zEpid

Dec 30, 2020
## Contents

1 Contents: ................................................................. 3  
   1.1 Causal Graphs .................................................. 3  
      1.1.1 Directed Acyclic Graphs .................................. 3  
   1.2 Time-Fixed Exposure ........................................... 6  
      1.2.1 Which estimator should I use? ............................ 6  
      1.2.2 Binary Outcome .......................................... 7  
      1.2.3 Continuous Outcome ..................................... 13  
      1.2.4 Causal Survival Analysis ............................... 17  
   1.3 Time-Varying Exposure ......................................... 20  
      1.3.1 Time-to-Event Data ....................................... 20  
      1.3.2 Longitudinal Data ....................................... 27  
      1.3.3 Summary ................................................ 29  
   1.4 Generalizability ............................................... 29  
      1.4.1 Generalizability ......................................... 29  
      1.4.2 Transportability ......................................... 33  
      1.4.3 Summary ................................................ 34  
   1.5 Missing Data ................................................... 34  
      1.5.1 IPMW ..................................................... 34  
      1.5.2 AIPMW ................................................... 36  
      1.5.3 IPCW ..................................................... 36  
      1.5.4 Summary ................................................ 37  
   1.6 Graphics ........................................................ 37  
      1.6.1 Functional Form Assessment ............................... 37  
      1.6.2 P-value Plot ............................................. 43  
      1.6.3 Spaghetti Plot .......................................... 44  
      1.6.4 Effect Measure Plots ..................................... 45  
      1.6.5 Receiver-Operator Curves ............................... 46  
      1.6.6 Dynamic Risk Plots ..................................... 48  
      1.6.7 L’Abbe Plots ............................................. 50  
      1.6.8 Zipper Plot .............................................. 52  
   1.7 Sensitivity Analyses ........................................... 53  
      1.7.1 Trapezoidal Distribution ................................ 53  
      1.7.2 Monte Carlo Risk Ratio ................................ 54  
   1.8 Reference ......................................................... 55  
      1.8.1 Measures ................................................ 56  
      1.8.2 Calculations ............................................. 56
1.8.3 Graphics .......................................................... 57
1.8.4 Causal .......................................................... 57
1.8.5 Super Learner .................................................. 59
1.8.6 Sensitivity analyses .......................................... 59
1.8.7 Data sets ......................................................... 59

2 Installation: .......................................................... 61

3 Source code and Issue Tracker ................................. 63
**zEpid** is a Python 3.5+ epidemiology analysis toolkit. 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.
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 are supported by single-world intervention graphs will be added.

Note that this branch requires installation of NetworkX since that library is used to analyses 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()
pd.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).

```python
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

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

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.
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

```python
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.

\[ ACE = \Pr(Y^{a=1}) - \Pr(Y^{a=0}) \]

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

\[ \Pr(Y^{a=1}) - \Pr(Y^{a=0}) \]

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 \textit{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

```python
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 is no confounding variables. However, this is an unreasonable assumption for our observational data. However, the \textit{RiskDifference} gives us some useful information. In the summary, we find \textit{LowerBound} and \textit{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
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.

```python
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 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.

```python
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 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

```r
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

```r
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
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 my 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 is 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 +...
                       ,
                       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 +...
                     ,
                     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 data is no 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.

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

# Specify treatment model
smn.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dv10 -')
```

(continues on next 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 \text{i} \text{ 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 complete 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

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

```python
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 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.

```python
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

```python
iaipw = AIPTW(df, exposure='art', outcome='cd4_wk45')
iaipw.exposure_model('male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0 →')
iaipw.outcome_model('art + male + age0 + age_rs1 + age_rs2 + cd40 + cd4_rs1 + cd4_rs2 + dvl0')
iaipw.fit()
iaipw.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.

```python
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 SuperLearner with a GLM, a stepwise selection, and a random forest.

```python
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
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)
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 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.

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

Overall, the SNM results are similar to the other models (ATE = 227.2). Instead, we interpret this estimate as: the mean 45-week CD4 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

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 compare 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:

```python
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)
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:

```python
# 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)

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, \textit{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 \textit{Time-Varying Treatment} tutorial for further discussion.
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 allow 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:

```python
import numpy as np
import pandas as pd
from lifelines import KaplanMeierFitter
from zepid import load_sample_data, spline
```
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  # cd4 baseline cubic
df['cd4_sq'] = df['cd4'] ** 2  # cd4 current cubic
df['cd4_cu'] = df['cd4'] ** 3  # cd4 current cubic
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

```python
mcgf = MonteCarloGFormula(df,
    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

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

(continues on next page)
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

```r
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.

```r
# Accessing predicted outcome values
gf = mcgf.predicted_outcomes
```
From this we can see that our natural course predictions (green) follow the observed data pretty well (black). 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.

1.3. Time-Varying Exposure
```python
# 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 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 previously 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 + dv10 + cd40 + ' 'cd40_q + cd40_c + dv1 + cd4 + cd4_q + cd4_c')

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

(continues on next 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.

```python
tm = KaplanMeierFitter()
dfe = df.loc[df['art']==1].copy()
tm.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(tm.event_table.index, 1 - tm.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:

```python
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 $t=3$.

```r
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

```r
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

```r
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 $Y_2$. Below is an example of estimating $Y$ at $t=2$ for a treat-all strategy

```r
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).

```python
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

```python
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 \( \text{RD}=0.10, \text{RR}=1.38 \). 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

```python
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

```python
gtf = GTransportFormula(df, exposure='A', outcome='Y', selection='S', generalize=True)
```

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

```python
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.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

---

**1.4. Generalizability**
confounder on at least 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

```python
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

```python
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

```r
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

```r
gtf = GTransportFormula(df, exposure='A', outcome='Y', selection='S', generalize=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

```r
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)
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

```r
aipw.fit()
aipw.summary()
```
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 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:

```python
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)

```python
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

```python
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

```python
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.

```python
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

```python
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

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

```python
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)
```

```python
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.

```python
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](https://github.com/pzivich/Python-for-Epidemiologists/blob/master/3_Epidemiology_Analysis/b_missing_data/4_IPCW.ipynb) 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.

```python
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
No. Iterations:          5      Covariance Type:  nonrobust
==============================================================================
                      coef    std err          z     P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept             -3.627    0.537      -6.760    0.000     -4.679      -2.575
age0                  0.0507    0.013       4.012    0.000       0.026       0.075
==============================================================================
AIC: 482.5073872152573
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 $f_{\text{form}}$ argument must be supplied. Note that any terms specified in the $f_{\text{form}}$ argument must be part of the data set. We can assess a quadratic functional form like the following:

```python
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.

```python
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), then data is binned into categories automatically. The number of categories is determined via the maximum value minus the minimum divided by 5.

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

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.

```python
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

```python
from zepid.graphics import pvalue_plot

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

![P-value Plot](image)

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.
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.

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

measure = [np.nan, 0.94, np.nan, np.nan, 1.22, np.nan, np.nan, 0.59, np.nan, np.nan, np.nan, 1.09]
```

(continues on next 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

```python
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 changes 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)
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).
The log-transformed risk ratio is also available

```python
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 used 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.

```python
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:

```python
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.

```python
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).

```python
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.
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
```
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
RiskDifference
NNT
OddsRatio
IncidenceRateRatio
IncidenceRateDifference
interaction_contrast
interaction_contrast_ratio

Diagnostics

Sensitivity
Specificity

Diagnostics

Others

spline
create_spline_transform
table1_generator

1.8.2 Calculations

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

Measures

risk_ci
incidence_rate_ci
risk_ratio
risk_difference
number_needed_to_treat
odds_ratio
incidence_rate_ratio
incidence_rate_difference
attributable_community_risk
population_attributable_fraction
Diagnostics

sensitivity
specificity
ppv_converter
npv_converter
screening_cost_analyzer

Others

probability_to_odds
odds_to_probability
counternull_pvalue
semibayes
rubins_rules
s_value

1.8.3 Graphics

Below is documentation for each of the implemented graphic generators.

Data Diagnostics

functional_form_plot
spaghetti_plot
roc

Displaying Results

EffectMeasurePlot
pvalue_plot
dynamic_risk_plot
labbe_plot
zipper_plot

1.8.4 Causal

Documentation for each of the causal inference methods implemented in zEpid

Causal Diagrams

DirectedAcyclicGraph

Inverse Probability Weights
<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPTW</td>
</tr>
<tr>
<td>StochasticIPTW</td>
</tr>
<tr>
<td>IPMW</td>
</tr>
<tr>
<td>IPCW</td>
</tr>
</tbody>
</table>

**Time-Fixed Treatment G-Formula**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeFixedGFormula</td>
</tr>
<tr>
<td>SurvivalGFormula</td>
</tr>
</tbody>
</table>

**Time-Varying Treatment G-Formula**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonteCarloGFormula</td>
</tr>
<tr>
<td>IterativeCondGFormula</td>
</tr>
</tbody>
</table>

**Augmented Inverse Probability Weights**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIPTW</td>
</tr>
<tr>
<td>SingleCrossfitAIPTW</td>
</tr>
<tr>
<td>DoubleCrossfitAIPTW</td>
</tr>
</tbody>
</table>

**Targeted Maximum Likelihood Estimator**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMLE</td>
</tr>
<tr>
<td>StochasticTMLE</td>
</tr>
<tr>
<td>SingleCrossfitTMLE</td>
</tr>
<tr>
<td>DoubleCrossfitTMLE</td>
</tr>
</tbody>
</table>

**G-estimation of SNM**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEstimationSNM</td>
</tr>
</tbody>
</table>

**Generalizability / Transportability**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPSW</td>
</tr>
<tr>
<td>GTransportFormula</td>
</tr>
<tr>
<td>AIPSW</td>
</tr>
</tbody>
</table>
1.8.5 Super Learner

Details for super learner and associated candidate estimators within zEpid.

Super Learners

SuperLearner

Candidate Estimators

EmpiricalMeanSL
GLMSL
StepwiseSL

1.8.6 Sensitivity analyses

Details for sensitivity analysis tools implemented within zEpid.

Distributions

trapezoidal

Sensitivity analyzers

MonteCarloRR

1.8.7 Data sets

Descriptions of the data sets included within zEpid

load_sample_data
load_ewing_sarcoma_data
load_gvhd_data
load_sciatica_data
load_leukemia_data
load_longitudinal_data
load_binge_drinking_data
CHAPTER 2

Installation:

Dependencies are from the typical Python data-stack: Numpy, Pandas, Scipy, Statsmodels, and Matplotlib. Additionally, it requires Tabulate, so nice looking tables can be easily generated. Install using:

```
pip install zepid
```
Chapter 2. Installation:
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 (@zepidpy)