW

class zepid.causal.ipw.IPMW.**IPMW**(*df*, *missing_variable*, *stabilized=False*, *monotone=True*)

Calculates inverse probability of missing weights. *IPMW* automatically codes a missingness indicator (based on np.nan), so data can be directly input, without creation of missingness indicator before inputting data

The formula for stabilized IPMW is

$$\pi_i = \frac{\Pr(M=0)}{\Pr(M=0|L=l)}$$

where M=0 indicates observed data. For unstabilized IPMW

$$\pi_i = \frac{1}{\Pr(M=0|L=l)}$$

IPMW currently supports weights for a single missing variable, or a list of variables that are monotonically missing. For data to be missing monotonically, there is some ordering of the variables with missing data such that the previous variable must be observed for the later to be observed. A simple example is censoring in longitudinal data without late entry. To be observed at time *t*, the individual must be observed at time *t-1*

For multiple variables with missing data, *IPMW* determines if the two variables are uniform missing. This is a special case of monotonic missing data. As a result, *IPMW* will only need to calculate IPMW for one of the variables. See the references for further details on this

Parameters

- **df** (*DataFrame*) – Pandas Dataframe object containing all variables of interest
- **missing_variable** (*str, list*) – Column name for missing data. numpy.nan values should indicate missing observations. For multiple missing variables, a list of strings (indicating column labels) can be added
- **stabilized** (*bool, optional*) – Whether to return the stabilized or unstabilized IPMW. Default is to return unstabilized weights
- **monotone** (*bool, optional*) – Whether missing data is monotonic or nonmonotonic. This option is only used for when multiple missing variables are provided. monotone=False will give an error (for now)

Note: Nonmonotonic missing data is arguably more common in practice. Sun and Tchetgen Tchetgen recently proposed a way to estimate IPMW under nonmonotonic missing data. I plan on implementing this in a future

release. Until then *IPMW* only supports monotonic missing data

Examples

Setting up the environment

```
>>> from zepid import load_sample_data, load_monotone_missing_data
>>> from zepid.causal.ipw import IPMW
>>> df = load_sample_data(timevary=False)
```

Calculating unstabilized Inverse Probability of Missingness Weights

```
>>> ipm = IPMW(df, missing='dead', stabilized=False)
>>> ipm.regression_models(model_denominator='age0 + art + male')
>>> ipm.fit()
```

Extracting calculated weights

```
>>> ipm.Weight
```

Calculating IPMW for monotone missing variables

```
>>> df = load_monotone_missing_data()
>>> ipm = IPMW(df, missing_variable=['B', 'C'], monotone=True)
>>> ipm.regression_models(model_denominator=['L + A', 'L + B'])
>>> ipm.fit()
>>> ipm.Weight
```

References

Sun B, et al. (2017). Inverse-probability-weighted estimation for monotone and nonmonotone missing data. American Journal of Epidemiology, 187(3), 585-591.

Perkins, NJ et al. (2017). Principled approaches to missing data in epidemiologic studies. American Journal of Epidemiology, 187(3), 568-575.

Li L, Shen C, Li X, Robins JM. (2013). On weighting approaches for missing data. Statistical Methods in Medical Research, 22(1), 14-30.

Greenland S, & Finkle WD. (1995). A critical look at methods for handling missing covariates in epidemiologic regression analyses. American journal of epidemiology, 142(12), 1255-1264.

Seaman SR., White IR. (2013). Review of inverse probability weighting for dealing with missing data. Statistical Methods in Medical Research, 22(3), 278-295.

__init__(*df*, *missing_variable*, *stabilized=False*, *monotone=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, missing_variable[, stabilized, ...])	Initialize self.
fit()	Calculates the IPMW based on the predicted probabilities from the fitted logistic regression models.

Continued on next page

Table 24 – continued from previous page

regression_models(model_denominator[, ...])	Regression model to generate predicted probabilities of censoring, conditional on specified variables.

IPCW(df, idvar, time, event[, flat_df, enter])	Calculates inverse probability of censoring weights.

zepid.causal.ipw.IPCW.IPCW

class zepid.causal.ipw.IPCW.**IPCW**(*df*, *idvar*, *time*, *event*, *flat_df=False*, *enter=None*)

Calculates inverse probability of censoring weights. Note that this function will accept either a flat file (one row per individual) or a long format (multiple rows per individual). If a flat file is provided, it must be converted to a long format. This will be done automatically if *flat_df=True*. Additionally, a warning and some comparison statistics are provided. Please verify that they match. In general, it is recommended to convert the data set yourself

IPCW are calculated via logistic regression and weights are cumulative products per unique ID. IPCW can be used to correct for missing at random data by the generated model in weighted Kaplan-Meier curves. The formula used to generate the unstabilized IPCW is

$$\pi_i(t) = \prod_{R_k \leq t} \frac{1}{\Pr(C_i > R_k | \bar{L} = \bar{l}, C_i > R_{k-1})}$$

The stabilized IPCW substitutes predicted probabilities under the specified numerator model into the numerator of the previous equation. In general, it is recommended to stabilize IPCW by the time.

$$\pi_i(t) = \prod_{R_k \leq t} \frac{\Pr(C_i > R_k)}{\Pr(C_i > R_k | \bar{L} = \bar{l}, C_i > R_{k-1})}$$

Note: IPCW no longer support late-entry. The reason is that the pooled logistic regression model approach does not correctly accumulate the weights. As such, either all occurrences of late-entries need to be dropped (called the new-user design) or rows need to be back-propagated (unobserved rows are filled in). The second approach requires filling in the missing observed covariates and for time-varying variables will require imputation. The new-user design is a safer bet and generally what I will currently recommend

---

Parameters

- **df** (*DataFrame*) – Pandas DataFrame object containing all the variables of interest
- **idvar** (*str*) – String that indicates the column name for a unique identifier for each individual
- **time** (*str*) – Column name for the ending observation time
- **event** (*str*) – Column name for the event of interest
- **flat_df** (*bool, optional*) – Whether the input dataframe only contains a single row per participant. If so, the flat dataframe is converted to a long dataframe. Default is False (for multiple rows per person)
- **enter** (*str, optional*) – Time participant began being observed. Default is None. This option is only needed when flat_df=True. Late-entries are no longer supported and specifying this will lead to a ValueError

Example

Setting up the environment

```
>>> from zepid import load_sample_data
>>> from zepid.causal.ipw import IPCW
>>> df = load_sample_data(timevary=True)
>>> df['enter_q'] = df['enter'] ** 2
>>> df['enter_c'] = df['enter'] ** 3
>>> df['age0_q'] = df['age0'] ** 2
>>> df['age0_c'] = df['age0'] ** 3
```

Calculating stabilized IPCW with a long data set

```
>>> ipc = IPCW(df, idvar='id', time='enter', event='dead')
>>> ipc.regression_models(model_denominator='enter + enter_q + enter_c + male +␣
↪age0 + age0_q + age0_c',
>>>                       model_numerator='enter + enter_q + enter_c')
>>> ipc.fit()
```

Extracting calculated stabilized IPCW

```
>>> ipc.Weight
```

Calculating stabilized IPCW with a wide data set

```
>>> df = load_sample_data(False)
>>> ipc = IPCW(df, idvar='id', time='t', event='dead', flat_df=True)
>>> ipc.regression_models(model_denominator='enter + enter_q + enter_c + male +␣
↪age0 + age0_q + age0_c',
>>>                       model_numerator='enter + enter_q + enter_c')
>>> ipc.fit()
```

References

Howe CJ et al. (2016) Selection bias due to loss to follow up in cohort studies. Epidemiology, 27(1), 91-97.

__init__ (*df*, *idvar*, *time*, *event*, *flat_df=False*, *enter=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, idvar, time, event[, flat_df, ...])	Initialize self.
fit()	Calculates IPCW for each observation period for each observation.
regression_models(model_denominator, ...[, ...])	Regression model to generate predicted probabilities of censoring, conditional on specified variables.

Time-Fixed Treatment G-Formula

TimeFixedGFormula(df, exposure, outcome[, ...])	G-formula for time-fixed exposure and single endpoint, also referred to as the g-computation algorithm formula.
SurvivalGFormula(df, idvar, exposure, ...[, ...])	G-formula for time-to-event data where the exposure is fixed at baseline.

zepid.causal.gformula.TimeFixed.TimeFixedGFormula

class zepid.causal.gformula.TimeFixed.**TimeFixedGFormula**(*df*, *exposure*, *outcome*, *exposure_type='binary'*, *outcome_type='binary'*, *standardize='population'*, *weights=None*)

G-formula for time-fixed exposure and single endpoint, also referred to as the g-computation algorithm formula. Uses the Snowden trick to calculate the marginal treatment under the specified exposure plan

The g-formula can be expressed as

$$E[Y^a] = \sum_l E[Y|A = a, L = l] \times \Pr(L = l)$$

When L is continuous, the summation becomes an integral.

Currently, *TimeFixedGFormula* only supports binary or continuous outcomes. For binary outcomes a logistic regression model to predict probabilities of outcomes via statsmodels. For continuous outcomes a linear regression or a Poisson regression model can be used to predict outcomes.

Binary and multivariate exposures are supported. For binary exposures, a string object of the column name for the exposure of interest should be provided. For multivariate exposures, a list of string objects corresponding to disjoint indicator terms for the exposure should be provided. Multivariate exposures require the user to custom specify treatments when fitting the g-formula. A list of the custom treatment must be provided and be the same length as the number of disjoint indicator columns. See https://github.com/pzivich/Python-for-Epidemiologists/tree/master/3_Epidemiology_Analysis/c_causal_inference/1_time-fixed-treatments for examples (highly recommended)

Key options for treatments:

- *'all'* -all individuals are given treatment
- *'none'* -no individuals are given treatment
- custom treatments -create a custom treatment. When specifying this, the dataframe must be referred to as 'g'. The following is an example that selects those whose age is 30 or younger and are females: `treatment="((g['age0']<=30) & (g['male']==0))`

Note: Custom treatments use a "magic-g" parameter. Internally, the g-formula implementation names the data set as *g*. Therefore, when using custom treatment specifications, the data set must be referred to as *g* when following the pandas selection syntax

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **exposure** (*str, list*) – Column name for exposure variable label or a list of disjoint indicator exposures
- **outcome** (*str*) – Column name for outcome variable

- **outcome_type** (*str, optional*) – Outcome variable type. Currently only 'binary', 'normal', and 'poisson variable types are supported
- **standardize** (*str, optional*) – Who the estimate corresponds to. Options are the entire population, the exposed, or the unexposed. See Sato & Matsuyama Epidemiology (2003) for details on weighting to exposed/unexposed. Weighting to the exposed or unexposed is also referred to as SMR weighting. Options for standardization are: * 'population' : weight to entire population * 'exposed' : weight to exposed individuals * 'unexposed' : weight to unexposed individuals
- **weights** (*str, optional*) – Column name for weights. Default is None, which assumes every observations has the same weight (i.e. 1)

Examples

Setting up the environment

```
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.gformula import TimeFixedGFormula
>>> df = load_sample_data(timevary=False)
>>> df[['cd4_rs1', 'cd4_rs2']] = spline(df, 'cd40', n_knots=3, term=2,␣
↪restricted=True)
>>> df[['age_rs1', 'age_rs2']] = spline(df, 'age0', n_knots=3, term=2,␣
↪restricted=True)
```

G-formula with a binary treatment and outcome

```
>>> g = TimeFixedGFormula(df, exposure='art', outcome='dead')
>>> g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1␣
↪+ cd4_rs2 + dvl0')
```

```
>>> # Return the estimated marginal outcome under treat-all
>>> g.fit(treatment='all')
>>> g.marginal_outcome
```

```
>>> # Return the estimated marginal outcome under treat-none
>>> g.fit(treatment='all')
>>> g.marginal_outcome
```

```
>>> # Return the estimated marginal outcome under custom treatment (treat all␣
↪females under 40)
>>> g.fit(treatment="((g['male']==0) & (g['age0']<=40))")
>>> g.marginal_outcome
```

G-formula with a categorical treatment and binary outcome

```
>>> # Creating categorical variable for CD4 count
>>> df['cd4_1'] = np.where(((df['cd40'] >= 200) & (df['cd40'] < 400)), 1, 0)
>>> df['cd4_2'] = np.where(df['cd40'] >= 400, 1, 0)
```

```
>>> g = TimeFixedGFormula(df,exposure=['art_male', 'art_female'], outcome='dead',␣
↪exposure_type='categorical')
>>> g.outcome_model(model='cd4_1 + cd4_2 + art + male + age0 + age_rs1 + age_rs2␣
↪+ dvl0')
```

```
>>> # Return marginal outcome under all in reference category (CD4 < 200)
>>> g.fit(treatment=["False", "False"])
```

```
>>> # Return marginal outcome under all in category 1 (CD4 >= 200 & CD4 < 400)
>>> g.fit(treatment=["True", "False"])
```

```
>>> # Return marginal outcome under all in category 2 (CD4 > 400)
>>> g.fit(treatment=["False", "True"])
```

G-formula with binary exposure and continuous (normal-distributed) outcome

```
>>> g = TimeFixedGFormula(df,exposure='art', outcome='cd4', outcome_type='normal')
>>> g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + dvl0  + cd40 +␣
↪cd4_rs1 + cd4_rs2')
```

G-formula with binary exposure and continuous (Poisson-distributed) outcome

```
>>> g = TimeFixedGFormula(df,exposure='art', outcome='cd4', outcome_type='poisson
↪')
>>> g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + dvl0  + cd40 +␣
↪cd4_rs1 + cd4_rs2')
```

G-formula with binary outcome and exposure. With a stochastic treatment/intervention

```
>>> g = TimeFixedGFormula(df,exposure='art', outcome='cd4', outcome_type='poisson
↪')
>>> g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + dvl0  + cd40 +␣
↪cd4_rs1 + cd4_rs2')
>>> g.fit_stochastic(p=0.75)
```

G-formula with binary outcome and exposure. With a conditional stochastic treatment/intervention

```
>>> g = TimeFixedGFormula(df,exposure='art', outcome='cd4')
>>> g.outcome_model(model='art + male + age0 + age_rs1 + age_rs2 + dvl0  + cd40 +␣
↪cd4_rs1 + cd4_rs2')
>>> g.fit_stochastic(p=[0.65, 0.85], conditional=["g['male']==1", "g['male']==0"])
```

References

JM Snowden, S Rose, and KM Mortimer. "Implementation of G-computation on a simulated data set: demonstration of a causal inference technique." American Journal of Epidemiology 173.7 (2011): 731-738.

J Ahern, KE Colson, C Margerson-Zilko, A Hubbard, & S Galea. (2016). Predicting the population health impacts of community interventions: the case of alcohol outlets and binge drinking. American Journal of Public Health, 106(11), 1938-1943.

J Ahern, A Hubbard, & S Galea. (2009). Estimating the effects of potential public health interventions on population disease burden: a step-by-step illustration of causal inference methods. American Journal of Epidemiology, 169(9), 1140-1147.

__init__(*df*, *exposure*, *outcome*, *exposure_type='binary'*, *outcome_type='binary'*, *standardize='population'*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(df, exposure, outcome[, ...])	Initialize self.
`fit`(treatment[, predict_missing])	Fit the parametric g-formula as specified.
`fit_stochastic`(p[, conditional, samples, ...])	Fits the g-formula for a stochastic intervention.
`outcome_model`(model[, print_results])	Build the outcome regression model.
`plot_kde`([bw_method, fill, color])	Generates a Kernel Density plot of the accuracy of the model predicted outcomes.
`run_diagnostics`([decimal])	Runs diagnostics for the g-formula regression model used.

zepid.causal.gformula.TimeFixed.SurvivalGFormula

class zepid.causal.gformula.TimeFixed.**SurvivalGFormula**(*df*, *idvar*, *exposure*, *outcome*, *time*, *weights=None*)

G-formula for time-to-event data where the exposure is fixed at baseline. Only supports binary exposures and outcomes. Outcomes are predicted using a logistic regression model. Input data set should be in a long format, where each row corresponds to an individual observed for one unit of time

Key options for treatments:

- *'all'* -all individuals are given treatment
- *'none'* -no individuals are given treatment
- custom treatments -create a custom treatment. When specifying this, the dataframe must be referred to as 'g'. The following is an example that selects those whose age is 30 or younger and are females: `treatment="((g['age0']<=30) & (g['male']==0))`

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **idvar** (*str*) – Column name for the ID label
- **exposure** (*str, list*) – Column name for exposure variable label or a list of disjoint indicator exposures
- **outcome** (*str*) – Column name for outcome variable
- **time** (*str*) – Column name for time variable
- **weights** (*str, optional*) – Column name for weights. Default is None, which assumes every observations has the same weight (i.e. 1)

Note: Custom treatments use a "magic-g" parameter. Internally, the g-formula implementation names the data set as *g*. Therefore, when using custom treatment specifications, the data set must be referred to as *g* when following the pandas selection syntax

---

Examples

Setting up data in long format

```
>>> from zepid import load_sample_data
>>> from zepid.causal.gformula import SurvivalGFormula
>>> import matplotlib.pyplot as plt
>>> df = load_sample_data(False).drop(columns=['cd4_wk45'])
```

```
>>> df['t'] = np.round(df['t']).astype(int)
>>> df = pd.DataFrame(np.repeat(df.values, df['t'], axis=0), columns=df.columns)
>>> df['t'] = df.groupby('id')['t'].cumcount() + 1
>>> df.loc[((df['dead'] == 1) & (df['id'] != df['id'].shift(-1))), 'd'] = 1
>>> df['d'] = df['d'].fillna(0)
>>> df['t_sq'] = df['t']**2
>>> df['t_cu'] = df['t']**3
```

Estimating the time-to-event mean effect under treat-all plan

```
>>> sgf = SurvivalGFormula(df.drop(columns=['dead']), idvar='id', exposure='art',␣
↪outcome='d', time='t')
>>> sgf.outcome_model(model='art + male + age0 + cd40 + dvl0 + t + t_sq + t_cu')
>>> sgf.fit(treatment='all')
>>> print(sgf.marginal_outcome)
```

Plotting cumulative incidence function

```
>>> sgf.plot(color='r')
>>> plt.show()
```

Estimating the time-to-event mean effect under treat-none plan

```
>>> sgf = SurvivalGFormula(df.drop(columns=['dead']), idvar='id', exposure='art',␣
↪outcome='d', time='t')
>>> sgf.outcome_model(model='art + male + age0 + cd40 + dvl0 + t + t_sq + t_cu')
>>> sgf.fit(treatment='none')
```

Estimating the time-to-event mean effect under custom treatment plan

```
>>> sgf = SurvivalGFormula(df.drop(columns=['dead']), idvar='id', exposure='art',␣
↪outcome='d', time='t')
>>> sgf.outcome_model(model='art + male + age0 + cd40 + dvl0 + t + t_sq + t_cu')
>>> sgf.fit(treatment="((g['age0']>=25) & (g['male']==0))")
```

Notes

The following process is used to estimate the cumulative incidence function. (1) A pooled logistic regression model is fit to the data. The model should predict the outcome conditional on treatment, baseline confounders, and time. Time should be modeled using flexible functional forms (e.g. splines) (2) Survival probabilities are estimated by predicting values at each time from the pooled logistic model and taking the cumulative product. The survival probabilities are predicted under the treatment plan of interest (3) Average the cumulative incidence function for each time period from all the subjects.

References

Hernán MA. (2010). The hazards of hazard ratios. Epidemiology, 21(1), 13–15. doi:10.1097/EDE.0b013e3181c1ea43

__init__(*df*, *idvar*, *exposure*, *outcome*, *time*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, idvar, exposure, outcome, time)	Initialize self.
fit(treatment)	Fit the parametric g-formula for time-to-event data.
outcome_model(model[, print_results])	Build the pooled logistic model.
plot(**plot_kwargs)	Plots the estimated cumulative incidence function

Time-Varying Treatment G-Formula

MonteCarloGFormula(df, idvar, exposure, ...)	Time-varying implementation of the Monte Carlo g-formula.
IterativeCondGFormula(df, exposures, outcomes)	Iterative conditional g-formula estimator.

zepid.causal.gformula.TimeVary.MonteCarloGFormula

class zepid.causal.gformula.TimeVary.**MonteCarloGFormula**(*df*, *idvar*, *exposure*, *outcome*, *time_in*, *time_out*, *weights=None*)

Time-varying implementation of the Monte Carlo g-formula. The Monte Carlo estimator is useful for survival data. For an extensive walkthrough of the Monte Carlo g-formula, see Keil et al. 2014 and other listed references. This implementation has four options for the treatment courses:

Options for treatments * all : all individuals are given treatment * none : no individuals are given treatment * natural : individuals retain their observed treatment * custom : create a custom treatment. When specifying this, the dataframe must be referred to as 'g' The

following is an example that selects those whose age is 25 or older and are females Ex) treatment="((g['age0']>=25) & (g['male']==0))

Note: Custom treatments use a "magic-g" parameter. Internally, the g-formula implementation names the data set as *g*. Therefore, when using custom treatment specifications, the data set must be referred to as *g* when following the pandas selection syntax

Currently, only binary exposures and a binary outcomes are supported. Logistic regression models are used to predict exposures and outcomes via statsmodels. See http://zepid.readthedocs.io/en/latest/ for an example (highly recommended)

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **idvar** (*str*) – ID column label
- **exposure** (*str*) – Treatment column label
- **outcome** (*str*) – Outcome column label
- **time_out** (*str*) – End of follow-up period time column label
- **time_in** (*str*) – Start of follow-up period time label

- **weights** (*str, optional*) – Column label for weights. Default is None, which assumes every observations has the same weight (i.e. 1)

Notes

1) Monte Carlo increases by time units of one. Input dataset should reflect this
2) Only binary exposures and binary outcomes are supported
3) Binary and continuous covariates are supported
4) The labeling of the covariate models is important. They are fit in the order that they are labeled!
5) Fit the natural course model first and compare to the observed data. They should be similar

Process for the Monte Carlo g-formula

1) run lines in "in_recode"
2) time-varying covariates, order ascending in from "labels"
    a) predict time-varying covariate
    b) run lines in "recode" from "add_covariate_model()"
3) predict exposure / apply exposure pattern
4) predict outcome
5) run lines in "out_recode"
6) lag variables in "lags"
7) append current-time rows to full dataframe
8) Repeat till t_max is met

Examples

Setting up the environment

```
>>> import numpy as np
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.gformula import MonteCarloGFormula
>>> df = load_sample_data(timevary=True)
>>> df['lag_art'] = df['art'].shift(1)
>>> df['lag_art'] = np.where(df.groupby('id').cumcount() == 0, 0, df['lag_art'])
>>> df['lag_cd4'] = df['cd4'].shift(1)
>>> df['lag_cd4'] = np.where(df.groupby('id').cumcount() == 0, df['cd40'], df[
↪'lag_cd4'])
>>> df['lag_dvl'] = df['dvl'].shift(1)
>>> df['lag_dvl'] = np.where(df.groupby('id').cumcount() == 0, df['dvl0'], df[
↪'lag_dvl'])
>>> df[['age_rs0', 'age_rs1', 'age_rs2']] = spline(df, 'age0', n_knots=4, term=2,␣
↪restricted=True)
>>> df['cd40_sq'] = df['cd40'] ** 2
>>> df['cd40_cu'] = df['cd40'] ** 3
>>> df['cd4_sq'] = df['cd4'] ** 2
>>> df['cd4_cu'] = df['cd4'] ** 3
>>> df['enter_sq'] = df['enter'] ** 2
>>> df['enter_cu'] = df['enter'] ** 3
```

Estimating the g-formula with the Monte Carlo estimator

```
>>> g = MonteCarloGFormula(df, idvar='id', exposure='art', outcome='dead', time_
↪in='enter', time_out='out')
```

```
>>> # Specifying the exposure/treatment model
>>> exp_m = 'male + age0 + age_rs0 + age_rs1 + age_rs2 + cd40 + cd40_sq + cd40_cu␣
↪+ dvl0 + cd4 + cd4_sq +' +
>>>         'cd4_cu + dvl + enter + enter_sq + enter_cu'
>>> g.exposure_model(exp_m, restriction="g['lag_art']==0")  # restriction␣
↪enforces intent-to-treat
```

```
>>> # Specifying the outcome model
>>> out_m = 'art + male + age0 + age_rs0 + age_rs1 + age_rs2 + cd40 + cd40_sq +␣
↪cd40_cu + dvl0 + cd4 +' +
>>>         'cd4_sq + cd4_cu + dvl + enter + enter_sq + enter_cu'
>>> g.outcome_model(out_m, restriction="g['drop']==0")  # restriction enforces␣
↪loss-to-follow-up
```

```
>>> # Specifying the time-varying confounder models
>>> dvl_m = 'male + age0 + age_rs0 + age_rs1 + age_rs2 + cd40 + cd40_sq + cd40_cu␣
↪+ dvl0 + lag_cd4 + ' +
>>>         'lag_dvl + lag_art + enter + enter_sq + enter_cu'
>>> g.add_covariate_model(label=1, covariate='dvl', model=dvl_m, var_type='binary
↪')
>>> cd4_m = 'male + age0 + age_rs0 + age_rs1 + age_rs2 +  cd40 + cd40_sq + cd40_
↪cu + dvl0 + lag_cd4 + ' +
>>>         'lag_dvl + lag_art + enter + enter_sq + enter_cu'
>>> cd4_recode_scheme = ("g['cd4'] = np.maximum(g['cd4'],1);"  # Recode scheme␣
↪makes sure variables are recoded
>>>                      "g['cd4_sq'] = g['cd4']**2;"
>>>                      "g['cd4_cu'] = g['cd4']**3")
>>> g.add_covariate_model(label=2, covariate='cd4', model=cd4_m, recode=cd4_
↪recode_scheme, var_type='continuous')
```

```
>>> # Specifying a model for informative censoring
>>> cens_m = "male + age0 + age_rs0 + age_rs1 + age_rs2 +  cd40 + cd40_sq + cd40_
↪cu + dvl0 + lag_cd4 +" +
>>>          "lag_dvl + lag_art + enter + enter_sq + enter_cu"
>>> g.censoring_model(cens_m)
```

```
>>> # Estimating outcomes under a simulated Markov Chain Monte Carlo for natural␣
↪course
>>> g.fit(treatment="((g['art']==1) | (g['lag_art']==1))",  # Treatment plan␣
↪(natural course in this case)
>>>        lags={'art': 'lag_art',  # Creating variables to lag in the process
>>>              'cd4': 'lag_cd4',
>>>              'dvl': 'lag_dvl'},
>>>        sample=50000,  # Number of resamples to use (should be large number to␣
↪reduce simulation error)
>>>        t_max=None,  # Maximum time to simulate to (None uses data set maximum␣
↪time)
>>>        in_recode=("g['enter_sq'] = g['enter']**2;"
>>>                   "g['enter_cu'] = g['enter']**3"))  # How to recode time in␣
↪each time-step
>>> # See website documentation for further instructions
```

(continues on next page)

(continued from previous page)

```
>>> # (https://zepid.readthedocs.io/en/latest/Causal.html#g-computation-algorithm-
↪monte-carlo)
```

References

Keil, AP, Edwards, JK, Richardson, DB, Naimi, AI, Cole, SR (2014). The Parametric g-Formula for Time-to-Event Data: Intuition and a Worked Example. Epidemiology 25(6), 889-897

__init__ (*df*, *idvar*, *exposure*, *outcome*, *time_in*, *time_out*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, idvar, exposure, outcome, ...)	Initialize self.
add_covariate_model(label, covariate, model)	Add a specified regression model for time-varying confounders.
censoring_model(model[, restriction, ...])	Add a specified regression model for censoring.
exposure_model(model[, restriction, ...])	Add a specified regression model for the exposure.
fit(treatment[, lags, sample, t_max, ...])	Estimate the counterfactual outcomes under the specified treatment plan using the previously specified regression models.
outcome_model(model[, restriction, ...])	Add a specified regression model for the outcome.

zepid.causal.gformula.TimeVary.IterativeCondGFormula

class zepid.causal.gformula.TimeVary.**IterativeCondGFormula**(*df*, *exposures*, *outcomes*)

Iterative conditional g-formula estimator. This time-varying parametric g-formula uses the iterative conditional approach (also referred to as the sequential regression). The iterative conditional estimator is useful for longitudinal data and requires less model specification than the Monte Carlo g-formula. The iterative conditional uses a mathematical trick to estimate the marginal outcome distribution at the end of follow-up

Unlike other implementations of the g-formula, the *IterativeCondGFormula* takes input data in a wide format. Additionally, treatments are specified by explicitly specifying the treatment plan array. See the examples for details

Currently, only binary exposures and a binary outcomes are supported. Logistic regression models are used to predict exposures and outcomes via statsmodels. See Kreif et al. 2017 for a good description of the iterative conditional g-formula. See http://zepid.readthedocs.io/en/latest/ for an example

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **exposures** (*list, array*) – Treatment column label
- **outcomes** (*list, array*) – Outcome column label

Notes

Process for the sequential regression g-formula 1) Identify individuals who followed the counterfactual treatment plan and had the outcome 2) Fit a regression model for the outcome at time t for Y 3) Predict outcomes under

the observed treatment and the counterfactual treatment 4) Repeat regression model fitting for t-1 to min(t) 5) Take the mean predicted Y at the end to obtain the cumulative probability

Examples

Setting up the environment

```
>>> from zepid import load_longitudinal_data
>>> from zepid.causal.gformula import IterativeCondGFormula
>>> df = load_longitudinal_data()
```

Estimating the g-formula with the Monte Carlo estimator

```
>>> icgf = IterativeCondGFormula(df, exposures=['A1', 'A2', 'A3'], outcomes=['Y1',
↪ 'Y2', 'Y3'])
```

```
>>> # Specifying regression models for each treatment-outcome pair
>>> icgf.outcome_model(models=['A1 + L1', 'A2 + A1 + L2', 'A3 + A2 + L3'], print_
↪results=False)
```

```
>>> # Estimating marginal 'Y3' under treat-all at every time
>>> icgf.fit(treatments=[1, 1, 1])
>>> print(icgf.marginal_outcome)
```

```
>>> # Estimating marginal 'Y3' under treat-none at every time
>>> icgf.fit(treatments=[0, 0, 0])
>>> print(icgf.marginal_outcome)
```

Custom treatments can be specified. Below is an example of treating everyone at the first and last time points

```
>>> # Estimating marginal 'Y3' under custom treatment plan
>>> icgf.fit(treatments=[1, 0, 1])
>>> print(icgf.marginal_outcome)
```

To estimate 'Y2', we can use a similar procedure but restrict our list of exposures and outcomes

```
>>> icgf = IterativeCondGFormula(df, exposures=['A1', 'A2'], outcomes=['Y1', 'Y2
↪'])
>>> icgf.outcome_model(models=['A1 + L1', 'A2 + A1 + L2'], print_results=False)
>>> icgf.fit(treatments=[1, 1])
>>> print(icgf.marginal_outcome)
```

References

Kreif, N., Tran, L., Grieve, R., De Stavola, B., Tasker, R. C., & Petersen, M. (2017). Estimating the comparative effectiveness of feeding interventions in the pediatric intensive care unit: a demonstration of longitudinal targeted maximum likelihood estimation. American Journal of Epidemiology, 186(12), 1370-1379.

__init__ (*df*, *exposures*, *outcomes*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposures, outcomes)	Initialize self.
fit(treatments)	Estimate the counterfactual outcomes under the specified treatment plan using the previously specified regression models
outcome_model(models[, print_results])	Add a specified regression model for the outcome.

Augmented Inverse Probability Weights

AIPTW(df, exposure, outcome[, weights, alpha])	Augmented inverse probability of treatment weight estimator.

zepid.causal.doublyrobust.AIPW.AIPTW

class zepid.causal.doublyrobust.AIPW.**AIPTW**(*df*, *exposure*, *outcome*, *weights=None*, *alpha=0.05*)

Augmented inverse probability of treatment weight estimator. This implementation calculates AIPTW for a time-fixed exposure and a single time-point outcome. *AIPTW* supports correcting for informative censoring (missing outcome data) through inverse probability of censoring/missingness weights.

AIPTW is a doubly robust estimator, with a desirable property. Both of the the g-formula and IPTW require that our parametric regression models are correctly specified. Instead, AIPTW allows us to have two 'chances' at getting the model correct. If either our outcome-model or treatment-model is correctly specified, then our estimate will be unbiased. This property does not hold for the variance (i.e. the variance will not be doubly robust)

The augment-inverse probability weight estimator is calculated from the following formula

$$\widehat{DR}(a) = \frac{YA}{\widehat{\Pr}(A=a|L)} - \frac{\hat{Y}^a * (A - \widehat{\Pr}(A=a|L)}{\widehat{\Pr}(A=a|L)}$$

The risk difference and risk ratio are calculated using the following formulas, respectively

$$\widehat{RD} = \widehat{DR}(a=1) - \widehat{DR}(a=0)$$

$$\widehat{RR} = \frac{\widehat{DR}(a=1)}{\widehat{DR}(a=0)}$$

Confidence intervals for the risk difference come from the influence curve. Confidence intervals for the risk ratio are less straight-forward. To get confidence intervals for the risk ratio, a bootstrap procedure should be used.

Parameters

- **df** (*DataFrame*) – Pandas DataFrame object containing all variables of interest
- **exposure** (*str*) – Column name of the exposure variable. Currently only binary is supported
- **outcome** (*str*) – Column name of the outcome variable. Currently only binary is supported
- **weights** (*str, optional*) – Column name of weights. Weights allow for items like sampling weights to be used to estimate effects
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05, returning the 95% CL

Examples

Set up the environment and the data set

```
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.doublyrobust import AIPTW
>>> df = load_sample_data(timevary=False).drop(columns=['cd4_wk45'])
>>> df[['cd4_rs1','cd4_rs2']] = spline(df,'cd40',n_knots=3,term=2,restricted=True)
>>> df[['age_rs1','age_rs2']] = spline(df,'age0',n_knots=3,term=2,restricted=True)
```

Estimate the base AIPTW model

```
>>> aipw = AIPTW(df, exposure='art', outcome='dead')
>>> aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> aipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0')
>>> aipw.fit()
>>> aipw.summary()
```

Estimate AIPTW accounting for missing outcome data

```
>>> aipw = AIPTW(df, exposure='art', outcome='dead')
>>> aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> aipw.missing_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0')
>>> aipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0')
>>> aipw.fit()
>>> aipw.summary()
```

AIPTW for continuous outcomes

```
>>> df = load_sample_data(timevary=False).drop(columns=['dead'])
>>> df[['cd4_rs1','cd4_rs2']] = spline(df,'cd40',n_knots=3,term=2,restricted=True)
>>> df[['age_rs1','age_rs2']] = spline(df,'age0',n_knots=3,term=2,restricted=True)
```

```
>>> aipw = AIPTW(df, exposure='art', outcome='cd4_wk45')
>>> aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> aipw.missing_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0')
>>> aipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 +␣
↪cd4_rs2 + dvl0')
>>> aipw.fit()
>>> aipw.summary()
```

```
>>> aipw = AIPTW(df, exposure='art', outcome='cd4_wk45')
>>> ymodel = 'art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 +␣
↪dvl0'
>>> aipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> aipw.missing_model(ymodel)
>>> aipw.outcome_model(ymodel, continuous_distribution='poisson')
>>> aipw.fit()
>>> aipw.summary()
```

References

Funk MJ, Westreich D, Wiesen C, Stürmer T, Brookhart MA, & Davidian M. (2011). Doubly robust estimation of causal effects. American Journal of Epidemiology, 173(7), 761-767.

Lunceford JK, Davidian M. (2004). Stratification and weighting via the propensity score in estimation of causal treatment effects: a comparative study. Statistics in medicine, 23(19), 2937-2960.

__init__ (*df*, *exposure*, *outcome*, *weights=None*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, weights, alpha])	Initialize self.
exposure_model(model[, custom_model, bound, ...])	Specify the propensity score / inverse probability weight model.
fit()	Calculate the augmented inverse probability weights and effect measures from the predicted exposure probabilities and predicted outcome values.
missing_model(model[, custom_model, bound, ...])	Estimation of Pr(M=0|A,L), which is the missing data mechanism for the outcome.
outcome_model(model[, custom_model, ...])	Specify the outcome model.
plot_kde(to_plot[, bw_method, fill, color, ...])	Generates density plots that can be used to check predictions qualitatively.
plot_love([color_unweighted, ...])	Generates a Love-plot to detail covariate balance based on the IPTW weights.
positivity([decimal])	Use this to assess whether positivity is a valid assumption for the exposure model / calculated IPTW.
run_diagnostics([decimal])	Run all currently implemented diagnostics for the exposure and outcome models.
standardized_mean_differences()	Calculates the standardized mean differences for all variables based on the inverse probability weights.
summary([decimal])	Prints a summary of the results for the doubly robust estimator.

SingleCrossfitAIPTW(df, exposure, outcome[, ...])	Implementation of the Augmented Inverse Probability Weighting estimator with a cross-fit procedure.
DoubleCrossfitAIPTW(df, exposure, outcome[, ...])	Implementation of the augmented inverse probability weighted estimator with a double cross-fit procedure.

zepid.causal.doublyrobust.crossfit.SingleCrossfitAIPTW

class zepid.causal.doublyrobust.crossfit.**SingleCrossfitAIPTW**(*df*, *exposure*, *outcome*, *alpha=0.05*)

Implementation of the Augmented Inverse Probability Weighting estimator with a cross-fit procedure. The purpose of the cross-fit procedure is to all for non-Donsker nuisance function estimators. Some of machine learning algorithms are non-Donsker. In practice this means that confidence interval coverage can be incorrect when certain nuisance function estimators are used. Additionally, bias may persist as well. Cross-fitting is meant to alleviate this issue, therefore cross-fitting with a doubly-robust estimator is recommended when using machine learning.

SingleCrossfitAIPTW uses a single cross-fit procedure, where the data set is paritioned into at least two non-overlapping splits. The nuisance function estimators are then estimated in each split. The estimated nuisance functions are then used to predict values in a non-overlapping split. This decouple the nuisance function estimation from the data used to estimate it

---

Note: Because of the repetitions of the procedure are needed to reduce variance determined by a particular partition, it can take a long time to run this code.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all necessary variables
- **exposure** (*str*) – Label for treatment column in the pandas data frame
- **outcome** (*str*) – Label for outcome column in the pandas data frame
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05

Examples

Setting up environment

```
>>> from sklearn.linear_model import LogisticRegression
>>> from zepid import load_sample_data
>>> from zepid.causal.doublyrobust import SingleCrossfitAIPTW
>>> df = load_sample_data(False).drop(columns='cd4_wk45').dropna()
```

Estimating the single cross-fit AIPTW

```
>>> scaipw = SingleCrossfitAIPTW(df, exposure='art', outcome='dead')
>>> scaipw.exposure_model("male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> scaipw.outcome_model("art + male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> scaipw.fit(n_splits=5, n_partitions=100)
>>> scaipw.summary()
```

References

Chernozhukov V, Chetverikov D, Demirer M, Duflo E, Hansen C, Newey W, & Robins J. (2018). "Double/debiased machine learning for treatment and structural parameters". The Econometrics Journal 21:1; pC1–C6

__init__(*df*, *exposure*, *outcome*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, alpha])	Initialize self.
exposure_model(covariates, estimator[, bound])	Specify the treatment nuisance model variables and estimator(s) to use.

Continued on next page

Table 36 – continued from previous page

fit([n_splits, n_partitions, method, . . .])	Runs the crossfit estimation procedure with augmented inverse probability weighting estimator.
outcome_model(covariates, estimator)	Specify the outcome nuisance model variables and estimator(s) to use.
run_diagnostics([color])	Runs available diagnostics for the plots.
summary([decimal])	Prints summary of model results

zepid.causal.doublyrobust.crossfit.DoubleCrossfitAIPTW

class zepid.causal.doublyrobust.crossfit.**DoubleCrossfitAIPTW**(*df*, *exposure*, *outcome*, *alpha=0.05*)

Implementation of the augmented inverse probability weighted estimator with a double cross-fit procedure. The purpose of the cross-fit procedure is to all for non-Donsker nuisance function estimators. Some of machine learning algorithms are non-Donsker. In practice this means that confidence interval coverage can be incorrect when certain nuisance function estimators are used. Additionally, bias may persist as well. Cross-fitting is meant to alleviate this issue, therefore cross-fitting with a doubly-robust estimator is recommended when using machine learning.

DoubleCrossfitAIPTW allows for double cross-fitting, where the data set is partitioned into at least three non-overlapping splits. The nuisance function estimators are then estimated in each split. The estimated nuisance functions are then used to predict values in the opposing split. Different splits are used for each nuisance function. A double cross-fit procedure further de-couples the nuisance function estimation compared to single cross-fit procedures.

Note: Because of the repetitions of the procedure are needed to reduce variance determined by a particular partition, it can take a long time to run this code. On a data set of 3000 observations with 100 different partitions it takes about an hour. The advantage is that the code can be ran in parallel. See the documentation for an example.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all necessary variables
- **exposure** (*str*) – Label for treatment column in the pandas data frame
- **outcome** (*str*) – Label for outcome column in the pandas data frame
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05

Examples

Setting up environment

```
>>> from sklearn.linear_model import LogisticRegression
>>> from zepid import load_sample_data
>>> from zepid.causal.doublyrobust import SingleCrossfitAIPTW
>>> df = load_sample_data(False).drop(columns='cd4_wk45').dropna()
```

Estimating the double cross-fit AIPTW

```
>>> dcaipw = DoubleCrossfitAIPTW(df, exposure='art', outcome='dead')
>>> dcaipw.exposure_model("male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
```

(continues on next page)

(continued from previous page)

```
>>> dcaipw.outcome_model("art + male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> dcaipw.fit(n_splits=5, n_partitions=100)
>>> dcaipw.summary()
```

References

Newey WK, Robins JR. (2018) "Cross-fitting and fast remainder rates for semiparametric estimation". arXiv:1801.09138

Zivich PN, & Breskin A. (2020). Machine learning for causal inference: on the use of cross-fit estimators. arXiv preprint arXiv:2004.10337.

Chernozhukov V, Chetverikov D, Demirer M, Duflo E, Hansen C, Newey W, & Robins J. (2018). "Double/debiased machine learning for treatment and structural parameters". The Econometrics Journal 21:1; pC1–C6

__init__ (*df*, *exposure*, *outcome*, *alpha=0.05*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, alpha])	Initialize self.
exposure_model(covariates, estimator[, bound])	Specify the treatment nuisance model variables and estimator(s) to use.
fit([n_splits, n_partitions, method, ...])	Runs the crossfit estimation procedure with augmented inverse probability weighted estimator.
outcome_model(covariates, estimator)	Specify the outcome nuisance model variables and estimator(s) to use.
run_diagnostics([color])	Runs available diagnostics for the plots.
summary([decimal])	Prints summary of model results

Targeted Maximum Likelihood Estimator

TMLE(df, exposure, outcome[, alpha, ...])	Implementation of target maximum likelihood estimator.
StochasticTMLE(df, exposure, outcome[, ...])	Implementation of target maximum likelihood estimator for stochastic treatment plans.

zepid.causal.doublyrobust.TMLE.TMLE

class zepid.causal.doublyrobust.TMLE.**TMLE**(*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*)

Implementation of target maximum likelihood estimator. This implementation calculates TMLE for a time-fixed exposure and a single time-point outcome. By default standard parametric regression models are used to calculate the estimate of interest. The TMLE estimator allows users to instead use machine learning algorithms from sklearn and PyGAM.

Note: Valid confidence intervals are only attainable with certain machine learning algorithms. These algorithms must be Donsker class for valid confidence intervals. GAM and LASSO are examples of alogorithms that are

Donsker class

Note: TMLE is a doubly-robust substitution estimator. TMLE obtains the target estimate in a single step. The single-step TMLE is described further by van der Laan. For further details, see the listed references.

Continuous outcomes must be bounded between 0 and 1. TMLE does this automatically for the user. Additionally, the average treatment effect is estimate is back converted to the original scale of Y. When scaling Y as Y*, some values may take the value of 0 or 1, which breaks a logit(Y*) transformation. To avoid this issue, Y* is bounded by the *continuous_bound* argument. The default is 0.0005, the same as R's tmle

The following is a general outline of the estimation process for TMLE

1. Initial estimates for Y are predicted from a regression model. Expected values for each individual are generated under the scenarios of all treated vs all untreated

$$E(Y|A, L)$$

2. Predicted probabilities are generated from a regression model

$$\pi_1 = \Pr(A = 1|L)$$

3. The 'clever covariate' is calculated by

$$H_a(A = a, L) = \frac{I(A = 1)}{\pi_1} - \frac{I(A = 0)}{\pi_0}$$

for each individual. Afterwards, the predicted Y is set as an offset in the following logit model and used to predict values under each treatment strategy after fitted

$$\text{logit}(E(Y|A, L)) = \text{logit}(Y_a) + \sigma H_a$$

4. The targeted Psi is estimated, representing the causal effect of all treated vs. all untreated

Confidence intervals are constructed using influence curves.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **exposure** (*str*) – Column label for the exposure of interest
- **outcome** (*str*) – Column label for the outcome of interest
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05
- **continuous_bound** (*float, optional*) – Optional argument to control the bounding feature for continuous outcomes. The bounding process may result in values of 0,1 which are undefined for logit(x). This parameter adds or substracts from the scenarios of 0,1 respectively. Default value is 0.0005

Examples

Setting up environment

```
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.doublyrobust import TMLE
>>> df = load_sample_data(False).dropna()
>>> df[['cd4_rs1', 'cd4_rs2']] = spline(df, 'cd40', n_knots=3, term=2,␣
↪restricted=True)
```

Estimating TMLE using logistic regression

```
>>> tmle = TMLE(df, exposure='art', outcome='dead')
>>> # Specifying exposure/treatment model
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
>>> # Specifying outcome model
>>> tmle.outcome_model('art + male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
>>> # TMLE estimation procedure
>>> tmle.fit()
>>> # Printing main results
>>> tmle.summary()
>>> # Extracting risk difference and confidence intervals, respectively
>>> tmle.risk_difference
>>> tmle.risk_difference_ci
```

Estimating TMLE with machine learning algorithm from sklearn

```
>>> from sklearn.linear_model import LogisticRegression
>>> log1 = LogisticRegression(penalty='l1', random_state=201)
>>> tmle = TMLE(df, 'art', 'dead')
>>> # custom_model allows specification of machine learning algorithms
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0', custom_
↪model=log1)
>>> tmle.outcome_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0', custom_
↪model=log1)
>>> tmle.fit()
```

Demonstration of estimating g-model with symmetric bounds

```
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0', bound=0.
↪05)
```

Demonstration of estimating g-model with asymmetric bounds

```
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0', bound=[0.
↪05, 0.9])
```

References

Schuler MS, and Sherri R. "Targeted maximum likelihood estimation for causal inference in observational studies." American journal of epidemiology 185.1 (2017): 65-73.

Van der Laan, MJ, and Sherri R. Targeted learning: causal inference for observational and experimental data. Springer Science & Business Media, 2011.

Van Der Laan, MJ, Rubin D. "Targeted maximum likelihood learning." The International Journal of Biostatistics 2.1 (2006).

Gruber S, van der Laan, MJ. (2011). tmle: An R package for targeted maximum likelihood estimation.

__init__ (*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, alpha, ...])	Initialize self.
exposure_model(model[, custom_model, bound, ...])	Estimation of Pr(A=1|L), which is termed as g(A=1|L) in the literature
fit()	Calculate the effect measures from the predicted exposure probabilities and predicted outcome values using the TMLE procedure.
missing_model(model[, custom_model, bound, ...])	Estimation of Pr(M=1|A,L), which is the missing data mechanism for the outcome.
outcome_model(model[, custom_model, bound, ...])	Estimation of E(Y|A,L,M=1), which is also written sometimes as Q(A,W,M=1) or Pr(Y=1|A,W,M=1).
plot_kde(to_plot[, bw_method, fill, color, ...])	Generates density plots that can be used to check predictions qualitatively.
plot_love([color_unweighted, ...])	Generates a Love-plot to detail covariate balance based on the IPTW weights.
positivity([decimal])	Use this to assess whether positivity is a valid assumption for the exposure model / calculated IPTW.
run_diagnostics([decimal])	Run all currently implemented diagnostics for the exposure and outcome models.
standardized_mean_differences()	Calculates the standardized mean differences for all variables based on the inverse probability weights.
summary([decimal])	Prints summary of the estimated average causal effects

zepid.causal.doublyrobust.TMLE.StochasticTMLE

class zepid.causal.doublyrobust.TMLE.**StochasticTMLE**(*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*, *verbose=False*)

Implementation of target maximum likelihood estimator for stochastic treatment plans. This implementation calculates TMLE for a time-fixed exposure and a single time-point outcome under a stochastic treatment plan of interest. By default, standard parametric regression models are used to calculate the estimate of interest. The StochasticTMLE estimator allows users to instead use machine learning algorithms from sklearn and PyGAM.

Note: Valid confidence intervals are only attainable with certain machine learning algorithms. These algorithms must be Donsker class for valid confidence intervals. GAM and LASSO are examples of alogorithms that are Donsker class

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing the variables of interest
- **exposure** (*str*) – Column label for the exposure of interest
- **outcome** (*str*) – Column label for the outcome of interest
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05
- **continuous_bound** (*float, optional*) – Optional argument to control the bounding feature for continuous outcomes. The bounding process may result in values of 0,1 which are undefined for logit(x). This parameter adds or substracts from the scenarios of 0,1 respectively. Default value is 0.0005

- **verbose** (*bool, optional*) – Optional argument for verbose estimation. With verbose estimation, the model fits for each result are printed to the console. It is highly recommended to turn this parameter to True when conducting model diagnostics

---

Note: TMLE is a doubly-robust substitution estimator. TMLE obtains the target estimate in a single step. The single-step TMLE is described further by van der Laan. For further details, see the listed references.

Continuous outcomes must be bounded between 0 and 1. TMLE does this automatically for the user. Additionally, the average treatment effect is estimate is back converted to the original scale of Y. When scaling Y as Y*, some values may take the value of 0 or 1, which breaks a logit(Y*) transformation. To avoid this issue, Y* is bounded by the *continuous_bound* argument. The default is 0.0005, the same as R's tmle

Following is a general narrative of the estimation procedure for TMLE with stochastic treatments

1. Initial estimators for g-model (IPTW) and Q-model (g-formula) are fit. By default these estimators are based on parametric regression models. Additionally, machine learning algorithms can be used to estimate the g-model and Q-model.

2. The auxiliary covariate is calculated (i.e. IPTW).

$$H = \frac{p}{\widehat{\Pr}(A = a)}$$

where p is the probability of treatment a under the stochastic intervention of interest.

3. Targeting step occurs through estimation of e via a logistic regression model. Briefly a weighted logistic regression model (weighted by the auxiliary covariates) with the dependent variable as the observed outcome and an offset term of the Q-model predictions under the observed treatment (A).

$$\text{logit}(Y) = \text{logit}(Q(A, W)) + \epsilon$$

4. Stochastic interventions are evaluated through Monte Carlo integration for binary treatments. The different treatment plans are randomly applied and evaluated through the Q-model and then the targeting step via

$$E[\text{logit}(Q(A = a, W)) + \hat{\epsilon}]$$

This process is repeated a large number of times and the point estimate is the average of those individual treatment plans.

Examples

Setting up environment

```
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.doublyrobust import StochasticTMLE
>>> df = load_sample_data(False).dropna()
>>> df[['cd4_rs1', 'cd4_rs2']] = spline(df, 'cd40', n_knots=3, term=2,␣
↪restricted=True)
```

Estimating TMLE for 0.2 being treated with ART

```
>>> tmle = StochasticTMLE(df, exposure='art', outcome='dead')
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
>>> tmle.outcome_model('art + male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
>>> tmle.fit(p=0.2)
>>> tmle.summary()
```

Estimating TMLE for conditional plan

```
>>> tmle = StochasticTMLE(df, exposure='art', outcome='dead')
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
>>> tmle.outcome_model('art + male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
>>> tmle.fit(p=[0.6, 0.4], conditional=["df['male']==1", "df['male']==0"])
>>> tmle.summary()
```

Estimating TMLE with machine learning algorithm from sklearn

```
>>> from sklearn.linear_model import LogisticRegression
>>> log1 = LogisticRegression(penalty='l1', random_state=201)
>>> tmle = StochasticTMLE(df, 'art', 'dead')
>>> tmle.exposure_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0', custom_
↪model=log1)
>>> tmle.outcome_model('male + age0 + cd40 + cd4_rs1 + cd4_rs2 + dvl0', custom_
↪model=log1)
>>> tmle.fit(p=0.75)
```

References

Muñoz ID, and Van Der Laan MJ. Population intervention causal effects based on stochastic interventions. Biometrics 68.2 (2012): 541-549.

van der Laan MJ, and Sherri R. Targeted learning in data science: causal inference for complex longitudinal studies. Springer Science & Business Media, 2011.

__init__ (*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*, *verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, alpha, ...])	Initialize self.
est_conditional_variance(haw, y_obs, y_pred)	
est_marginal_variance(haw, y_obs, y_pred, ...)	
exposure_model(model[, custom_model, bound])	Estimation of Pr(A=1|L), which is termed as g(A=1|L) in the literature.
fit(p[, conditional, samples, seed])	Calculate the effect from the predicted exposure probabilities and predicted outcome values using the TMLE procedure.
outcome_model(model[, custom_model, bound, ...])	Estimation of E(Y|A,L), which is also written sometimes as Q(A,W) or Pr(Y=1|A,W).
run_diagnostics([decimal])	Provides some summary diagnostics for *StochasticTMLE*.
summary([decimal])	Prints summary of the estimated incidence under the specified treatment plan
targeting_step(y, q_init, iptw, verbose)	

SingleCrossfitTMLE(df, exposure, outcome[, ...])	Implementation of the Targeted Maximum Likelihood Estimator with a single cross-fit procedure.
DoubleCrossfitTMLE(df, exposure, outcome[, ...])	Implementation of the Targeted Maximum Likelihood Estimator with a double cross-fit procedure.

zepid.causal.doublyrobust.crossfit.SingleCrossfitTMLE

class zepid.causal.doublyrobust.crossfit.**SingleCrossfitTMLE**(*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*)

Implementation of the Targeted Maximum Likelihood Estimator with a single cross-fit procedure. The purpose of the cross-fit procedure is to all for non-Donsker nuisance function estimators. Some of machine learning algorithms are non-Donsker. In practice this means that confidence interval coverage can be incorrect when certain nuisance function estimators are used. Additionally, bias may persist as well. Cross-fitting is meant to alleviate this issue, therefore cross-fitting with a doubly-robust estimator is recommended when using machine learning.

SingleCrossfitTMLE uses a single cross-fit, where the data set is paritioned into at least two non-overlapping splits. The nuisance function estimators are then estimated in each split. The estimated nuisance functions are then used to predict values in a non-overlapping split. This decouple the nuisance function estimation from the data used to estimate it

---

Note: Because of the repetitions of the procedure are needed to reduce variance determined by a particular partition, it can take a long time to run this code.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all necessary variables
- **exposure** (*str*) – Label for treatment column in the pandas data frame
- **outcome** (*str*) – Label for outcome column in the pandas data frame
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05
- **continuous_bound** (*float, optional*) – Optional argument to control the bounding feature for continuous outcomes. The bounding process may result in values of 0,1 which are undefined for logit(x). This parameter adds or substracts from the scenarios of 0,1 respectively. Default value is 0.0005

Examples

Setting up environment

```
>>> from sklearn.linear_model import LogisticRegression
>>> from zepid import load_sample_data
>>> from zepid.causal.doublyrobust import SingleCrossfitTMLE
>>> df = load_sample_data(False).drop(columns='cd4_wk45').dropna()
```

Estimating the single cross-fit TMLE

```
>>> sctmle = SingleCrossfitTMLE(df, exposure='art', outcome='dead')
>>> sctmle.exposure_model("male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> sctmle.outcome_model("art + male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> sctmle.fit(n_splits=5, n_partitions=100)
>>> sctmle.summary()
```

References

Chernozhukov V, Chetverikov D, Demirer M, Duflo E, Hansen C, Newey W, & Robins J. (2018). "Double/debiased machine learning for treatment and structural parameters". The Econometrics Journal 21:1; pC1–C6

__init__(*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, alpha, ...])	Initialize self.
exposure_model(covariates, estimator[, bound])	Specify the treatment nuisance model variables and estimator(s) to use.
fit([n_splits, n_partitions, method, ...])	Runs the crossfit estimation procedure with the targeted maximum likelihood estimator.
outcome_model(covariates, estimator)	Specify the outcome nuisance model variables and estimator(s) to use.
run_diagnostics([color])	Runs available diagnostics for the plots.
summary([decimal])	Prints summary of model results

zepid.causal.doublyrobust.crossfit.DoubleCrossfitTMLE

class zepid.causal.doublyrobust.crossfit.**DoubleCrossfitTMLE**(*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*)

Implementation of the Targeted Maximum Likelihood Estimator with a double cross-fit procedure. The purpose of the cross-fit procedure is to all for non-Donsker nuisance function estimators. Some of machine learning algorithms are non-Donsker. In practice this means that confidence interval coverage can be incorrect when certain nuisance function estimators are used. Additionally, bias may persist as well. Cross-fitting is meant to alleviate this issue, therefore cross-fitting with a doubly-robust estimator is recommended when using machine learning.

DoubleCrossfitTMLE uses a double cross-fit, where the data set is paritioned into at least three non-overlapping split. The nuisance function estimators are then estimated in each split. The estimated nuisance functions are then used to predict values in a non-overlapping split. This decouple the nuisance function estimation from the data used to estimate it

Note: Because of the repetitions of the procedure are needed to reduce variance determined by a particular partition, it can take a long time to run this code.

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all necessary variables
- **exposure** (*str*) – Label for treatment column in the pandas data frame
- **outcome** (*str*) – Label for outcome column in the pandas data frame
- **alpha** (*float, optional*) – Alpha for confidence interval level. Default is 0.05
- **continuous_bound** (*float, optional*) – Optional argument to control the bounding feature for continuous outcomes. The bounding process may result in values of 0,1 which are undefined for logit(x). This parameter adds or substracts from the scenarios of 0,1 respectively. Default value is 0.0005

Examples

Setting up environment

```
>>> from sklearn.linear_model import LogisticRegression
>>> from zepid import load_sample_data
>>> from zepid.causal.doublyrobust import DoubleCrossfitTMLE
>>> df = load_sample_data(False).drop(columns='cd4_wk45').dropna()
```

Estimating the double cross-fit TMLE

```
>>> dctmle = DoubleCrossfitTMLE(df, exposure='art', outcome='dead')
>>> dctmle.exposure_model("male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> dctmle.outcome_model("art + male + age0 + cd40 + dvl0",␣
↪estimator=LogisticRegression(solver='lbfgs'))
>>> dctmle.fit(n_splits=5, n_partitions=100)
>>> dctmle.summary()
```

References

Zivich PN, & Breskin A. (2020). Machine learning for causal inference: on the use of cross-fit estimators. arXiv preprint arXiv:2004.10337.

Newey WK, Robins JR. (2018) "Cross-fitting and fast remainder rates for semiparametric estimation". arXiv:1801.09138

Chernozhukov V, Chetverikov D, Demirer M, Duflo E, Hansen C, Newey W, & Robins J. (2018). "Double/debiased machine learning for treatment and structural parameters". The Econometrics Journal 21:1; pC1–C6

__init__ (*df*, *exposure*, *outcome*, *alpha=0.05*, *continuous_bound=0.0005*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, alpha, ...])	Initialize self.
exposure_model(covariates, estimator[, bound])	Specify the treatment nuisance model variables and estimator(s) to use.

Continued on next page

Table 43 – continued from previous page

fit([n_splits, n_partitions, method, ...])	Runs the crossfit estimation procedure with the targeted maximum likelihood estimator.
outcome_model(covariates, estimator)	Specify the outcome nuisance model variables and estimator(s) to use.
run_diagnostics([color])	Runs available diagnostics for the plots.
summary([decimal])	Prints summary of model results

G-estimation of SNM

GEstimationSNM(df, exposure, outcome[, weights])	G-estimation for structural nested mean models.

zepid.causal.snm.g_estimation.GEstimationSNM

class zepid.causal.snm.g_estimation.**GEstimationSNM**(*df*, *exposure*, *outcome*, *weights=None*)

G-estimation for structural nested mean models. G-estimation is distinct from the other g-methods (inverse probability weights and g-formula) in the parameter it estimates. Rather than estimating the average causal effect of treating everyone versus treating no one, g-estimation estimates the average causal effect within strata of L. It does this by specifying a structural nested model. The structural nested mean model looks like the following for additive effects

$$E[Y^a|A=a,V] - E[Y^{a=0}|A=a,V] = \psi a + \psi a * V$$

There are two items to note in the structural nested model; (1) there is no intercept or term for L, and (2) we need the potential outcomes to solve for psi. The first item means that we are estimating fewer parameters, making g-estimation less susceptible to model misspecification than the g-formula. The second means we cannot solve the above equation directly.

Under the assumption of conditional exchangeability, we can solve for psi using another equation. Specifically, we can work to solve the following model

$$logit(\Pr(A=1|Y^{a=0},L)) = alpha + alphaY^{a=0} + alphaYa = 0V + alphaL$$

Under the assumption of conditional exchangeability, the alpha term for the potential outcome Y should be equal to zero! Therefore, we need to find the value of psi that results in that alpha term equaling zero. For the additive model, we can solve for psi in the first equation by

$$H(\psi) = Y - (\psi A + \psi AL)$$

meaning we solve for when alpha is approximately zero under

$$logit(\Pr(A=1|Y^{a=0},L)) = alpha + alphaH(\psi) + alphaH(\psi)V + alphaL$$

To find the values for the psi's where the alpha for those terms is approximately zero, we have two options; (1) grid-search or (2) closed form. The closed form is ultimately faster since we are only required to do some basic matrix manipulation to solve. For the grid search, we need to search across the potential values that minimize the values of alphas. We use SciPy's Nelder-Mead optimization procedure for the heavy lifting.

Parameters

- **df** (*DataFrame*) – Pandas DataFrame object containing all variables of interest
- **exposure** (*str*) – Column name of the exposure variable. Currently only binary is supported

- **outcome** (*str*) – Column name of the outcome variable. Either continuous or binary outcomes are supported
- **weights** – Column name of weights. Weights allow for items like sampling weights, missing weights, and censoring weights to estimate effects

Notes

Similar to marginal structural models, g-estimation cannot inherently account for missing at random data. To account for missing outcome data, inverse probability of missing weights should be used

The grid-search approach does allow for some unique sensitivity analyses that are not incorporated into the closed-form. Specifically, we can imagine that there is some unobserved confounding. With unobserved confounding, we know that the alpha value will not exactly equal zero. We can optimize for slightly different alphas to see how sensitive our results are to some assumptions regarding unobserved confounding. For further details on translating unobserved confounding to alpha values, see Scharfstein et al. 1999 in the references

If you continuous variable takes on large values, you may see the closed-form and grid-search start to diverge in results. This is because of the tolerance value. If you have large outcome values, I recommend rescaling them to prevent any issues with the grid-search

Examples

Set up the environment and the data set

```
>>> from zepid import load_sample_data, spline
>>> from zepid.causal.snm import GEstimationSNM
>>> df = load_sample_data(timevary=False).drop(columns=['dead'])
>>> df[['cd4_rs1','cd4_rs2']] = spline(df,'cd40',n_knots=3,term=2,restricted=True)
>>> df[['age_rs1','age_rs2']] = spline(df,'age0',n_knots=3,term=2,restricted=True)
```

One-parameter structural nested mean model via closed-form solution

```
>>> snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
>>> snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> snm.structural_nested_model(model='art')
>>> snm.fit()
>>> snm.summary()
```

One-parameter structural nested mean model via grid-search

```
>>> snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
>>> snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> snm.structural_nested_model(model='art')
>>> snm.fit(solver='search')
```

One-parameter structural nested mean model via grid-search with different alphas

```
>>> snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
>>> snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> snm.structural_nested_model(model='art')
>>> snm.fit(solver='search', alpha_value=0.03)
```

Two-parameter structural nested mean model via closed-form

```
>>> snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
>>> snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> snm.structural_nested_model(model='art + art:male')
>>> snm.fit()
```

Two-parameter structural nested mean model via grid-search and starting values

```
>>> snm = GEstimationSNM(df, exposure='art', outcome='cd4_wk45')
>>> snm.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_
↪rs2 + dvl0')
>>> snm.structural_nested_model(model='art + art:male')
>>> snm.fit(solver='search', starting_value=[-0.05, 0.0])
```

References

Naimi AI, Cole SR, Kennedy EH. (2017). An introduction to g methods. International journal of epidemiology, 46(2), 756-762.

Robins JM. (2000). Marginal structural models versus structural nested models as tools for causal inference. In Statistical models in epidemiology, the environment, and clinical trials (pp. 95-133). Springer, New York, NY.

Vansteelandt S, Joffe M. (2014). Structural nested models and G-estimation: the partially realized promise. Statistical Science, 29(4), 707-731.

Wallace MP, Moodie EE, Stephens DA. (2017). An R package for G-estimation of structural nested mean models. Epidemiology, 28(2), e18-e20.

Scharfstein DO, Rotnitzky A, Robins JM. (1999). Adjusting for nonignorable drop-out using semiparametric nonresponse models. Journal of the American Statistical Association, 94(448), 1096-1120.

__init__(*df*, *exposure*, *outcome*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome[, weights])	Initialize self.
exposure_model(model[, print_results])	Specify the treatment model to satisfy conditional exchangeability.
fit([solver, starting_value, alpha_value, ...])	Using the treatment model and the format of the structural nested mean model, the solutions for psi are calculated.
missing_model(model_denominator[, ...])	Estimation of Pr(M=0|A=a,L), which is the missing data mechanism for the outcome.
structural_nested_model(model)	Specify the structural nested mean model to fit.
summary([decimal])	Summary of results

Generalizability / Transportability

IPSW(df, exposure, outcome, selection[, ...])	Calculate inverse probability of sampling weights through logistic regression.

Continued on next page

Table 46 – continued from previous page

GTransportFormula(df, exposure, outcome, ...)	Calculate the g-transport-formula using a observed study sample and a sample from the target population.
AIPSW(df, exposure, outcome, selection[, ...])	Doubly robust estimator for generalizability.

zepid.causal.generalize.estimators.IPSW

class zepid.causal.generalize.estimators.**IPSW**(*df*, *exposure*, *outcome*, *selection*, *generalize=True*, *weights=None*)

Calculate inverse probability of sampling weights through logistic regression. Inverse probability of sampling weights are an extension of inverse probability weights to allow for the generalizability or the transportability of results.

For generalizability, inverse probability of sampling weights take the following form

$$IPSW = \frac{1}{\Pr(S=1|W)}$$

where *W* is all the factors related to the sample selection process

For transportability, the inverse probability of sampling weights are actually inverse odds of sampling weights. They take the following form

$$IPSW = \frac{\Pr(S=0|W}{\Pr(S=1|W)}$$

Confidence intervals should be obtained by using a non-parametric bootstrapping procedure

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all variables required for generalization/transportation. Should include all features related to sample selection, indicator for selection into the sample, and treatment/outcome information for the sample (selection == 1)
- **exposure** (*str*) – Column label for exposure/treatment of interest. Can be nan for all those not in sample
- **outcome** (*str*) – Column label for outcome of interest. Can be nan for all those not in sample
- **selection** (*str*) – Column label for indicator of selection into the sample. Should be 1 if individual comes from the study sample and 0 if individual is from random sample of source population
- **generalize** (*bool, optional*) – Whether the problem is a generalizability (True) problem or a transportability (False) problem. See notes for further details on the difference between the two estimation methods
- **weights** (*None, str, optional*) – For conditionally randomized trials, or observational research, inverse probability of treatment weights can be used to adjust for confounding. Before estimating the effect measures, this weight vector and the IPSW are multiplied to calculate new weights When weights is None, the data is assumed to come from a randomized trial, and does not need to be adjusted

Note: There are two related concepts; generalizability and transportability. Generalizability is when your study sample is part of your target population. For example, you want to generalize results from California to the entire United States. Transportability is when your study sample is not part of your target population. For example,

we want to apply our results from California to Canada. Depending on the scenario, the IPSW have slightly different forms. *IPSW* allows for both of these problems

Examples

Setting up the environment

```
>>> from zepid import load_generalize_data
>>> from zepid.causal.generalize import IPSW
>>> df = load_generalize_data(False)
```

Generalizability for RCT results

```
>>> ipsw = IPSW(df, exposure='A', outcome='Y', selection='S', generalize=True)
>>> ipsw.sampling_model('L + W + L:W', print_results=False)
>>> ipsw.fit()
>>> ipsw.summary()
```

Transportability for RCT results

```
>>> ipsw = IPSW(df, exposure='A', outcome='Y', selection='S', generalize=False)
>>> ipsw.sampling_model('L + W + L:W', print_results=False)
>>> ipsw.fit()
>>> ipsw.summary()
```

For observational studies, IPTW can be used to account for confounders via the *treatment_model()* function

```
>>> ipsw = IPSW(df, exposure='A', outcome='Y', selection='S')
>>> ipsw.sampling_model('L + W + L:W', print_results=False)
>>> ipsw.treatment_model('L', print_results=False)
>>> ipsw.fit()
>>> ipsw.summary()
```

References

Lesko CR, Buchanan AL, Westreich D, Edwards JK, Hudgens MG, & Cole SR. (2017). Generalizing study results: a potential outcomes perspective. Epidemiology (Cambridge, Mass.), 28(4), 553.

Westreich D, Edwards JK, Lesko CR, Stuart E, & Cole SR. (2017). Transportability of trial results using inverse odds of sampling weights. AJE, 186(8), 1010-1014.

Dahabreh IJ, Robertson SE, Stuart EA, Hernan MA (2018). Transporting inferences from a randomized trial to a new target population. arXiv preprint arXiv:1805.00550.

__init__ (*df*, *exposure*, *outcome*, *selection*, *generalize=True*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome, selection[, ...])	Initialize self.
fit()	Uses the calculated IPSW to obtain the risk difference and risk ratio from the sample.

Continued on next page

Table 47 – continued from previous page

sampling_model(model_denominator[, ...])	Logistic regression model(s) for estimating sampling weights.
summary([decimal])	Prints a summary of the results for the IPSW estimator
treatment_model(model_denominator[, ...])	Logistic regression model(s) for estimating inverse probability of treatment weights (IPTW).

zepid.causal.generalize.estimators.GTransportFormula

class zepid.causal.generalize.estimators.**GTransportFormula**(*df*, *exposure*, *outcome*, *selection*, *outcome_type='binary'*, *generalize=True*, *weights=None*)

Calculate the g-transport-formula using a observed study sample and a sample from the target population. Broadly, the process for fitting the g-transport-formula is similar to the g-formula (as implemented in *TimeFixedGFormula*). Instead of predicting the potential outcomes of only the sample, the g-transport-formula predicts potential outcomes for the full target population

For generalizability, we first fit a Q-model predicting the outcome as a function of the treatment and any modifiers (along with confounders if in observation data). Afterwards, we predict the potential outcomes for the entire population (S=1 and S=0). To obtain the marginal effect measure, we take the mean of the entire population (S=1 and S=0)

For transportability, we similarly fit a Q-model in the observed sample and generate predictions for the entire sample. However, for transportability our sample is not part of the target population. Therefore, we only take the marginal of the S=0 group.

Confidence intervals should be obtained by using a non-parametric bootstrapping procedure

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all variables required for generalization/transportation. Should include all features related to sample selection, indicator for selection into the sample, and treatment/outcome information for the sample (selection == 1)
- **exposure** (*str*) – Column label for exposure/treatment of interest. Can be nan for all those not in sample. Only binary exposures are currently supported
- **outcome** (*str*) – Column label for outcome of interest. Can be nan for all those not in sample
- **selection** (*str*) – Column label for indicator of selection into the sample. Should be 1 if individual comes from the study sample and 0 if individual is from random sample of source population
- **outcome_type** (*str, optional*) – Outcome variable type. Currently only 'binary', 'normal', and 'poisson variable types are supported
- **generalize** (*bool, optional*) – Whether the problem is a generalizability (True) problem or a transportability (False) problem. See notes for further details on the difference between the two estimation methods
- **weights** (*None, str, optional*) – Optional argument for weights. Can be used to input inverse probability of missing weights

Note: There are two related concepts; generalizability and transportability. Generalizability is when your study sample is part of your target population. For example, you want to generalize results from California to the entire United States. Transportability is when your study sample is not part of your target population. For example, we want to apply our results from California to Canada. Depending on the scenario, how the marginal risk difference is calculated is slightly different. *GTransportFormula* allows for both of these problems

Examples

Setting up the environment

```
>>> from zepid import load_generalize_data
>>> from zepid.causal.generalize import GTransportFormula
>>> df = load_generalize_data(False)
```

Generalizability

```
>>> gtf = GTransportFormula(df, exposure='A', outcome='Y', selection='S',␣
↪generalize=True)
>>> gtf.outcome_model('A + L + L:A + W + W:A + W:A:L')
>>> gtf.fit()
>>> gtf.summary()
```

Transportability

```
>>> gtf = GTransportFormula(df, exposure='A', outcome='Y', selection='S',␣
↪generalize=False)
>>> gtf.outcome_model('A + L + L:A + W + W:A + W:A:L')
>>> gtf.fit()
>>> gtf.summary()
```

For observational studies, confounders should be included in the Q-model

References

Lesko CR, Buchanan AL, Westreich D, Edwards JK, Hudgens MG, & Cole SR. (2017). Generalizing study results: a potential outcomes perspective. Epidemiology (Cambridge, Mass.), 28(4), 553.

Dahabreh IJ, Robertson SE, Stuart EA, Hernan MA (2018). Transporting inferences from a randomized trial to a new target population. arXiv preprint arXiv:1805.00550.

__init__(*df*, *exposure*, *outcome*, *selection*, *outcome_type='binary'*, *generalize=True*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome, selection[, ...])	Initialize self.
fit()	Uses the g-transport formula to obtain the risk difference and risk ratio from the sample.
outcome_model(model[, print_results])	Build the model for the outcome.
summary([decimal])	Prints a summary of the results for the g-transport estimator

zepid.causal.generalize.estimators.AIPSW

class zepid.causal.generalize.estimators.**AIPSW**(*df*, *exposure*, *outcome*, *selection*, *generalize=True*, *weights=None*)

Doubly robust estimator for generalizability. I haven't found a good name for it in the literature yet, so I am naming it augmented-IPSW (in honor of other doubly robust estimators like AIPTW and AIPMW).

The process of estimating AIPSW follows other doubly robust estimators. We need to specify both the IPSW model and the g-transport model. From this information, the AIPSW is calculated via the following

$$\psi = \frac{1}{n}\sum\left(E[Y|A=a,L,S=1] + \frac{I(S=1,A=a)}{\Pr(S=1|L)}(Y - E[Y|A=a,L,S=1])\right)$$

For transportability problems, AIPSW takes the following form

$$\psi = \frac{\sum IPSW \times I(S=1,A=a)(Y - E[Y|A=a,L,S=1]) + (1-S)E[Y|A=a,L,S=1]}{\Pr(S=0)}$$

For generalizability, we first fit a Q-model predicting the outcome as a function of the treatment and any modifiers (along with confounders if in observation data). Next we calculate IPSW (with IPTW if there is any confounders). Afterwards, we predict the potential outcomes for the entire population (S=1 and S=0). We then use the above formula to calculate the marginal effect

A similar process is done for transportability. Instead we merge g-transport and inverse odds of sampling weights

Confidence intervals should be obtained by using a non-parametric bootstrapping procedure

Parameters

- **df** (*DataFrame*) – Pandas dataframe containing all variables required for generalization/transportation. Should include all features related to sample selection, indicator for selection into the sample, and treatment/outcome information for the sample (selection == 1)
- **exposure** (*str*) – Column label for exposure/treatment of interest. Can be nan for all those not in sample. Only binary exposures are currently supported
- **outcome** (*str*) – Column label for outcome of interest. Can be nan for all those not in sample
- **selection** (*str*) – Column label for indicator of selection into the sample. Should be 1 if individual comes from the study sample and 0 if individual is from random sample of source population
- **generalize** (*bool, optional*) – Whether the problem is a generalizability (True) problem or a transportability (False) problem. See notes for further details on the difference between the two estimation methods
- **weights** (*None, str, optional*) – Optional argument for weights. Can be used to input inverse probability of missing weights

---

Note: There are two related concepts; generalizability and transportability. Generalizability is when your study sample is part of your target population. For example, you want to generalize results from California to the entire United States. Transportability is when your study sample is not part of your target population. For example, we want to apply our results from California to Canada. Depending on the scenario, how the marginal risk difference is calculated is slightly different. *AIPSW* allows for both of these problems

---

Examples

Setting up the environment

```
>>> from zepid import load_generalize_data
>>> from zepid.causal.generalize import AIPSW
>>> df = load_generalize_data(False)
```

Generalizability for RCT

```
>>> aipw = AIPSW(df, exposure='A', outcome='Y', selection='S', generalize=True)
>>> aipw.sampling_model('L + W_sq', print_results=False)
>>> aipw.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
>>> aipw.fit()
>>> aipw.summary()
```

Transportability for RCT

```
>>> aipw = AIPSW(df, exposure='A', outcome='Y', selection='S', generalize=False)
>>> aipw.sampling_model('L + W_sq', print_results=False)
>>> aipw.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
>>> aipw.fit()
>>> aipw.summary()
```

Transportability for Observational study

```
>>> df = load_generalize_data(True)
>>> aipw = AIPSW(df, exposure='A', outcome='Y', selection='S', generalize=False)
>>> aipw.sampling_model('L + W_sq', print_results=False)
>>> aipw.treatment_model('L', print_results=False)
>>> aipw.outcome_model('A + L + L:A + W + W:A + W:A:L', print_results=False)
>>> aipw.fit()
>>> aipw.summary()
```

References

Dahabreh IJ, Robertson SE, Stuart EA, Hernan MA (2018). Transporting inferences from a randomized trial to a new target population. arXiv preprint arXiv:1805.00550.

Dahabreh IJ, Hernan MA, Robertson SE, Buchanan A, Steingrimsson JA. (2019). Generalizing trial findings in nested trial designs with sub-sampling of non-randomized individuals. arXiv preprint arXiv:1902.06080.

__init__(*df*, *exposure*, *outcome*, *selection*, *generalize=True*, *weights=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df, exposure, outcome, selection[, ...])	Initialize self.
fit()	Uses AIPSW formula to obtain the risk difference and risk ratio from the sample.
outcome_model(model[, outcome_type, ...])	Build the g-transport model for the outcome.
sampling_model(model_denominator[, ...])	Logistic regression model(s) for estimating IPSW.
summary([decimal])	Prints a summary of the results for the AIPSW estimator

Continued on next page

Table 49 – continued from previous page

treatment_model(model_denominator[, ...])	Logistic regression model(s) for estimating inverse probability of treatment weights (IPTW).

1.8.5 Super Learner

Details for super learner and associated candidate estimators within zEpid.

Super Learners

SuperLearner(estimators, estimator_labels[, ...])	*SuperLearner* is an implementation of the super learner algorithm, which is a generalized stacking algorithm.

Candidate Estimators

EmpiricalMeanSL()	Empirical mean estimator in the format of SciKit learn.
GLMSL(family[, verbose])	Generalized Linear Model for use with SuperLearner.
StepwiseSL(family[, selection, ...])	Step-wise model selection for Generalized Linear Model selection for use with SuperLearner.

1.8.6 Sensitivity analyses

Details for sensitivity analysis tools implemented within zEpid.

Distributions

trapezoidal(mini, mode1, mode2, maxi[, size])	Generates random data following a trapezoidal distribution.

Sensitivity analyzers

MonteCarloRR(observed_RR[, sd, sample])	Monte Carlo simulation to assess the impact of an unmeasured binary confounder on the results of a study.

zepid.sensitivity_analysis.distributions.**trapezoidal**(*mini*, *mode1*, *mode2*, *maxi*, *size=None*)

Generates random data following a trapezoidal distribution. This function can be used to generate distributions of probabilities and effect measures for sensitivity analyses. It is particularly useful when used in conjunction with *rr_corr* to determine a distribution of potential results due to a single unadjusted confounder

Parameters

- **mini** (*float*) – Minimum value of trapezoidal distribution
- **mode1** (*float*) – Start of uniform distribution
- **mode2** (*float*) – End of uniform distribution
- **maxi** (*float*) – Maximum value of trapezoidal distribution

- **size** (*int, optional*) – Number of observations to generate. Default is None, which returns a single draw

Returns Returns either a single float from the distribution or an array of floats

Return type float or array

Examples

Single draw from a trapezoidal distribution

>>>from zepid.sensitivity_analysis import trapezoidal >>>trapezoidal(mini=0.2, mode1=0.3, mode2=0.5, maxi=0.6)

100 draws from a trapezoidal distribution

>>>trapezoidal(mini=0.2, mode1=0.3, mode2=0.5, maxi=0.6, size=100)

References

Fox MP, Lash TL, Hamer DH. (2005). A sensitivity analysis of a randomized controlled trial of zinc in treatment of falciparum malaria in children. Contemporary clinical trials, 26(3), 281-289.

Fox MP, Lash TL Greenland S. (2005). A method to automate probabilistic sensitivity analyses of misclassified binary variables. International journal of epidemiology, 34(6), 1370-1376.

class zepid.sensitivity_analysis.Simple.**MonteCarloRR**(*observed_RR*, *sd=None*, *sample=10000*)

Monte Carlo simulation to assess the impact of an unmeasured binary confounder on the results of a study. Observed RR comes from the data analysis, while the RR between the unmeasured confounder and the outcome should be obtained from prior literature or constitute an reasonable guess. Probability of exposure between the groups should also be reasonable numbers.

The Monte Carlo corrected Risk Ratio is calculated in each iteration by

$$RR_{MC} = \frac{RR_{obs}}{\frac{p_1(RR_c-1)+1}{p_0(RR_c-1)+1}}$$

Parameters

- **observed_RR** (*float*) – Observed RR from the data, not accounting for some binary unmeasured confounder
- **sd** (*float, optional*) – Standard deviation of the observed log(risk ratio). This parameter is optional. If specified, then random error is incorporated into the bias analysis estimates
- **sample** (*integer, optional*) – Number of MC simulations to run. It is important that the specified size of later distributions matches this number of samples

Examples

Monte Carlo bias analysis with trapezoidal distributions

```
>>> from zepid.sensitivity_analysis import MonteCarloRR, trapezoidal
>>> mcrr = MonteCarloRR(observed_RR=0.73322, sample=10000)
>>> mcrr.confounder_RR_distribution(trapezoidal(mini=0.9, mode1=1.1, mode2=1.7,␣
↪maxi=1.8, size=10000))
```

(continues on next page)

(continued from previous page)

```
>>> mcrr.prop_confounder_exposed(trapezoidal(mini=0.25, mode1=0.28, mode2=0.32,␣
↪maxi=0.35, size=10000))
>>> mcrr.prop_confounder_unexposed(trapezoidal(mini=0.55, mode1=0.58, mode2=0.62,␣
↪maxi=0.65, size=10000))
>>> mcrr.fit()
```

Printing a summarization of the bias analysis to the console

```
>>> mcrr.summary()
```

Creating a density plot of the bias analysis results

```
>>> import matplotlib.pyplot as plt
>>> mcrr.plot()
>>> plt.show()
```

confounder_RR_distribution(*dist*, *seed=None*)

Distribution of the risk ratio between the unmeasured confounder and the outcome. This value should come from prior literature or a reasonable guess. Any numpy random distribution can be based to this function. Alternatively, the trapezoid distribution within this library can also be used

Parameters

- **dist** – Distribution from which the confounder-outcome Risk Ratio is pulled from. Input should be something like *numpy.random.triangular(left=0.9,mode=1.2,right=1.6)* or *zepid.sensitivity_analysis.trapezoidal*
- **seed** (*int, optional*) – NumPy seed for the generated distribution. Default is None

fit()

After the observed Risk Ratio, distribution of the confounder-outcome Risk Ratio, proportion of the unmeasured confounder in exposed, proportion of the unmeasured confounder in the unexposed.

$$RR* = RR/dd = (p1 * (RRc - 1) + 1)/(p0 * (RRc - 1) + 1)$$

Where RR* is the corrected risk ratio, RR is the observed risk ratio in the data set, RRc is the risk ratio between unmeasured confounder and outcome, p1 is the probability/proportion of unmeasured confounder in exposed, and p0 is the probability/proportion of unmeasured confounder in unexposed

plot(*bw_method='scott'*, *fill=True*, *color='b'*)

Generate a Gaussian kernel density plot of the corrected risk ratio distribution. The kernel density used is SciPy's Gaussian kernel. Either Scott's Rule or Silverman's Rule can be implemented

Parameters

- **bw_method** (*str, optional*) – Method used to estimate the bandwidth. Following SciPy, either 'scott' or 'silverman' are valid options
- **fill** (*bool, optional*) – Whether to color the area under the density curves. Default is true
- **color** (*str, optional*) – Color of the line/area for the treated group. Default is Blue

Returns

Return type matplotlib axes

prop_confounder_exposed(*dist*, *seed=None*)

Distribution of the proportion of the unmeasured confounder in the exposed group. This value should come from prior literature or a reasonable guess. Any numpy random distribution can be based to this function. Alternatively, the trapezoid distribution within this library can also be used

Parameters

- **dist** – Distribution from which the confounder-exposure probability is pulled from. Input should be something like *numpy.random.triangular(left=0.9,mode=1.2,right=1.6)* or *zepid.sensitivity_analysis.trapezoidal*
- **seed** (*int, optional*) – NumPy seed for the generated distribution. Default is None

prop_confounder_unexposed(*dist*, *seed=None*)

Distribution of the proportion of the unmeasured confounder in the unexposed group. This value should come from prior literature or a reasonable guess. Any numpy random distribution can be based to this function. Alternatively, the trapezoid distribution within this library can also be used

Parameters

- **dist** – Distribution from which the confounder-no exposure probability is pulled from. Input should be something like *numpy.random.triangular(left=0.9,mode=1.2,right=1.6)* or *zepid.sensitivity_analysis.trapezoidal*
- **seed** (*int, optional*) – NumPy seed for the generated distribution. Default is None

summary(*decimal=3*)

Generate the summary information after the corrected risk ratio distribution is generated. fit() must be run before this

Parameters **decimal** (*int, optional*) – Decimal places to display in output. Default is 3

1.8.7 Data sets

Descriptions of the data sets included within zEpid

load_sample_data(timevary)	Load data that is part of the zepid package.
load_ewing_sarcoma_data()	Loads Ewing's Sarcoma survival data from Glaubiger DL, Makuch R, Schwarz J, Levine AS, Johnson RE.
load_gvhd_data()	Loads bone marrow transplant recipient data from Keil AP, Edwards JK, Richardson DB, Naimi AI, Cole SR.
load_sciatica_data()	Loads the Sciatica Trial data published in; Mertens, BJA, Jacobs, WCH, Brand, R, and Peul, WC.
load_leukemia_data()	Loads data from Freireich EJ et al., "The Effect of 6-Mercaptopurine on the Duration of Steriod-induced Remissions in Acute Leukemia: A Model for Evaluation of Other Potentially Useful Therapy" Blood 1963
load_longitudinal_data()	Loads simulated longitudinal data.
load_binge_drinking_data()	Loads data from Ahren J et al., "Predicting the Population Health Impacts of Community Interventions: The Case of Alcohol Outlets and Binge Drinking" AJPH 2016.

zepid.datasets.load_sample_data

zepid.datasets.**load_sample_data**(*timevary*)

Load data that is part of the zepid package. This data set comes from simulated data from Jessie Edwards (thanks Jess!). This data is used for examples on zepid.readthedocs

Parameters **timevary** (*bool*) – Whether to return the time-varying data set or the time fixed. If True then returns data set with repeated visits. If False then a data set with single observation

per subject representing the 45-week risk is returned

Notes

For the time-varying data set, the following variables are returned;

- id - participant unique ID
- enter - start of follow-up period
- out - end of time period
- male - indicator variable for male (1 = yes)
- age0 - age at enter = 0
- cd40 - CD4 T cell count at enter = 0
- dvl0 - detectable viral load data at enter = 0
- cd4 - CD4 T cell count at enter = t
- dvl - viral load at enter = t
- art - indicator of whether ART was prescribed at enter = t
- drop - indicator of whether individual dropped out of the study at enter = t (1 = yes)
- dead - indicator for death at out = t (1 = yes)

For the time-fixed data set, the following variables are returned

- id - participant unique ID
- male - indicator variable for male (1 = yes)
- age0 - age at enter = 0
- cd40 - CD4 T cell count at enter = 0
- dvl0 - detectable viral load data at enter = 0
- art - indicator of whether ART was prescribed at enter = 0
- t - total time contributed

Returns Returns either a time-varying or time-fixed pandas DataFrame

Return type DataFrame

Examples

Load the time-fixed exposure data set

```
>>> from zepid import load_sample_data
>>> load_sample_data(timevary=False)
```

Load the time-varying exposure data set

```
>>> load_sample_data(timevary=True)
```

zepid.datasets.load_ewing_sarcoma_data

zepid.datasets.**load_ewing_sarcoma_data**()

Loads Ewing's Sarcoma survival data from Glaubiger DL, Makuch R, Schwarz J, Levine AS, Johnson RE. Determination of prognostic factors and their influence on therapeutic results in patients with Ewing's sarcoma. Cancer. 1980;45(8):2213-9. In total, data for 76 patients is loaded. Further descriptions of the 76 patients and the data can be found in Makuch RW. Adjusted survival curve estimation using covariates. J Chronic Dis. 1982;35(6):437-43. or Cole SR, Hernán MA. Adjusted survival curves with inverse probability weights. Comput Methods Programs Biomed. 2004;75(1):45-9.

Notes

Variables within the dataset are

- treat - treatment (1 is novel treatment; 0 is one of three standard treatments)
- ldh - pre-treatment serum lactic acid dehydrogenase (LDH) (1 is >= 200 international units; 0 is <200)
- time - days till recurrence or censoring (continuous)
- outcome - sarcoma recurrence (1 is recurrence; 0 is censored)

Returns Returns pandas DataFrame

Return type DataFrame

zepid.datasets.load_gvhd_data

zepid.datasets.**load_gvhd_data**()

Loads bone marrow transplant recipient data from Keil AP, Edwards JK, Richardson DB, Naimi AI, Cole SR. The parametric g-formula for time-to-event data: intuition and a worked example. Epidemiology. 2014;25(6):889-97. Patients were followed until death or administrative censoring at 5-years.

Notes

Variables are formatted exactly as described in Keil et al. 2014

- id: unique ID for each participant
- age: participant baseline age
- agesq: squared baseline age
- agecurs1: restricted cubic spline knot 1 for baseline age
- agecurs2: restricted cubic spline knot 2 for basline age
- male: participant gender (1 is male, 0 is female)
- cmv: cytomegalovirus baseline immune status (1 is yes, 0 is no)
- all: at this time, I am unsure what this variable indicates (1, 0)
- wait: wait time from diagnosis to transplantation (months)
- day: day since transplantation
- daysq: squared day since transplantation
- daycu: cubic day since transplantation

- daycurs1: restricted cubic spline knot 1 for days since transplantation
- daycurs2: restricted cubic spline knot 2 for days since transplantation
- yesterday: previous day
- tomorrow: day after
- gvhd: indicator for Graph-versus-Host Disease (1 is yes, 0 is no)
- d: indicator of death (1 is yes, 0 is no)
- relapse: indicator for relapse (1 is yes, 0 is no)
- platnorm: indicator for normal platelet count (1 is yes, 0 is no)
- censlost: indicator for censoring due to loss-to-follow-up (1 is yes, 0 is no)
- gvhdm1: indicator for previous day diagnosis of GvHD (1 is yes, 0 is no)
- relapsem1: indicator for previous day relapse (1 is yes, 0 is no)
- platnormm1: indicator for previous day normal platelet count (1 is yes, 0 is no)
- daysnogvhd: number of consecutive days without a GvHD diagnosis
- daysnorelapse: number of consecutive days without relapse
- daysnoplatnorm: number of consecutive days without normal platelet count
- daysgvhd: number of consecutive days with GvHD
- daysrelapse: number of consecutive days after relapse
- daysplatnorm: number of consecutive days with normal platelet count

Returns Returns pandas DataFrame

Return type DataFrame

zepid.datasets.load_sciatica_data

zepid.datasets.**load_sciatica_data**()

Loads the Sciatica Trial data published in; Mertens, BJA, Jacobs, WCH, Brand, R, and Peul, WC. Assessment of patient-specific surgery effect based on weighted estimation and propensity scoring in the re-analysis of the Sciatica Trial. PLOS One 2014. Details of the original Sciatica Trial are available in; Peul WC, van Houwelingen HC, van den Hout WB, et al. Surgery versus Prolonged Conservative Treatment for Sciatica. NEJM 2007 DOI: 10.1177/0962280214545529

Notes

Variables included are

- id: unique identifier for patient
- tpoints: follow-up time period
- time: follow-up time
- age_b: age at follow-up time
- age_t: age at randomization
- vas1_t: VAS score

- vas2_t: VAS score
- roland_t: Roland score
- likert_t: Likert score
- vas1_b: VAS1 at baseline
- vas2_b: VAS2 at baseline
- roland_b: Roland score at baseline
- likert_b: Likert score at baseline
- male: participant gender (1 is male, 0 is female)
- weight: participant weight in kilograms
- height: participant height in meters
- surgery: whether participant received the surgery (1 is surgery, 0 is not yet surgery)

Returns Returns pandas DataFrame

Return type DataFrame

zepid.datasets.load_leukemia_data

zepid.datasets.**load_leukemia_data**()

Loads data from Freireich EJ et al., "The Effect of 6-Mercaptopurine on the Duration of Steriod-induced Remissions in Acute Leukemia: A Model for Evaluation of Other Potentially Useful Therapy" Blood 1963

Notes

Variables included are t: time status: event indicator (0: censored, 1: relapsed) sex: male, female logwbc: log-transformed white blood cell count treat: treatment indicator

Returns Returns pandas DataFrame

Return type DataFrame

zepid.datasets.load_longitudinal_data

zepid.datasets.**load_longitudinal_data**()

Loads simulated longitudinal data. This longitudinal data is used to demonstrate the sequential regression time-varying g-formula, and the longitudinal targeted maximum likelihood estimator. The format of the returned file is a long data set

Notes

Variables included are

- A: treatment of interest
- Y: outcome of interest
- t: time-point
- W: baseline variable

- L: time-varying variable
- id: unique identifier for each subject

Returns Returns pandas DataFrame

Return type DataFrame

zepid.datasets.load_binge_drinking_data

zepid.datasets.**load_binge_drinking_data**()

Loads data from Ahren J et al., "Predicting the Population Health Impacts of Community Interventions: The Case of Alcohol Outlets and Binge Drinking" AJPH 2016. Below is some notes taken from the supplementary materials detailed by the paper authors;

"The data provided for use with this sample code are simulated. The data are designed to be similar to the real data and associations examined in the main paper. There are 4000 observations representing individuals who are nested in 44 communities (variable name: neighborhood_id). The exposure of interest is neighborhood alcohol outlet density (alc_outlet_density), with values ranging from 39 to 168. The outcome of interest is a binary indicator of binge drinking (binge_drink), and covariates include gender (male), age (age_categorical), marital status (married), education (education_categorical), and race/ethnicity (race_categorical). Alcohol outlet density and binge drinking were simulated as simple linear functions of the covariates. Thus, unlike the applied example, the relation of outlet density with binge drinking has a linear shape."

Notes

Variables included are

- male: gender (0: female, 1: male)
- age_categorical: age groups (not clearly defined as to what they refer to)
- married: marital status
- education_categorical: categories of education levels
- race_categorical: categories of race
- alc_outlet_density: density of alcohol outlets in the neighborhood (continuous)
- binge_drink: whether individual binge drinks (1: yes, 0: no)
- neighborhood_id: identifier for groups that individuals are nested in

Returns Returns pandas DataFrame

Return type DataFrame

CHAPTER 2

Installation:

Dependencies are from the typical Python data-stack: Numpy, Pandas, Scipy, Statsmodels, and Matplotlib. Addtionally, it requires Tabulate, so nice looking tables can be easily generated. Install using:

```
pip install zepid
```

CHAPTER 3

Source code and Issue Tracker

Available on Github pzivich/zepid Please report bugs, issues, and feature extensions there.

Also feel free to contact us via Gitter email (gmail: zepidpy) or on Twitter (@PausalZ)

Python Module Index

z

Index

Symbols

A

C

D

T

Z