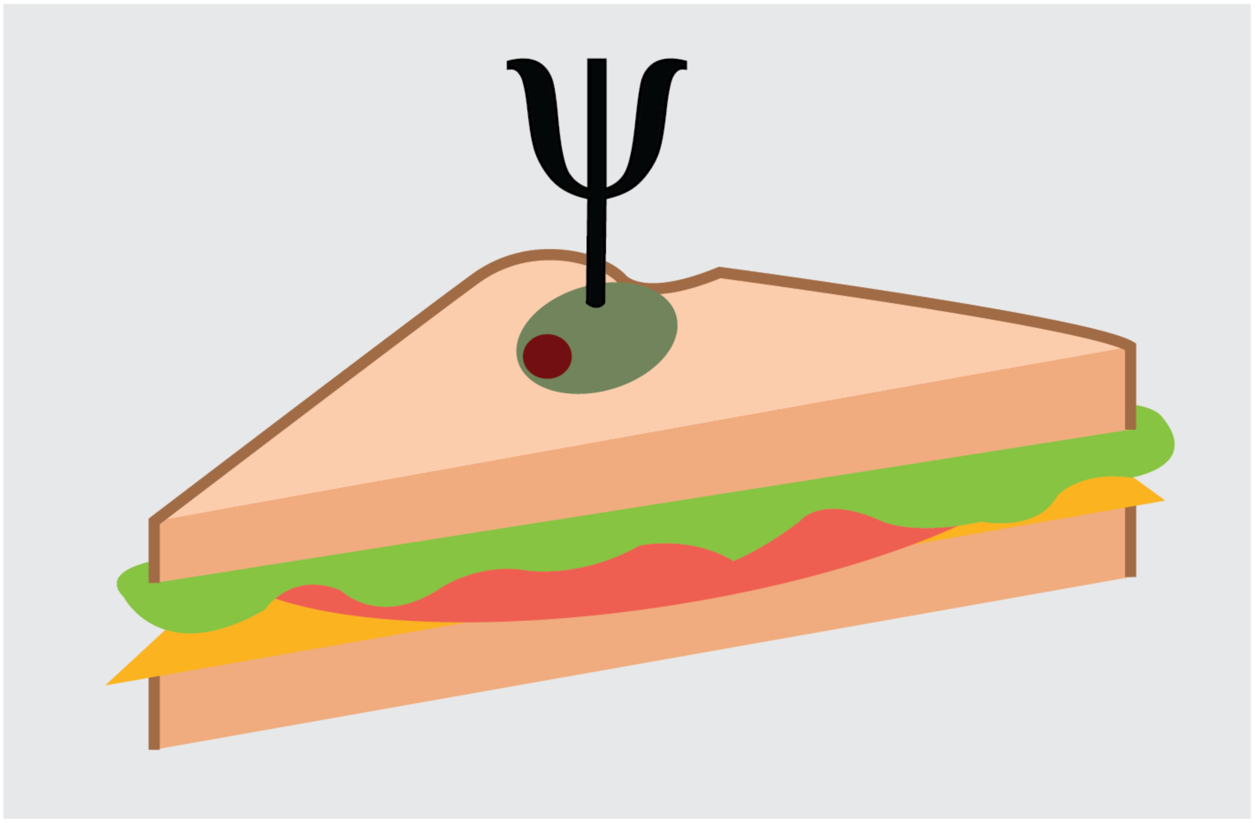
Delicatessen Documentation

Release 2.2

Paul Zivich, Mark Klose

Apr 24, 2024

CONTENTS



`delicatessen` is a one-stop shop for all your sandwich (variance) needs. This Python 3.8+ library supports M-estimation, which is a general statistical framework for estimating unknown parameters. If you are an R user, I recommend looking into `geex` (Saul & Hudgens (2020)). `delicatessen` supports a variety of pre-built estimating equations as well as custom, user built estimating equations.

Here, we provide a brief overview of M-Estimation. For a more detailed, please refer to Ross et al. (2024), Stefanski & Boos (2002), or Boos & Stefanski (2013). M-estimation was developed to study the large sample properties of robust statistics. However, many common large-sample statistics can be expressed with estimating equations, so M-estimation provides a unified structure and a streamlined approach to estimation. Let the parameter of interest be the vector $\theta = (\theta_1, \theta_2, ..., \theta_v)$ and data is observed for n independent units $O_1, O_2, \ldots, O_n$. An M-estimator, $\hat{\theta}$, is the solution to the estimating equation $\sum_{i=1}^{n} \psi(O_i, \hat{\theta}) = 0$ where ψ is a known $v \times 1$-dimension estimating function. M-estimators further provides a convenient and automatic method of calculating large-sample variance estimators via the empirical sandwich variance estimator:

$$V_n(O, \hat{\theta}) = B_n(O, \hat{\theta})^{-1} F_n(O, \hat{\theta}) \left(B_n(O, \hat{\theta})^{-1} \right)^T$$

where the ‘bread’ is

$$B_n(O, \hat{\theta}) = n^{-1} \sum_{i=1}^{n} -\psi'(O_i, \hat{\theta})$$

where the ψ' indicates the partial derivative, and the ‘filling’ is

$$F_n(O, \hat{\theta}) = n^{-1} \sum_{i=1}^{n} \psi(O_i, \hat{\theta}) \psi(O_i, \hat{\theta})^T$$

While M-Estimation is a general approach, widespread application is hindered by the corresponding derivative and matrix calculations. To circumvent these barriers, `delicatessen` automates M-estimators using numerical approximation methods.

The following description is a high-level overview. The user provides a $v \times n$ array of estimating function(s) to the `MEstimator` class object. Next, the `MEstimator` object solves for $\hat{\theta}$ using a root-finding algorithm. After successful completion of the root-finding, the bread is computed by numerically approximating the partial derivatives and the filling is calculated. Finally, the empirical sandwich variance is computed.

CHAPTER ONE

INSTALLATION:

To install `delicatessen`, use the following command in terminal or command prompt

```
python -m pip install delicatessen
```

Only two dependencies for `delicatessen` are: `NumPy`, `SciPy`.

While `pandas` is not a dependency, several examples are demonstrated with pandas for ease of data processing. To replicate the tests in `tests/` you will need to also install `pandas`, `statsmodels` and `pytest` (but these are not necessary for use of the package).

CHAPTER TWO

CITATION:

Please use the following citation for `delicatessen`: Zivich PN, Klose M, Cole SR, Edwards JK, & Shook-Sa BE. (2022). Delicatessen: M-Estimation in Python. *arXiv preprint arXiv:2203.11300*. URL

```
@article{zivich2022,
  title={Delicatessen: M-estimation in Python},
  author={Zivich, Paul N and Klose, Mark and Cole, Stephen R and Edwards, Jessie K and␣
↪Shook-Sa, Bonnie E},
  journal={arXiv preprint arXiv:2203.11300},
  year={2022}
}
```

CHAPTER

THREE

CONTENTS:

3.1 Basics

Here, the basics of M-estimator will be reviewed. An M-estimator, $\hat{\theta}$, is defined as the solution to the estimating equation

$$\sum_{i=1}^{n} \psi(O_i, \hat{\theta}) = 0$$

where ψ is a known $v \times 1$-dimension estimating function, O_i indicates the observed unit $i \in \{1, ..., n\}$, and the parameters are the vector $\theta = (\theta_1, \theta_2, ..., \theta_v)$. Note that v is finite-dimensional and the number of parameters matches the dimension of the estimating functions.

3.1.1 Point Estimation

To implement the point estimation of θ, we use a *root-finding* algorithm. Root-finding algorithms are procedures for finding the zeroes (i.e., roots) of an equation. This is accomplished in `delicatessen` by using SciPy's root-finding algorithms.

3.1.2 Variance Estimation

To estimate the variance for θ, the M-estimator uses the empirical sandwich variance estimator:

$$V_n(O, \hat{\theta}) = B_n(O, \hat{\theta})^{-1} F_n(O, \hat{\theta}) \left(B_n(O, \hat{\theta})^{-1} \right)^T$$

where the 'bread' is

$$B_n(O, \hat{\theta}) = n^{-1} \sum_{i=1}^{n} -\psi'(O_i, \hat{\theta})$$

where the ψ' indicates the partial derivative, and the 'filling' is

$$F_n(O, \hat{\theta}) = n^{-1} \sum_{i=1}^{n} \psi(O_i, \hat{\theta}) \psi(O_i, \hat{\theta})^T$$

The sandwich variance requires finding the derivative of the estimating functions and some matrix algebra. Again, we can get the computer to complete all these calculations for us. For the derivative, `delicatessen` offers two options. First, the derivatives can be numerically approximated using the central difference method. This is done using SciPy's `approx_fprime` functionality. As of `v2.0`, the derivatives can also be computed using forward-mode automatic differentiation. This approach provides the exact derivative (as opposed to an approximation). This is implemented by-hand in `delicatessen` via operator overloading. Finally, we use forward-mode because there is no time advantage of backward-mode because the Jacobian is square.

Automatic Differentiation Caveats

There are some caveats to the use of automatic differentiation. First, some NumPy functionalities are not fully supported. For example, `np.log(x, where=0<x)` will result in an error since there is an attempt to evaluate a log at zero internally. When using these specialty functions are necessary, it is better to use numerical approximation for differentiation. The second is regarding discontinuities. Consider the following function $f(x) = x**2$ if $x \geq 1$ and $f(x) = 0$ otherwise. Because of how automatic differentiation operates, the derivative at $x = 1$ will result in $2x$ (this is the same behavior as other automatic differentiation software, like autograd).

After computing the derivatives, the filling is computed via a dot product. The bread is then inverted using NumPy. Finally, the bread and filling matrices are combined via dot products.

This introduction has all been a little abstract. In the following Examples section, we will see how M-estimators can be used to address specific estimation problems.

3.2 Applied Examples

This section provides illustrative examples of application of M-estimators. This includes the replication of examples from textbooks and published scientific articles. These examples include built-in estimating equations and use of user-built estimating equations.

3.2.1 Ross et al. (2024): Introduction to M-estimation

The following is a replication of the cases described in Ross et al. (2024). The original paper provides a tutorial on M-estimation and provides several introductory examples. Examples are provided in the context of regression, standardization, and measurement error. Here, we recreate the cases described in the paper. For finer details, see the original publication.

Setup

[1]:
```python
import numpy as np
import pandas as pd

import delicatessen
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_regression, ee_rogan_gladen
from delicatessen.utilities import inverse_logit

print("Versions")
print("NumPy:        ", np.__version__)
print("pandas:       ", pd.__version__)
print("Delicatessen: ", delicatessen.__version__)
```

```
Versions
NumPy:        1.25.2
pandas:       1.4.1
Delicatessen:  2.1
```

The following data is used for the first and second cases.

[2]:
```
# From Table 1
d = pd.DataFrame()
d['X'] = [0, 0, 0, 0, 1, 1, 1, 1]            # X values
d['W'] = [0, 0, 1, 1, 0, 0, 1, 1]            # W values
d['Y'] = [0, 1, 0, 1, 0, 1, 0, 1]            # Y values
d['n'] = [496, 74, 113, 25, 85, 15, 15, 3]   # Counts
d['intercept'] = 1                           # Intercept term (always 1)

# Expanding rows by n
d = pd.DataFrame(np.repeat(d.values,         # Converting tabled data
                           d['n'], axis=0),  # ... by replicating counts
                 columns=d.columns)          # ... into rows for each X,W,Y
d = d[['intercept', 'X', 'W', 'Y']].copy()   # Dropping extra rows
```

Example 1: Logistic Regression

For the first example, we fit a logistic regression model for the variable Y given X, W.

[3]:
```
# Extracting arrays for easier coding later on
X = np.asarray(d[['intercept', 'X', 'W']])   # Design matrix for regression
y = np.asarray(d['Y'])                       # Outcome in regression
```

For estimation, we can use the built-in function for regression

[4]:
```
def psi(theta):
    return ee_regression(theta=theta,
                         y=y, X=X,
                         model='logistic')
```

[5]:
```
estr = MEstimator(psi, init=[0, 0, 0,])
estr.estimate()
```

[6]:
```
# Formatting results into a nice table
result = pd.DataFrame()
result['Param'] = ['beta_0', 'beta_1', 'beta_2']
result['Coef'] = estr.theta
ci = estr.confidence_intervals()
result['LCL'] = ci[:, 0]
result['UCL'] = ci[:, 1]
result.round(2)
```

[6]:
```
    Param  Coef   LCL   UCL
0  beta_0 -1.89 -2.13 -1.66
1  beta_1  0.12 -0.43  0.67
2  beta_2  0.36 -0.11  0.83
```

These results match those reported in the paper. They also match the results if you were to fit this using `statsmodels` GLM instead.

Example 2: Estimating the marginal risk difference

In this example, we build on the prior example. Using the logistic model, we can generate predictions and use those predictions to estimate the marginal risk difference by X via g-computation. Alternatively, we can use a logistic model to estimate propensity score and then estimate the marginal risk difference with inverse probability weighting.

First, let's apply g-computation

[7]:
```
# Copies of data with policies applied
d1 = d.copy()
d1['X'] = 1
X1 = np.asarray(d1[['intercept', 'X', 'W']])
d0 = d.copy()
d0['X'] = 0
X0 = np.asarray(d0[['intercept', 'X', 'W']])
```

[8]:
```
def psi(theta):
    # Dividing parameters into corresponding parts and labels from slides
    beta = theta[0:3]              # Logistic model coefficients
    mu0, mu1 = theta[3], theta[4]  # Causal risks
    delta1 = theta[5]              # Causal contrast

    # Logistic regression model for outcome
    ee_logit = ee_regression(theta=beta,
                             y=y, X=X,
                             model='logistic')

    # Transforming logistic model coefficients into causal parameters
    y0_hat = inverse_logit(np.dot(X0, beta))  # Prediction under a=0
    y1_hat = inverse_logit(np.dot(X1, beta))  # Prediction under a=1

    # Estimating function for causal risk under a=1
    ee_r1 = y1_hat - mu1             # Simple mean

    # Estimating function for causal risk under a=0
    ee_r0 = y0_hat - mu0             # Simple mean

    # Estimating function for causal risk difference
    ee_rd = np.ones(d.shape[0])*((mu1 - mu0) - delta1)

    # Returning stacked estimating functions in order of parameters
    return np.vstack([ee_logit,   # EF of logistic model
                      ee_r0,      # EF of causal risk a=0
                      ee_r1,      # EF of causal risk a=1
                      ee_rd])     # EF of causal contrast
```

[9]:
```
estr = MEstimator(psi, init=[0, 0, 0, 0.5, 0.5, 0])
estr.estimate()
```

[10]:
```
# Formatting results into a nice table
result = pd.DataFrame()
result['Param'] = ['beta_0', 'beta_1', 'beta_2', 'mu_0', 'mu_1', 'delta']
result['Coef'] = estr.theta
```

(continues on next page)

(continued from previous page)

```
ci = estr.confidence_intervals()
result['LCL'] = ci[:, 0]
result['UCL'] = ci[:, 1]
result.round(2)
```

[10]:

```
    Param  Coef   LCL   UCL
0  beta_0 -1.89 -2.13 -1.66
1  beta_1  0.12 -0.43  0.67
2  beta_2  0.36 -0.11  0.83
3    mu_0  0.14  0.11  0.17
4    mu_1  0.15  0.09  0.22
5   delta  0.01 -0.06  0.09
```

These results match those reported in the publication (some differences in rounding by the chosen root-finding algorithm).

Now consider estimating the marginal risk difference using inverse probability weighting. The following code implements this estimator by-hand

[11]:

```
a = np.asarray(d['X'])
W = np.asarray(d[['intercept', 'W']])
```

[12]:

```
def psi(theta):
    # Dividing parameters into corresponding parts and labels from slides
    alpha = theta[0:2]              # Logistic model coefficients
    mu0, mu1 = theta[2], theta[3]   # Causal risks
    delta1 = theta[4]               # Causal contrast

    # Logistic regression model for propensity score
    ee_logit = ee_regression(theta=alpha,        # Regression model
                             y=a,                # ... for exposure
                             X=W,                # ... given confounders
                             model='logistic')   # ... logistic model

    # Transforming logistic model coefficients into causal parameters
    pscore = inverse_logit(np.dot(W, alpha))     # Propensity score
    wt = d['X']/pscore + (1-d['X'])/(1-pscore)   # Corresponding weights

    # Estimating function for causal risk under a=1
    ee_r1 = d['X']*d['Y']*wt - mu1         # Weighted conditional mean

    # Estimating function for causal risk under a=0
    ee_r0 = (1-d['X'])*d['Y']*wt - mu0     # Weighted conditional mean

    # Estimating function for causal risk difference
    ee_rd = np.ones(d.shape[0])*((mu1 - mu0) - delta1)

    # Returning stacked estimating functions in order of parameters
    return np.vstack([ee_logit,   # EF of logistic model
                      ee_r0,      # EF of causal risk a=0
                      ee_r1,      # EF of causal risk a=1
                      ee_rd])     # EF of causal contrast
```

[13]:
```
estr = MEstimator(psi, init=[0, 0, 0.5, 0.5, 0])
estr.estimate()
```

[14]:
```
# Formatting results into a nice table
result = pd.DataFrame()
result['Param'] = ['alpha_0', 'alpha_1', 'mu_0', 'mu_1', 'delta']
result['Coef'] = estr.theta
ci = estr.confidence_intervals()
result['LCL'] = ci[:, 0]
result['UCL'] = ci[:, 1]
result.round(2)
```

[14]:
```
     Param  Coef   LCL   UCL
0  alpha_0 -1.74 -1.95 -1.53
1  alpha_1 -0.30 -0.83  0.24
2     mu_0  0.14  0.11  0.17
3     mu_1  0.15  0.09  0.22
4    delta  0.01 -0.06  0.08
```

Again, these results match those reported in the publication

Example 3: Outcome misclassification

For the final example

[15]:
```
# Loading in data for the fusion example
d = pd.DataFrame()
d['R'] = [1, 1, 0, 0, 0, 0]           # R or population indicator
d['Y'] = [0, 0, 1, 1, 0, 0]           # True outcome
d['W'] = [1, 0, 1, 0, 1, 0]           # Measured outcome
d['n'] = [680, 270, 204, 38, 18, 71]  # Counts
d['intercept'] = 1                    # Intercept is always 1

# Expanding out data
d = pd.DataFrame(np.repeat(d.values, d['n'], axis=0),  # Expanding compact data frame
                 columns=d.columns)                    # ... keeping column names
d = d[['intercept', 'R', 'W', 'Y']].copy()             # Dropping the n column

# Converting to arrays to simplify process
r = np.asarray(d['R'])
w = np.asarray(d['W'])
y = np.asarray(d['Y'])
```

[16]:
```
def psi(theta):
    return ee_rogan_gladen(theta=theta,     # Parameter of interest
                           y=d['Y'],        # ... correct measure
                           y_star=d['W'],   # ... mismeasure
                           r=d['R'])        # ... sample indicator
```

[17]:
```
estr = MEstimator(psi, init=[0.75, 0.75, 0.75, 0.75])
estr.estimate()
```

```
[18]: # Formatting results into a nice table
result = pd.DataFrame()
result['Param'] = ['Corrected', 'Mismeasured', 'Sensitivity', 'Specificity']
result['Coef'] = estr.theta
ci = estr.confidence_intervals()
result['LCL'] = ci[:, 0]
result['UCL'] = ci[:, 1]
result.round(2)
```

```
[18]:          Param  Coef   LCL   UCL
0    Corrected  0.80  0.72  0.88
1  Mismeasured  0.72  0.69  0.74
2  Sensitivity  0.84  0.80  0.89
3  Specificity  0.80  0.71  0.88
```

These results match Ross et al.

Conclusions

Here, we replicated the introduction to M-estimation tutorial provided in Ross et al. The basic ideas illustrated in that paper and replicated here can be extended in numerous directions.

References

Ross RK, Zivich PN, Stringer JS, & Cole SR. (2024). M-estimation for common epidemiological measures: introduction and applied examples. *International Journal of Epidemiology*, 53(2), dyae030.

3.2.2 Zivich et al. (2022): Delicatessen Publication

The following is the code to replicate the examples provided in Zivich et al. (2022). These examples are some common use cases of M-estimation in life science research. Specifically, we provide three case studies.

Zivich PN, Klose M, Cole SR, Edwards JK, & Shook-Sa BE. (2022). Delicatessen: M-Estimation in Python. arXiv preprint arXiv:2203.11300.

Setup

```
[1]: # Loading packages for examples
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline
mpl.rcParams['figure.dpi']= 300

import delicatessen
from delicatessen import MEstimator
from delicatessen.data import load_robust_regress, load_inderjit
```

(continues on next page)

(continued from previous page)

```
from delicatessen.estimating_equations import (ee_regression,
                                               ee_robust_regression,
                                               ee_3p_logistic,
                                               ee_effective_dose_delta)
from delicatessen.utilities import inverse_logit

decimal_places = 3
np.random.seed(51520837)

print("NumPy version:       ", np.__version__)
print("SciPy version:       ", sp.__version__)
print("Pandas version:      ", pd.__version__)
print("Matplotlib version:  ", matplotlib.__version__)
print("Delicatessen version:", delicatessen.__version__)
```

```
NumPy version:        1.25.2
SciPy version:        1.11.2
Pandas version:       1.4.1
Matplotlib version:   3.5.1
Delicatessen version: 2.1
```

Case Study 1: Linear Regression

The first example considers application of linear regression with an outlier. We will use some simulated data to compare simple linear regression to robust linear regression.

Benchmark

Since the outlier was simulated, we will first load the data and use simple linear regression as a benchmark. The following block of code implements this

[2]:
```
# Loading the data without the outlier to generate reference
d = load_robust_regress(outlier=False)       # Loads data without outlier
x = d[:, 0]                                  # Extract X-values (height)
y = d[:, 1]                                  # Extract Y-values (weight)
X = np.vstack((np.ones(x.shape[0]), x)).T    # Convert to array


def psi_simple_linear(theta):
    return ee_regression(theta=theta,
                         X=X,
                         y=y,
                         model='linear')


# Linear regression without the outlier for reference
lme = MEstimator(psi_simple_linear,
                 init=[0., 0.])
lme.estimate(solver='hybr')
```

Now we can load the data with the outlier and reuse the defined estimating equation above to estimate the simple linear regression model

```
[3]: # Loading the data with the outlier
d = load_robust_regress(outlier=True)       # Loads data with outlier
y = d[:, 1]                                 # Extract Y with outlier

# Linear regression with the outlier
ulme = MEstimator(psi_simple_linear,
                  init=[0., 0.])
ulme.estimate(solver='hybr')
```

Finally, robust linear regression is used with the Huber method

```
[4]: def psi_case1_robust(theta):
    return ee_robust_regression(theta=theta,
                                X=X,
                                y=y,
                                k=1.345,
                                model='linear')


# Notice: the theta from the previous regression is used since
#   robust regression can fail when initial values are too far.
rlme = MEstimator(psi_case1_robust,
                  init=ulme.theta)
rlme.estimate(solver='hybr')
```

To distinguish between our results, we will generate a figure of the three estimated regression lines

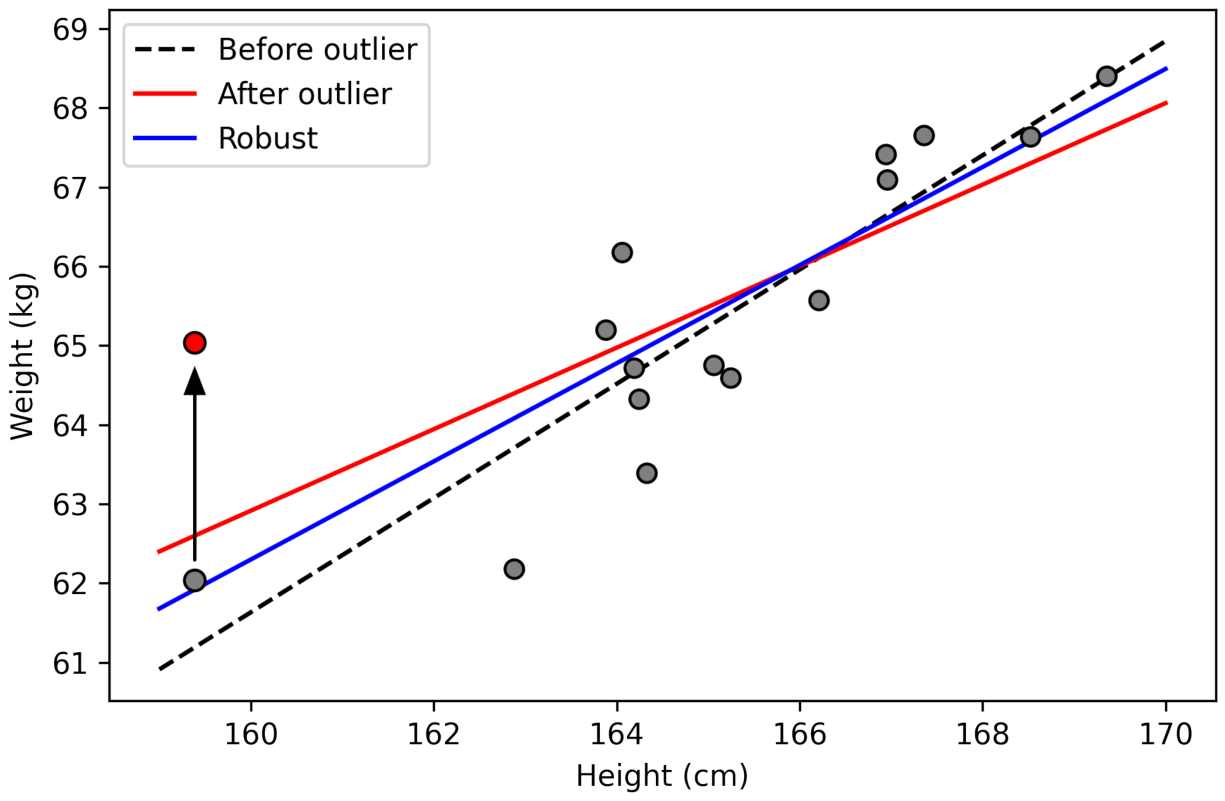
```
[5]: # Displaying results
plt.figure(figsize=[6, 4])

xlin = np.linspace(159, 170, 100)

# Plotting linear model results
plt.plot(xlin, lme.theta[0] + xlin*lme.theta[1],
         '--', color='k', label='Before outlier')
plt.plot(xlin, ulme.theta[0] + xlin*ulme.theta[1],
         '-', color='red', label='After outlier')
plt.plot(xlin, rlme.theta[0] + xlin*rlme.theta[1],
         '-', color='blue', label='Robust')

# Plotting the data points (including the outlier)
plt.scatter(x, y, s=40, c='gray', edgecolors='k',
            label=None, zorder=4)
plt.scatter(159.386, y[x == 159.386] - 3, s=50, c='gray',
            edgecolors='k', zorder=4)
plt.scatter(159.386, y[x == 159.386], s=50, c='red',
            edgecolors='k', zorder=5)
plt.arrow(159.386, float(y[x == 159.386]) - 2.75, 0, 2.1,
          head_width=0.2, facecolor='k', zorder=5)

# Making nice labels for graph
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Height (cm)")
plt.ylabel("Weight (kg)")
plt.legend()

# Outputting result
plt.tight_layout()
```

```
C:\Users\zivic\AppData\Local\Temp\ipykernel_10668\1702613546.py:21: DeprecationWarning:␣
↪Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in␣
↪future. Ensure you extract a single element from your array before performing this␣
↪operation. (Deprecated NumPy 1.25.)
  plt.arrow(159.386, float(y[x == 159.386]) - 2.75, 0, 2.1,
```



As seen, the simple linear regression model is sensitive to the outlier. Robust linear regression is less sensitive (and is closer to the benchmark, the dashed line).

Case Study 2: Dose-Response Curve

Next, we consider estimation of the dose-response curve using some publicly available data that come pre-packaged with `delicatessen`.

```
[6]: d = load_inderjit()
```

The following code implements a 3-parameter log-logistic model (the lower limit is set to zero, i.e., it is not estimated). Additionally, we append an estimating equation to compute the 20% effective dose.

[7]:
```python
def psi(theta):
    # Asserting that the lower limit is zero
    lower_limit = 0

    # Estimating equations for the 3PL model
    pl3 = ee_3p_logistic(theta=theta[0:3],
                         X=d[:, 1], y=d[:, 0],
                         lower=lower_limit)

    # Estimating equations for the effective concentrations
    ed20 = ee_effective_dose_delta(theta[3], y=d[:, 0], delta=0.20,
                                   steepness=theta[0], ed50=theta[1],
                                   lower=lower_limit, upper=theta[2])

    # Returning stacked estimating equations
    return np.vstack((pl3,
                      ed20))



# Optimization procedure
mest = MEstimator(psi, init=[3.3, 2.5, 8., 2.])
mest.estimate(solver='lm', maxiter=2000)
ci_theta = mest.confidence_intervals()
```

```
c:\users\zivic\documents\open-source\delicatessen\delicatessen\estimating_equations\dose_
↪response.py:131: RuntimeWarning: divide by zero encountered in log
  nested_log = np.where(X > 0, np.log(X / theta[1]), 0)  # Handling when dose = 0
```

[8]:
```python
# Printing the results to the console
print("==================================")
print("Case Study 2: Results")
print("==================================")
print("ED(50)")
print("----------------------------------")
print("Est:   ", np.round(mest.theta[0], decimal_places))
print("95% CI:", np.round(ci_theta[0],
                          decimal_places))
print("----------------------------------")
print("Steepness")
print("----------------------------------")
print("Est:   ", np.round(mest.theta[1], decimal_places))
print("95% CI:", np.round(ci_theta[1],
                          decimal_places))
print("----------------------------------")
print("Upper Limit")
print("----------------------------------")
print("Est:   ", np.round(mest.theta[2], decimal_places))
print("95% CI:", np.round(ci_theta[2],
                          decimal_places))
print("----------------------------------")
print("ED(20)")
print("----------------------------------")
```

(continues on next page)

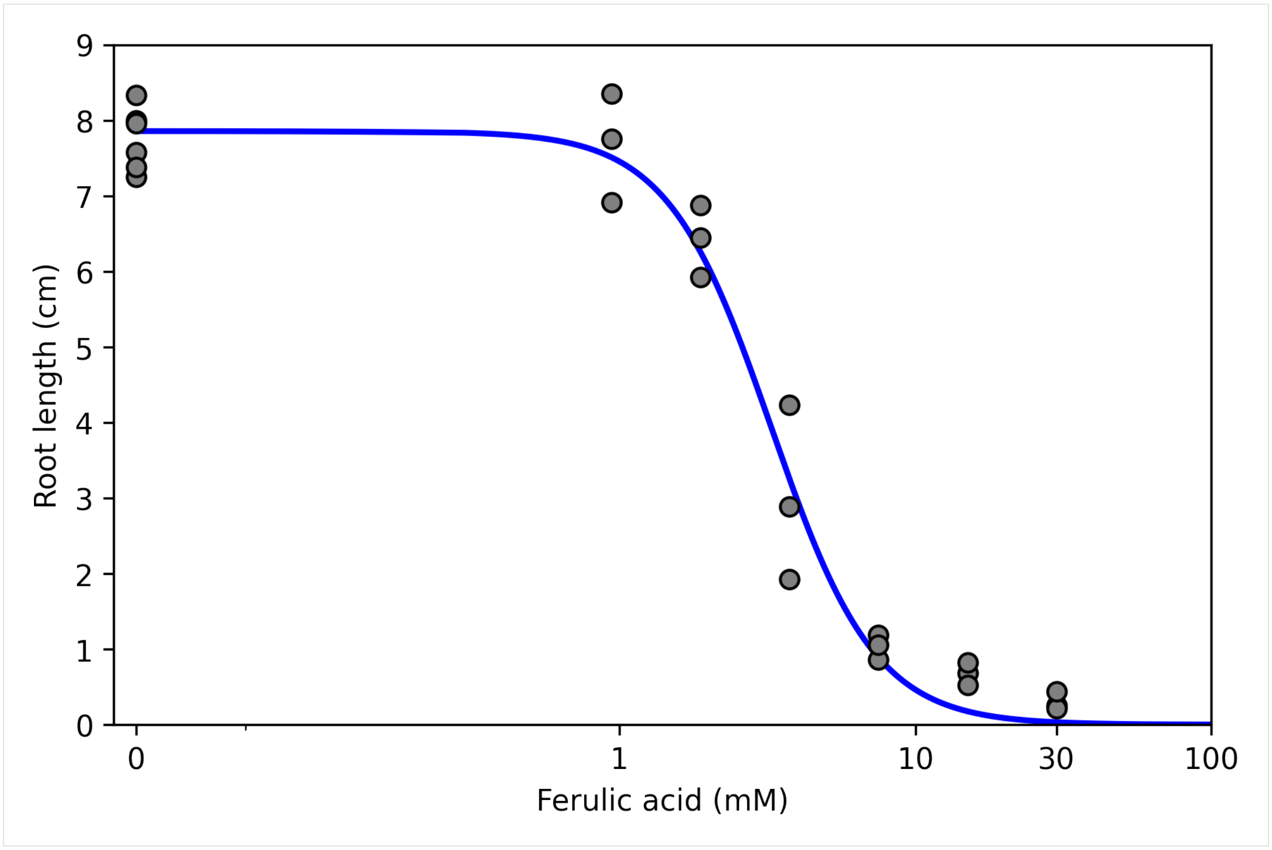
(continued from previous page)

```
print("Est:    ", np.round(mest.theta[3], decimal_places))
print("95% CI:", np.round(ci_theta[3],
                          decimal_places))
print("=========================================")
```

```
==================================
Case Study 2: Results
==================================
ED(50)
----------------------------------
Est:    3.263
95% CI: [2.743 3.784]
----------------------------------
Steepness
----------------------------------
Est:    2.47
95% CI: [1.897 3.043]
----------------------------------
Upper Limit
----------------------------------
Est:    7.855
95% CI: [7.554 8.157]
----------------------------------
ED(20)
----------------------------------
Est:    1.862
95% CI: [1.581 2.143]
=========================================
```

The results can also be shown through a visualization of the estimated dose-response function.

[9]:
```
# Displaying results
plt.figure(figsize=[6, 4])

x = np.linspace(0, 100, 5000)
theta = mest.theta
y = 0 + (theta[2] - 0) / (1 + (x/theta[0])**theta[1])

# Plotting points and drawing dose-response line
plt.plot(x, y, '-', color='blue', linewidth=2)
plt.scatter(d[:, 1], d[:, 0], s=40, c='gray', edgecolors='k', zorder=5)

plt.ylim([0, 9])
plt.ylabel("Root length (cm)")
plt.xscale('symlog', linthresh=0.3)
plt.xlim([-0.02, 100])
plt.xticks([0, 1, 10, 30, 100],
           ["0", "1", "10", "30", "100"])
plt.xlabel("Ferulic acid (mM)")
plt.tight_layout()
```



As we can see here, the higher the dose, the shorter the root length.

Case Study 3: Standardization to external population

For the final example, we use inverse odds weights to standardize scientific results to an external population by drug use. This approach helps to generalize study results beyond a particular convenience sample

[10]:
```
# Loading Kamat et al. 2012
d1 = pd.read_csv("data/kamat.et.al.2012_biomarkers.csv")
d1['drug_use'] = np.where(d1['Cocaine'] + d1['Opiate'] > 0, 1, 0)
d1['S'] = 1
biomarkers = ['IFN_alpha', 'CXCL9', 'CXCL10', 'sIL-2R', 'IL12']
d1 = d1[['drug_use', 'S', ] + biomarkers].copy()
for bm in biomarkers:
    d1[bm] = np.log(d1[bm])


# MACS/ WIHS in 2018-2019 of cocaine or heroin use in previous 6 months
d0 = pd.DataFrame()
d0['drug_use'] = [1]*300 + [0]*(4016-300)
d0['S'] = 0


# Stacking data together and adding constant for model
d = pd.concat([d0, d1])           # Stacking data sets together
```

(continues on next page)

(continued from previous page)

```
d['constant'] = 1                    # Creating intercept for the model
d[biomarkers] = d[biomarkers].fillna(9999)
```

The following code calculates the means for the study population. Note that these may not be generalizable

[11]:
```
def psi_standard_mean(theta):
    # Returning stacked estimating equations
    return (np.asarray(d1['IFN_alpha']) - theta[0],
            np.asarray(d1['CXCL9']) - theta[1],
            np.asarray(d1['CXCL10']) - theta[2],
            np.asarray(d1['sIL-2R']) - theta[3],
            np.asarray(d1['IL12']) - theta[4], )


nm = MEstimator(psi_standard_mean, init=[1., ]*5)
nm.estimate()
```

The following code generalizes the results from Kamat et al. using data from the Women's Interagency HIV Study (WIHS). The WIHS study is considered to be more generalizable, but we don't have access to these biomarkers in the WIHS data.

[12]:
```
# Weighted means to standardize to population
x = np.asarray(d[['constant', 'drug_use']])
s = np.asarray(d['S'])


def psi_weighted_mean(theta):
    global x, y, s

    # Estimating weights
    nuisance = ee_regression(theta=theta[:2],
                             X=x, y=s, model='logistic')
    pi = inverse_logit(np.dot(x, theta[:2]))
    weight = np.where(s == 1, (1-pi)/pi, 0)

    # Returning stacked estimating equations
    return np.vstack((nuisance,
                      s*weight*(np.asarray(d['IFN_alpha']) - theta[2]),
                      s*weight*(np.asarray(d['CXCL9']) - theta[3]),
                      s*weight*(np.asarray(d['CXCL10']) - theta[4]),
                      s*weight*(np.asarray(d['sIL-2R']) - theta[5]),
                      s*weight*(np.asarray(d['IL12']) - theta[6]), )
                     )


wm = MEstimator(psi_weighted_mean, init=[0, 0] + [1., ]*5)
wm.estimate()
```

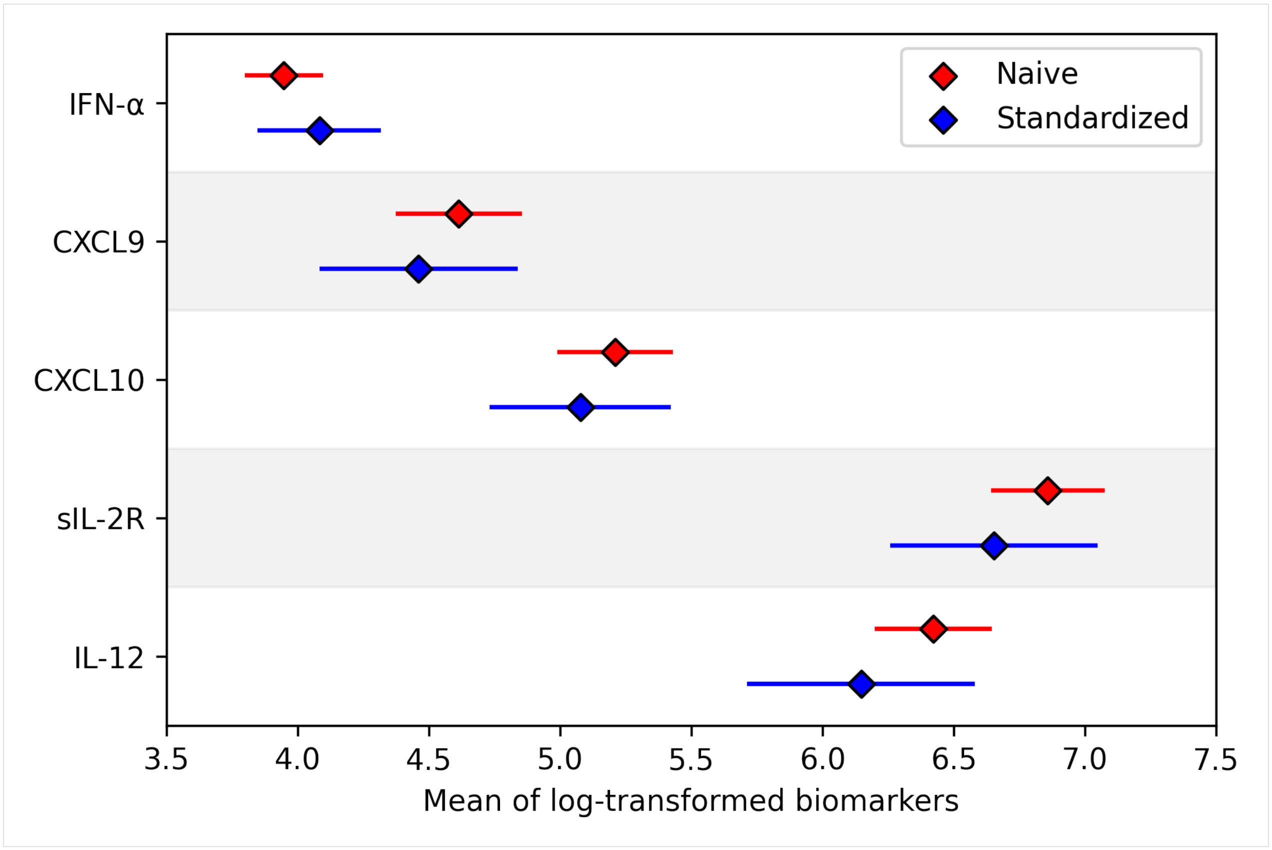
Finally we can show the difference between the raw results and the results generalized to the WIHS population.

[13]:
```
# Displaying results
plt.figure(figsize=[6, 4])
```

(continues on next page)

(continued from previous page)

```
# Plotting point estimates
plt.scatter(nm.theta, [i-0.2 for i in range(len(biomarkers))],
            s=40, color='red', edgecolors='k', marker='D',
            zorder=3, label='Naive')
plt.scatter(wm.theta[2:], [i+0.2 for i in range(len(biomarkers))],
            s=40, color='blue', edgecolors='k', marker='D',
            zorder=3, label='Standardized')

# Plotting confidence intervals
plt.hlines([i-0.2 for i in range(len(biomarkers))],
           nm.theta - 1.96*np.diag(nm.variance)**0.5,
           nm.theta + 1.96 * np.diag(nm.variance)**0.5,
           colors='red')
plt.hlines([i+0.2 for i in range(len(biomarkers))],
           wm.theta[2:] - 1.96*np.diag(wm.variance)[2:]**0.5,
           wm.theta[2:] + 1.96 * np.diag(wm.variance)[2:]**0.5,
           colors='blue')

# Some shaded regions to make it easier to examine
plt.fill_between([3, 7.5], [3.5, 3.5], [2.5, 2.5], color='gray', alpha=0.1)
plt.fill_between([3, 7.5], [1.5, 1.5], [0.5, 0.5], color='gray', alpha=0.1)

plt.yticks([i for i in range(len(biomarkers))],
           ['IFN-', 'CXCL9', 'CXCL10', 'sIL-2R', 'IL-12'])
plt.ylim([4.5, -0.5])
plt.xlim([3.5, 7.5])
plt.xlabel("Mean of log-transformed biomarkers")
plt.legend()
plt.tight_layout()
```



As shown here, there are some differences in biomarkers after standardizing the data (particularly IL-12 and sIL-2R).

This concludes the examples provided in the paper.

References

Ritz, C., Baty, F., Streibig, J. C. & Gerhard, D. Dose-Response Analysis Using R. PLOS ONE 10, e0146021 (2015).

Inderjit, Streibig, J. C. & Olofsdotter, M. Joint action of phenolic acid mixtures and its significance in allelopathy research. Physiol Plant 114, 422–428 (2002).

Kamat, A. et al. A Plasma Biomarker Signature of Immune Activation in HIV Patients on Antiretroviral Therapy. PLOS ONE 7, e30881 (2012).

3.2.3 Cole et al. (2023): Fusion Designs

The following is a replication of the cases described in Cole et al. (2023) and the rejoinder. The original paper considered fusion study designs, whereby multiple data sources are used together to address a single question. Examples are provided in the context of transporting a proportion, measurement error, and the joint occurrence of measurement error and transporting. In that paper, M-estimators are proposed as a general solution to estimation in fusion designs. Importantly, the empirical sandwich variance estimator allows for uncertainty estimation in this setting, unlike standard methods or approximations (like the GEE trick for inverse probability weighting). Further, these procedures are less computationally demanding than competing methods that do provide appropriate inference (e.g., bootstrapping, Monte Carlo methods).

Here, we recreate the cases described in the paper. For finer details, see the original publications.

Setup

[1]:
```
import numpy as np
import pandas as pd

import delicatessen
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_regression, ee_rogan_gladen
from delicatessen.utilities import inverse_logit

print("Versions")
print("NumPy:        ", np.__version__)
print("pandas:       ", pd.__version__)
print("Delicatessen: ", delicatessen.__version__)
```

```
Versions
NumPy:         1.25.2
pandas:        1.4.1
Delicatessen:  2.1
```

Case 1: Transporting the proportion

The goal is to estimate $\Pr(Y = 1)$ in the target population. However, we only have the outcome measured in a sample of a secondary population ($S = 1$). To estimate the proportion in the target population, we will transport from the secondary population to the target population conditional on a covariate, W. For estimation, we use inverse odds of sampling weights.

First, we build the data set described in Cole et al.

[2]:
```
d = pd.DataFrame()
d['Y'] = [0, 1, 0, 1] + [np.nan, np.nan]
d['W'] = [0, 0, 1, 1] + [0, 1]
d['n'] = [266, 67, 400, 267] + [333, 167]
d['S'] = [0, 0, 0, 0] + [1, 1]

# Expanding data set out
d = pd.DataFrame(np.repeat(d.values, d['n'], axis=0), columns=d.columns)
d['C'] = 1

# Setting up for delicatessen
y_no_nan = np.asarray(d['Y'].fillna(-999))
s = np.asarray(d['S'])
W = np.asarray(d[['C', 'W']])
```

For estimation, we stack the naive proportion, inverse odds weighted proportion, and sampling models together.

[3]:
```
def psi(theta):
    # Subsetting the input parameters
    mu_w = theta[0]     # Weighted proportion
    mu_n = theta[1]     # Naive proportion
    beta = theta[2:]    # Sampling model parameters

    # Sampling model for transporting
```

(continues on next page)

(continued from previous page)

```
    ee_sm = ee_regression(beta,
                          X=W, y=d['S'],
                          model='logistic')

    # Calculating inverse odds of sampling weights
    pi_s = inverse_logit(np.dot(W, beta))
    iosw = (1-s) * pi_s / (1-pi_s)

    # Weighted mean
    ee_wprop = iosw * (y_no_nan - mu_w)

    # Naive mean
    ee_nprop = (1-s) * (y_no_nan - mu_n)

    # Returning the stacked estimating equations
    return np.vstack([ee_wprop, ee_nprop, ee_sm])
```

[4]:
```
estr = MEstimator(psi, init=[0.5, 0.5, 0., 0.])
estr.estimate()
ci = estr.confidence_intervals()
```

[5]:
```
fmt = '{:.2f}, 95% CI: {:.2f}, {:.2f}'
print("Weighted proportion:", fmt.format(estr.theta[0], ci[0, 0], ci[0, 1]))
print("Naive proportion:   ", fmt.format(estr.theta[1], ci[1, 0], ci[1, 1]))
fmt = "{:.3f}, {:.3f}"
print("Model coefficients: ", fmt.format(estr.theta[2], estr.theta[3]))
```

```
Weighted proportion: 0.27, 95% CI: 0.24, 0.30
Naive proportion:    0.33, 95% CI: 0.30, 0.36
Model coefficients:  -0.000, -1.385
```

Note that this example differs in number from that reported in Cole et al. Since the corresponding data was not provided in full in the publication, we instead adapt the data from case three here. Regardless, the estimating equation setup is the same between examples.

Case 2: Estimating a Misclassified Proportion

The goal is to use external sensitivity and specificity data to correct for measurement error in the main sample. We do this by using the Rogan-Gladen correction

$$\hat{\mu} = \frac{\hat{\mu}^* + \widehat{Sp} - 1}{\widehat{Se} + \widehat{Sp} - 1}$$

where μ is the corrected mean, μ^* is the mismeasured mean, Se is the sensitivity, and Sp is the specificity. Hats indicate estimated quantities.

First, we build the data set described in Cole et al.

[6]:
```
# Compact data frame expression of data
d = pd.DataFrame()
d['R'] = [1, 1, 0, 0, 0, 0]
d['Y'] = [np.nan, np.nan, 1, 1, 0, 0]
```

(continues on next page)

(continued from previous page)

```
d['Y_star'] = [1, 0, 1, 0, 1, 0]
d['n'] = [680, 270, 204, 38, 18, 71]

# Expanding data set out
d = pd.DataFrame(np.repeat(d.values, d['n'], axis=0),
                 columns=d.columns)
d = d[['R', 'Y_star', 'Y']].copy()
```

For estimation, we use the built-in estimating equation for the Rogan-Gladen correction, `ee_rogan_gladen`

[7]:
```
def psi(theta):
    return ee_rogan_gladen(theta=theta, y=d['Y'],
                           y_star=d['Y_star'], r=d['R'])
```

[8]:
```
estr = MEstimator(psi, init=[0.5, 0.5, .75, .75])
estr.estimate()
ci = estr.confidence_intervals()
```

[9]:
```
fmt = '{:.2f}, 95% CI: {:.2f}, {:.2f}'
print("Corrected proportion:", fmt.format(estr.theta[0], ci[0, 0], ci[0, 1]))
print("Naive proportion:    ", fmt.format(estr.theta[1], ci[1, 0], ci[1, 1]))
print("Sensitivity:         ", fmt.format(estr.theta[2], ci[2, 0], ci[2, 1]))
print("Specificity:         ", fmt.format(estr.theta[3], ci[3, 0], ci[3, 1]))
```

```
Corrected proportion: 0.80, 95% CI: 0.72, 0.88
Naive proportion:     0.72, 95% CI: 0.69, 0.74
Sensitivity:          0.84, 95% CI: 0.80, 0.89
Specificity:          0.80, 95% CI: 0.71, 0.88
```

These results match those provided in Cole et al.

Case 3: Estimating a Misclassified and Transported Proportion

In the rejoinder, Cole et al. combine the previous examples. Specifically, we now transport the main sample and then correct for measurement error afterwards.

First, we build the data described

[10]:
```
# Compact data frame expression of data
d = pd.DataFrame()
d['Y_star'] = [0, 1, 0, 1] + [np.nan, np.nan] + [1, 0, 1, 0]
d['Y'] = [np.nan, np.nan, np.nan, np.nan] + [np.nan, np.nan] + [1, 1, 0, 0]
d['W'] = [0, 0, 1, 1] + [0, 1] + [np.nan, np.nan, np.nan, np.nan]
d['n'] = [266, 67, 400, 267] + [333, 167] + [180, 20, 60, 240]
d['S'] = [1, 1, 1, 1] + [2, 2] + [3, 3, 3, 3]

# Expanding data set out
d = pd.DataFrame(np.repeat(d.values, d['n'], axis=0), columns=d.columns)
d['C'] = 1

# Some extra data processing
y_no_nan = np.asarray(d['Y'].fillna(-1))
```

(continues on next page)

(continued from previous page)

```
ystar_no_nan = np.asarray(d['Y_star'].fillna(-1))
w_no_nan = np.asarray(d[['C', 'W']].fillna(-1))
s1 = np.where(d['S'] == 1, 1, 0)
s2 = np.where(d['S'] == 2, 1, 0)
s3 = np.where(d['S'] == 3, 1, 0)
```

[11]:
```
def psi(theta):
    # Subsetting the input parameters
    param = theta[:4]     # Measurement error parameters
    beta = theta[4:]      # Sampling model parameters

    # Sampling model for transporting
    ee_sm = ee_regression(beta,
                          X=w_no_nan,
                          y=s2,
                          model='logistic')
    ee_sm = ee_sm * (1 - s3)    # Only S=1 or S=2 contribute

    # Calculating inverse odds of sampling weights
    pi_s = inverse_logit(np.dot(w_no_nan, beta))
    # Note: iosw is the odds weight if S=1, zero if S=2 and 1 if S=3.
    #       So S=2 don't contribute to measurement error model, the
    #       naive mean among S=1 is reweighted appropriately, and S=3
    #       observations all contribute equally (we can't estimate
    #       weights for them since W was not measured)
    iosw = s1 * pi_s / (1-pi_s) + s3

    # Rogan-Gladen correction
    ee_rg = ee_rogan_gladen(param,
                            y=y_no_nan,
                            y_star=ystar_no_nan,
                            r=s1,
                            weights=iosw)
    ee_rg = ee_rg * (1 - s2)  # Only S=1 or S=3 contribute

    # Returning the stacked estimating equations
    return np.vstack([ee_rg, ee_sm])
```

[12]:
```
estr = MEstimator(psi, init=[0.5, 0.5, .75, .75, 0., 0.])
estr.estimate(solver='lm')
se = np.diag(estr.variance)**0.5
```

[13]:
```
fmt = '{:.3f} (SE={:.3f})'
print("Corrected weighted proportion:", fmt.format(estr.theta[0], se[0]))
print("Naive weighted proportion:    ", fmt.format(estr.theta[1], se[1]))
print("Sensitivity:                  ", fmt.format(estr.theta[2], se[2]))
print("Specificity:                  ", fmt.format(estr.theta[3], se[3]))
fmt = "{:.3f}, {:.3f}"
print("Sampling model coefficients:  ", fmt.format(estr.theta[4], estr.theta[5]))
```

```
Corrected weighted proportion: 0.097 (SE=0.038)
Naive weighted proportion:     0.268 (SE=0.016)
```

(continues on next page)

(continued from previous page)

```
Sensitivity:                 0.900 (SE=0.021)
Specificity:                 0.800 (SE=0.023)
Sampling model coefficients:  -0.000, -1.385
```

These results match those reported in the rejoinder. Note that the coefficients for the sampling model have a negative sign in the front since we are modeling $\Pr(S = 1|W)$, but the paper models $\Pr(S = 0|W)$ so there is no negative sign.

Conclusion

We replicated the cases from Cole et al. (2023) to showcase the basics of M-estimators for fusion designs and how they are implemented in `delicatessen`.

References

Cole SR, Edwards JK, Breskin A, Rosin S, Zivich PN, Shook-Sa BE, & Hudgens MG. (2023). “Illustration of 2 Fusion Designs and Estimators”. *American Journal of Epidemiology*, 192(3), 467-474.

Cole SR, Edwards JK, Breskin A, Rosin S, Zivich PN, Shook-Sa BE, & Hudgens MG. (2023). “Rejoinder: Combining Information with Fusion Designs, and Roses by Other Names”. *American Journal of Epidemiology*, kwad084.

3.2.4 Boos & Stefanski (2013): M-Estimation (Estimating Equations)

Here, we will implement some of the examples described in Chapter 7 of Boos & Stefanski (2013). If you have the book (or access to it), then reading along with each section may be helpful. Here, we code each of the estimating equations by-hand (rather than using the pre-built options offered).

Examples of M-Estimation provided in that chapter are replicated here using `delicatessen`. Reading the chapter and looking at the corresponding implementations is likely to be the best approach to learning both the theory and application of M-Estimation.

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In *Essential Statistical Inference* (pp. 297-337). Springer, New York, NY.

Setup

[1]:
```
import numpy as np
import scipy as sp
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

import delicatessen
from delicatessen import MEstimator

np.random.seed(80950841)

print("NumPy version:       ", np.__version__)
print("SciPy version:       ", sp.__version__)
print("Pandas version:      ", pd.__version__)
print("Delicatessen version:", delicatessen.__version__)
```

```
NumPy version:        1.25.2
SciPy version:        1.11.2
Pandas version:       1.4.1
Delicatessen version: 2.1
```

7.2 The Basic Approach

7.2.2 Sample Mean and Variance

The first example is the estimating equations for the mean and variance. To demonstrate the example, we will use some generic data for Y. Below is an example data set that will be used up to Section 7.2.6

[2]:
```
n = 200
data = pd.DataFrame()
data['Y'] = np.random.normal(loc=10, scale=2, size=n)
data['X'] = np.random.normal(loc=5, size=n)
data['C'] = 1
```

Here, estimating equations for both the mean and variance are stacked together:

$$\psi(Y_i, \theta) = \begin{bmatrix} Y_i - \theta_1 \\ (Y_i - \theta_1)^2 - \theta_2 \end{bmatrix}$$

The top estimating equation is the mean, and the bottom estimating equation is the (asymptotic) variance. The following is a by-hand implementation of these estimating equations

[3]:
```
def psi_mean_var(theta):
    """By-hand stacked estimating equations"""
    return (data['Y'] - theta[0],
            (data['Y'] - theta[0])**2 - theta[1])



estr = MEstimator(psi_mean_var, init=[0, 0])
estr.estimate()

print("=========================================================")
print("Mean & Variance")
print("=========================================================")
print("M-Estimation: by-hand")
print("Theta:", estr.theta)
print("Var:  \n", estr.asymptotic_variance)
print("---------------------------------------------------------")
print("Closed-Form")
print("Mean: ", np.mean(data['Y']))
print("Var:  ", np.var(data['Y'], ddof=0))
print("=========================================================")
```

```
=========================================================
Mean & Variance
=========================================================
M-Estimation: by-hand
Theta: [10.16284625  4.11208477]
```

(continues on next page)

(continued from previous page)

```
Var:
 [[ 4.11208477 -1.6739995 ]
 [-1.6739995  36.16386927]]
----------------------------------------------------------
Closed-Form
Mean:  10.162846250198633
Var:   4.112084770881207
==========================================================
```

The M-Estimator solves for $\hat{\theta}$ via a root finding procedure given the initial values in `init`. Since the variance must be > 0, we provide a positive initial value. For the sandwich variance, `delicatessen` uses a numerical approximation procedure for the bread matrix. This is different from the closed-form variance estimator provided in Chapter 7, but both should return approximately the same answer. The advantage of the numerically approximating the derivatives is that this process can be done for arbitrary estimating functions.

Notice that θ_2 also matches the first element of the (asymptotic) variance matrix. These two values should match (since they are estimating the same thing). Further, as shown the closed-form solutions for the mean and variance are equal to the M-Estimation approach.

The following uses the built-in estimating equation for the mean and variance

```
[4]: from delicatessen.estimating_equations import ee_mean_variance

def psi_mean_var_default(theta):
    """Built-in stacked estimating equations"""
    return ee_mean_variance(y=np.asarray(data['Y']), theta=theta)


estr = MEstimator(psi_mean_var_default, init=[0, 0])
estr.estimate()

print("==========================================================")
print("Mean & Variance")
print("==========================================================")
print("M-Estimation: built-in")
print("Theta:", estr.theta)
print("Var:  \n", estr.asymptotic_variance)
print("==========================================================")
```

```
==========================================================
Mean & Variance
==========================================================
M-Estimation: built-in
Theta: [10.16284625  4.11208477]
Var:
 [[ 4.11208477 -1.6739995 ]
 [-1.6739995  36.16386927]]
==========================================================
```

7.2.3 Ratio Estimator

Now consider if we wanted to estimate the ratio between two means. This can be expressed as a single estimating equation

$$\psi(Y_i, \theta) = \left[Y_i - X_i \times \theta_1\right]$$

and is implemented by-hand as

[5]:
```
def psi_ratio(theta):
    return data['Y'] - data['X']*theta


estr = MEstimator(psi_ratio, init=[0, ])
estr.estimate()

print("===========================================================")
print("Ratio Estimator")
print("===========================================================")
print("M-Estimation: single estimating equation")
print("Theta:", estr.theta)
print("Var:  ",estr.asymptotic_variance)
print("-----------------------------------------------------------")
print("Closed-Form")

theta = np.mean(data['Y']) / np.mean(data['X'])
b = 1 / np.mean(data['X'])**2
c = np.mean((data['Y'] - theta*data['X'])**2)
var = b * c
print("Ratio:",theta)
print("Var:  ",var)

print("===========================================================")
```

```
===========================================================
Ratio Estimator
===========================================================
M-Estimation: single estimating equation
Theta: [2.08234516]
Var:   [[0.33842324]]
-----------------------------------------------------------
Closed-Form
Ratio: 2.0823451609959682
Var:   0.33842329733168625
===========================================================
```

As you may notice, only a single initial value is provided (since only a single array is being returned). Furthermore, we provide an initial value > 0 since we are estimating a ratio.

However, there is another set of stacked estimating equations we can consider for the ratio. Specifically, we can estimate each of the means and then take the ratio of those means. Below is this alternative set of estimating equations

$$\psi(Y_i, \theta) = \begin{bmatrix} Y_i - \theta_1 \\ X_i - \theta_2 \\ \theta_1 - \theta_2\theta_3 \end{bmatrix}$$

Note that the last element is the ratio

[6]:
```python
def psi_ratio_three(theta):
    return (data['Y'] - theta[0],
            data['X'] - theta[1],
            np.ones(data.shape[0])*theta[0] - theta[1]*theta[2])


estr = MEstimator(psi_ratio_three, init=[0.1, 0.1, 0.1])
estr.estimate()

print("========================================================")
print("Ratio Estimator")
print("========================================================")
print("M-Estimation: three estimating equations")
print("Theta:", estr.theta)
print("Var:  \n", estr.asymptotic_variance)
print("========================================================")
```

```
========================================================
Ratio Estimator
========================================================
M-Estimation: three estimating equations
Theta: [10.16284625  4.88048112  2.08234516]
Var: 
 [[ 4.11208477  0.04326814  0.82409608]
 [ 0.04326814  0.95223639 -0.39742316]
 [ 0.82409608 -0.39742316  0.3384232 ]]
========================================================
```

Here, we used a trick to make sure the dimension of `ratio` stays as n, we use `np.ones`. Without multiplying by the array of ones, `ratio` would be a single value. However, `MEstimator` expects a $3 \times n$ array. Multiplying the 3rd equation by an array of 1's ensure the same dimension.

Also notice this form requires the use of 3 `init` values, unlike the other ratio estimator. As before, the ratio initial value is set > 0 to be nice to the root-finding algorithm.

7.2.4 Delta Method via M-Estimation

The delta method has been used in a variety of contexts, including estimating the variance for transformations of parameters. Instead of separately estimating the parameters, transforming the parameters, and then using the delta method to estimate the variance of the transformed parameters; we can apply the transformation in an estimating equation and automatically estimate the variance for the transformed parameter(s) via the sandwich variance estimator. To do this, we stack the estimating equation for the transformation into our set of estimating equations. Below is the mean-variance estimating equations stacked with two transformations of the variance

$$\psi(Y_i, \theta) = \begin{bmatrix} Y_i - \theta_1 \\ (Y_i - \theta_1)^2 - \theta_2 \\ \sqrt{\theta_2} - \theta_3 \\ \log(\theta_2) - \theta_4 \end{bmatrix}$$

These equations can be expressed programmatically as

[7]:
```python
def psi_delta(theta):
    return (data['Y'] - theta[0],
```

(continues on next page)

(continued from previous page)

```
            (data['Y'] - theta[0])**2 - theta[1],
            np.ones(data.shape[0])*np.sqrt(theta[1]) - theta[2],
            np.ones(data.shape[0])*np.log(theta[1]) - theta[3])


estr = MEstimator(psi_delta, init=[10., 2., 1., 1.])
estr.estimate()

print("========================================================")
print("Delta Method")
print("========================================================")
print("M-Estimation")
print("Theta:", estr.theta)
print("Var:  \n", estr.variance)
print("========================================================")
```

```
========================================================
Delta Method
========================================================
M-Estimation
Theta: [10.16284625  4.11208477  2.0278276   1.41393014]
Var:
 [[ 0.02056042 -0.00837    -0.00206379 -0.00203546]
 [-0.00837     0.18081935  0.04458452  0.04397267]
 [-0.00206379  0.04458452  0.01099318  0.01084232]
 [-0.00203546  0.04397267  0.01084232  0.01069352]]
========================================================
```

Notice the use of the np.ones trick to ensure that the final equations are the correct shapes. Here, there are 4 parameters, so init must be provided 4 values.

7.2.6 Instrumental Variable Estimation

Consider the following instrumental variable approach to correcting for measurement error of a variable. Here, Y is the outcome of interest, X is the true unmeasured variable, X^* is the possibly mismeasured variables, and I is the instrument for X.We are interested in estimating β_1 of

$$Y_i = \beta_0 + \beta_1 X_i + e_{i,j}$$

Since X^* is mismeasured, we can't immediately estimated β_1. Instead, we need to use an instrumental variable approach. Below is some generated data consistent with this measurment error story

[8]:
```
# Generating some data
n = 500
data = pd.DataFrame()
data['X'] = np.random.normal(size=n)
data['Y'] = 0.5 + 2*data['X'] + np.random.normal(loc=0, size=n)
data['X-star'] = data['X'] + np.random.normal(loc=0, size=n)
data['T'] = -0.75 - 1*data['X'] + np.random.normal(loc=0, size=n)
```

Two variations on the estimating equations for instrumental variable analyses are provided in the book. The first estimating equation is

$$\psi(Y_i, X_i^*, T_i, \theta) = \begin{bmatrix} T_i - \theta_1 \\ (Y_i - \theta_2 X_i^*)(\theta_1 - T_i) \end{bmatrix}$$

where θ_1 is the mean of the instrument, and θ_2 corresponds to β_1. The previous estimating equations can be translated as

[9]:
```python
def psi_instrument(theta):
    return (theta[0] - data['T'],
            (data['Y'] - data['X-star']*theta[1])*(theta[0] - data['T']))


estr = MEstimator(psi_instrument, init=[0.1, 0.1])
estr.estimate()

print("======================================================")
print("Instrumental Variable")
print("======================================================")
print("M-Estimation")
print("Theta:", estr.theta)
print("Var:  \n", estr.variance)
print("======================================================")
```

```
======================================================
Instrumental Variable
======================================================
M-Estimation
Theta: [-0.89989957  2.01777751]
Var:
 [[ 0.00430115 -0.0006694 ]
 [-0.0006694   0.023841  ]]
======================================================
```

As mentioned in the chapter, certain joint distributions may be of interest. To capture these additional distributions, the estimating equations were updated to

$$\psi(Y_i, X_i^*, T_i, \theta) = \begin{bmatrix} T_i - \theta_1 \\ \theta_2 - X_i^* \\ (Y_i - \theta_3 X_i^*)(\theta_2 - X_i^*) \\ (Y_i - \theta_4 X_i^*)(\theta_1 - T_i) \end{bmatrix}$$

This set of estimating equations further allows for inference on the difference between β_1 minus the coefficient for Y given X^*. Here, θ_1 is the mean of the instrument, θ_2 is the mean of the mismeasured value of X, and θ_3 corresponds to the coefficient for Y given X^*, and θ_4 is β_1.

Again, we can easily translate these equations for `delicatessen`

[10]:
```python
def psi(theta):
    return (theta[0] - data['T'],
            theta[1] - data['X-star'],
            (data['Y'] - data['X-star']*theta[2])*(theta[1] - data['X-star']),
            (data['Y'] - data['X-star']*theta[3])*(theta[0] - data['T'])
            )

```

(continues on next page)

(continued from previous page)

```
estr = MEstimator(psi, init=[0.1, 0.1, 0.1, 0.1])
estr.estimate()

print("=========================================================")
print("Instrumental Variable")
print("=========================================================")
print("M-Estimation")
print("Theta:", estr.theta)
print("Var:  \n", estr.variance)
print("=========================================================")
```

```
=========================================================
Instrumental Variable
=========================================================
M-Estimation
Theta: [-0.89989957  0.02117577  0.95717618  2.01777751]
Var:
 [[ 0.00430115 -0.00207361 -0.00011136 -0.0006694 ]
 [-0.00207361  0.0041239   0.00023703  0.00039778]
 [-0.00011136  0.00023703  0.00302462  0.00171133]
 [-0.0006694   0.00039778  0.00171133  0.023841  ]]
=========================================================
```

7.4 Nonsmooth Estimating Functions

7.4.1 Robust Location Estimation

To begin, we generate some generic data used for this example

[11]:
```
y = np.random.normal(size=250)
n = y.shape[0]
```

The robust location estimator reduces the influence of outliers by applying bounds. The robust mean proposed by Huber (1964) is

$$\psi(Y_i, \theta_1) = g_k(Y_i) - \theta_1$$

where k indicates the bound, such that if $Y_i > k$ then k, or $Y_i < -k$ then $-k$, otherwise Y_i.

Below is the estimating equation translated into code

[12]:
```
def psi_robust_mean(theta):
    k = 3                          # Bound value
    yr = np.where(y > k, k, y)     # Applying upper bound
    yr = np.where(y < -k, -k, y)   # Applying lower bound
    return yr - theta


estr = MEstimator(psi_robust_mean, init=[0.])
estr.estimate()

print("=========================================================")
```

(continues on next page)

(continued from previous page)

```
print("Robust Location")
print("=========================================================")
print("M-Estimation")
print("Theta:", estr.theta)
print("Var:  \n", estr.variance)
print("=========================================================")
```

```
=========================================================
Robust Location
=========================================================
M-Estimation
Theta: [0.03056108]
Var:
 [[0.00370521]]
=========================================================
```

Notice that the estimating equation here is not smooth (i.e., non-differentiable at k).

7.5 Regression M-estimators

7.5.1 Linear Model with Random X

For the next examples, the following simulated data is used

[13]:
```
n = 500
data = pd.DataFrame()
data['X'] = np.random.normal(size=n)
data['Z'] = np.random.normal(size=n)
data['Y'] = 0.5 + 2*data['X'] - 1*data['Z'] + np.random.normal(size=n)
data['C'] = 1
```

Here, we are interested in estimating the relationship between Y and X, Z. We will do this via linear regression. Note that we need to manually add an intercept (the column `C` in the data).

It is also worthwhile to note that the variance here is robust (to violations of the homoscedastic assumption). As comparison, we provide the equivalent using `statsmodels` generalized linear model with heteroscedastic-corrected variances.

As with all the preceding estimating equations, there are multiple ways to code these. Since linear regression involves matrix manipulations for the programmed estimating equations to return the correct format for `delicatessen`, we highlight two variations here.

First, we present a for-loop implementation of the estimating equation

[14]:
```
def psi(theta):
    # Transforming to arrays
    x = np.asarray(data[['C', 'X', 'Z']])
    y = np.asarray(data['Y'])
    beta = np.asarray(theta)[:, None]
    n = x.shape[0]

    # Where to store each of the resulting estimates
    est_vals = []
```

(continues on next page)

(continued from previous page)

```
    # Looping through each observation
    for i in range(n):
        v_i = (y[i] - np.dot(x[i], beta)) * x[i]
        est_vals.append(v_i)

    # returning 3-by-n object
    return np.asarray(est_vals).T

mestimator = MEstimator(psi, init=[0.1, 0.1, 0.1])
mestimator.estimate()

print("=========================================================")
print("Linear Model")
print("=========================================================")
print("M-Estimation: by-hand")
print(mestimator.theta)
print(mestimator.variance)
print("---------------------------------------------------------")

print("GLM Estimator")
glm = smf.glm("Y ~ X + Z", data).fit(cov_type="HC1")
print(np.asarray(glm.params))
print(np.asarray(glm.cov_params()))
print("=========================================================")
```

```
=========================================================
Linear Model
=========================================================
M-Estimation: by-hand
[ 0.41082601  1.96289222 -1.02663555]
[[ 2.18524086e-03  7.28169364e-05  1.54216649e-04]
 [ 7.28169364e-05  2.08315683e-03 -4.09520393e-05]
 [ 1.54216649e-04 -4.09520393e-05  2.14573772e-03]]
---------------------------------------------------------
GLM Estimator
[ 0.41082601  1.96289222 -1.02663555]
[[ 2.18524092e-03  7.28169947e-05  1.54216630e-04]
 [ 7.28169947e-05  2.08315690e-03 -4.09519947e-05]
 [ 1.54216630e-04 -4.09519947e-05  2.14573770e-03]]
=========================================================
```

As the second approach, a vectorized version is used

[15]:
```
def psi_regression(theta):
    x = np.asarray(data[['C', 'X', 'Z']])
    y = np.asarray(data['Y'])[:, None]
    beta = np.asarray(theta)[:, None]
    return ((y - np.dot(x, beta)) * x).T


mestimator = MEstimator(psi_regression, init=[0.1, 0.1, 0.1])
mestimator.estimate()
```

(continues on next page)

(continued from previous page)

```
print("==========================================================")
print("Linear Model")
print("==========================================================")
print("M-Estimation: by-hand")
print(mestimator.theta)
print(mestimator.variance)
print("==========================================================")
```

```
==========================================================
Linear Model
==========================================================
M-Estimation: by-hand
[ 0.41082601  1.96289222 -1.02663555]
[[ 2.18524113e-03  7.28169589e-05  1.54216655e-04]
 [ 7.28169589e-05  2.08315685e-03 -4.09520174e-05]
 [ 1.54216655e-04 -4.09520174e-05  2.14573766e-03]]
==========================================================
```

While these two approaches give the same answer, vectorized versions will generally be faster than for-loop variations (but may be less 'human readable'). Having said that, it is easier to make a mistake with a vectorized version. We would generally recommend creating a for-loop version first (and then creating a vectorized version if that for-loop is too slow).

The following uses the built-in linear regression functionality (which uses the vectorized implementation)

[16]:
```
from delicatessen.estimating_equations import ee_regression

def psi_regression(theta):
    return ee_regression(theta=theta,
                         X=data[['C', 'X', 'Z']],
                         y=data['Y'],
                         model='linear')


mestimator = MEstimator(psi_regression, init=[0.1, 0.1, 0.1])
mestimator.estimate()

print("==========================================================")
print("Linear Model")
print("==========================================================")
print("M-Estimation: built-in")
print(mestimator.theta)
print(mestimator.variance)
print("==========================================================")
```

```
==========================================================
Linear Model
==========================================================
M-Estimation: built-in
[ 0.41082601  1.96289222 -1.02663555]
[[ 2.18524113e-03  7.28169589e-05  1.54216655e-04]
 [ 7.28169589e-05  2.08315685e-03 -4.09520174e-05]
 [ 1.54216655e-04 -4.09520174e-05  2.14573766e-03]]
```

(continues on next page)

(continued from previous page)

```
==============================================================
```

7.5.4 Robust Regression

The next example is robust regression, where the standard linear regression model is made robust to outliers. We use :math:`f_k()` from 7.4.1 but now apply it to the residuals of the regression model.

[17]:
```
def psi_robust_regression(theta):
    k = 1.345
    x = np.asarray(data[['C', 'X', 'Z']])
    y = np.asarray(data['Y'])[:, None]
    beta = np.asarray(theta)[:, None]
    preds = np.clip(y - np.dot(x, beta), -k, k)
    return (preds * x).T


mestimator = MEstimator(psi_robust_regression, init=[0.5, 2., -1.])
mestimator.estimate()

print("==============================================================")
print("Robust Linear Model")
print("==============================================================")
print("M-Estimation: by-hand")
print(mestimator.theta)
print(mestimator.variance)
print("==============================================================")
```

```
==============================================================
Robust Linear Model
==============================================================
M-Estimation: by-hand
[ 0.41223641  1.95577495 -1.02508413]
[[ 2.31591830e-03  1.82105969e-04  2.57209766e-04]
 [ 1.82105969e-04  2.12098818e-03 -6.95783670e-05]
 [ 2.57209766e-04 -6.95783670e-05  2.38212585e-03]]
==============================================================
```

The following uses the built-in robust linear regression functionality

[18]:
```
from delicatessen.estimating_equations import ee_robust_regression

def psi_robust_regression(theta):
    return ee_robust_regression(theta=theta,
                                X=data[['C', 'X', 'Z']],
                                y=data['Y'],
                                model='linear',
                                loss='huber', k=1.345)

mestimator = MEstimator(psi_robust_regression, init=[0.5, 2., -1.])
mestimator.estimate()
```

(continues on next page)

(continued from previous page)

```
print("=========================================================")
print("Robust Linear Model")
print("=========================================================")
print("M-Estimation: built-in")
print(mestimator.theta)
print(mestimator.variance)
print("=========================================================")
```

```
=========================================================
Robust Linear Model
=========================================================
M-Estimation: built-in
[ 0.41223641  1.95577495 -1.02508413]
[[ 2.31591830e-03  1.82105969e-04  2.57209766e-04]
 [ 1.82105969e-04  2.12098818e-03 -6.95783670e-05]
 [ 2.57209766e-04 -6.95783670e-05  2.38212585e-03]]
=========================================================
```

You'll notice that the coefficients have changed slightly here. That is because we have reduced the extent of outliers on the estimation of the linear regression parameters (however, our simulated data mechanism doesn't really result in major outliers, so the change is small here).

7.5.5 Generalized Linear Models

The last category of models consider here are generalized linear models (GLMs). These models are much more flexible than the previous regression models in terms of the distributions being assumed. To illustrate this, we will use GLMs to estimate the odds ratio, risk ratio, and risk difference (since the book does not provide specific examples).

The following is some generic data for the example

```
[19]: d = pd.DataFrame()
d['X'] = [1, -1, 0, 1, 2, 1, -2, -1, 0, 3, -3, 1, 1, -1, -1, -2, 2, 0, -1, 0]
d['Z'] = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
d['Y'] = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0]
d['I'] = 1
```

For the subsequent examples, we will use the built-in GLM functionality.

Odds Ratio

To estimate odds ratios, we will use logistic regression. Logistic regression can be implemented through a GLM with a binomial distribution and the logit link. Below is the implemenentation

```
[20]: from delicatessen.estimating_equations import ee_glm

def psi(theta):
    return ee_glm(theta, X=d[['I', 'X', 'Z']], y=d['Y'],
                  distribution='binomial', link='logit')

mestr = MEstimator(psi, init=[0., 0., 0.])
mestr.estimate()
```

(continues on next page)

(continued from previous page)

```
print("==========================================================")
print("Logistic")
print("==========================================================")
print("ln(OR):", mestr.theta)
print("OR:    ", np.exp(mestr.theta))
print("==========================================================")
```

```
==========================================================
Logistic
==========================================================
ln(OR): [-0.34613077  0.16182415  0.28150058]
OR:     [0.70741997 1.17565348 1.32511677]
==========================================================
```

To estimate the risk ratios, we can use a log-binomial regression model. This can be implemented through a GLM with the binomial distribution and the log identity. Below is code to do this

[21]:
```
def psi(theta):
    return ee_glm(theta, X=d[['I', 'X', 'Z']], y=d['Y'],
                  distribution='binomial', link='log')

mestr = MEstimator(psi, init=[-.9, 0., 0.])
mestr.estimate(solver='lm')

print("==========================================================")
print("Log-binomial")
print("==========================================================")
print("ln(RR):", mestr.theta)
print("RR:    ", np.exp(mestr.theta))
print("==========================================================")
```

```
==========================================================
Log-binomial
==========================================================
ln(RR): [-0.89140821  0.06939184  0.16163837]
RR:     [0.41007787 1.07185612 1.1754351 ]
==========================================================
```

As seen here, the odds ratio and risk ratios differ. This is expected since the outcome is not that rare.

Note: the log-binomial can be a bit difficult to estimate as it is not bounded (but the log-RR is bounded). What this means is that decent starting values might need to be provided. As seen here, the intercept is given a good starting value.

Finally, we can estimate the risk difference via linear-binomial regression. This can be done with the binomial distribution but with the identity link. Below is code for this model

[22]:
```
def psi(theta):
    return ee_glm(theta, X=d[['I', 'X', 'Z']], y=d['Y'],
                  distribution='binomial', link='identity')

mestr = MEstimator(psi, init=[.2, 0., 0.])
mestr.estimate(solver='lm')
```

(continues on next page)

(continued from previous page)

```
print("=========================================================")
print("Risk difference")
print("=========================================================")
print("RD:     ", mestr.theta)
print("=========================================================")
```

```
=========================================================
Risk difference
=========================================================
RD:     [0.41621376 0.03456562 0.06753619]
=========================================================
```

Like the log-binomial model this too can be difficult to fit since the risk differences are bounded but the model is not. This is especially a concern for the linear-binomial model, since we need to ensure the starting values don't produce risk differences outside the range of $[-1, 1]$. To ensure this, a starting value of 0.2 was used for the intercept.

This concludes the replication of the examples provided in Boos & Stefanski (2002).

3.2.5 Bonate (2011): Pharmacokinetic-Pharmacodynamic Modeling and Simulations

Here, we replicate some of the examples described in Bonate (2011). I recommend following along with the 2nd edition of the book. The purpose of this notebook is to illustrate the versatility of `delicatessen` by illustrating its application for pharmacokinetic modeling. This can easily be done by using the built-in estimating equations, as will be shown.

Bonate PL. (2011). Pharmacokinetic-Pharmacodynamic Modeling and Simulations. 2nd Edition. Springer, New York, NY.

Setup

[1]:
```
import numpy as np
import scipy as sp
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

import delicatessen
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_glm

np.random.seed(80950841)

print("NumPy version:       ", np.__version__)
print("SciPy version:       ", sp.__version__)
print("Pandas version:      ", pd.__version__)
print("Delicatessen version:", delicatessen.__version__)
```

```
NumPy version:        1.25.2
SciPy version:        1.11.2
Pandas version:       1.4.1
Delicatessen version: 2.2
```

Chapter 11: Generalized Linear Models and Its Extensions

Adverse Events Case Study

The first example from Chapter 11 is the *Case Study: Assessing the Relationship Between Drug Concentrations and Adverse Events Using Logistic Regression*. Data comes from Table 2 of the book. In the book, a variety of different models for different adverse events are considered. Here, we only consider nausea (and vomiting) by AUC. For the one observations with a missing AUC value, they are dropped from the data set (same as the book). For ease of examining the coefficients, we will also divide the AUC value by 1000. This means the coefficients for AUC are rescaled from those reported in the book.

First, we load the data set and transform the coefficients and add an intercept column to the data set

[2]:
```
d = pd.read_csv("data/bonate.csv").dropna()
d['intercept'] = 1   # Adding intercept to data
d['auc'] /= 1000     # Rescaling AUC
d['c_max'] /= 1000   # Rescaling C_max
d.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 42 entries, 0 to 42
Data columns (total 12 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   id         42 non-null     int64
 1   c_max      42 non-null     float64
 2   auc        42 non-null     float64
 3   age        42 non-null     int64
 4   sex        42 non-null     int64
 5   ps         42 non-null     int64
 6   myalgia    42 non-null     int64
 7   phlebitis  42 non-null     int64
 8   asthenia   42 non-null     int64
 9   diarrhea   42 non-null     int64
 10  nausea     42 non-null     int64
 11  intercept  42 non-null     int64
dtypes: float64(2), int64(10)
memory usage: 4.3 KB
```

[3]:
```
table = pd.DataFrame(columns=["Model", "Intercept", "AUC", "Sex", "Age", "PS"])
```

To begin, we will fit a null (intercept-only) logistic regression model. This is easily done by using the built-in `ee_glm` estimating equation. For the logistic model, we specify a binomial distribution with the logit link. Below is code to setup the estimating equation and then estimate the parameters using `MEstimator`

[4]:
```
def psi(theta):
    # Estimating equation for null model
    return ee_glm(theta=theta,
                  y=d['nausea'],
                  X=d[['intercept', ]],
                  distribution='binomial',
                  link='logit')
```

(continues on next page)

(continued from previous page)

```
# Estimate the parameters of the logit model
estr_null = MEstimator(psi, init=[0., ])
estr_null.estimate()

# Adding results to the output table
table.loc[len(table)] = ["Null", estr_null.theta[0], ] + [np.nan, ]*4
```

Next we fit a logistic regression model that includes linear terms for all the independent variables in the data set. This is easily done by modifying the previous design matrix (i.e., `X`). Below is code to fit the full model

[5]:
```
def psi(theta):
    # Estimating equation for full model
    return ee_glm(theta=theta,
                  y=d['nausea'],
                  X=d[['intercept', 'auc', 'sex', 'age', 'ps']],
                  distribution='binomial',
                  link='logit')


# Estimate the parameters of the logit model
estr_full = MEstimator(psi, init=[0., ]*5)
estr_full.estimate()

# Adding results to the output table
table.loc[len(table)] = ["Full", ] + list(estr_full.theta)
```

In the book, Bonate performs some variable selection. In general, we would not recommend use of backwards-selection procedures (like those done in the book). Such procedures complicate inference (P-values and confidence intervals after these procedures are no longer valid). For comparison purposes, we estimate the reduced model reported in the book. Again, this is easily done by modifying the `X` argument for `ee_glm`

[6]:
```
def psi(theta):
    # Estimating equation for reduced model
    return ee_glm(theta=theta,
                  y=d['nausea'],
                  X=d[['intercept', 'auc', 'sex']],
                  distribution='binomial',
                  link='logit')


# Estimate the parameters of the logit model
estr_redu = MEstimator(psi, init=[0., ]*3)
estr_redu.estimate()

# Adding results to the output table
table.loc[len(table)] = ["Reduced", ] + list(estr_redu.theta) + [np.nan, ]*2
```

Finally, two alternative models are considered: a probit regression model and a complimentary log-log model. Again, these models are easily implemented using `ee_glm`. For the probit model, we set the link equal to `probit`

[7]:
```
def psi(theta):
    # Estimating equation for reduced probit model
    return ee_glm(theta=theta,
```

(continues on next page)

(continued from previous page)

```
                  y=d['nausea'],
                  X=d[['intercept', 'auc', 'sex']],
                  distribution='binomial',
                  link='probit')


# Estimate the parameters of the probit model
estr_prob = MEstimator(psi, init=[0., ]*3)
estr_prob.estimate()

# Adding results to the output table
table.loc[len(table)] = ["Probit", ] + list(estr_prob.theta) + [np.nan, ]*2
```

Similarly, the complimentary log-log model only requires setting the link to `cloglog`

[8]:
```
def psi(theta):
    # Estimating equation for reduced C-log-log model
    return ee_glm(theta=theta,
                  y=d['nausea'],
                  X=d[['intercept', 'auc', 'sex']],
                  distribution='binomial',
                  link='cloglog')


# Estimate the parameters of the cloglog model
estr_clog = MEstimator(psi, init=[0., ]*3)
estr_clog.estimate()

# Adding results to the output table
table.loc[len(table)] = ["Probit", ] + list(estr_clog.theta) + [np.nan, ]*2
```

Now we can view the results across the different models

[9]:
```
table.set_index("Model")
```

[9]:
```
         Intercept       AUC       Sex       Age        PS
Model
Null     -0.587787       NaN       NaN       NaN       NaN
Full     -5.602899  0.289785  1.730299  0.049538  0.220545
Reduced  -2.663522  0.303502  1.772238       NaN       NaN
Probit   -1.629729  0.184030  1.080253       NaN       NaN
Probit   -2.233720  0.193108  1.212290       NaN       NaN
```

These point estimates match those reported in Tables 3 and 4 in the book (note the null model differs slightly, since we dropped the one observation with the missing AUC value to fit this model, but the book does not). These results highlight how `delicatessen` allows one to easily fit a variety of different models.

END

This is the end of the current replication.

3.2.6 Hernan & Robins (2023): Causal Inference with Models

The following replicates selected examples from the textbook "Causal Inference: What If" by Hernan and Robins. These replications focus on Part II of the textbook. I recommend reading Part I prior to looking through the following code. It is a great, approachable, and freely available resource.

Here, we demonstrate application of `delicatessen` for causal inference. Throughout, we use the sandwich variance estimator. This is not described in the textbook, but is an alternative to the bootstrap. Importantly, it is a consistent estimator of the variance that is computationally simpler (in terms of the computers computational effort). `delicatessen` automates the whole procedure, so it is also simpler for us.

Broadly, interest is in estimating the average causal effect of stopping smoking (variable name: `qsmk`) on 10-year weight change (variable name: `wt82_71`). If we let Y^a indicate the potential weight change under smoking status a, then the average causal effect can be written as

$$E[Y^1] - E[Y^0]$$

Herafter, we assume that the interest parameter is identified (see the book for what this means). Our focus will be on estimators for this quantity.

Setup

Data for these replications is available at the following Harvard School of Public Health website. Data comes from the National Health and Nutrition Examination Survey Data I Epidemiologic Follow-up Study (NHEFS). The NHEFS was jointly initiated by the National Center for Health Statistics and the National Institute on Aging in collaboration with other agencies of the United States Public Health Service. A detailed description of the NHEFS, together with publicly available data sets and documentation, can be found at wwwn.cdc.gov/nchs/nhanes/nhefs/

The data set used in the book and this tutorial is a subset of the full NHEFS. First, we will load the data and run some basic variable manipulations.

[1]:
```python
import numpy as np
import scipy as sp
import pandas as pd

import delicatessen as deli
from delicatessen import MEstimator
from delicatessen.estimating_equations import (ee_regression, ee_glm, ee_ipw_msm,
                                               ee_gformula, ee_gestimation_snmm)
from delicatessen.utilities import inverse_logit, regression_predictions

print("Versions")
print("NumPy:        ", np.__version__)
print("SciPy:        ", sp.__version__)
print("pandas:       ", pd.__version__)
print("Delicatessen: ", deli.__version__)
```

```
Versions
NumPy:         1.25.2
SciPy:         1.11.2
```

(continues on next page)

(continued from previous page)

```
pandas:        1.4.1
Delicatessen:  2.1
```

[2]:
```
df = pd.read_csv('data/nhefs.csv')
df.dropna(subset=['sex', 'age', 'race', 'ht',
                  'school', 'alcoholpy', 'smokeintensity'],
         inplace=True)

# recoding some variables
df['inactive'] = np.where(df['active'] == 2, 1, 0)
df['no_exercise'] = np.where(df['exercise'] == 2, 1, 0)
df['university'] = np.where(df['education'] == 5, 1, 0)

# Subsetting only variables of interest
df = df[['wt82_71', 'qsmk', 'sex', 'age', 'race', 'wt71', 'wt82', 'ht',
         'school', 'alcoholpy', 'smokeintensity', 'smokeyrs', 'smkintensity82_71',
         'education', 'exercise', 'active', 'death']]

# creating quadratic terms
for col in ['age', 'wt71', 'smokeintensity', 'smokeyrs']:
    df[col+'_sq'] = df[col] * df[col]

df['I'] = 1

# Indicator terms
df['educ_2'] = np.where(df['education'] == 2, 1, 0)
df['educ_3'] = np.where(df['education'] == 3, 1, 0)
df['educ_4'] = np.where(df['education'] == 4, 1, 0)
df['educ_5'] = np.where(df['education'] == 5, 1, 0)
df['exer_1'] = np.where(df['exercise'] == 1, 1, 0)
df['exer_2'] = np.where(df['exercise'] == 2, 1, 0)
df['active_1'] = np.where(df['active'] == 1, 1, 0)
df['active_2'] = np.where(df['active'] == 2, 1, 0)

# Interaction terms
df['qsmk_smkint'] = df['qsmk'] * df['smokeintensity']

# Complete-case data
dc = df.dropna(subset=['wt82_71']).copy()
```

Chapter 12: IP weighting and marginal structural models

The first estimation approach is inverse probability weighting. Inverse probability weights are defined as

$$\frac{1}{\Pr(A = a|W)}$$

where W is the set of confounders. We will estimate these weights and then use them to estimate the parameters of a marginal structural model. An example of a marginal structural model is

$$E[Y^a] = \alpha_0 + \alpha_1 a$$

where α are the parameters to estimate. Here, α_1 represents the average causal effect. We will estimate the marginal structural model using the observed data and a regression model weighted by the inverse probability weights (see the books for details).

12.1: The causal question

Chapter 12 starts out with estimating the crude association between qsmk and wt82_71 (12.1). We will do this by fitting a linear regression model with delicatessen.

```
[3]: def psi_ols(theta):
         return ee_regression(theta=theta, X=dc[['I', 'qsmk']],
                              y=dc['wt82_71'], model='linear')
```

```
[4]: estr = MEstimator(psi_ols, init=[0., 0.])
     estr.estimate(solver='hybr')
```

```
[5]: print("Point:", estr.theta[1])
     print("95% CI:", estr.confidence_intervals()[1, :])
```

```
Point: 2.540581454955888
95% CI: [1.58620716 3.49495575]
```

The book reports an estimate of 2.5 (95% CI: 1.7, 3.4)

You may notice that the confidence interval differs slightly. That is because we are using the sandwich variance (the book does not). While the variance estimators used here and in the book are expected to be asymptotically equivalent in this case, they can produce different results in finite samples as we see here.

12.2: Estimating inverse probability weights via modeling

Now we will estimate the parameters of the marginal structural model using unstabilized inverse probability weights.

To do this with delicatessen, we will define the corresponding design matrices. Then we will define the stacked estimating equations. Then we will estimate the parameters and covariance using MEstimator and present the output.

```
[6]: # Design matrix for the propensity score model
     W = dc[['I', 'sex', 'race', 'age', 'age_sq',
             'educ_2', 'educ_3', 'educ_4', 'educ_5',
             'smokeintensity', 'smokeintensity_sq',
             'smokeyrs', 'smokeyrs_sq',
             'exer_1', 'exer_2', 'active_1', 'active_2',
             'wt71', 'wt71_sq']]
     # Design matrix for the marginal structural model
     msm = dc[['I', 'qsmk']]
     # treatment variable
     a = dc['qsmk']
     # outcome variable
     y = dc['wt82_71']
```

```
[7]: def psi_ipw_msm1(theta):
         # Dividing parameters into corresponding estimation equations
         alpha = theta[0:2]
```

(continues on next page)

(continued from previous page)

```
    beta = theta[2:]

    # Estimating the propensity scores
    ee_ps = ee_regression(theta=beta,          # Estimate propensity scores
                          X=W, y=a,            # ... given observed A,W
                          model='logistic')    # ... with logit model
    pi = inverse_logit(np.dot(W, beta))        # Get Pr(A = 1 | W)
    ipw = 1 / np.where(a == 1, pi, 1-pi)       # Convert to IPW

    # Estimating the MSM using a weighted linear model
    ee_msm = ee_regression(theta=alpha,        # MSM parameters
                           X=msm, y=y,         # ... observed data
                           model='linear',     # ... with linear model
                           weights=ipw)        # ... but weighted by IPW

    # Stacking the estimating equations and returning
    return np.vstack([ee_msm, ee_ps])
```

[8]:
```
# M-estimator
init_vals = [0., 0., ] + [0.,]*W.shape[1]
estr = MEstimator(psi_ipw_msm1, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[9]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:2])
print("95% CI")
print(ci[0:2, :].T)
```

```
[1.77997819 3.44053543]
95% CI
[[1.35249881 2.48589079]
 [2.20745757 4.39518006]]
```

which are the estimates of the α_0 and α_1 parameters. The book provides the point estimate for `qsmk` ($\hat{\alpha}_1$) as 3.4 (95% CI: 2.4, 4.5).

The point estimates presented here are the same as the book. However, the confidence intervals differ slightly. The confidence intervals in the book use the 'GEE trick' which provides overly conservative confidence intervals. With `delicatessen` and the sandwich variance, we can estimate the variance *without being overly conservative*. So, the variance reported above would be preferred over the approach described in the book. This highlights the advantage of M-estimators for computation of the variance with nuisance parameters (like the IPW estimator when the propensity score is estimated).

Instead of coding this by-hand, we can also use the built-in `ee_ipw_msm` function. This function estimates a marginal structural model using inverse probability weights, as done above. Below is how to apply this functionality

[10]:
```
def psi_ipw_msm1a(theta):
    # Built-in estimating equation
    return ee_ipw_msm(theta, y=y, A=a, W=W, V=msm,
                      distribution='normal',
                      link='identity')
```

[11]:
```
# M-estimator
init_vals = [0., 0., ] + [0.,]*W.shape[1]
estr = MEstimator(psi_ipw_msm1a, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[12]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:2])
print("95% CI")
print(ci[0:2, :].T)
```

```
[1.77997819 3.44053543]
95% CI
[[1.35249881 2.48589079]
 [2.20745757 4.39518006]]
```

As expected, this built-in functionality produces the same results as the by-hand version.

12.3: Stabilized inverse probability weights

Next, we are going to use stabilized weights. The stabilized weights will require us to estimate an additional parameter. We will accomplish this by stacking an estimating equation for that parameter. This extra estimating equation is for an intercept-only model for the probability of `qsmk`.

[13]:
```
def psi_ipw_msm2(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[0:2]                 # MSM parameters
    gamma = np.array([theta[2], ])     # Numerator parameter
    beta = theta[3:]                   # Propensity score parameters

    # Estimating the propensity scores using a logit model
    ee_ps = ee_regression(theta=beta,        # Propensity score model
                          X=W, y=a,          # ... with observed data
                          model='logistic')  # ... and logit model
    pi = inverse_logit(np.dot(W, beta))      # Predicted probability of A=1

    # Estimating intercept-only for numerator
    ee_num = ee_regression(theta=gamma,            # Numerator model
                           X=dc[['I']], y=a,       # ... intercept-only
                           model='logistic')       # ... logit model
    num = inverse_logit(np.dot(dc[['I']], gamma))  # Marginal probability

    # Construct stabilized weights
    ipw = np.where(a == 1, num/pi, (1-num)/(1-pi))

    # Estimating the MSM using a weighted linear model
    ee_msm = ee_regression(theta=alpha,     # MSM
                           X=msm, y=y,      # ... with observed data
                           model='linear',  # ... linear
                           weights=ipw)     # ... weighted by stabilized

    # Stacking the estimating equations and returning
    return np.vstack([ee_msm, ee_num, ee_ps])
```

[14]:
```
# M-estimator
init_vals = [0., 0., ] + [0., ] + [0.,]*W.shape[1]
estr = MEstimator(psi_ipw_msm2, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[15]:
```
ci = estr.confidence_intervals()
print("Intercept:", estr.theta[0])
print("95% CI:", ci[0, :])
print("qsmk:", estr.theta[1])
print("95% CI:", ci[1, :])
```

```
Intercept: 1.77997819053316
95% CI: [1.35249875 2.20745763]
qsmk: 3.44053542964726
95% CI: [2.48589062 4.39518024]
```

The estimate is the same in the previous section. This is expected because as long as the marginal structural model is saturated, the unstabilized and stabilized IPTW should produce the same answer. (note there are some differences out in the smaller decimals places, but this is due to floating point errors, not differences between the estimators).

Note: `ee_ipw_msm` does not support the computation of stabilized weights (results are equivalent in this setting, so we can be more computationally efficient by opting for the unstabilized weights).

12.4: Marginal structural models

Now we will consider the IPW estimator for a continuous action. We will look at `smokeintensity` on `wt82_71`.

[16]:
```
# Restricting data by smoking intensity
ds = dc.loc[dc['smokeintensity'] <= 25].copy()

# Design matrix for the propensity score model
W = ds[['I', 'sex', 'race', 'age', 'age_sq',
        'educ_2', 'educ_3', 'educ_4', 'educ_5',
        'smokeintensity', 'smokeintensity_sq',
        'smokeyrs', 'smokeyrs_sq',
        'exer_1', 'exer_2', 'active_1', 'active_2',
        'wt71', 'wt71_sq']]
# Design matrix for the marginal structural model
ds['smkint_sq'] = ds['smkintensity82_71']**2
msm = ds[['I', 'smkintensity82_71', 'smkint_sq']]
# Treatment array
a = ds['smkintensity82_71']
# Outcome array
y = ds['wt82_71']
```

[17]:
```
def psi_ipw_msm3(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[0:3]                # Marginal structural model
    gamma = np.array([theta[3], ])    # Numerator
    beta = theta[4:]                  # Propensity score model
    div_ps = W.shape[0] - len(beta)   # Divisor for PS SD
    div_nm = W.shape[0] - len(gamma)  # Divisor for Num SD
```

(continues on next page)

(continued from previous page)

```
    # Estimating the propensity scores using a logit model
    ee_ps = ee_regression(theta=beta,            # Generalized PS model
                          X=W, y=a,              # ... for observed data
                          model='linear')        # ... linear regression
    mu = np.dot(W, beta)                         # Predicted values
    mu_resid = np.sum((a - mu)**2) / div_ps      # Standard deviation
    fAL = sp.stats.norm.pdf(a, mu,               # PDF from normal
                            np.sqrt(mu_resid))   # ... with estimates

    # Estimating intercept-only for numerator
    ee_num = ee_regression(theta=gamma,          # Numerator for stabilized
                           X=ds[['I', ]], y=a,   # ... for observed data
                           model='linear')       # ... linear regression
    num = np.dot(ds[['I', ]], gamma)             # Predicted values
    num_resid = np.sum((a - num)**2) / div_nm    # Standard deviation
    fA = sp.stats.norm.pdf(a, num,               # PDF from normal
                           np.sqrt(num_resid))   # ... with estimates

    # Stabilized weights
    ipw = fA / fAL

    # Estimating the MSM using a weighted linear model
    ee_msm = ee_regression(theta=alpha,      # Marginal structural model
                           X=msm, y=y,       # ... observed data
                           model='linear',   # ... linear model
                           weights=ipw)      # ... weighted by IPW

    # Stacking the estimating equations and returning
    return np.vstack([ee_msm, ee_num, ee_ps])
```

```
[18]: # M-estimator
init_vals = [0., 0., 0., ] + [0., ] + [0.,]*W.shape[1]
estr = MEstimator(psi_ipw_msm3, init=init_vals)
estr.estimate(solver='hybr', tolerance=1e-12, maxiter=5000)
```

```
[19]: ci = estr.confidence_intervals()
print(estr.theta[0:3])
print("95% CI:")
print(ci[0:3, :].T)
```

```
[ 2.00452474 -0.10898888  0.00269494]
95% CI:
[[ 1.44058085e+00 -1.66863862e-01 -1.75081112e-03]
 [ 2.56846862e+00 -5.11138978e-02  7.14069266e-03]]
```

The book reports coefficients of: 2.005, 0.109, 0.003. This matches the output shown above.

As done in the book, we want to know the weight change for no change in smoking intensity and a +20 in smoking intensity. Rather than add corresponding estimation equations for those parts, we can directly manipulate the point and covariance estimates to get this. The function `regression_predictions` will do this for us given the estimated parameters and the values of interest

[20]:
```python
# Creating dataframe for combinations to predict
p = pd.DataFrame()
p['smkintensity82_71'] = [0, 20]
p['smkint_sq'] = [0**2, 20**2]
p['I'] = 1
vals = np.asarray(p[['I', 'smkintensity82_71', 'smkint_sq']])

# Getting predicted values and variance for combinations
pred_y = regression_predictions(vals,
                                estr.theta[0:3],
                                estr.variance[0:3, 0:3])

# Displaying results
print("No change in smoking")
print(pred_y[0, 0])
print("95% CI:", pred_y[0, 2:])
print()
print("+20 smoking intensity")
print(pred_y[1, 0])
print("95% CI:", pred_y[1, 2:])
```

```
No change in smoking
2.004524735075991
95% CI: [1.44058085 2.56846862]

+20 smoking intensity
0.9027234481603187
95% CI: [-1.49966211  3.305109  ]
```

Again, we get similar results to those reported in the book: 2.0 (95% CI: 1.4, 2.6) and 0.9 (95% CI: 1.7, 3.5). However, our confidence intervals are slightly more narrow. Again this would be expected since we are using a variance estimator that is not overly conservative.

Note: `ee_ipw_msm` does not support non-binary treatments. Allowing for generic distributions for the generalized propensity score model is difficult. So, you will need to implement them by-hand (using a similar approach to above).

Binary outcome

We now repeat the process, but for a binary outcome and treatment. We will use a GLM with the binomial distribution and logistic link (this will estimate the causal odds ratio). This is avaible in `ee_glm`. We will also be using `qsmk` again.

[21]:
```python
# Design matrix for propensity scores
W = dc[['I', 'sex', 'race', 'age', 'age_sq',
        'educ_2', 'educ_3', 'educ_4', 'educ_5',
        'smokeintensity', 'smokeintensity_sq',
        'smokeyrs', 'smokeyrs_sq',
        'exer_1', 'exer_2', 'active_1', 'active_2',
        'wt71', 'wt71_sq']]
# Design matrix for marginal structural model
msm = dc[['I', 'qsmk']]
# Treatment array
a = dc['qsmk']
```

(continues on next page)

(continued from previous page)

```
# Outcome array
y = dc['death']
```

[22]:
```
def psi_ipw_msm4(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[0:2]              # Marginal structural model
    gamma = np.array([theta[2], ])  # Numerator model
    beta = theta[3:]                # Propensity score model

    # Estimating the propensity scores using a logit model
    ee_ps = ee_regression(theta=beta,         # Propensity score
                          X=W, y=a,           # ... observed data
                          model='logistic')   # ... logistic model
    pi = inverse_logit(np.dot(W, beta))       # Predicted probability

    # Estimating intercept-only for numerator
    ee_num = ee_regression(theta=gamma,            # Numerator model
                           X=dc[['I']], y=a,       # ... observed data
                           model='logistic')       # ... logit model
    num = inverse_logit(np.dot(dc[['I']], gamma)) # Predicted prob

    # Stabilized inverse probability weights
    ipw = np.where(a == 1, num/pi, (1-num)/(1-pi))

    # Estimating the MSM using a weighted linear model
    ee_msm = ee_glm(theta=alpha,                # MSM
                    X=msm, y=y,                 # ... observed data
                    link='logit',               # ... logit link
                    distribution='binomial',    # ... binomial dist
                    weights=ipw)                # ... weighted

    # Stacking the estimating equations and returning
    return np.vstack([ee_msm, ee_num, ee_ps])
```

[23]:
```
# M-estimator
init_vals = [0., 0., ] + [0., ] + [0.,]*W.shape[1]
estr = MEstimator(psi_ipw_msm4, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[24]:
```
ci = estr.confidence_intervals()
print("qsmk:", np.exp(estr.theta[1]))
print("95% CI:", np.exp(ci[1, :]))
```

```
qsmk: 1.030577892676402
95% CI: [0.78938458 1.34546687]
```

The previous results are for the causal odds ratio. Again, they are similar to the book with slight differences in the confidence intervals (i.e., 1.0; 95% CI: 0.8, 1.4).

We can replicate this approach using `ee_ipw_msm`. To simplify the process, I am going to use the previous propensity score model parameters as the starting values.

[25]:
```
init_ps = list(estr.theta[3:])
```

[26]:
```
def psi_ipw_msm4a(theta):
    # Built-in estimating equation
    return ee_ipw_msm(theta, y=y, A=a, W=W, V=msm,
                      distribution='binomial',
                      link='logit')
```

[27]:
```
# M-estimator
init_vals = [0., 0., ] + init_ps
estr = MEstimator(psi_ipw_msm4a, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[28]:
```
ci = estr.confidence_intervals()
print(np.exp(estr.theta[0:2]))
print("95% CI")
print(np.exp(ci[0:2, :]).T)
```

```
[0.22527109 1.030578  ]
95% CI
[[0.19459809 0.78938459]
 [0.26077884 1.34546713]]
```

Which are the same (exponentiated) coefficients as the previous approach. There is a slight discrepancy you may notice in the confidence intervals, but this is a floating point error resulting from a difference between the stabilized weights (by-hand) and the unstabilized weights (`ee_ipw_msm`).

12.5: Effect modification and marginal structural models

We will now use marginal structural models to study effect measure modification. We will look at effect modification by sex (`sex`) of quitting smoking (`qsmk`) on 10-year weight change (`wt82_71`).

[29]:
```
# Design matrix for propensity scores
W = dc[['I', 'sex', 'race', 'age', 'age_sq',
        'educ_2', 'educ_3', 'educ_4', 'educ_5',
        'smokeintensity', 'smokeintensity_sq',
        'smokeyrs', 'smokeyrs_sq',
        'exer_1', 'exer_2', 'active_1', 'active_2',
        'wt71', 'wt71_sq']]
# Design matrix for marginal structural model
dc['qsmk_sex'] = dc['qsmk']*dc['sex']
msm = dc[['I', 'qsmk', 'sex', 'qsmk_sex']]
# Treatment array
a = dc['qsmk']
# Outcome array
y = dc['wt82_71']
```

[30]:
```
def psi_ipw_msm5(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[0:4]               # Marginal structural model
    gamma = np.array([theta[4], ])  # Numerator parameter
```

(continues on next page)

(continued from previous page)

```
    beta = theta[5:]                # Propensity score

    # Estimating the propensity scores using a logit model
    ee_ps = ee_regression(theta=beta,        # Propensity score
                          X=W, y=a,          # ... observed data
                          model='logistic')  # ... logit model
    pi = inverse_logit(np.dot(W, beta))      # Predicted prob

    # Estimating intercept-only for numerator
    ee_num = ee_regression(theta=gamma,            # Numerator model
                           X=dc[['I']], y=a,       # ... observed data
                           model='logistic')       # ... logit model
    num = inverse_logit(np.dot(dc[['I']], gamma))  # Predicted prob

    # Stabilized inverse probability weights
    ipw = np.where(a == 1, num/pi, (1-num)/(1-pi))

    # Estimating the MSM using a weighted linear model
    ee_msm = ee_regression(theta=alpha,     # Marginal structural model
                           X=msm, y=y,      # ... observed data
                           model='linear',  # ... linear model
                           weights=ipw)     # ... weighted

    # Stacking the estimating equations and returning
    return np.vstack([ee_msm, ee_num, ee_ps])
```

[31]:
```
# M-estimator
init_vals = [0., 0., 0., 0., ] + [0., ] + [0.,]*W.shape[1]
estr = MEstimator(psi_ipw_msm5, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[32]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:4])
print("95% CI")
print(ci[0:4, :].T)
```

```
[ 1.78444688  3.52197763 -0.00872478 -0.15947852]
95% CI
[[ 1.19225612  2.28240175 -0.88125781 -2.15657223]
 [ 2.37663763  4.76155352  0.86380824  1.83761518]]
```

While not reported in the book, other online references report the folloiwing coefficients: 1.7844, 3.5220, -0.0087, -0.1595.

As before, we can replicate this result using the built-in `ee_ipw_msm` function.

[33]:
```
def psi_ipw_msm5a(theta):
    # Built-in estimating equation
    return ee_ipw_msm(theta, y=y, A=a, W=W, V=msm,
                      distribution='normal',
                      link='identity')
```

[34]:
```
# M-estimator
init_vals = [0., ]*msm.shape[1] + init_ps
estr = MEstimator(psi_ipw_msm5a, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[35]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:4])
print("95% CI")
print(ci[0:4, :].T)
```

```
[ 1.78444688  3.52197763 -0.00872478 -0.15947852]
95% CI
[[ 1.19225614  2.28240246 -0.88125778 -2.15657182]
 [ 2.37663761  4.7615528   0.86380821  1.83761477]]
```

which again replicates the by-hand results.

12.6: Censoring and missing data

To conclude, we will now consider the missing outcomes that were ignored earlier. To do this, we will use stabilized inverse probability of missingness weights (IPCW in the book). As we have done so many times already, we will use `delicatessen` to stack additional nuisance models together.

[36]:
```
# Design matrix for propensity score model
W = df[['I', 'sex', 'race', 'age', 'age_sq',
        'educ_2', 'educ_3', 'educ_4', 'educ_5',
        'smokeintensity', 'smokeintensity_sq',
        'smokeyrs', 'smokeyrs_sq',
        'exer_1', 'exer_2', 'active_1', 'active_2',
        'wt71', 'wt71_sq']]
# Design matrix for missing model
X = df[['I', 'qsmk', 'sex', 'race', 'age', 'age_sq',
        'educ_2', 'educ_3', 'educ_4', 'educ_5',
        'smokeintensity', 'smokeintensity_sq',
        'smokeyrs', 'smokeyrs_sq',
        'exer_1', 'exer_2', 'active_1', 'active_2',
        'wt71', 'wt71_sq']]
# Design matrix for marginal structural model
msm = df[['I', 'qsmk']]
# Treatment array
a = df['qsmk']
# Outcome array
y = df['wt82_71']
# Missing indicator
r = np.where(df['wt82_71'].isna(), 0, 1)
```

[37]:
```
def psi_ipw_msm6(theta):
    # Dividing parameters up for their corresponding estimation equations
    alpha = theta[0:2]                  # MSM
    gamma_n = np.array([theta[2], ])  # Numerator PS
    beta_n = np.array([theta[3], ])   # Numerator MW
    gamma_d = theta[4:4+W.shape[1]]   # Propensity score
```

(continues on next page)

(continued from previous page)

```
    beta_d = theta[4+W.shape[1]:]     # Missing model

    # Estimating the propensity scores using a logit model
    ee_ps = ee_regression(theta=gamma_d,      # Propensity score
                          X=W, y=a,           # ... observed data
                          model='logistic')   # ... logit model
    pi_a = inverse_logit(np.dot(W, gamma_d))  # Predicted prob

    # Estimating intercept-only for numerator of IPTW
    ee_num = ee_regression(theta=gamma_n,     # Numerator
                           X=df[['I']], y=a,  # ... observed data
                           model='logistic')  # ... logit model
    num_a = inverse_logit(np.dot(df[['I']], gamma_n))

    # Estimating the missing scores using a logit model
    ee_ms = ee_regression(theta=beta_d,      # Missing score
                          X=X, y=r,          # ... observed data
                          model='logistic')  # ... logit model
    pi_m = inverse_logit(np.dot(X, beta_d))  # Predicted prob

    # Estimating intercept-only for numerator of IPMW
    ee_sms = ee_regression(theta=beta_n,      # Numerator
                           X=df[['I']], y=r,  # ... observed data
                           model='logistic')  # ... logit model
    num_m = inverse_logit(np.dot(df[['I']], beta_n))

    # Stabilized inverse probability weights
    iptw = np.where(a == 1, num_a/pi_a, (1-num_a)/(1-pi_a))
    ipmw = np.where(r == 1, num_m/pi_m, 0)
    ipw = iptw*ipmw

    # Estimating the MSM using a weighted linear model
    ee_msm = ee_regression(theta=alpha,     # MSM
                           X=msm, y=y,      # ... observed data
                           model='linear',  # ... linear model
                           weights=ipw)     # ... weighted
    # Setting rows with missing Y's as zero (no contribution)
    ee_msm = np.nan_to_num(ee_msm, copy=False, nan=0.)

    # Stacking the estimating equations and returning
    return np.vstack([ee_msm, ee_num, ee_sms, ee_ps, ee_ms])
```

[38]:
```
# M-estimator
init_vals = [0., 0., ] + [0., 0., ] + [0.,]*W.shape[1] + [0.,]*X.shape[1]
estr = MEstimator(psi_ipw_msm6, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[39]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:2])
print("95% CI")
print(ci[0:2, :].T)
```

```
[1.66199003 3.49649333]
95% CI
[[1.22590804 2.54496958]
 [2.09807203 4.44801708]]
```

Here, the book reports 3.5 (95% CI: 2.5, 4.5).

Again, we will replicate this using `ee_ipw_msm` instead. To incorporate the missingness weights, we will fit a separate model and then use the optional `weights` argument. Below is how this can be done

```
[40]: def psi_ipw_msm6a(theta):
          # Separating parameters out
          alpha = theta[:2+W.shape[1]]  # MSM & PS
          gamma = theta[2+W.shape[1]:]  # Missing score

          # Estimating equation for IPMW
          ee_ms = ee_regression(theta=gamma,         # Missing score
                                X=X, y=r,            # ... observed data
                                model='logistic')    # ... logit model
          pi_m = inverse_logit(np.dot(X, gamma))     # Predicted prob
          ipmw = r / pi_m                            # Missing weights

          # Estimating equations for MSM and PS
          ee_msm = ee_ipw_msm(theta=alpha, y=y, A=a, W=W, V=msm,
                              distribution='normal',
                              link='identity',
                              weights=ipmw)
          # Setting rows with missing Y's as zero (no contribution)
          ee_msm = np.nan_to_num(ee_msm, copy=False, nan=0.)

          # Return the stacked estimating equations
          return np.vstack([ee_msm, ee_ms])
```

```
[41]: # M-estimator
      init_vals = [0., ]*msm.shape[1] + init_ps + [0., ]*X.shape[1]
      estr = MEstimator(psi_ipw_msm6a, init=init_vals)
      estr.estimate(solver='hybr', maxiter=5000)
```

```
[42]: ci = estr.confidence_intervals()
      print(estr.theta[0:2])
      print("95% CI")
      print(ci[0:2, :].T)
```

```
[1.66199003 3.49649333]
95% CI
[[1.22590805 2.54496955]
 [2.09807201 4.4480171 ]]
```

Again, we are able to replicate the by-hand implementation. This concludes chapter 12.

Chapter 13: Standardization and the Parametric G-Formula

For Chapter 13, the book reviews the g-formula. Unlike IPW, the g-formula relies on modeling the outcome process. The g-computation algorithm estimator is

$$\hat{E}[Y^a] = n^{-1} \sum_{i=1}^{n} m(a, W_i; \beta)$$

where m is a statistical model for $E[Y|A, W]$ and β are the parameters defining the model. This version is slightly different from the standardization form given in the book, but it is equivalent. Broadly, we apply the g-computation algorithm via (1) estimate an outcome model, (2) predict the outcomes had everyone been assigned a, and (3) compute the mean of those predictions.

13.2: Estimating the mean outcome via modeling

First, we will fit a linear model for `wt82_71` conditional on `qsmk` and the set of confounding variables. To begin, we will ignore the missing outcomes.

[43]:
```
# Design matrix for outcome model
X = dc[['I', 'qsmk', 'qsmk_smkint',
        'sex', 'race', 'age', 'age_sq',
        'educ_2', 'educ_3', 'educ_4', 'educ_5',
        'smokeintensity', 'smokeintensity_sq',
        'smokeyrs', 'smokeyrs_sq',
        'exer_1', 'exer_2', 'active_1', 'active_2',
        'wt71', 'wt71_sq']]
# Outcome array
y = dc['wt82_71']
```

[44]:
```
def psi_regression(theta):
    # Estimating the linear outcome model
    return ee_regression(theta=theta,
                         X=X, y=y,
                         model='linear')
```

[45]:
```
# M-estimator
init_vals = [0., ]*X.shape[1]
estr = MEstimator(psi_regression, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[46]:
```
print("Regression coefficients")
print(estr.theta)
```

```
Regression coefficients
[-1.58816566e+00  2.55959409e+00  4.66628395e-02 -1.43027166e+00
  5.60109604e-01  3.59635262e-01 -6.10095538e-03  7.90444028e-01
  5.56312403e-01  1.49156952e+00 -1.94977036e-01  4.91364717e-02
 -9.90651192e-04  1.34368569e-01 -1.86642949e-03  2.95975357e-01
  3.53912774e-01 -9.47569487e-01 -2.61377894e-01  4.55017905e-02
 -9.65325105e-04]
```

To ease application of later steps, we are going to save a list of these optimized values for starting values of the root-finding procedure.

```
[47]: init_reg = list(estr.theta)
```

13.3: Standardizing the mean outcome to the confounder distribution

Now we can apply the g-computation algorithm (a way to evaluate the g-formula). To do this, we are going to create a copy of our data set. In that copy we are going to set `qsmk=1` for all observations. We will then save the design matrix. We will then repeat this process for `qsmk=0`.

```
[48]: # Copy of the data that we will updated qsmk in
dca = dc.copy()
```

```
[49]: # Setting qsmk to 1
dca['qsmk'] = 1
dca['qsmk_smkint'] = dca['qsmk'] * dca['smokeintensity']
# Design matrix from qsmk=1 data
X1 = dca[['I', 'qsmk', 'qsmk_smkint',
          'sex', 'race', 'age', 'age_sq',
          'educ_2', 'educ_3', 'educ_4', 'educ_5',
          'smokeintensity', 'smokeintensity_sq',
          'smokeyrs', 'smokeyrs_sq',
          'exer_1', 'exer_2', 'active_1', 'active_2',
          'wt71', 'wt71_sq']]
```

```
[50]: # Setting qsmk to 0
dca['qsmk'] = 0
dca['qsmk_smkint'] = dca['qsmk'] * dca['smokeintensity']
# Design matrix from qsmk=0 data
X0 = dca[['I', 'qsmk', 'qsmk_smkint',
          'sex', 'race', 'age', 'age_sq',
          'educ_2', 'educ_3', 'educ_4', 'educ_5',
          'smokeintensity', 'smokeintensity_sq',
          'smokeyrs', 'smokeyrs_sq',
          'exer_1', 'exer_2', 'active_1', 'active_2',
          'wt71', 'wt71_sq']]
```

Now, we will use the design matrices to generate predicted outcomes (pseudo-outcomes) for each individual. The average causal effect can then be calculated from those pseudo-outcomes.

The following is the corresponding estimating equations

```
[51]: def psi_gcomp1(theta):
    # Dividing parameters into corresponding estimation equations
    rd, r1, r0 = theta[0], theta[1], theta[2]    # Causal means
    beta = theta[3:]                             # Outcome model

    # Estimating the linear model
    ee_reg = ee_regression(theta=beta,       # Outcome model
                           X=X, y=y,         # ... observed data
                           model='linear')   # ... linear model

    # Generating pseudo-outcome using the model
```

(continues on next page)

(continued from previous page)

```
    y1hat = np.dot(X1, beta)  # Predicted Y when qsmk=1
    y0hat = np.dot(X0, beta)  # Predicted Y when qsmk=0

    # Causal means
    ee_r1 = y1hat - r1   # Causal mean for qsmk=1
    ee_r0 = y0hat - r0   # Causal mean for qsmk=0

    # Average causal effect
    ee_rd = np.ones(y.shape[0]) * ((r1 - r0) - rd)

    # Stacking the estimating equations and returning
    return np.vstack([ee_rd, ee_r1, ee_r0, ee_reg])
```

[52]:
```
# M-estimator
init_vals = [0., 0., 0.,] + init_reg
estr = MEstimator(psi_gcomp1, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[53]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:3])
print("95% CI")
print(ci[0:3, :].T)
```

```
[3.5173742  5.27358732 1.75621312]
95% CI
[[2.58132997 4.42099134 1.33029619]
 [4.45341844 6.1261833  2.18213004]]
```

The first estimate is for the average causal effect (the second are the causal means under all quit smoking and all don't quit smoking). Here, the confidence intervals match the book, but we were able to avoid the computational complexity of the bootstrap (another advantage of the sandwich variance estimator).

In the book, they report 3.5 (95% CI: 2.6, 4.5). The confidence interval in the book was generated via the bootstrap, so there is potential for random error in the estimates.

Rather than implement g-computation by-hand, we can also use the built-in estimating equations. Here is an example of that

[54]:
```
def psi_gcomp1a(theta):
    # Built-in g-formula estimating equation
    return ee_gformula(theta,
                       y=y, X=X,
                       X1=X1, X0=X0)
```

[55]:
```
# M-estimator
init_vals = [0., 0., 0.,] + init_reg
estr = MEstimator(psi_gcomp1a, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[56]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:3])
print("95% CI")
print(ci[0:3, :].T)
```

```
[3.5173742  5.27358732 1.75621312]
95% CI
[[2.58132997 4.42099134 1.33029619]
 [4.45341844 6.1261833  2.18213004]]
```

which provides the same answers (as we would expect!)

Chapter 14: G-Estimation of Structural Nested Models

G-estimation differs from the previous approaches in what parameter it targets. Rather than the average causal effect, we will estimate the parameters of a structural nested model. The structural nested mean model is

$$E[Y^a - Y^{a=0} | A = a, W] = \varphi_0 a$$

Here, φ_0 represents the difference by a (which is equivalent to average causal effect). However, we can also study effect measure modification easily with structural nested models. Consider

$$E[Y^a - Y^{a=0} | A = a, W] = \varphi_0 a + \varphi_1 a V$$

where $V \in W$. Therefore, the structural nested model described effect measure modification by V. Importantly, structural nested models assume that we have correctly specified this model (hence the difference in interest parameters that occurs between methods).

G-estimation is a bit less straightforward to understand compare to the previous methods. Note that we will be using the propensity score for estimation and we still assume the same identification conditions. See the book for a much more in-depth discussion.

14.5 G-estimation

For solve our g-estimator, we are going to use a different approach than the one described in the main text of the book. Instead, we are going to use the approach (the estimating equations) described in Technical Point 14.2. As mentioned there, this approach is equivalent to the process described in the main text.

[57]:
```
# Design matrix for propensity scores
W = np.asarray(df[['I', 'sex', 'race', 'age', 'age_sq',
                   'educ_2', 'educ_3', 'educ_4', 'educ_5',
                   'smokeintensity', 'smokeintensity_sq',
                   'smokeyrs', 'smokeyrs_sq',
                   'exer_1', 'exer_2', 'active_1', 'active_2',
                   'wt71', 'wt71_sq']])
# Design matrix for missing model
X = np.asarray(df[['I', 'qsmk', 'sex', 'race', 'age', 'age_sq',
                   'educ_2', 'educ_3', 'educ_4', 'educ_5',
                   'smokeintensity', 'smokeintensity_sq',
                   'smokeyrs', 'smokeyrs_sq',
                   'exer_1', 'exer_2', 'active_1', 'active_2',
                   'wt71', 'wt71_sq']])
# Design matrix for structural nested model
snm = np.asarray(df[['I', ]])
# Treatment array
a = np.asarray(df['qsmk'])
# Outcome array
```

(continues on next page)

(continued from previous page)

```
y = np.asarray(df['wt82_71'])
# Missing indicator
r = np.where(df['wt82_71'].isna(), 0, 1)
```

[58]:
```
def psi_snm1(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[:snm.shape[1]]              # SNM parameters
    beta = theta[snm.shape[1]:                # Propensity score
                 snm.shape[1]+W.shape[1]]
    gamma = theta[snm.shape[1]+W.shape[1]:]  # Missing score

    # Estimating equation for IPMW
    ee_ms = ee_regression(theta=gamma,        # Missing score
                          X=X, y=r,           # ... observed data
                          model='logistic')  # ... logit model
    pi_m = inverse_logit(np.dot(X, gamma))    # Predicted prob
    ipmw = r / pi_m                           # Missing weights

    # Estimating equations for PS
    ee_log = ee_regression(theta=beta,         # Propensity score
                           X=W, y=a,           # ... observed data
                           model='logistic',  # ... logit model
                           weights=ipmw)      # ... weighted
    pi = inverse_logit(np.dot(W, beta))        # Predicted prob

    # H(psi) equation for linear models
    h_psi = y - np.dot(snm*a[:, None], alpha)

    # Estimating equation for the structural nested mean model
    ee_snm = ipmw*(h_psi[:, None] * (a - pi)[:, None] * snm).T
    # Setting rows with missing Y's as zero (no contribution)
    ee_snm = np.nan_to_num(ee_snm, copy=False, nan=0.)

    return np.vstack([ee_snm, ee_log, ee_ms])
```

[59]:
```
# M-estimator
init_vals = [0., ] + init_ps + [0., ]*X.shape[1]
estr = MEstimator(psi_snm1, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[60]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:1])
print("95% CI")
print(ci[0, :])
```

```
[3.4458988]
95% CI
[2.52709776 4.36469984]
```

The book reported 3.4 (95% CI: 2.5, 4.5). This confidence interval was generated by inverting the test results. As such, there is expected differences (different estimators are being used). As in the inverse probability weighting examples, these confidence intervals are expected due to be conservative due to the use of the 'GEE trick'.

Now consider how we can use the built-in g-estimation functionality, `ee_gestimation_snmm`.

[61]:
```
def psi_snm1a(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[:snm.shape[1]+W.shape[1]]  # SNM and PS
    gamma = theta[snm.shape[1]+W.shape[1]:]  # Missing scores

    # Estimating equation for IPMW
    ee_ms = ee_regression(theta=gamma,       # Missing score
                          X=X, y=r,          # ... observed data
                          model='logistic')  # ... logit model
    pi_m = inverse_logit(np.dot(X, gamma))   # Predicted prob
    ipmw = r / pi_m                          # Missing weights

    # Estimating equations for PS
    ee_snm = ee_gestimation_snmm(theta=alpha,
                                 y=y, A=a,
                                 W=W, V=snm,
                                 weights=ipmw)
    # Setting rows with missing Y's as zero (no contribution)
    ee_snm = np.nan_to_num(ee_snm, copy=False, nan=0.)

    return np.vstack([ee_snm, ee_ms])
```

[62]:
```
# M-estimator
init_vals = [0., ] + init_ps + [0., ]*X.shape[1]
estr = MEstimator(psi_snm1a, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[63]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:1])
print("95% CI")
print(ci[0, :])
```

```
[3.4458988]
95% CI
[2.52709776 4.36469984]
```

The structural nested model parameters match between implementations, as expected.

14.6: Structural nested models with two or more parameters

We now adapt the previous code to consider more than 2 parameters in the structural nested model. Thankfully, this is pretty straightforward for how we coded the estimating equations since `snm` is a global object.

[64]:
```
snm = np.asarray(df[['I', 'smokeintensity']])
```

[65]:
```
def psi_snm2(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[:snm.shape[1]]              # SNM parameters
    beta = theta[snm.shape[1]:               # Propensity score
                 snm.shape[1]+W.shape[1]]
```

(continues on next page)

(continued from previous page)

```
    gamma = theta[snm.shape[1]+W.shape[1]:]  # Missing score

    # Estimating equation for IPMW
    ee_ms = ee_regression(theta=gamma,       # Missing score
                          X=X, y=r,          # ... observed data
                          model='logistic')  # ... logit model
    pi_m = inverse_logit(np.dot(X, gamma))   # Predicted prob
    ipmw = r / pi_m                          # Missing weights

    # Estimating equations for PS
    ee_log = ee_regression(theta=beta,        # Propensity score
                           X=W, y=a,          # ... observed data
                           model='logistic',  # ... logit model
                           weights=ipmw)      # ... weighted
    pi = inverse_logit(np.dot(W, beta))       # Predicted prob

    # H(psi) equation for linear models
    h_psi = y - np.dot(snm*a[:, None], alpha)

    # Estimating equation for the structural nested mean model
    ee_snm = ipmw*(h_psi[:, None] * (a - pi)[:, None] * snm).T
    # Setting rows with missing Y's as zero (no contribution)
    ee_snm = np.nan_to_num(ee_snm, copy=False, nan=0.)

    return np.vstack([ee_snm, ee_log, ee_ms])
```

[66]:
```
# M-estimator
init_vals = [0., 0., ] + init_ps + [0., ]*X.shape[1]
estr = MEstimator(psi_snm2, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[67]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:2])
print("95% CI")
print(ci[:2, :].T)
```

```
[2.85947039 0.03004128]
95% CI
[[ 1.0291672  -0.05951923]
 [ 4.68977357  0.11960179]]
```

which provides the same results as the book (2.86 and 0.03). Unlike the book, we provide confidence intervals. Inverting the test is not simple in this case, and the book does not use a bootstrapping procedure. Our confidence intervals are computed using the sandwich variance estimator.

Again, we can apply the built-in estimating equations for g-estimation. In truth, the new estimating equation is the same as the previous g-estimation version (since `snm` is a global object as called). Regardless, we provide again here.

[68]:
```
def psi_snm2a(theta):
    # Dividing parameters into corresponding estimation equations
    alpha = theta[:snm.shape[1]+W.shape[1]]  # SNM and PS
    gamma = theta[snm.shape[1]+W.shape[1]:]  # Missing scores
```

(continues on next page)

(continued from previous page)

```
    # Estimating equation for IPMW
    ee_ms = ee_regression(theta=gamma,       # Missing score
                          X=X, y=r,          # ... observed data
                          model='logistic')  # ... logit model
    pi_m = inverse_logit(np.dot(X, gamma))   # Predicted prob
    ipmw = r / pi_m                          # Missing weights

    # Estimating equations for PS
    ee_snm = ee_gestimation_snmm(theta=alpha,
                                 y=y, A=a,
                                 W=W, V=snm,
                                 weights=ipmw)
    # Setting rows with missing Y's as zero (no contribution)
    ee_snm = np.nan_to_num(ee_snm, copy=False, nan=0.)

    return np.vstack([ee_snm, ee_ms])
```

[69]:
```
# M-estimator
init_vals = [0., 0., ] + init_ps + [0., ]*X.shape[1]
estr = MEstimator(psi_snm2a, init=init_vals)
estr.estimate(solver='hybr', maxiter=5000)
```

[70]:
```
ci = estr.confidence_intervals()
print(estr.theta[0:2])
print("95% CI")
print(ci[:2, :].T)
```

```
[2.85947039 0.03004128]
95% CI
[[ 1.0291672  -0.05951923]
 [ 4.68977357  0.11960179]]
```

This concludes chapter 14.

Replication of the following chapters 15-17 are not yet available.

3.2.7 Morris et al. (2022): Precision in Randomized Trials

The following is a replication of the analyses described in Morris et al. (2022)., which uses data from Wilson E et al. (2017). Morris et al. examine three different approaches to adjust for prognostic factors in randomized trials. The three ways are direct adjustment, standardization, and inverse probability weighting. This example highlights how causal inference methods (like those in the Hernan & Robins (2023)) can be used in randomized trials.

Here, we are not going to consider direct adjustment. Attention is further restricted to the any-test outcome. For finer details and comparisons between the covariate adjustment approaches, see the original publication.

Setup

```
[1]: import numpy as np
     import pandas as pd
     import patsy

     import delicatessen
     from delicatessen import MEstimator
     from delicatessen.estimating_equations import ee_regression
     from delicatessen.utilities import inverse_logit

     print("Versions")
     print("NumPy:        ", np.__version__)
     print("pandas:       ", pd.__version__)
     print("Delicatessen: ", delicatessen.__version__)
```

```
Versions
NumPy:         1.25.2
pandas:        1.4.1
Delicatessen:  2.1
```

Data is available from Supplement S1 of Wilson et al. (2017) (referenced below).

```
[2]: d = pd.read_excel("data/GetTestedData.xls", sheet_name=1)

     # Subsetting the desired columns
     cols = ['group',      # Randomized arm
             'anytest',    # Outcome 1 (any test)
             'anydiag',    # Outcome 2 (any diagnosis)
             'gender',     # gender (male, female, transgender)
             'msm',        # MSM, other
             'age',        # age (continuous)
             'partners',   # Number of partners in <12 months
             'ethnicgrp']  # Ethnicity (5 categories)
     d = d[cols].copy()

     # Re-coding columns as numbers
     d['group_n'] = np.where(d['group'] == 'SH:24', 1, np.nan)
     d['group_n'] = np.where(d['group'] == 'Control', 0, d['group_n'])
     d['gender_n'] = np.where(d['gender'] == 'Female', 0, np.nan)
     d['gender_n'] = np.where(d['gender'] == 'Male', 1, d['gender_n'])
     d['gender_n'] = np.where(d['gender'] == 'Transgender', 2, d['gender_n'])
     d['msm_n'] = np.where(d['msm'] == 'other', 0, np.nan)
     d['msm_n'] = np.where(d['msm'] == 'msm', 1, d['msm_n'])
     d['msm_n'] = np.where(d['msm'] == '99', 2, d['msm_n'])
     d['partners_n'] = np.where(d['partners'] == '1', 0, np.nan)
     categories = ['2', '3', '4', '5', '6', '7', '8', '9', '10+']
     for index in range(len(categories)):
         d['partners_n'] = np.where(d['partners'] == categories[index],
                                    index, d['partners_n'])

     d['ethnicgrp_n'] = np.where(d['ethnicgrp'] == 'White/ White British', 0, np.nan)
     d['ethnicgrp_n'] = np.where(d['ethnicgrp'] == 'Black/ Black British', 1, d['ethnicgrp_n
     ↪'])
```

(continues on next page)

(continued from previous page)

```
d['ethnicgrp_n'] = np.where(d['ethnicgrp'] == 'Mixed/ Multiple ethnicity', 2, d[
↪'ethnicgrp_n'])
d['ethnicgrp_n'] = np.where(d['ethnicgrp'] == 'Asian/ Asian British', 3, d['ethnicgrp_n
↪'])
d['ethnicgrp_n'] = np.where(d['ethnicgrp'] == 'Other', 4, d['ethnicgrp_n'])

# Dropping old columns and renaming new ones
d = d.drop(columns=['group', 'gender', 'msm', 'partners', 'ethnicgrp'])
relabs = dict()
for c in cols:
    relabs[c + "_n"] = c

d = d.rename(columns=relabs)
```

As done in the main paper, we conduct a complete-case analysis (i.e., discard all the observations with missing outcomes).

As `delicatessen` interacts with NumPy arrays, we will further extract the covariates from the complete-case data set. Here, we use `patsy`, which allows for R-like formulas, to make the covariate manipulations easier. Specifically, the `C(...)` functionality generates an array of indicator variables (saving us the trouble of hand-coding disjoint indicators).

Finally, we generate some copies of the data set and set the corresponding `group` values to all-1 and all-0. These sets of covariates will be used to generate the predicted outcome under each treatment for the standardization approach

```
[3]: ds = d.dropna().copy()

# Outcome
y = np.asarray(ds['anytest'])

# Treatment
a = np.asarray(ds['group'])

# Observed covariates
prop_score_covs = patsy.dmatrix("age + C(gender) + C(msm) + C(partners) + C(ethnicgrp)",
                                ds)
outcome_covs = patsy.dmatrix("group + age + C(gender) + C(msm) + C(partners) +␣
↪C(ethnicgrp)",
                                ds)

# Setting group=a and getting matrices
dsa = ds.copy()
dsa['group'] = 1
outcome_a1_covs = patsy.dmatrix("group + age + C(gender) + C(msm) + C(partners) +␣
↪C(ethnicgrp)",
                                dsa)
dsa['group'] = 0
outcome_a0_covs = patsy.dmatrix("group + age + C(gender) + C(msm) + C(partners) +␣
↪C(ethnicgrp)",
                                dsa)
```

Standardization

For standardization (or g-computation), we stack several estimating equations together. Let Y indicate the outcome, A indicate the treatment, and W are the prognostic factors included in the model (age, gender, sexual orientation, number of partners, ethnicity). The parameters are θ, which we further divide into the interest parameters $(\mu_0, \mu_1, \mu_2, \mu_3)$ and the nuisance parameters (β). As their name implies, nuisance parameters are not of interest but are necessary for estimation for our interest parameters. Importantly, by stacking the estimating equations, we can account for the uncertainty in the estimation of the nuisance parameters into the uncertainty in our interest parameters automatically.

The estimating equations are

$$\psi(Y_i, A_i, W_i; \theta) = \begin{bmatrix} (\mu_1 - \mu_0) - \mu_2 \\ \log(\mu_1/\mu_0) - \mu_3 \\ \hat{Y}_i^{a=1} - \mu_1 \\ \hat{Y}_i^{a=0} - \mu_0 \\ (Y_i - \text{expit}(g(A_i, W_i)^T \beta)) g(A_i, W_i)_i \end{bmatrix}$$

where $g(A_i, W_i)$ is a specific function to generate the design matrix, and $\hat{Y}_i^a = \text{expit}(g(a, W_i)^T \beta)$. The first equation in the stack is the risk difference (RD) and the second is the log-transformed risk ratio (RR). Notice that these are defined in terms of other parameters. Specifically, parameters from the third and fourth equations, which correspond to the risk under all-1 and the risk under all-0. The predicted values of the outcome are generated using the parameters from the final equation, which is the logistic model used to estimate the probability of Y_i conditional on A_i, W_i.

An astute reader may notice that we could actually shorten this stack of equations. Rather than estimate μ_2, μ_3, we could have plugged the corresponding $\hat{Y}_i^a$ into the first and second equations (thereby removing the need for the third and fourth). While this is correct, we prefer the process above for three reasons: 1. The above stacked equations give us the correspond risk under each arm, which also may be of interest (what was the risk under a=0). Therefore, no extra steps are necessary to get this estimated parameter. 2. This second argument is more technical. Here, we use a root-finding procedure to get $\hat{\theta}$. During this process we plug in different values for $\hat{\theta}$ until we find the (approximate) zeroes. For the RR, some values during the exploration of values may result in $\hat{Y}_i^{a=0} = 0$. This causes a division by zero and may break the root-finding procedure. By instead calculating the average of $\hat{Y}_i^{a=0}$ across all i's, we reduce the opporuntity for a rogue zero. 3. Related to the previous reason, the optimization procedure will run much faster since we only need to take the log of a single number as opposed to as an array. While this won't make much of a difference in run-time here, it is good practice (and may be important in problems with more complex estimating equations).

Below we write out the estimating equation by-hand (except for the regression component, which we use a built-in functionality), estimate with our M-estimator, and then provide the results (similar to Table 3).

[4]:
```python
def psi(theta):
    # Extracting parameters from theta for ease
    risk_diff = theta[0]
    log_risk_ratio = theta[1]
    risk_a1 = theta[2]
    risk_a0 = theta[3]
    beta = theta[4:]

    # Estimating nuisance model (outcome regression)
    ee_log = ee_regression(theta=beta,          # beta coefficients
                           X=outcome_covs,      # X covariates
                           y=y,                 # y
                           model='logistic')    # logit

    # Generating predicted outcome values
    ee_ya1 = inverse_logit(np.dot(outcome_a1_covs, beta)) - risk_a1
    ee_ya0 = inverse_logit(np.dot(outcome_a0_covs, beta)) - risk_a0
```

(continues on next page)

(continued from previous page)

```
    # Estimating interest parameters
    ee_risk_diff = np.ones(y.shape[0])*(risk_a1 - risk_a0) - risk_diff
    ee_risk_ratio = np.ones(y.shape[0])*np.log(risk_a1 / risk_a0) - log_risk_ratio

    # Returning stacked estimating equations (order matters)
    return np.vstack((ee_risk_diff,  # risk difference
                      ee_risk_ratio, # risk ratio
                      ee_ya1,        # risk a=1
                      ee_ya0,        # risk a=0
                      ee_log,))      # logistic model
```

[5]:
```
estr = MEstimator(psi,
                  init=[0, 0, 0.5, 0.5, ] + [0, ]*outcome_covs.shape[1])
estr.estimate(solver='lm')
```

[6]:
```
std = np.sqrt(np.diag(estr.variance))
fmt = '{:.3f}'

print("Risk Difference:", fmt.format(estr.theta[0]), "("+fmt.format(std[0])+")")
print("Risk Ratio:     ", fmt.format(estr.theta[1]), "("+fmt.format(std[1])+")")
```

```
Risk Difference: 0.260 (0.021)
Risk Ratio:      0.795 (0.075)
```

At this point, you may look at the results for the risk ratio and notice some differences in the third decimal place. We reserve conversation regarding this point till the end.

Inverse Probability Weighting

Now, we will consider the case of inverse probability weighting. Inverse probability weighting relies on a separate nuisance model. To clarify this, our nuisance parameters will be indicated by α here.

The estimating equations are

$$\psi(Y_i, A_i, W_i; \theta) = \begin{bmatrix} (\mu_3 - \mu_4) - \mu_1 \\ \log(\mu_3/\mu_4) - \mu_2 \\ \frac{Y_i A_i}{\text{expit}(g(W_i)^T \beta)} - \mu_3 \\ \frac{Y_i (1-A_i)}{1-\text{expit}(g(W_i)^T \beta)} - \mu_4 \\ (A_i - \text{expit}(g(W_i)^T \beta)) g(W_i) \end{bmatrix}$$

where $g(W_i)$ is a specific function to generate the design matrix. As before, the first and second equations are the risk difference and risk ratio, respectively. Third and fourth equations correspond to the risk under all-1 and the risk under all-0. The final estimating equation is the propensity score model and is used to generate the propensity scores given W_i.

Again, we write out the estimating equation by-hand (except for the regression component, which we use a built-in functionality), estimate with our M-estimator, and then provide the results (similar to Table 3).

[7]:
```
def psi(theta):
    # Extracting parameters from theta for ease
    risk_diff = theta[0]
```

(continues on next page)

(continued from previous page)

```
    log_risk_ratio = theta[1]
    risk_a1 = theta[2]
    risk_a0 = theta[3]
    beta = theta[4:]

    # Estimating nuisance model (outcome regression)
    ee_log = ee_regression(theta=beta,          # beta coefficients
                           X=prop_score_covs,   # W covariates
                           y=a,                 # a
                           model='logistic')      # logit

    # Generating predicted propensity score
    prop_score = inverse_logit(np.dot(prop_score_covs, beta))

    # Calculating weighted pieces
    ee_ya1 = (a*y) / prop_score - risk_a1
    ee_ya0 = ((1-a)*y) / (1-prop_score) - risk_a0

    # Estimating interest parameters
    ee_risk_diff = np.ones(a.shape[0])*(risk_a1 - risk_a0) - risk_diff
    ee_risk_ratio = np.ones(a.shape[0])*np.log(risk_a1 / risk_a0) - log_risk_ratio

    # Returning stacked estimating equations (order matters)
    return np.vstack((ee_risk_diff,  # risk difference
                      ee_risk_ratio, # risk ratio
                      ee_ya1,        # risk a=1
                      ee_ya0,        # risk a=0
                      ee_log,))      # logistic model
```

[8]:
```
estr = MEstimator(psi, init=[0., 0., 0.5, 0.5] + [0, ]*prop_score_covs.shape[1])
estr.estimate(solver='lm')
```

[9]:
```
std = np.sqrt(np.diag(estr.variance))
fmt = '{:.3f}'

print("Risk Difference:", fmt.format(estr.theta[0]), "("+fmt.format(std[0])+")")
print("Risk Ratio:     ", fmt.format(estr.theta[1]), "("+fmt.format(std[1])+")")
```

```
Risk Difference: 0.261 (0.021)
Risk Ratio:      0.805 (0.075)
```

Results are the same as Morris et al. up to the third decimal place.

Differences in Results

As promised, let's return to the differences in the answers. Why does this occur? The most likely culprit is the differences in the procedures for finding $\hat{\theta}$ across software.

`delicatessen` operates by a root-finding procedure. Essentially, we are looking for the values of $\hat{\theta}$ that lead to the estimating equations all being approximately zero. Specifically, I am using the Levenberg-Marquardt algorithm in these examples. `delicatessen` uses a root-finding procedure because it simplifies building stacked estimating equations. However, this comes at the cost of speed and robustness of other approaches.

Other regression software tends to proceed under a different approach. Generally, most software aims to maximize the log-likelihood. As before, there are a variety of algorithms that can be used to find the maximize the likelihood. However, most algorithms take advantage of both the log-likelihood and the score function (the derivative of the log-likelihood) to find the maximum. This means they are quite robust to sparse data while also being fast. As there is sparse data (there are many categorical variables for number of partners), this is the most likely explanation for the small differences.

Conclusion

We replicated selected analyses in Morris et al. (2022) to showcase the basics of stacking estimating equations and how they are implemented in `delicatessen`. While `delicatessen` has built-in functionalities for standardization and inverse probability weighting, we opted to present them by-hand here. By-hand takes some extra work but it provides a better idea of how `delicatessen` can be used to build estimating equations. For example, each of the above approaches could further be extended to account for missing outcomes under a weaker assumption.

References

Morris TP, Walker AS, Williamson EJ, & White IR. (2022). "Planning a method for covariate adjustment in individually-randomised trials: a practical guide". *Trials*, 23:328

Wilson E, Free C, Morris TP, Syred J, Ahamed I, Menon-Johansson AS et al.. (2017). "Internet-accessed sexually transmitted infection (e-STI) testing and results service: a randomised, single-blind, controlled trial". *PLoS Medicine*, 14(12), e1002479.

3.2.8 Generalized Additive Model

The following is a brief illustrative example of how generalized additive models (GAMs) can be implemented in `delicatessen`. For demonstration, we use data on motorcycle impacts from Silverman (1985).

Setup

[1]:
```python
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

import delicatessen as deli
from delicatessen import MEstimator
from delicatessen.estimating_equations import (ee_regression,
```

(continues on next page)

(continued from previous page)

```
                                                ee_ridge_regression,
                                                ee_additive_regression)
from delicatessen.utilities import additive_design_matrix, regression_predictions

%matplotlib inline
mpl.rcParams['figure.dpi']= 200
np.random.seed(202301)

print("NumPy version:       ", np.__version__)
print("SciPy version:       ", sp.__version__)
print("Pandas version:      ", pd.__version__)
print("Matplotlib version:  ", mpl.__version__)
print("Delicatessen version:", deli.__version__)
```

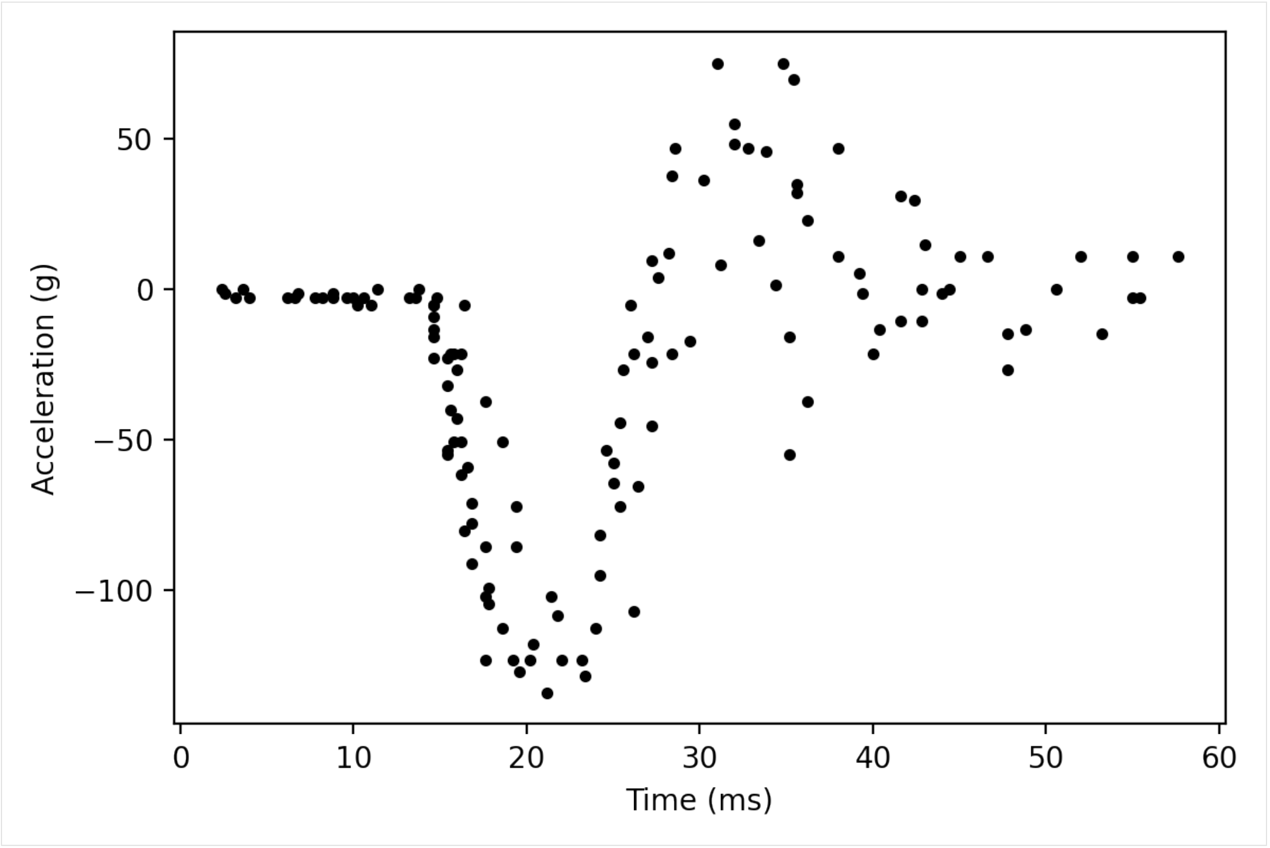
```
NumPy version:        1.25.2
SciPy version:        1.11.2
Pandas version:       1.4.1
Matplotlib version:   3.5.1
Delicatessen version: 2.1
```

Loading the motorcycle data (the data can be found in a few different R libraries and other online resources about GAMs).

```
[2]: # Loading data
d = pd.read_csv("data/silverman.csv")
d['intercept'] = 1

# Formating data for delicatessen
X = np.asarray(d[['intercept', 'times']])
y = np.asarray(d['accel'])

# Formating data for plots
p = pd.DataFrame()
p['times'] = np.linspace(np.min(d['times']), np.max(d['times']), 200)
p['intercept'] = 1
Xp = np.asarray(p[['intercept', 'times']])
```

Now let’s visualize the relationship in the data

```
[3]: plt.plot(d['times'], d['accel'], '.', color='k')
plt.xlabel("Time (ms)")
plt.ylabel("Acceleration (g)")
plt.tight_layout()
```



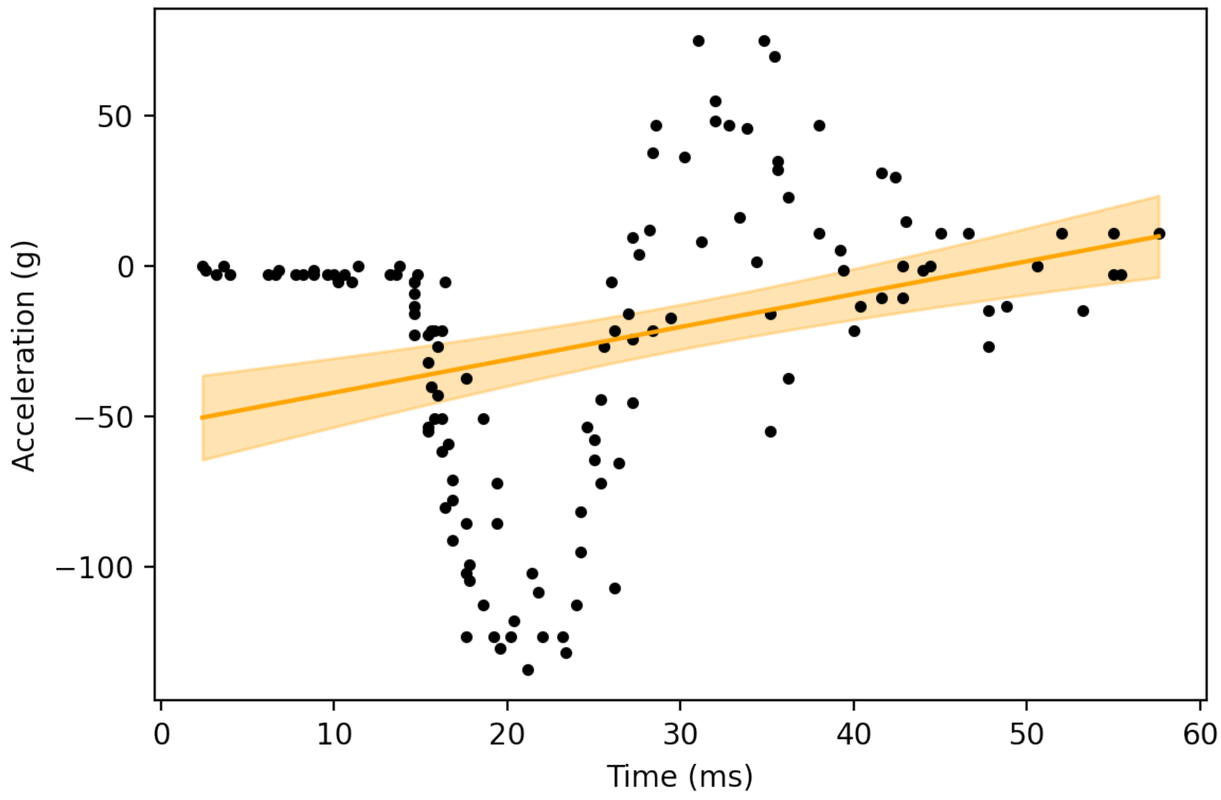
As easily seen in the scatterplot, the relationship between time and acceleration is non-linear.

We can show this even further by fitting a regression model with a linear relationship between the variables. We will do that here

[4]:
```
def psi(theta):
    return ee_regression(theta=theta, X=X, y=y, model='linear')
```

[5]:
```
estr = MEstimator(psi, init=[0, 0])
estr.estimate(solver='lm')
```

[6]:
```
# Predicted values and confidence intervals for plot
pred_y = regression_predictions(Xp, estr.theta, estr.variance)

# Plot
plt.plot(d['times'], d['accel'], '.', color='k')
plt.plot(p['times'], pred_y[:, 0], color='orange')
plt.fill_between(p['times'],
                 pred_y[:, 2], pred_y[:, 3],
                 color='orange', alpha=0.3)
plt.xlabel("Time (ms)")
plt.ylabel("Acceleration (g)")
plt.tight_layout()
```



As we can clearly see, a linear relationship is very different from the observed data

Implementing a GAM

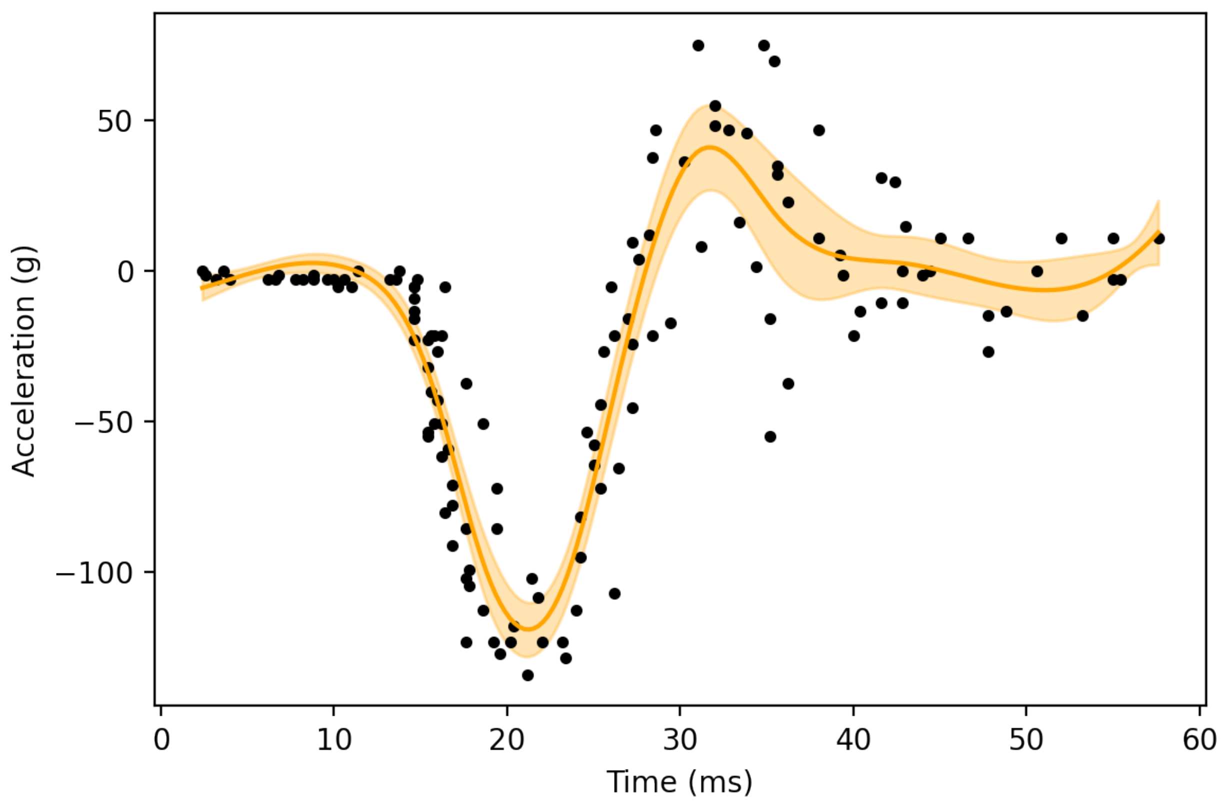
As of v1.1, `delicatessen` has a built-in functionality for GAMs. We will use that built-in functionality here. GAMs are implemented through the use of L_2-penalized splines. For implementation, the user needs to provide the knots (number or locations). Other optional arguments include the power of the splines, whether to restrict, and the strength of the penalty.

First, we define our estimating equations. The `ee_additive_regression` model builds the additive model design matrix for us, so we only need to pass the design matrix for the linear model above. However, we also need to specify some spline options. For the intercept term, we do not want any splines. For times, we will specify use of 12 knots and a rather strong penalty of 2000. This penalty will prevent the GAM from being too 'wiggly'.

```
[7]: # Defining specifications for splines to built the additive design matrix
knot_locs = np.percentile(d['times'], q=np.linspace(2.5, 97.5, 11)).tolist()
specs = [None,                                                # Spec for 1st term␣
↪(intercept)
         {"knots": knot_locs, "penalty": 2000, "power": 3}]  # Spec for 2nd term (times)

# Defining the estimating equation
def psi(theta):
    return ee_additive_regression(theta=theta,
                                  X=X, y=y,
                                  model='linear',
                                  specifications=specs)
```

[8]:
```
estr = MEstimator(psi, init=[0, 0, ] + [0, ]*10)
estr.estimate(solver='lm', maxiter=2000)
```

[9]:
```
# Creating the corresponding additive design matrix
Xpa = additive_design_matrix(p[['intercept', 'times']],
                             specifications=specs)

# Generating predicted values
pred_y = regression_predictions(Xpa, estr.theta, estr.variance)

# Plot
plt.plot(d['times'], d['accel'], '.', color='k')
plt.plot(p['times'], pred_y[:, 0], color='orange')
plt.fill_between(p['times'],
                 pred_y[:, 2], pred_y[:, 3],
                 color='orange', alpha=0.3)
plt.xlabel("Time (ms)")
plt.ylabel("Acceleration (g)")
plt.tight_layout()
```

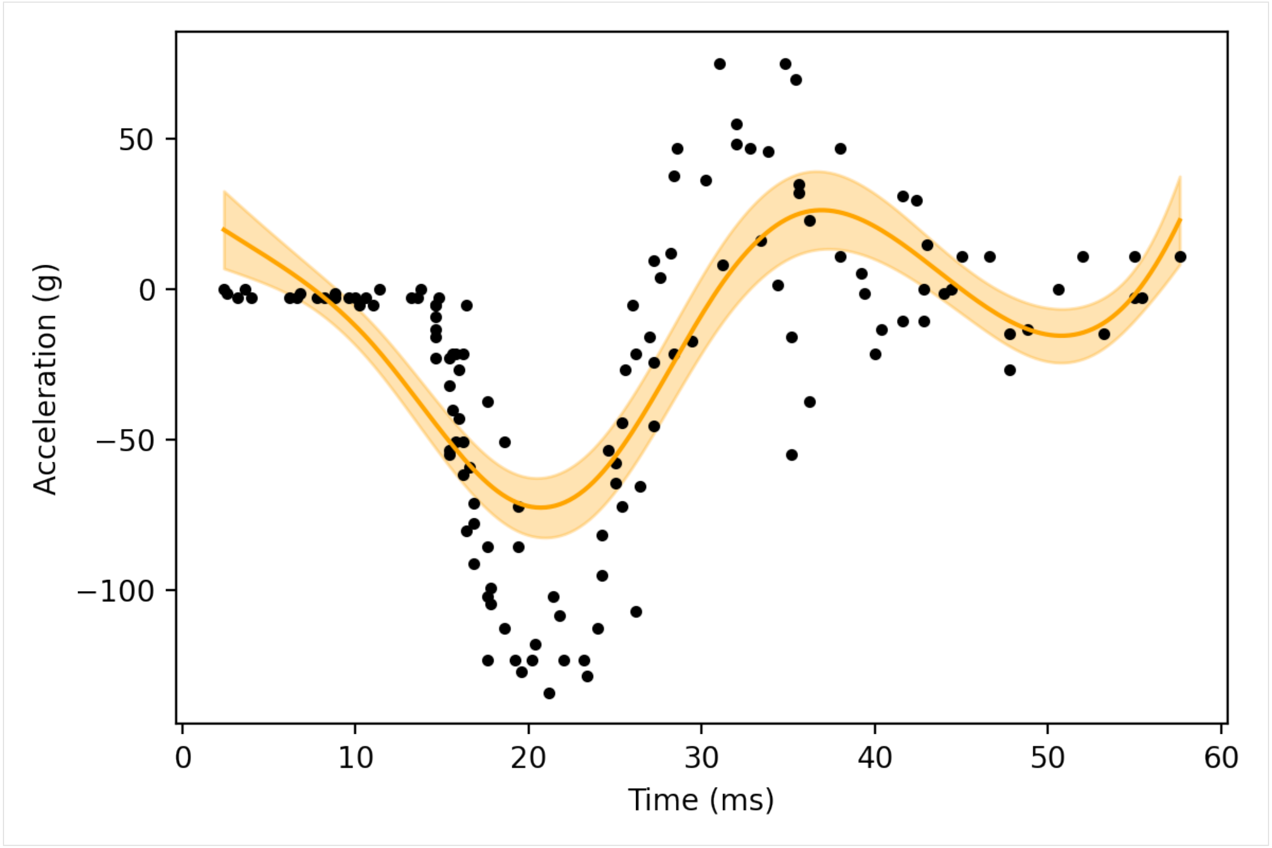


Here, we can see that the GAM does a much better job of following the shape of the scatterplot.

Impact of the Penalty

To show how the penalty impacts the wiggliness of the GAM, let's increase the GAM penalty by a large amount. As this penalty is only applied to the spline terms, we would expect to see the regression line become smoother.

[10]:
```
# Defining specifications for splines to built the additive design matrix
knot_locs = np.percentile(d['times'], q=np.linspace(2.5, 97.5, 11)).tolist()
specs = [None,                                                    # Spec for 1st term␣
↪(intercept)
         {"knots": knot_locs, "penalty": 1000000, "power": 3}]  # Spec for 2nd term␣
↪(times)

# Defining the estimating equation
def psi(theta):
    return ee_additive_regression(theta=theta,
                                  X=X, y=y,
                                  model='linear',
                                  specifications=specs)

estr = MEstimator(psi, init=[0, 0, ] + [0, ]*10)
estr.estimate(solver='lm', maxiter=2000)

# Creating the corresponding additive design matrix
Xpa = additive_design_matrix(p[['intercept', 'times']],
                            specifications=specs)

# Generating predicted values
pred_y = regression_predictions(Xpa, estr.theta, estr.variance)

# Plot
plt.plot(d['times'], d['accel'], '.', color='k')
plt.plot(p['times'], pred_y[:, 0], color='orange')
plt.fill_between(p['times'],
                 pred_y[:, 2], pred_y[:, 3],
                 color='orange', alpha=0.3)
plt.xlabel("Time (ms)")
plt.ylabel("Acceleration (g)")
plt.tight_layout()
```



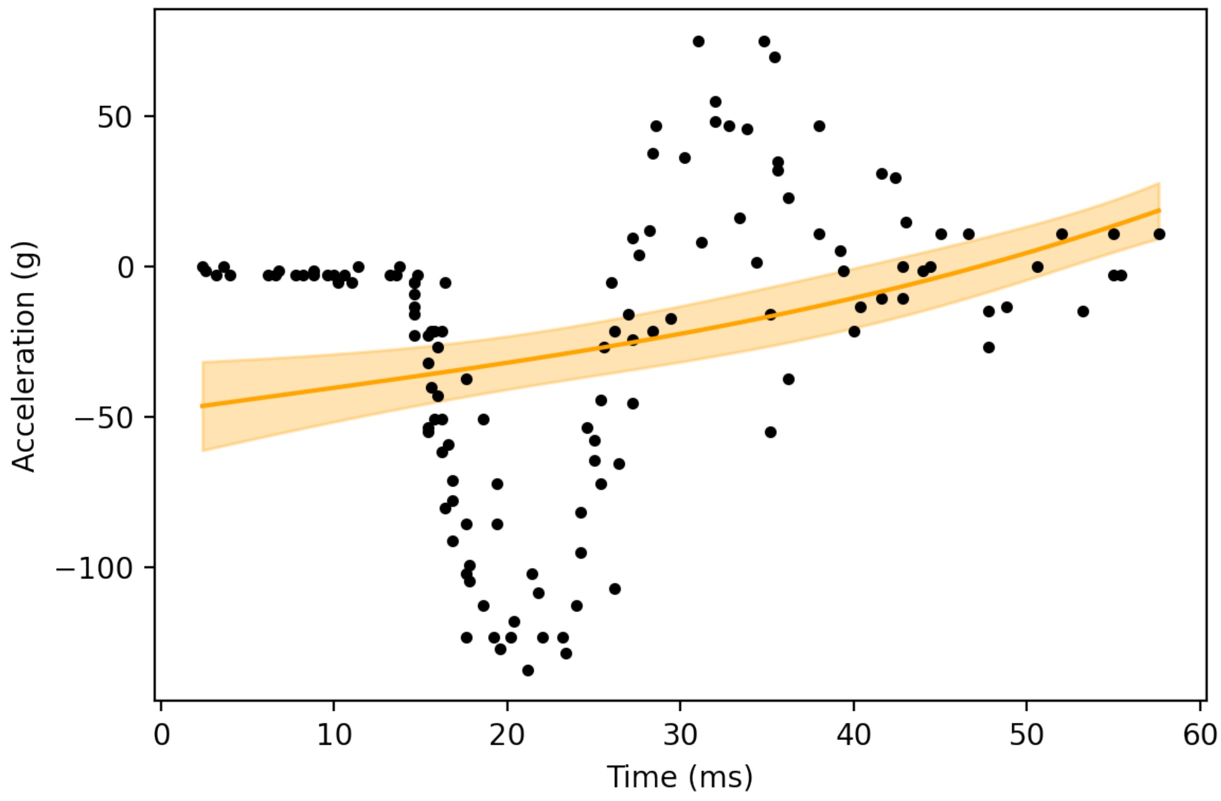
Here, we can see the estimated function from the GAM is smoother than the previous GAM. To continue along this trend, let's crank up the penalty further.

```
[11]: # Defining specifications for splines to built the additive design matrix
knot_locs = np.percentile(d['times'], q=np.linspace(2.5, 97.5, 11)).tolist()
specs = [None,                                                  # Spec for 1st term␣
↪(intercept)
         {"knots": knot_locs, "penalty": 1e11, "power": 3}]  # Spec for 2nd term (times)

# Defining the estimating equation
def psi(theta):
    return ee_additive_regression(theta=theta,
                                  X=X, y=y,
                                  model='linear',
                                  specifications=specs)

estr = MEstimator(psi, init=[0, 0, ] + [0, ]*10)
estr.estimate(solver='lm', maxiter=2000)

# Creating the corresponding additive design matrix
Xpa = additive_design_matrix(p[['intercept', 'times']],
                            specifications=specs)

# Generating predicted values
pred_y = regression_predictions(Xpa, estr.theta, estr.variance)
```

(continues on next page)

(continued from previous page)

```
# Plot
plt.plot(d['times'], d['accel'], '.', color='k')
plt.plot(p['times'], pred_y[:, 0], color='orange')
plt.fill_between(p['times'],
                 pred_y[:, 2], pred_y[:, 3],
                 color='orange', alpha=0.3)
plt.xlabel("Time (ms)")
plt.ylabel("Acceleration (g)")
plt.tight_layout()
```



Here, the GAM is very similar to the linear regression we started with. This helps to show the more general relationship: as the penalty goes to infinity, the GAM will coincide with the linear model from before.

To summarize, we want a penalty that prevents overfitting to the data but does not prevent the GAM from approximating the data. This selection of penalization is commonly done via cross-validation. However, use of cross-validation invalidates the variance estimation (including with the sandwich variance estimator). Therefore, I would recommend sacrificing the optimal penalty parameter (as determined by cross-validation) in favor of variance estimation with `delicatessen`.

Conclusions

GAMs are a flexible method that place fewer constraints on the functional forms of relationships via splines. These splines don't require us to know the true underlying functional form. Instead, the higher-order splines (with well-chosen knots) can serve as close approximations.

References

Silverman BW. (1985). Some aspects of the spline smoothing approach to non-parametric regression curve fitting. *Journal of the Royal Statistical Society: Series B (Methodological)*, 47(1), 1-21.

3.2.9 Code and Issue Tracker

Please report any bugs, issues, or feature requests on GitHub at pzivich/Delicatessen.

Otherwise, you may contact me via email (gmail: zivich.5).

3.3 Built-in Estimating Equations

Here, we provide an overview of some of the built-in estimating equations with `delicatessen`. This documentation is split into several sections based on topic areas.

All built-in estimating equations need to be 'wrapped' inside an outer function. Below is a generic example of an outer function, where `psi` is the wrapper function and `ee` is the generic estimating equation example (`ee` is not a valid built-in estimating equation).

```
def psi(theta):
    return ee(theta=theta, data=data)
```

Here, `ee` takes two inputs `theta` and `data`. `theta` is the general theta vector that is present in all stacked estimating equations expected by `delicatessen`. `data` is an argument that takes an input source of data. The `data` provided should be **in the local scope** of the `.py` file this function lives in.

After wrapped in an outer function, the function can be passed to `MEstimator`. See the examples below for further details and examples.

To replicate the following examples, please load the following libraries as shown

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from delicatessen import MEstimator
```

3.3.1 Basic Equations

Some basic estimating equations are provided.

Mean

The most basic available estimating equation is for the mean: ee_mean. To illustrate, consider we wanted to estimated the mean for the following data

```
obs_vals = [1, 2, 1, 4, 1, 4, 2, 4, 2, 3]
```

To use ee_mean with MEstimator, this function will be wrapped in an outer function. Below is an illustration of this wrapper function

```
from delicatessen.estimating_equations import ee_mean

def psi(theta):
    return ee_mean(theta=theta, y=obs_vals)
```

Note that obs_vals must be available in the scope of the defined function.

After creating the wrapper function, the corresponding M-Estimator can be called like the following

```
from delicatessen import MEstimator

estr = MEstimator(stacked_equations=psi, init=[0, ])
estr.estimate()

print(estr.theta)   # [2.4, ])
```

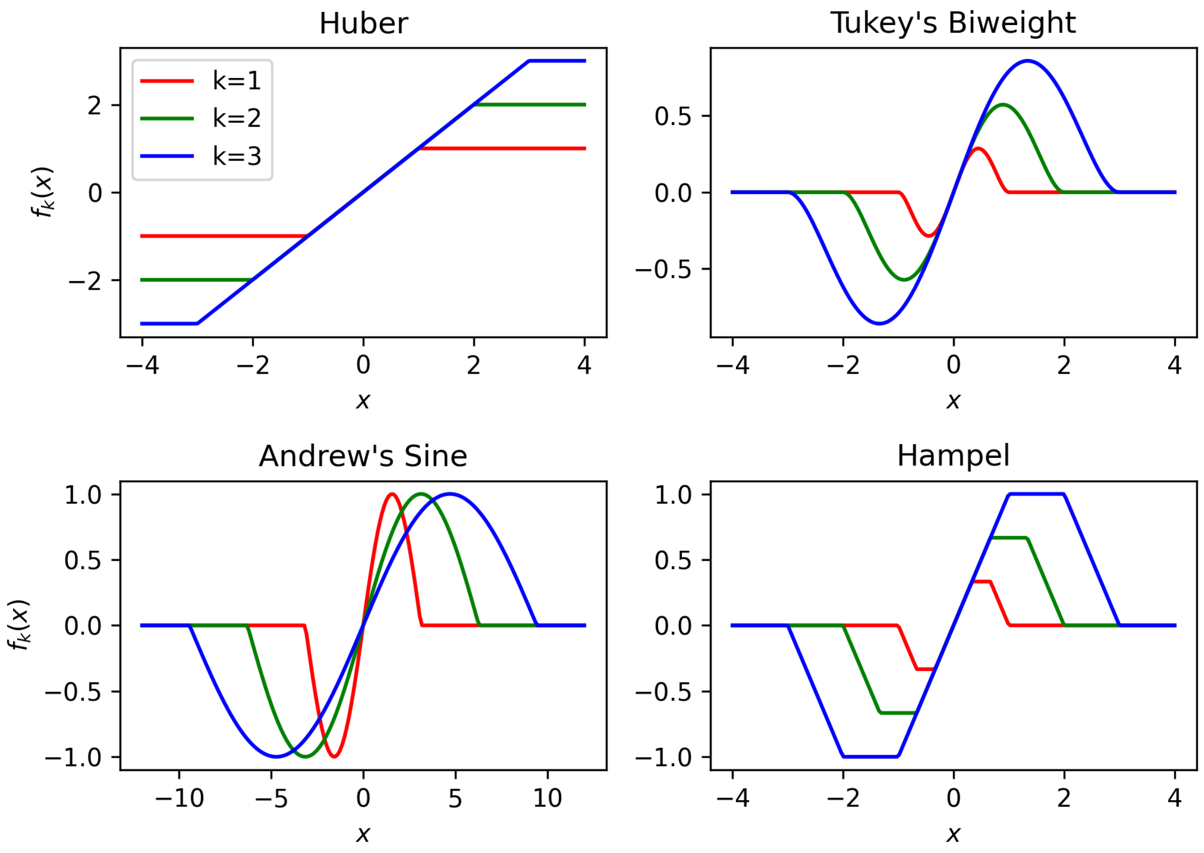
Since ee_mean consists of a single parameter, only a single init value is provided.

Robust Mean

Sometimes extreme observations, termed outliers, occur. The mean is generally sensitive to these outliers. A common approach to handling outliers is to exclude them. However, exclusion ignores all information contributed by outliers, and should only be done when outliers are a result of experimental error. Robust statistics have been proposed as middle ground, whereby outliers contribute to estimation but their overall influence is constrained.

```
obs_vals = [1, -10, 2, 1, 4, 1, 4, 2, 4, 2, 3, 12]
```

Instead, the robust mean can be used instead. The robust mean estimating equation is available in ee_mean_robust, with several different options for the loss function. The following is a plot showcasing the influence functions for the available robust loss functions.



```python
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_mean_robust

def psi(theta):
    return ee_mean_robust(theta=theta, y=obs_vals, loss='huber', k=6)

estr = MEstimator(stacked_equations=psi, init=[0, ])
estr.estimate()

print(estr.theta)
```

Mean and Variance

A stacked estimating equation, where the first value is the mean and the second is the variance, is also provided. Returning to the previous data,

```python
obs_vals = [1, 2, 1, 4, 1, 4, 2, 4, 2, 3]
```

The mean-variance estimating equation can be implemented as follows

```python
from delicatessen.estimating_equations import ee_mean_variance

def psi(theta):
```

(continues on next page)

(continued from previous page)

```
    return ee_mean_variance(theta=theta, y=obs_vals)

estr = MEstimator(stacked_equations=psi, init=[0, 1, ])
estr.estimate()

print(estr.theta)  # [2.4, 1.44]
```

Note `init` here takes two values because there are two parameters. The first value of `theta` is the mean and the second is the variance. Now, the variance output provides a 2-by-2 covariance matrix. The leading diagonal of that matrix are the variances (where the first is the estimated variance of the mean and the second is the estimated variance of the variance).

3.3.2 Regression

Several common regression models are provided as built-in estimating equations.

Linear Regression

The estimating equations for linear regression predict a continuous outcome as a function of provided covariates.

To demonstrate application, consider the following simulated data set

```
import numpy as np
import pandas as pd

n = 500
data = pd.DataFrame()
data['X'] = np.random.normal(size=n)
data['Z'] = np.random.normal(size=n)
data['Y1'] = 0.5 + 2*data['X'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(0.5 + 2*data['X'] - 1*data['Z']),␣
↪size=n)
data['Y3'] = data['Y3'] = np.random.poisson(lam=np.exp(0.5 + 2*data['X'] - 1*data['Z']),␣
↪size=n)
data['C'] = 1
```

In this case, `X` and `Z` are the independent variables and `Y` is the dependent variable. Here the column `C` is created to be the intercept column, since the intercept needs to be manually provided (this may be different from other formula-based packages that automatically add the intercept to the regression).

For this data, we can now create the wrapper function for the `ee_regression` estimating equations

```
from delicatessen.estimating_equations import ee_regression

def psi(theta):
    return ee_regression(theta=theta,
                         X=data[['C', 'X', 'Z']],
                         y=data['Y1'],
                         model='linear')
```

After creating the wrapper function, we can now call the M-Estimation procedure to estimate the regression coefficients and their variance

```
estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.])
estr.estimate(solver='lm')
```

Note that there are 3 independent variables, meaning `init` needs 3 starting values. The linear regression done here should match the `statsmodels` generalized linear model with their robust covariance estimate. Below is code on how to compare to `statsmodels.glm`.

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

glm = smf.glm("Y ~ X + Z", data).fit(cov_type="HC1")
print(np.asarray(glm.params))         # Point estimates
print(np.asarray(glm.cov_params()))   # Covariance matrix
```

While `statsmodels` likely runs faster, the benefit of M-estimation and `delicatessen` is that multiple estimating equations can be stacked together (including multiple regression models).

Logistic Regression

In the case of a binary dependent variable, logistic regression can instead be performed. Consider the following simulated data set

In this case, `X` and `Z` are the independent variables and `Y` is the dependent variable. Here the column `C` is created to be the intercept column, since the intercept needs to be manually provided (this may be different from other formula-based packages that automatically add the intercept to the regression).

For this data, we can now create the wrapper function for the `ee_regression` estimating equations

```
def psi(theta):
    return ee_regression(theta=theta,
                         X=data[['C', 'X', 'Z']],
                         y=data['Y2'],
                         model='logistic')
```

After creating the wrapper function, we can now call the M-Estimation procedure to estimate the regression coefficients and their variance

```
estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.])
estr.estimate(solver='lm')
```

Note that there are 3 independent variables, meaning `init` needs 3 starting values. The logistic regression done here should match the `statsmodels` generalized linear model with a robust variance estimate. Below is code on how to compare to `statsmodels.glm`.

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

glm = smf.glm("Y2 ~ X + Z", data,
              family=sm.families.Binomial()).fit(cov_type="HC1")
print(np.asarray(glm.params))         # Point estimates
print(np.asarray(glm.cov_params()))   # Covariance matrix
```

While `statsmodels` likely runs faster, the benefit of M-estimation and `delicatessen` is that multiple estimating equations can be stacked together (including multiple regression models). This advantage will become clearer in the causal section.

Poisson Regression

In the case of a count dependent variable, Poisson regression can instead be performed. Consider the following simulated data set

In this case, `X` and `Z` are the independent variables and `Y` is the dependent variable. Here the column `C` is created to be the intercept column, since the intercept needs to be manually provided (this may be different from other formula-based packages that automatically add the intercept to the regression).

For this data, we can now create the wrapper function for the `ee_regression` estimating equations

```
def psi(theta):
    return ee_regression(theta=theta,
                         X=data[['C', 'X', 'Z']],
                         y=data['Y3'],
                         model='poisson')
```

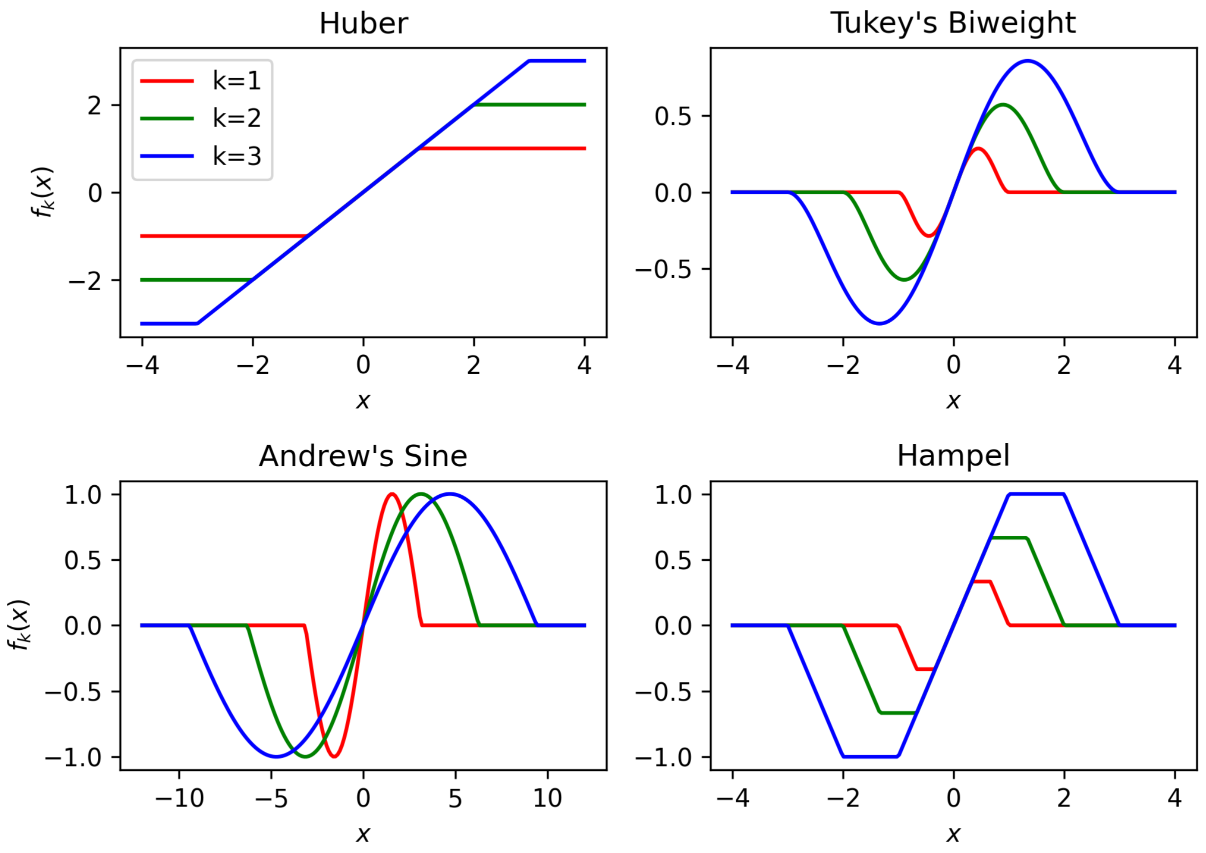
After creating the wrapper function, we can now call the M-Estimation procedure to estimate the regression coefficients and their variance

```
estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.])
estr.estimate(solver='lm')
```

Note that there are 3 independent variables, meaning `init` needs 3 starting values.

3.3.3 Robust Regression

Similar to the mean, linear regression can also be made robust to outliers. This is simply accomplished by placing a loss function on the residuals. Several loss functions are available. The following is a plot showcasing the influence functions for the available robust loss functions.



Continuing with the data generated in the previous example, robust linear regression with Huber's loss function can be implemented as follows

```python
from delicatessen.estimating_equations import ee_robust_regression

def psi(theta):
    return ee_robust_regression(theta=theta,
                                X=data[['C', 'X', 'Z']],
                                y=data['Y1'],
                                model='linear', loss='huber', k=1.345)
```

After creating the wrapper function, we can now call the M-Estimation procedure

```python
estr = MEstimator(stacked_equations=psi, init=[0.5, 2., -1.])
estr.estimate(solver='lm')
```

Note: to help the root-finding procedure, we generally recommend using the simple linear regression values as the initial values for robust linear regression.

Robust regression is only available for linear regression models.

3.3.4 Penalized Regression

There is also penalized regression models available. Here, we will demonstrate for linear regression, but logistic and Poisson penalized regression are also supported.

To demonstrate application of the penalized regression models, consider the following simulated data set

```
from delicatessen.estimating_equations import (ee_ridge_regression,
                                               ee_lasso_regression,
                                               ee_elasticnet_regression,
                                               ee_bridge_regression)


n = 500
data = pd.DataFrame()
data['V'] = np.random.normal(size=n)
data['W'] = np.random.normal(size=n)
data['X'] = data['W'] + np.random.normal(scale=0.25, size=n)
data['Z'] = np.random.normal(size=n)
data['Y'] = 0.5 + 2*data['W'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
data['C'] = 1
```

Here, there is both variables with no effect and collinearity in the data.

Ridge Penalty

The Ridge or L_2 penalty is intended to penalize collinear terms. The penalty term in the estimating equations is

$$2\frac{\lambda}{n}|\beta|\text{sign}(\beta)$$

where λ is the penalty term (and is scaled by n) and β are the regression coefficients.

To implement ridge regression, the estimating equations can be specified as

```
penalty_vals = [0., 10., 10., 10., 10.]
def psi(theta):
    x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y1']
    return ee_ridge_regression(theta=theta, X=x, y=y, model='linear',
                               penalty=penalty_vals)
```

Here, $\lambda = 10$ for all coefficients, besides the intercept. The M-estimator is then implemented via

```
estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.])
estr.estimate(solver='lm')
```

Different penalty terms can be assigned to each coefficient. Furthermore, the `center` argument can be used to penalize towards non-zero values for all or some of the coefficients.

Bridge Penalty

The bridge penalty is a generalization of the L_p penalty, with the Ridge ($p = 2$) and LASSO ($p = 1$) as special cases. In the estimating equations, the bridge penalty is

$$\gamma \frac{\lambda}{n} |\beta|^{\gamma-1} \text{sign}(\beta)$$

where $\gamma > 0$. However, only $\gamma \geq 1$ is supported in `delicatessen` (due to the no roots potentially existing when $\gamma < 1$). Additionally, the empirical sandwich variance estimator is not valid when $\gamma < 2$, and a nonparametric bootstrap should be used to estimate the variance instead

To implement bridge regression, the estimating equations can be specified as

```python
penalty_vals = [0., 10., 10., 10., 10.]
def psi(theta):
    x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y']
    return ee_bridge_regression(theta=theta, X=x, y=y,
                                model='linear',
                                gamma=2.3, penalty=penalty_vals)
```

where γ is the p value in L_p. Setting $\gamma = 1$ is the LASSO penalty and $\gamma = 2$ is the Ridge penalty. Here, we use a value larger than 2 for demonstration.

```python
estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.])
estr.estimate(solver='lm')
```

Different penalty terms can be assigned to each coefficient. Furthermore, the `center` argument can be used to penalize towards non-zero values for all or some of the coefficients.

3.3.5 Flexible Regression

The previous regression models generally rely on strong parametric assumptions (unless explicitly relaxed by the user through the specified design matrix). An alternative is to use more flexible regression models, which place less strict parametric assumptions on the model. Here, we will demonstrate flexible models for linear regression, but logistic and Poisson regression are also supported.

To demonstrate application of the flexible regression models, consider the following simulated data set

```python
from delicatessen.estimating_equations import ee_additive_regression
from delicatessen.utilities import additive_design_matrix

n = 2000
d = pd.DataFrame()
d['X'] = np.random.uniform(-5, 5, size=n)
d['Z'] = np.random.binomial(n=1, p=0.5, size=n)
d['Y'] = 2*d['Z'] + np.exp(np.sin(d['X'] + 0.5)) + np.abs(d['X']) + np.random.
↪normal(size=n)
d['C'] = 1
```

Here, there the relationship between X and Y is nonlinear. The flexible regression models will attempt to capture this flexibility without the user having to directly specify the functional form.

Generalized Additive Model

Generalized Additive Models (GAMs) are an extension of Generalized Linear Models (GLMs) that replace linear terms in the model with an arbitrary (but user-specified) function. For the GLM, we might consider the following model

$$Y_i = \beta_0 + \beta_1 Z_i + \beta_2 X_i + \epsilon_i$$

However, this model assumes that the relationship between X and Y is linear (which we known to be untrue in this case). GAMs work by replacing the linear term with a spline function. For the GAM, we might consider the following model

$$Y_i = \beta_0 + \beta_1 Z_i + \beta_2 X_i + \sum_k \beta_k f_k(X_k) + \epsilon_i$$

Here, X was replaced with a set of function. Those functions define a pre-specified number of spline terms. These spline terms allow for the relationship between X and Y to be modeled in a flexible but smooth way. However, this flexibility is not free. If our splines are complex, the GAM can overfit the data. To help prevent this issue, GAMs generally use penalized splines, where the coefficients for the spline terms are penalized. `delicatessen` uses L2 penalization and allows various specifications for the splines.

The main trick of the GAM is to generate a new design matrix for the additive model based on some input design matrix and spline specifications. This is done (internally) by the `additive_design_matrix` function. This can also be directly called

```python
x_knots = np.linspace(-4.75, 4.75, 30)
specs = [None,                                  # No spline for intercept
         None,                                  # No spline for Z
         {"knots": x_knots, "penalty": 20},    # Spline specs for X
         ]
Xa = additive_design_matrix(X=data[['C', 'Z', 'X']], specifications=specs)
```

Here, a design matrix is return where the first two columns (C and Z) have no spline terms generated. For the last column (X), a natural cubic spline with 30 evenly spaced knots and a penalty of 20 is generated. So the output design matrix will consist of the C,Z,X columns followed by the 29 column basis of the splines.

To implement a GAM, the estimating equations can be specified as

```python
def psi(theta):
    return ee_additive_regression(theta=theta,
                                  X=d[['C', 'Z', 'X']], y=d['Y'],
                                  specifications=specs,
                                  model='linear')
```

Here, the previously defined spline specifications are provided. Internally, `ee_additive_regression` calls the `additive_design_matrix`, so this design matrix does not have to be provided by the user. However, pre-computing the design matrix is helpful for determining the number of initial values. To determine the number of initial values to provide `MEstimator`, we can check the number of columns in `Xa`. In the following, we use the number of columns to generate a list of starting values.

```python
estr = MEstimator(psi, init=[0, ]*Xa.shape[1])
estr.estimate(solver='lm', maxiter=10000)
```

Multiple splines, different types of splines, or varying penalty strengths can also be specified. These specifications are all done through the list of dictionaries provided in the `specifications` arguments. Any element with a dictionary will have splines generated and any `None` element will only have the main term returned. See the `ee_additive_regression` and `additive_design_matrix` reference pages for further examples.

3.3.6 Survival

Suppose each person has two unique times: their event time (T_i) and their censoring time (C_i). However, we are only able to observe whichever one of those times occurs first. Therefore the observable data is $T_i^* = \min(T_i, C_i)$ and $\delta_i = I(T_i^* = T_i)$. However, we want to estimate some probability of events using T_i^*, δ_i For an introduction to survival analysis, I would recommend Collett D. (2015). "Modelling survival data in medical research".

Currently available estimating equations for parametric survival models are: exponential and Weibull models, and accelerated failure time models (AFT). For the basic survival models, we will use the following generated data set. In accordance with the description above, each person is assigned two possible times and then we generate the observed data (`t` and `delta` here).

```
n = 100
d = pd.DataFrame()
d['C'] = np.random.weibull(a=1, size=n)
d['C'] = np.where(d['C'] > 5, 5, d['C'])
d['T'] = 0.8 * np.random.weibull(a=0.75, size=n)
d['delta'] = np.where(d['T'] < d['C'], 1, 0)
d['t'] = np.where(d['delta'] == 1, d['T'], d['C'])
```

Exponential

The exponential model is a one-parameter model, that stipulates the hazard of the event of interest is constant. While often too restrictive of an assumption, we demonstrate application here.

```
from delicatessen.estimating_equations import ee_exponential_model, ee_exponential_
↪measure
```

The wrapper function for the exponential model should look like

```
def psi(theta):
    # Estimating equations for the exponential model
    return ee_exponential_model(theta=theta, t=d['t'], delta=d['delta'])
```

After creating the wrapper function, we can now call the M-Estimation procedure to estimate the parameter for the exponential model

```
estr = MEstimator(psi, init=[1., ])
estr.estimate(solver='lm')
```

Here, the parameter for the exponential model should be non-negative, so a positive value should be given to help the root-finding procedure.

While the parameter for the exponential model may be of interest, we are often more interested in the one of the functions over time. For example, we may want to plot the estimated survival function over time. `delicatessen` provides a function to estimate the survival (or other measures like density, risk, hazard, cumulative hazard) at provided time points.

Below is how we could further generate a plot of the survival function from the estimated exponential model

```
resolution = 50
time_spacing = list(np.linspace(0.01, 5, resolution))
fast_inits = [0.5, ]*resolution

```

(continues on next page)

(continued from previous page)

```python
def psi(theta):
    ee_exp = ee_exponential_model(theta=theta[0],
                                  t=times, delta=events)
    ee_surv = ee_exponential_measure(theta[1:], scale=theta[0],
                                     times=time_spacing, n=times.shape[0],
                                     measure="survival")
    return np.vstack((ee_exp, ee_surv))

estr = MEstimator(psi, init=[1., ] + fast_inits)
estr.estimate(solver="lm")

# Creating plot of survival times
ci = mestr.confidence_intervals()[1:, :]  # Extracting relevant CI
plt.fill_between(time_spacing, ci[:, 0], ci[:, 1], alpha=0.2)
plt.plot(time_spacing, mestr.theta[1:], '-')
plt.show()
```

Here, we set the `resolution` to be 50. The resolution determines how many points along the survival function we are evaluating (and thus determines how ‘smooth’ our plot will appear). As this involves the root-finding of multiple values, it is important to help the root-finder along by providing good starting values. Since survival is bounded between [0,1], we have all the initial values for those start at 0.5 (the middle). Furthermore, we could also consider pre-washing the exponential model parameter (i.e., use the solution from the previous estimating equation).

Weibull

The Weibull model is a generalization of the exponential model. The Weibull model allows for the hazard to vary over time (it can increase or decrease monotonically).

```python
from delicatessen.estimating_equations import ee_weibull_model, ee_weibull_measure
```

The wrapper function for the Weibull model should look like

```python
def psi(theta):
    # Estimating equations for the Weibull model
    return ee_weibull_model(theta=theta, t=d['t'], delta=d['delta'])
```

After creating the wrapper function, we can now call the M-Estimation procedure to estimate the parameters for the Weibull model

```python
estr = MEstimator(psi, init=[1., 1.])
estr.estimate(solver='lm')
```

Here, the parameters for the Weibull model should be non-negative (the optimizer does not know this), so a positive value should be given to help the root-finding procedure along.

While the parameters for the Weibull model may be of interest, we are often more interested in the one of the functions over time. For example, we may want to plot the estimated survival function over time. `delicatessen` provides a function to estimate the survival (or other measures like density, risk, hazard, cumulative hazard) at provided time points.

Below is how we could further generate a plot of the survival function from the estimated Weibull model

```
import matplotlib.pyplot as plt

resolution = 50
time_spacing = list(np.linspace(0.01, 5, resolution))
fast_inits = [0.5, ]*resolution

def psi(theta):
    ee_wbf = ee_weibull_model(theta=theta[0:2],
                              t=times, delta=events)
    ee_surv = ee_weibull_measure(theta[2:], scale=theta[0], shape=theta[1],
                                 times=time_spacing, n=times.shape[0],
                                 measure="survival")
    return np.vstack((ee_wbf, ee_surv))

estr = MEstimator(psi, init=[1., 1., ] + fast_inits)
estr.estimate(solver="lm")

# Creating plot of survival times
ci = mestr.confidence_intervals()[2:, :]  # Extracting relevant CI
plt.fill_between(time_spacing, ci[:, 0], ci[:, 1], alpha=0.2)
plt.plot(time_spacing, mestr.theta[2:], '-')
plt.show()
```

Here, we set the `resolution` to be 50. The resolution determines how many points along the survival function we are evaluating (and thus determines how 'smooth' our plot will appear). As this involves the root-finding of multiple values, it is important to help the root-finder along by providing good starting values. Since survival is bounded between [0,1], we have all the initial values for those start at 0.5 (the middle). Furthermore, we could also consider pre-washing the Weibull model parameter (i.e., use the solution from the previous estimating equation).

Accelerated Failure Time

Currently, only an AFT model with a Weibull (Weibull-AFT) is available for use. Unlike the previous exponential and Weibull models, the AFT models can further include covariates, where the effect of a covariate is interpreted as an 'acceleration' factor. In the two sample case, the AFT can be thought of as the following

$$S_1(t) = S_0(t/\sigma)$$

where $\sigma^{-1} > 0$ and is interpreted as the acceleration factor. One way to describe is that the risk of the event in group 1 at $t = 1$ is equivalent to group 0 at $t = \sigma^{-1}$. Alternatively, you can interpret the the AFT coefficient as the ratio of the mean survival times comparing group 1 to group 0. While requiring strong parametric assumptions, AFT models have the advantage of providing a single summary measure (compared to nonparametric methods, like Kaplan-Meier) but also being relatively easy to interpret (compared to semiparametric Cox models).

For the following examples, we generate some additional survival data with baseline covariates

```
n = 200
d = pd.DataFrame()
d['X'] = np.random.binomial(n=1, p=0.5, size=n)
d['W'] = np.random.binomial(n=1, p=0.5, size=n)
d['T'] = (1 / 1.25 + 1 / np.exp(0.5) * d['X']) * np.random.weibull(a=0.75, size=n)
d['C'] = np.random.weibull(a=1, size=n)
d['C'] = np.where(d['C'] > 10, 10, d['C'])
d['delta'] = np.where(d['T'] < d['C'], 1, 0)
d['t'] = np.where(d['delta'] == 1, d['T'], d['C'])
```

There are variations on the AFT model. These variations place parametric assumptions on the error distribution.

Weibull AFT

The Weibull AFT assumes that errors follow a Weibull distribution. Therefore, the Weibull AFT consists of a shape and scale parameter (like the Weibull model from before) but not it further includes parameters for each covariate included in the AFT model.

```
from delicatessen.estimating_equations import ee_aft_weibull, ee_aft_weibull_measure
```

The wrapper function for the Weibull AFT model should look like

```
def psi(theta):
    # Estimating equations for the Weibull AFT model
    return ee_aft_weibull(theta=theta,
                          t=d['t'], delta=d['delta'],
                          X=d[['X', 'W']])
```

After creating the wrapper function, we can now call the M-estimator to estimate the parameters for the Weibull model

```
estr = MEstimator(psi, init=[0., 0., 0., 0.])
estr.estimate(solver='lm')

print(estr.theta)
print(estr.variance)
```

Unlike the previous models, the Weibull AFT model parameters are log-transformed. Therefore, starting values of zero can be input for the root-finding procedure.

Here, `theta[0]` is the log-transformed intercept term for the shape parameter, and `theta[-1]` is the log-transformed scale parameter. The middle terms (`theta[1:3]` in this case) corresponds to the acceleration factors for the covariates in their input order. Therefore, `theta[1]` is the acceleration factor for `'X'` and `theta[2]` is the acceleration factor for `'W'`.

While the parameters for the Weibull model may be of interest, we are often more interested in the one of the functions over time. For example, we may want to plot the estimated survival function over time. `delicatessen` provides a function to estimate the survival (or other measures like density, risk, hazard, cumulative hazard) at specified time points.

Below is how we could further generate a plot of the survival function from the estimated Weibull AFT model. Unlike the other survival models, we also need to specify the covariate pattern of interest. Here, we will generate the survival function when both $X = 1$ and $W = 1$

```
import matplotlib.pyplot as plt

resolution = 50
time_spacing = list(np.linspace(0.01, 5, resolution))
fast_inits = [0.5, ]*resolution
dc = d.copy()
dc['X'] = 1
dc['W'] = 1

def psi(theta):
    ee_aft = ee_aft_weibull(theta=theta,
```

(continues on next page)

(continued from previous page)

```
                            t=d['t'], delta=d['delta'],
                            X=d[['X', 'W']])
    pred_surv_t = ee_aft_weibull_measure(theta=theta[4:], X=dc[['X', 'W']],
                                         times=time_spacing, measure='survival',
                                         mu=theta[0], beta=theta[1:3], sigma=theta[3])
    return np.vstack((ee_aft, pred_surv_t))

estr = MEstimator(psi, init=[0., 0., 0., 0., ] + fast_inits)
estr.estimate(solver="lm")

# Creating plot of survival times
ci = mestr.confidence_intervals()[4:, :]  # Extracting relevant CI
plt.fill_between(time_spacing, ci[:, 0], ci[:, 1], alpha=0.2)
plt.plot(time_spacing, mestr.theta[4:], '-')
plt.show()
```

Here, we set the `resolution` to be 50. The resolution determines how many points along the survival function we are evaluating (and thus determines how 'smooth' our plot will appear).

As this involves the root-finding of multiple values, it is important to help the root-finder along by providing good starting values. Since survival is bounded between [0,1], we have all the initial values for those start at 0.5. Furthermore, models like Weibull AFT should be used with pre-washing the AFT model parameters (i.e., use the solution from the previous estimating equation).

3.3.7 Dose-Response

Estimating equations for dose-response relationships are also included. The following examples use the data from Inderjit et al. (2002). This data can be loaded via

```
d = load_inderjit()   # Loading array of data
dose_data = d[:, 1]   # Dose data
resp_data = d[:, 0]   # Response data
```

4-Parameter Log-Logistic

The 4-parameter logistic model (4PL) consists of parameters for the lower-limit of the response, the effective dose, steepness of the curve, and the upper-limit of the response.

The wrapper function for the 4PL model should look like

```
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_4p_logistic

def psi(theta):
    # Estimating equations for the 4PL model
    return ee_4p_logistic(theta=theta, X=dose_data, y=resp_data)
```

After creating the wrapper function, we can now call the M-Estimation procedure to estimate the coefficients for the 4PL model and their variance

```
estr = MEstimator(psi, init=[np.min(resp_data),
                              (np.max(resp_data)+np.min(resp_data)) / 2,
                              (np.max(resp_data)+np.min(resp_data)) / 2,
                              np.max(resp_data)])
estr.estimate(solver='lm')

print(estr.theta)
print(estr.variance)
```

When you use 4PL, you may notice convergence errors. This estimating equation can be hard to optimize since it has implicit bounds the root-finder isn't aware of. To avoid these issues, we can give the root-finder good starting values.

First, the upper limit should *always* be greater than the lower limit. Second, the ED50 should be between the lower and upper limits. Third, the sign for the steepness depends on whether the response declines (positive) or the response increases (negative). Finally, some solvers may be better suited to the problem, so try a few different options. With decent initial values, we have found `lm` to be fairly reliable.

For the 4PL, good general starting values I have found are the following. For the lower-bound, give the minimum response value as the initial. For ED50, give the median response. The initial value for steepness is more difficult. Ideally, we would give a starting value of zero, but that will fail in this 4PL. Giving a larger starting value (between 2 to 8) works in this example. For the upper-bound, give the maximum response value as the initial.

To summarize, be sure to examine your data (e.g., scatterplot). This will help to determine the initial starting values for the root-finding procedure. Otherwise, you may come across a convergence error.

3-Parameter Log-Logistic

The 3-parameter logistic model (3PL) consists of parameters for the effective dose, steepness of the curve, and the upper-limit of the response. Here, the lower-limit is pre-specified and is no longer being estimated.

The wrapper function for the 3PL model should look like

```
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_3p_logistic

def psi(theta):
    # Estimating equations for the 3PL model
    return ee_3p_logistic(theta=theta, X=dose_data, y=resp_data,
                          lower=0)
```

Since the shortest a root of a plant could be zero, a lower limit of zero makes sense here.

After creating the wrapper function, we can now call the M-Estimation procedure to estimate the coefficients for the 3PL model and their variance

```
estr = MEstimator(psi, init=[(np.max(resp_data)+np.min(resp_data)) / 2,
                              (np.max(resp_data)+np.min(resp_data)) / 2,
                              np.max(resp_data)])
estr.estimate(solver='lm')

print(estr.theta)
print(estr.variance)
```

As before, you may notice convergence errors. This estimating equation can be hard to optimize since it has implicit bounds the root-finder isn't aware of. To avoid these issues, we can give the root-finder good starting values.

For the 3PL, good general starting values I have found are the following. For ED50, give the mid-point between the maximum response and the minimum response. The initial value for steepness is more difficult. Ideally, we would give a starting value of zero, but that will fail in this 3PL. Giving a larger starting value (between 2 to 8) works in this example. For the upper-bound, give the maximum response value as the initial.

To summarize, be sure to examine your data (e.g., scatterplot). This will help to determine the initial starting values for the root-finding procedure. Otherwise, you may come across a convergence error.

2-Parameter Log-Logistic

The 2-parameter logistic model (2PL) consists of parameters for the effective dose, and steepness of the curve. Here, the lower-limit and upper-limit are pre-specified and no longer being estimated.

The wrapper function for the 3PL model should look like

```
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_2p_logistic

def psi(theta):
    # Estimating equations for the 2PL model
    return ee_2p_logistic(theta=theta, X=dose_data, y=resp_data,
                          lower=0, upper=8)
```

While a lower-limit of zero makes sense in this example, the upper-limit of 8 is poorly motivated (and thus this should only be viewed as an example of the 2PL model and not how it should be applied in practice). Setting the limits as constants should be motivated by substantive knowledge of the problem.

After creating the wrapper function, we can now call the M-estimator to estimate the coefficients for the 2PL model and their variance

```
estr = MEstimator(psi, init=[(np.max(resp_data)+np.min(resp_data)) / 2,
                              (np.max(resp_data)+np.min(resp_data)) / 2])
estr.estimate(solver='lm')

print(estr.theta)
print(estr.variance)
```

As before, you may notice convergence errors. To avoid these issues, we can give the root-finder good starting values.

For the 2PL, good general starting values I have found are the following. For ED50, give the mid-point between the maximum response and the minimum response. The initial value for steepness is more difficult. Ideally, we would give a starting value of zero, but that will fail in this 2PL.

To summarize, be sure to examine your data (e.g., scatterplot). This will help to determine the initial starting values for the root-finding procedure.

ED(δ)

In addition to the x-parameter logistic models, an estimating equation to estimate a corresponding δ effective dose is available. Notice that this estimating equation should be stacked with one of the x-PL models. Here, we demonstrate with the 3PL model.

Here, our interest is in the following effective doses: 0.05, 0.10, 0.20, 0.80. The wrapper function for the 3PL model and estimating equations for these effective doses are

```python
def psi(theta):
    lower_limit = 0

    # Estimating equations for the 3PL model
    pl3 = ee_3p_logistic(theta=theta, X=d[:, 1], y=d[:, 0],
                         lower=lower_limit)

    # Estimating equations for the effective concentrations
    ed05 = ee_effective_dose_delta(theta[3], y=resp_data, delta=0.05,
                                   steepness=theta[0], ed50=theta[1],
                                   lower=lower_limit, upper=theta[2])
    ed10 = ee_effective_dose_delta(theta[4], y=resp_data, delta=0.10,
                                   steepness=theta[0], ed50=theta[1],
                                   lower=lower_limit, upper=theta[2])
    ed20 = ee_effective_dose_delta(theta[5], y=resp_data, delta=0.20,
                                   steepness=theta[0], ed50=theta[1],
                                   lower=lower_limit, upper=theta[2])
    ed80 = ee_effective_dose_delta(theta[6], y=resp_data, delta=0.80,
                                   steepness=theta[0], ed50=theta[1],
                                   lower=lower_limit, upper=theta[2])

    # Returning stacked estimating equations
    return np.vstack((pl3,
                      ed05,
                      ed10,
                      ed20,
                      ed80))
```

Notice that the estimating equations are stacked together in the order of the `theta` vector.

After creating the wrapper function, we can now estimate the coefficients for the 3PL model, the ED for the δ values, and their variance

```python
midpoint = (np.max(resp_data)+np.min(resp_data)) / 2
estr = MEstimator(psi, init=[midpoint,
                             midpoint,
                             np.max(resp_data),
                             midpoint,
                             midpoint,
                             midpoint,
                             midpoint])
estr.estimate(solver='lm')
print(estr.theta)
print(estr.variance)
```

Since the ED for δ's are transformations of the other parameters, there starting values are less important (the rootfinders are better at solving those equations). Again, we can make it easy on the solver by having the starting point for each being the mid-point of the response values.

3.3.8 Causal Inference

This next section describes available estimators for the average causal effect. These estimators all rely on specific identification conditions to be able to interpret the mean difference as an estimate of the causal mean. For information on these assumptions, I recommend this this paper as an introduction.

This section proceeds under the assumption that the identification conditions have been previously deliberated, and the average causal effect is identified and is estimable (see arXiv2108.11342 or arXiv1904.02826 for more information on estimability).

With that aside, let's proceed through the available estimators of the causal mean. In the following examples, we will use the generic data example here, where $Y(a)$ is independent of A conditional on W. Below is a sample data set

```
n = 200
d = pd.DataFrame()
d['W'] = np.random.binomial(1, p=0.5, size=n)
d['A'] = np.random.binomial(1, p=(0.25 + 0.5*d['W']), size=n)
d['Ya0'] = np.random.binomial(1, p=(0.75 - 0.5*d['W']), size=n)
d['Ya1'] = np.random.binomial(1, p=(0.75 - 0.5*d['W'] - 0.1*1), size=n)
d['Y'] = (1-d['A'])*d['Ya0'] + d['A']*d['Ya1']
d['C'] = 1
```

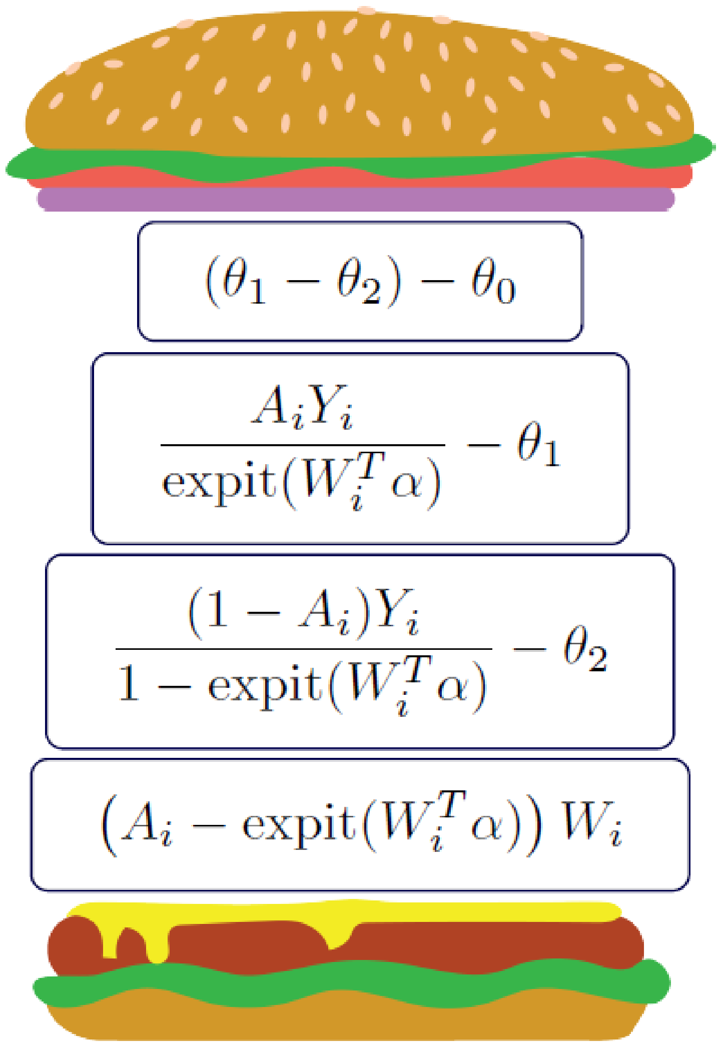
Here, we don't get to see the potential outcomes $Ya0$ or $Ya1$, but instead estimate the mean under different plans using the observed data, Y, A, W.

Inverse probability weighting

First, we use the inverse probability weighting (IPW) estimator, which models the probability of A conditional on W in order to estimate the average causal effect. In general, the Horvitz-Thompson IPW estimator for the mean difference can be written as

$$\frac{1}{n} \sum_{i=1}^{n} \frac{Y_i A_i}{Pr(A_i = 1 | W_i; \hat{\alpha})} - \frac{1}{n} \sum_{i=1}^{n} \frac{Y_i (1 - A_i)}{Pr(A_i = 0 | W_i; \hat{\alpha})}$$

In `delicatessen`, the built-in IPW estimator consists of 4 estimating equations, with both binary and continuous outcomes supported by `ee_ipw` (since we are using the Horwitz-Thompson estimator). The stacked estimating equations are


$(\theta_1 - \theta_2) - \theta_0$
$\frac{A_i Y_i}{\text{expit}(W_i^T \alpha)} - \theta_1$
$\frac{(1 - A_i) Y_i}{1 - \text{expit}(W_i^T \alpha)} - \theta_2$
$\left(A_i - \text{expit}(W_i^T \alpha)\right) W_i$

where θ_1 is the average causal effect, θ_2 is the mean under the plan where $A = 1$ for everyone, θ_3 is the mean under the plan where $A = 0$ for everyone, and α is the parameters for the logistic model used to estimate the propensity scores.

To load the pre-built IPW estimating equations,

```
from delicatessen.estimating_equations import ee_ipw
```

The estimating equation is then wrapped inside the wrapper psi function. Notice that the estimating equation has 4 non-optional inputs: the parameter values, the outcomes, the actions, and the covariates to model the propensity scores with.

```
def psi(theta):
    return ee_ipw(theta,                    # Parameters
                  y=d['Y'],                 # Outcome
                  A=d['A'],                 # Action (exposure, treatment, etc.)
                  W=d[['C', 'W']])          # Design matrix for PS model
```

Note that we add an intercept to the logistic model by adding a column of 1's via d['C'].

Here, the initial values provided must be 3 + b (where b is the number of columns in W). For binary outcomes, it will likely be best practice to have the initial values set as [0., 0.5, 0.5, ...]. followed by b 0.'s. For continuous outcomes, all `0.` can be used instead. Furthermore, a logistic model for the propensity scores could be optimized outside of `delicatessen` and those (pre-washed) regression estimates can be passed as initial values to speed up optimization.

Now we can call the M-estimator to solve for the values and the variance.

```
estr = MEstimator(psi, init=[0., 0.5, 0.5, 0., 0.])
estr.estimate(solver='lm')
```

After successful optimization, we can inspect the estimated values.

```
estr.theta[0]    # causal mean difference of 1 versus 0
estr.theta[1]    # causal mean under X1
estr.theta[2]    # causal mean under X0
estr.theta[3:]   # logistic regression coefficients
```

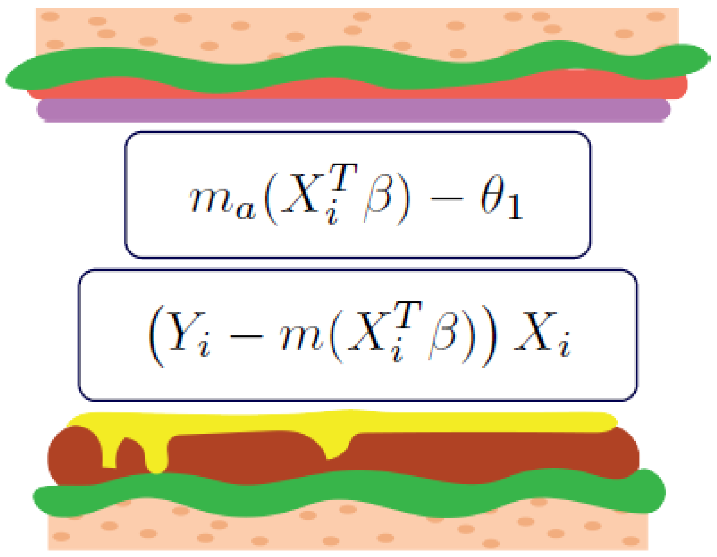
The IPW estimators demonstrates a key advantage of M-estimators. By stacking estimating equations, the sandwich variance estimator correctly incorporates the uncertainty in estimation of the propensity scores into the parameter(s) of interest (e.g., average causal effect). Therefore, we do not have to rely on the nonparametric bootstrap (computationally cumbersome) or the GEE-trick (conservative estimate of the variance for the average causal effect).

G-computation

Second, we use g-computation, which instead models Y conditional on A and W. In general, g-computation for the average causal effect can be written as

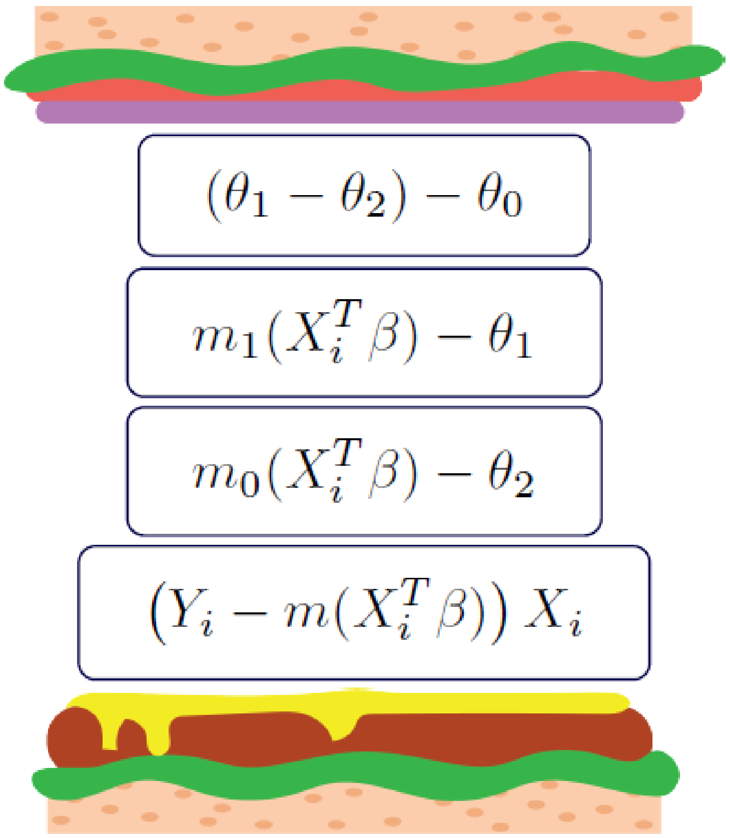
$$\frac{1}{n}\sum_{i=1}^{n} m_1(W_i; \hat{\beta}) - \frac{1}{n}\sum_{i=1}^{n} m_0(W_i; \hat{\beta})$$

where $m_a(W_i; \hat{\beta}) = E[Y_i|A_i = a, W_i; \hat{\beta}]$. In `delicatessen`, the built-in g-computation consists of either 2 estimating equations or 4 estimating equations, with both binary and continuous outcomes supported. The 2 stacked estimating equations are



where θ_1 is the mean under the action a, and β is the parameters for the regression model used to estimate the outcomes. Notice that the g-computation procedure supports generic deterministic plans (e.g., set $A = 1$ for all, set $A = 0$ for all, set $A = 1$ if $W = 1$ otherwise $A = 0$, etc.). These plans are more general than those allowed by either the built-in IPW or built-in AIPW estimating equations.

The 4 stacked estimating equations instead compare the mean difference between two action plans. The estimating equations are


$(\theta_1 - \theta_2) - \theta_0$
$m_1(X_i^T \beta) - \theta_1$
$m_0(X_i^T \beta) - \theta_2$
$(Y_i - m(X_i^T \beta))\, X_i$

where θ_0 is the average causal effect, θ_1 is the mean under the first plan, θ_2 is the mean under the second, and β is the parameters for the regression model used to predict the outcomes.

To load the pre-built g-computation estimating equations,

```
from delicatessen.estimating_equations import ee_gformula
```

The estimating equation is then wrapped inside the wrapper `psi` function. In the first example, we focus on estimating the average causal effect. Notice that for `ee_gformula` some additional data prep is necessary. Specifically, we need to create a copy of the data set where `A` is set to the value our plan dictates (e.g., `A=1`). Below is code that does this step and creates the wrapper function

```
# Creating data under the plans
d1 = d.copy()
d1['A'] = 1
d0 = d.copy()
d0['A'] = 0
```

(continues on next page)

(continued from previous page)

```
# Creating interaction terms
d['AxW'] = d['A'] * d['W']
d1['AxW'] = d1['A'] * d1['W']
d0['AxW'] = d0['A'] * d0['W']

def psi(theta):
    return ee_gformula(theta,                              # Parameters
                       y=d['Y'],                           # Outcome
                       X=d[['C', 'A', 'W', 'AxW']],        # Design matrix - observed
                       X=d1[['C', 'A', 'W', 'AxW']],       # Design matrix - plan 1
                       X=d0[['C', 'A', 'W', 'AxW']])       # Design matrix - plan 2
```

Note that we add an intercept to the outcome model by adding a column of 1's via `d['C']`.

Here, the initial values provided must be 3+*b* (where *b* is the number of columns in X). For binary outcomes, it will likely be best practice to have the initial values set as `[0., 0.5, 0.5, ...]`. followed by b `0.`'s. For continuous outcomes, all `0.` can be used instead. Furthermore, a regression model for the outcomes could be optimized outside of `delicatessen` and those (pre-washed) regression estimates can be passed as initial values to speed up optimization.

Now we can call the M-estimator to solve for the values and the variance.

```
estr = MEstimator(psi, init=[0., 0.5, 0.5, 0., 0., 0., 0.])
estr.estimate(solver='lm')
```

After successful optimization, we can inspect the estimated values.

```
estr.theta[0]    # causal mean difference of 1 versus 0
estr.theta[1]    # causal mean under X1
estr.theta[2]    # causal mean under X0
estr.theta[3:]   # regression coefficients
```

The variance and Wald-type confidence intervals can also be output via

```
estr.variance
estr.confidence_intervals()
```

Again, a key advantage of M-Estimation is demonstrated here. By stacking the estimating equations, the sandwich variance estimator correctly incorporates the uncertainty in estimation of the outcome model into the parameter(s) of interest (e.g., average causal effect). Therefore, we do not have to rely on the nonparametric bootstrap.

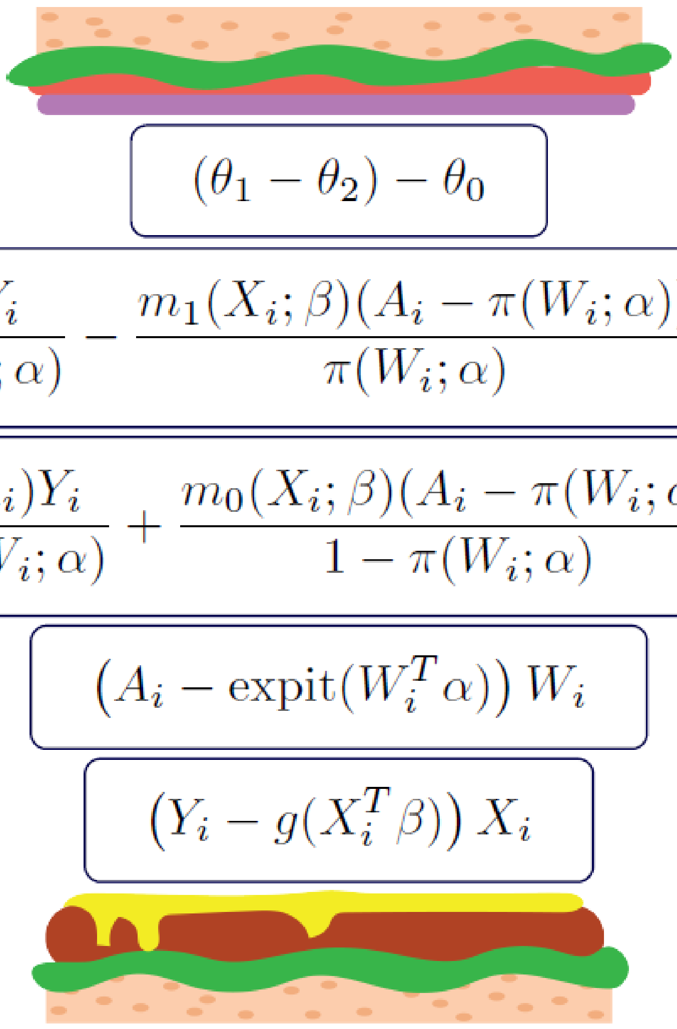
Augmented inverse probability weighting

Finally, we use the augmented inverse probability weighting (AIPW) esitmator, which incorporates both a model for Y conditional on A and W, and a model for A conditional on W. In other words, the AIPW estimator combines the g-computation and IPW estimators together in a clever way (which has some desireable statistical properties not reviewed here). The AIPW estimator for the average causal effect can be written as

$$\frac{1}{n} \sum_{i=1}^{n} \frac{A_i \times Y_i}{\pi_i} - \frac{m_1(W_i; \hat{\beta})(A_i - \pi_i}{\pi_i} - \frac{1}{n} \sum_{i=1}^{n} \frac{(1 - A_i) \times Y_i}{1 - \pi_i} + \frac{m_0(W_i; \hat{\beta})(A_i - \pi_i}{1 - \pi_i}$$

where $m_a(W_i; \hat{\beta}) = E[Y_i | A_i = a, W_i; \hat{\beta}]$, and $\pi_i = Pr(A_i = 1 | W_i; \hat{\alpha})$. In `delicatessen`, the built-in AIPW estimator consists of 5 estimating equations, with both binary and continuous outcomes supported. Similar to IPW (and unlike g-computation), the built-in AIPW estimator only supports the average causal effect as the parameter of interest.

The stacked estimating equations are



$$\boxed{(\theta_1 - \theta_2) - \theta_0}$$

$$\boxed{\frac{A_i Y_i}{\pi(W_i;\alpha)} - \frac{m_1(X_i;\beta)(A_i - \pi(W_i;\alpha))}{\pi(W_i;\alpha)} - \theta_1}$$

$$\boxed{\frac{(1-A_i) Y_i}{1-\pi(W_i;\alpha)} + \frac{m_0(X_i;\beta)(A_i - \pi(W_i;\alpha))}{1-\pi(W_i;\alpha)} - \theta_2}$$

$$\boxed{\left(A_i - \text{expit}(W_i^T \alpha)\right) W_i}$$

$$\boxed{\left(Y_i - g(X_i^T \beta)\right) X_i}$$

where θ_0 is the average causal effect, θ_1 is the mean under the first plan, θ_2 is the mean under the second, α is the parameters for the propensity score logistic model, and β is the parameters for the regression model used to predict the outcomes. For binary outcomes, the final estimating equation is replaced with the logistic model analog.

To load the pre-built AIPW estimating equations,

```python
from delicatessen.estimating_equations import ee_aipw
```

The estimating equation is then wrapped inside the wrapper `psi` function. Like `ee_gformula`, `ee_aipw` requires some additional data prep. Specifically, we need to create a copy of the data set where $A = 1$ for everyone and another copy where $A = 0$ for everyone. Below is code that does this step and creates the wrapper function

```python
# Creating data under the plans
d1 = d.copy()
```

(continues on next page)

(continued from previous page)

```
d1['A'] = 1
d0 = d.copy()
d0['A'] = 0

# Creating interaction terms
d['AxW'] = d['A'] * d['W']
d1['AxW'] = d1['A'] * d1['W']
d0['AxW'] = d0['A'] * d0['W']

def psi(theta):
    return ee_gformula(theta,                             # Parameters
                       y=d['Y'],                          # Outcome
                       A=d['A'],                          # Action
                       W=d[['C', 'W']],                   # Design matrix - PS
                       X=d[['C', 'A', 'W', 'AxW']],       # Design matrix - observed
                       X=d1[['C', 'A', 'W', 'AxW']],      # Design matrix - plan A=1
                       X=d0[['C', 'A', 'W', 'AxW']])      # Design matrix - plan A=0
```

Note that we add an intercept to the outcome model by adding a column of 1's via `d['C']`.

Here, the initial values provided must be 3 + b + c (where b is the number of columns in W and c is the number of columns in X). For binary outcomes, it will likely be best practice to have the initial values set as `[0., 0.5, 0.5, ...]`. followed by b `0.`'s. For continuous outcomes, all `0.` can be used instead. Furthermore, a regression models could be optimized outside of `delicatessen` and those (pre-washed) regression estimates can be passed as initial values to speed up optimization.

Now we can call the M-estimator to solve for the values and the variance.

```
estr = MEstimator(psi, init=[0., 0.5, 0.5,
                             0., 0.,
                             0., 0., 0., 0.])
estr.estimate(solver='lm')
```

After successful optimization, we can inspect the estimated values.

```
estr.theta[0]     # causal mean difference of 1 versus 0
estr.theta[1]     # causal mean under A=1
estr.theta[2]     # causal mean under A=0
estr.theta[3:5]   # propensity score regression coefficients
estr.theta[5:]    # outcome regression coefficients
```

The variance and Wald-type confidence intervals can also be output via

```
estr.variance
estr.confidence_intervals()
```

Additional Examples

For additional examples, we have replicated chapters 12-14 of the textbook “Causal Inference: What If” by Hernan and Robins (2023). These additional examples are provided here.

3.3.9 References and Further Readings

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Cole SR, & Hernán MA. (2008). Constructing inverse probability weights for marginal structural models. *American Journal of Epidemiology*, 168(6), 656-664.

Funk MJ, Westreich D, Wiesen C, Stürmer T, Brookhart MA, & Davidian M. (2011). Doubly robust estimation of causal effects. *American Journal of Epidemiology*, 173(7), 761-767.

Hernán MA, & Robins JM. (2006). Estimating causal effects from epidemiological data. *Journal of Epidemiology & Community Health*, 60(7), 578-586.

Huber PJ. (1992). Robust estimation of a location parameter. In Breakthroughs in statistics (pp. 492-518). Springer, New York, NY.

Inderjit, Streibig JC & Olofsdotter M. (2002). Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiol Plant* 114, 422–428.

Snowden JM, Rose S, & Mortimer KM. (2011). Implementation of G-computation on a simulated data set: demonstration of a causal inference technique. *American Journal of Epidemiology*, 173(7), 731-738.

3.4 Custom Estimating Equations

A key advantages of `delicatessen` is the flexibility in the estimating equations that can be specified. Here, I provide an overview and tips for how to build your own estimating equations using `delicatessen`.

In general, it will be best if you find an paper or book that directly provides the estimating equation(s) to you. Alternatively, if you can find the score function or gradient for a regression model, that is the corresponding estimating equation. This section does *not* address how to derive your own estimating equation(s). Rather, this section provides information on how to translate an estimating equation into code that is compatible with `delicatessen`, as `delicatessen` assumes you are giving it a valid estimating equation.

3.4.1 Building from scratch

First, we will go through the case of building an estimating equation completely from scratch. To do this, we will go through an example with linear regression.

First, we have the estimating equation (which is the score function) provided in Boos & Stefanski (2013)

$$\sum_{i}^{n} (Y_i - X_i^T \beta) X_i = 0$$

We will demonstrate using the following simulated data set

```
np.random.seed(8091421)
n = 200
d = pd.DataFrame()
```

(continues on next page)

(continued from previous page)

```
d['X'] = np.random.normal(size=n)
d['W'] = np.random.normal(size=n)
d['C'] = 1
d['Y'] = 5 + d['X'] + np.random.normal(size=n)
```

First, we can build the estimating equation using a for-loop where each `i` piece will be stacked together. While this for-loop approach will be 'slow', it is often a good strategy to implement a for-loop version that is easier to debug first.

Below calculates the estimating equation for each `i` in the for-loop. This function returns a stacked array of each `i` observation as a 3-by-n array. That array can then be passed to the `MEstimator`

```
def psi(theta):
    # Transforming to arrays
    X = np.asarray(d[['C', 'X', 'W']])   # Design matrix
    y = np.asarray(d['Y'])               # Dependent variable
    beta = np.asarray(theta)[:, None]    # Parameters
    n = X.shape[0]                       # Number of observations

    # Where to store each of the resulting estimating functions
    est_vals = []

    # Looping through each observation from 1 to n
    for i in range(n):
        v_i = (y[i] - np.dot(X[i], beta)) * X[i]
        est_vals.append(v_i)

    # returning 3-by-n NumPy array
    return np.asarray(est_vals).T
```

We can then apply this estimating equation via

```
mest = MEstimator(psi, init=[0., 0., 0.])
mest.estimate()
```

for which the coefficients match the coefficients from a ordinary least squares model (variance estimates may differ, since most OLS software use the inverse of the information matrix to estimate the variance, which is equivalent to the inverse of the bread matrix).

Here, we can vectorize the operations. The advantage of the vectorized-form is that it will run much faster. With some careful experimentation, the following is a vectorized version. Remember that `delicatessen` is expecting a 3-by-n array to be given by the `psi` function in this example. Failure to provide this is a common mistake when building custom estimating equations.

```
def psi(theta):
    X = np.asarray(d[['C', 'X', 'W']])    # Design matrix
    y = np.asarray(d['Y'])[:, None]       # Dependent variable
    beta = np.asarray(theta)[:, None]     # Parameters
    return ((y - np.dot(X, beta)) * X).T  # Computes all estimating functions
```

As before, we can run this chunk of code. Vectorizing (even parts of an estimating equation) can help to improve run-times if you find a M-estimator taking too long to solve.

3.4.2 Building with basics

Instead of building everything from scratch, you can also piece together built-in estimating equations with your custom estimating equations code. To demonstrate this, we will go through inverse probability weighting.

The inverse probability weighting estimator consists of four estimating equations: the difference between the weighted means, the weighted mean under $A = 1$, the weighted mean under $A = 0$, and the propensity score model. We can express this mathematically as

$$\sum_{i=1}^{n} \begin{bmatrix} (\theta_1 - \theta_2) - \theta_0 \\ \frac{A_i \times Y_i}{\pi_i} - \theta_1 \\ \frac{(1-A_i) \times Y_i}{1-\pi_i} - \theta_2 \\ (A_i - \text{expit}(W_i^T \alpha)) W_i \end{bmatrix} = 0$$

where A is the action of interest, Y is the outcome of interest, and W is the set of confounding variables.

Rather than re-code the logistic regression model (to estimate the propensity scores), we will use the built-in logistic regression functionality. Below is a stacked estimating equation for the inverse probability weighting estimator above

```python
def psi(theta):
    # Ensuring correct typing
    W = np.asarray(d['C', 'W'])      # Design matrix of confounders
    A = np.asarray(d['A'])           # Action
    y = np.asarray(y)                # Outocome
    beta = theta[3:]                 # Regression parameters

    # Estimating propensity score
    preds_reg = ee_regression(theta=beta,        # Built-in regression
                              X=W,               # Plug-in covariates for X
                              y=A,               # Plug-in treatment for Y
                              model='logistic')  # Specify logistic
    # Estimating weights
    pi = inverse_logit(np.dot(W, beta))          # Pr(A|W) using delicatessen.utilities

    # Calculating Y(a=1)
    ya1 = (A * y) / pi - theta[1]                # i's contribution is (AY) / \pi

    # Calculating Y(a=0)
    ya0 = ((1-A) * y) / (1-pi) - theta[2]        # i's contribution is ((1-A)Y) / (1-\pi)

    # Calculating Y(a=1) - Y(a=0) (using np.ones to ensure a 1-by-n array)
    ate = np.ones(y.shape[0]) * (theta[1] - theta[2]) - theta[0]

    # Output (3+b)-by-n stacked array
    return np.vstack((ate,             # theta[0] is for the ATE
                      ya1[None, :],    # theta[1] is for R1
                      ya0[None, :],    # theta[2] is for R0
                      preds_reg))      # theta[3:] is for the regression coefficients
```

This example demonstrates how estimating equations can easily be stacked together using `delicatessen`. Specifically, both built-in and user-specified functions can be specified together seamlessly. All it requires is specifying both in the estimating equation and returning a stacked array of the estimates.

One important piece to note here is that the returned array needs to be in the *same* order as the theta's are input. As done here, all the `theta` values are the 3rd are for the propensity score model. Therefore, the propensity score model values are last in the returned stack. Returning the values in a different order than input is a common mistake.

3.4.3 Handling np.nan

Sometimes, np.nan will be necessary to include in your data set. However, delicatessen does not naturally handle np.nan. In fact, delicatessen will return an error when there are np.nan's detected (this is by design). The following discusses how np.nan can be handled appropriately in the estimating equations.

In the first case, we will consider handling np.nan with a built-in estimating equation. When trying to fit a regression model where there are np.nan's present, the missing values will be set to a placeholder value and their contributions will be manually removed using an indicator function for missingness. Below is an example using the built-in logistic regression estimating equations:

```python
import numpy as np
import pandas as pd
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_logistic_regression

d = pd.DataFrame()
d['X'] = np.random.normal(size=100)
y = np.random.binomial(n=1, p=0.5 + 0.01 * d['X'], size=100)
d['y'] = np.where(np.random.binomial(n=1, p=0.9, size=100), y, np.nan)
d['C'] = 1

X = np.asarray(d[['C', 'X']])
y = np.asarray(d['y'])
y_no_nan = np.asarray(d['y'].fillna(0.5))
r = np.where(d['Y'].isna(), 0, 1)


def psi(theta):
    # Estimating logistic model values with filled-in Y's
    a_model = ee_logistic_regression(theta,
                                     X=X, y=y_no_nan)
    # Setting contributions with missing to zero manually
    a_model = a_model * r
    return a_model


mest = MEstimator(psi, init=[0, 0, ])
mest.estimate()
```

If the contribution to the estimating function with missing Y's had not been included, the optimized points would have been nan. Alternatively, had numpy.nan_to_num been used with deriv_method='exact', this would have also resulted in an error (numpy.nan_to_num is okay to use with deriv_method='approx').

As a second example, we will consider estimating the mean with missing data and correcting for informative missing by inverse probability weighting. To reduce random error, this example uses 10,000 observations. Here, we must set nan's to be zero's prior to subtracting off the mean. This is shown below:

```python
import numpy as np
import pandas as pd
from scipy.stats import logistic
from delicatessen import MEstimator
from delicatessen.estimating_equations import ee_logistic_regression
from delicatessen.utilities import inverse_logit

```

(continues on next page)

(continued from previous page)

```python
# Generating data
d = pd.DataFrame()
d['X'] = np.random.normal(size=100000)
y = 5 + d['X'] + np.random.normal(size=100000)
d['y'] = np.where(np.random.binomial(n=1, p=logistic.cdf(1 + d['X']), size=100000), y,␣
↪np.nan)
d['C'] = 1

X = np.asarray(d[['C', 'X']])
y = np.asarray(d['y'])
r = np.asarray(np.where(d['y'].isna(), 0, 1))


def psi(theta):
    # Estimating logistic model values
    a_model = ee_logistic_regression(theta[1:],
                                     X=X, y=r)
    pi = inverse_logit(np.dot(X, theta[1:]))

    y_w = np.where(r, y / pi, 0) - theta[0]  # nan-to-zero then subtract off
    return np.vstack((y_w[None, :],
                      a_model))

mest = MEstimator(psi, init=[0, 0, 0])
mest.estimate()
```

This will result in an estimate close to the truth (5). If we were to instead use `np.where(r, y/pi - theta[0], 0)`, then the wrong answer will be returned. When in doubt about the form to use (and where the subtraction should go), go back to the formula for the estimating function or estimator. Here, the IPW mean is

$$\sum_{i=1}^{n} \left(\frac{I(R_i = 1) Y_i}{\Pr(R_i = 1 | X_i)} - \theta \right) = 0$$

As seen with the indicator function, observations where Y is missing should contribute a zero *minus* θ. If we had instead used, the Hajek estimator

$$\sum_{i=1}^{n} \left((Y_i - \theta) \frac{I(R_i = 1)}{\Pr(R_i = 1 | X_i)} \right) = 0$$

The subtraction would have been on the inside of the `np.where` step.

3.4.4 Common Mistakes

Here is a list of common mistakes, most of which I have done myself.

1. The `psi` function doesn't return a NumPy array.
2. The `psi` function returns the wrong shape. Remember, it should be a b-by-n NumPy array!
3. The `psi` function is summing over n. `delicatessen` needs to do the sum internally (in order to compute the bread and filling), so do not sum over n in `psi`!
4. The `theta` values and b *must* be in the same order. If `theta[0]` is the mean, the 1st row of the returned array better be the estimating function for that mean!

5. Automatic differentiation with `np.nan_to_num`. This will result in the bread matrix having *nan* values.
6. Trying to use a SciPy function with automatic differentation (only some functionalities are supported. please open an issue on GitHub if you have one you would like to see added).

If you still have trouble, please open an issue at pzivich/Delicatessen. This will help me to add other common mistakes here and improve the documentation for custom estimating equations.

3.4.5 Additional Examples

Additional examples are provided here .

3.5 Optimization Advice

This section is meant to provide some guidance if you have trouble with root-finding for a given set of the estimating functions.

A weakness of `delicatessen` is that it does not have the fastest or most robust routines for estimating statistical parameters (in general maximizing the likelihood is easier computationally than root-finding of the score functions). This is the cost of the flexibility of the general M-Estimator (a cost that I believe to be worthwhile). When θ only consists of a few parameters, the root-finding will generally be robust. However, cases where θ consists of many parameters is likely to occur.

Below are a few recommendations on getting `MEstimator` to find the roots of the estimating equations.

3.5.1 Center initial values

Optimization will be improved when the starting values (those provided in `init`) are ‘close’ to the θ values. However, we don’t know what those values may be. Since the root-finding needs to look for the best values, it is best to start it in the ‘middle’ of the possible space.

As an example, consider the mean estimating equation for a binary variable. The starting value could be specified as 0, 1, or any other number. However, we can be nice to the root-finding by providing 0.5 as the starting value, since 0.5 is the middle of the possible space for a proportion.

For regression, starting values of 0 are likely to be preferred (without additional information about a problem).

If initial values are placed outside of the bounds of a particular θ, this can also break the optimization procedure. Returning to the proportion, providing a starting value of -10 may cause the root-finder trouble, since proportions are actually bound to [0,1]. So make sure your initial values are (1) reasonable, and (2) within the bounds of the measure θ.

3.5.2 Pre-wash initials

In the case of stacked estimating equations composed of multiple estimating functions (e.g., g-computation, IPW, AIPW), some parameters can be estimated indepedent of the others. Then the pre-optimized values can be passed as the initial values for the overall estimator. This ‘pre-washing’ of values allows the `delicatessen` root-finding to focus on values of θ that can’t be optimized outside.

This pre-washing approach is particularly useful for regression models, since more stable optimization strategies exist for most regression implementations. Pre-washing the initial values allows `delicatessen` to ‘borrow’ the strength of more stable methods.

Finally, `delicatessen` offers the option to run the root-finding procedure for a subset of the estimating functions. Therefore, some parameters can be solved outside of the procedure and only the remaining subset can be searched for. This option is particularly valuable when an estimator consists of hundreds of parameters.

3.5.3 Increase iterations

If neither of those works, increasing the number of iterations is a good next place. By default, `MEstimator` goes to 1000 iterations (far beyond SciPy's default value).

3.5.4 Different root-finding

By default, `delicatessen` uses the secant method available in `scipy.optimize.newton`. However, `delicatessen` also supports other algorithms in `scipy.optimize.root`, such as Levenberg-Marquette and Powell's Hybrid. Additionally, user-specified root-finding algorithms can also be used.

3.5.5 Non-smooth equations

As mentioned in the examples, non-smooth estimating equations (e.g., percentiles, positive mean deviation, etc.) can be difficult to optimize. In general, it is best to avoid using `delicatessen` with non-smooth estimating equations.

If one must use `delicatessen` with non-smooth estimating equations, some tricks we have found helpful are to: use `solver='hybr'` and increasing the tolerance (to the same order as n) help.

3.5.6 A warning

Before ending this section, I want to emphasize that reducing the `tolerance` is not really advised. While it may allow the optimization routine to succeed, it only allows the 'error' of the optimization to be greater. Therefore, the optimization will stop 'further' away from zero. Do **not** use this approach to getting `delicatessen` to succeed in the optimization unless you are absolutely sure that the new `tolerance` is within acceptable error tolerance for your problem. The default is `1e-9`.

3.6 Reference

Documentation for all available functions and arguments for those functions are provided here. *M-Estimator* contains documentation for the general M-Estimator procedure. *Estimating Equations* details the built-in estimating equations that come with `delicatessen`.

For a more narrative-driven description of M-Estimation and how to use `delicatessen`, please see the sections provided in the side-bar.

3.6.1 M-Estimator

Reference documentation for the M-Estimator available in `delicatessen`. This is the main utility in the `delicatessen` library. For implementation of your own estimating equations with `delicatessen`, see the documentation and examples provided in the 'Custom Equations' section.

M-Estimator

MEstimator(stacked_equations[, init, subset])	M-Estimator for stacked estimating equations.

delicatessen.mestimation.MEstimator

class MEstimator(*stacked_equations*, *init=None*, *subset=None*)

M-Estimator for stacked estimating equations.

Estimating equations are a general approach to point and variance estimation that consists of defining an estimator as the solution to a vector of equations that are equal to zero. The corresponding estimators, often called M-estimators or Z-estimators, satisify the following equation

$$\sum_{i=1}^{n} \psi(O_i, \hat{\theta}) = 0$$

where ψ is the v-dimensional vector of estimating equation(s), $\hat{\theta}$ is the v-dimensional parameter vector, and O_i is the observed data (where units are independent but not necessarily identically distributed).

Note: Estimating equations are advantageous in both theoretical and applied research. They simplifies proofs of consistency and asymptotic normality of estimators under a large-sample approximation framework. In application, this approach simplifies variance estimation and automates the delta-method.

M-Estimators consists of two broad step: point estimation and variance estimation. Point estimation is carried out by determining the values of θ where the sum of the estimating equations are zero. This is done via standard root-finding algorithms.

For variance estimation, sandwich variance estimator is used. The asymptotic sandwich variance estimator consists of

$$V_n(O, \hat{\theta}) = B_n(O, \hat{\theta})^{-1} F_n(O, \hat{\theta}) \left\{ B_n(O, \hat{\theta}^{-1}) \right\}^T$$

where B is the 'bread' and F is the 'filling' matrix. These matrices are defined as

$$B_n(O, \hat{\theta}) = n^{-1} \sum_{i=1}^{n} -\frac{\partial}{\partial \theta} \psi(O_i, \hat{\theta})$$

$$F_n(O, \hat{\theta}) = n^{-1} \sum_{i=1}^{n} \psi(O_i, \hat{\theta}) \psi(O_i, \hat{\theta})^T$$

respectively. The partial derivatives for the bread are calculated using either numerical approximation (e.g., forward difference method) or forward-mode automatic differentiation. Inverting the bread is done via NumPy's `linalg.pinv`. For the filling, the dot product is taken at $\hat{\theta}$.

Note: The difficult part (that must be done by the user) is to specify the estimating equations. Be sure to check the provided examples for the expected format. Pre-built estimating equations for common problems are also made available.

After completion of these steps, point and variance estimates are stored. These can be extracted from `MEstimator`. Further, confidence intervals, Z-scores, P-values, or S-values can all be generated.

Note: For complex regression problems, the root-finding algorithms are not as robust relative to maximization approaches. A simple solution for difficult problems is to 'pre-wash' or find the solution to the equations and provide those as the initial starting values.

Parameters

- **stacked_equations** (*function, callable*) – Function that returns a *v*-by-*n* NumPy array of the estimating equations. See provided examples in the documentation for how to construct a set of estimating equations.
- **init** (*list, set, array*) – Initial values for the root-finding algorithm. A total of *v* values should be provided.
- **subset** (*list, set, array, None, optional*) – Optional argument to conduct the root-finding procedure on a subset of parameters in the estimating equations. The input list is used to location index the parameter array via `np.take()`. The subset list will only affect the root-finding procedure (i.e., the sandwich variance estimator ignores the subset argument). Default is `None`, which runs the root-finding procedure for all parameters in the estimating equations.

Note: Because the root-finding procedure is NOT ran for parameters outside of the subset, those coefficients *must* be solved outside of `MEstimator`. In general, I do *NOT* recommend using the `subset` argument unless a series of complex estimating equations need to be solved. In general, this argument does not massively improve speed until the estimating equations consist of hundreds of parameters.

Examples

Loading necessary functions and building a generic data set for estimation of the mean

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_mean_variance
```

```
>>> y_dat = [1, 2, 4, 1, 2, 3, 1, 5, 2]
```

M-estimation with built-in estimating equation for the mean and variance. First, `psi`, or the stacked estimating equations, is defined

```
>>> def psi(theta):
>>>     return ee_mean_variance(theta=theta, y=y_dat)
```

Calling the M-estimation procedure

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, 0, ])
>>> estr.estimate()
```

Inspecting the output results

```
>>> estr.theta                                 # Point estimates
>>> estr.variance                              # Covariance
>>> estr.asymptotic_variance                   # Asymptotic covariance
>>> np.sqrt(np.diag(estr.asymptotic_variance)) # Standard deviation
>>> estr.variance                              # Covariance
>>> np.sqrt(np.diag(estr.variance))            # Standard error
>>> estr.confidence_intervals()                # Confidence intervals
>>> estr.z_scores()                            # Z-scores
>>> estr.p_values()                            # P-values
>>> estr.s_values()                            # S-values
```

Alternatively, a custom estimating equation can be specified. This is done by constructing a valid estimating equation for the `MEstimator`. The `MEstimator` expects the `psi` function to return a v-by-n array, where v is the number of parameters (length of `theta`) and n is the total number of observations. Below is an example of the mean and variance estimating equation from before, but implemented by-hand

```
>>> def psi(theta):
>>>     y = np.array(y_dat)
>>>     mean = y - theta[0]
>>>     variance = (y - theta[0]) ** 2 - theta[1]
>>>     return mean, variance
```

The M-estimation procedure is called using the same approach as before

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, 0, ])
>>> estr.estimate()
```

Note that `len(init)` should be equal to v. So in this case, two initial values are provided.

`MEstimator` can also be run with a user-provided root-finding algorithm. To specify a custom root-finder, a function must be created by the user that consists of two keyword arguments (`stacked_equations`, `init`) and must return only the optimized values. The following is an example with SciPy's Levenberg-Marquardt algorithm implemented in `root`.

```
>>> def custom_solver(stacked_equations, init):
>>>     options = {"maxiter": 1000}
>>>     opt = root(stacked_equations,
>>>                x0=np.asarray(init),
>>>                method='lm', tol=1e-9, options=options)
>>>     return opt.x
```

The provided custom root-finder can then be implemented like the following (continuing with the estimating equation from the previous example):

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, 0, ])
>>> estr.estimate(solver=custom_solver)
```

For more examples on how to apply `MEstimator`, see https://deli.readthedocs.io/en/latest/

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Ross RK, Zivich PN, Stringer JSA, & Cole SR. (2024). M-estimation for common epidemiological measures: introduction and applied examples. *International Journal of Epidemiology*, 53(2), dyae030.

Stefanski LA, & Boos DD. (2002). The calculus of M-estimation. *The American Statistician*, 56(1), 29-38.

`__init__`(*stacked_equations*, *init=None*, *subset=None*)

Methods

`__init__`(stacked_equations[, init, subset])	
`confidence_intervals`([alpha])	Calculate two-sided Wald-type $(1-\alpha)\times 100\%$ confidence intervals using the point and sandwich variance estimates.
`estimate`([solver, maxiter, tolerance, ...])	Run the point and variance estimation procedures for given estimating equation and starting values.
`p_values`([null])	Calculate two-sided Wald-type P-values using the Z-scores compute using the point and the sandwich variance estimates.
`s_values`([null])	Calculate two-sided Wald-type S-values using the point estimates and the sandwich variance.
`z_scores`([null])	Calculate the Z-score using the point estimates and the sandwich variance.

`estimate`(*solver='lm'*, *maxiter=5000*, *tolerance=1e-09*, *deriv_method='approx'*, *dx=1e-09*, *allow_pinv=True*)

Run the point and variance estimation procedures for given estimating equation and starting values. This function carries out the point and variance estimation of `theta`. After this procedure, the point estimates (in `theta`) and the covariance matrix (in `variance`) can be extracted from the `MEstimator` object.

Parameters

- **`solver`** (*`str, function, callable, optional`*) – Method to use for the root-finding procedure. Default is the Levenberg-Marquardt algorithm (`scipy.optimize.root(method='lm')`, specified by `solver='lm'`). Other built-in option is the secant method (`scipy.optimize.newton`, specified by `solver='newton'`), and a modification of the Powell hybrid method (`scipy.optimize.root(method='hybr')`, specified by `solver='hybr'`). Finally, any generic root-finding algorithm can be used via a user-provided callable object. The function must consist of two keyword arguments: `stacked_equations`, and `init`. Additionally, the function should return only the optimized values. Please review the provided example in the documentation for how to implement a custom root-finding algorithm.
- **`maxiter`** (*`int, optional`*) – Maximum iterations to consider for the root finding procedure. Default is 5000 iterations. For complex estimating equations, this value may need to be increased. This argument is not used when a custom root-finding method (e.g., `solver`) is provided.
- **`tolerance`** (*`float, optional`*) – Maximum tolerance for errors in the root finding in `scipy.optimize`. Default is 1e-9. This argument is not used when a custom root-finding

method (e.g., `solver`) is provided.

- **deriv_method** (*str, optional*) – Method to compute the derivative of the estimating equations for the bread matrix. Default is `'approx'`. Options include numerical approximation via the forward difference method via SciPy (`'approx'`), forward difference as implemented in delicatessen (*'fapprox'*), backward difference as implemented in delicatessen (*'bapprox'*), central difference implemented as in delicatessen (*'capprox'*), or forward-mode automatic differentiation as implemented in delicatessen(`'exact'`).
- **dx** (*float, optional*) – Spacing to use to numerically approximate the partial derivatives of the bread matrix. Default is 1e-9. Here, a small value for `dx` should be used, since some large values can result in poor approximations. This argument is only used with numerical approximation methods.
- **allow_pinv** (*bool, optional*) – Whether to allow for the pseudo-inverse (via `numpy.linalg.pinv`) if the bread matrix is determined to be non-invertible. If you want to disallow the pseudo-inverse (i.e., use `numpy.linalg.inv`), set this argument to `False`. Default is `True`, which is more robust to the possible bread matrices.

Returns

Return type None

`confidence_intervals`(*alpha=0.05*)

Calculate two-sided Wald-type $(1-\alpha)\times$ 100% confidence intervals using the point and sandwich variance estimates. The formula for the confidence intervals is

$$\hat{\theta} \pm z_{\alpha/2} \times \widehat{SE}(\hat{\theta})$$

Note: The `.estimate()` function must be called before the confidence intervals can be calculated.

Parameters **alpha** (*float, optional*) – The $0 < \alpha < 1$ level for the corresponding confidence intervals. Default is 0.05, which corresponds to 95% confidence intervals.

Returns b-by-2 array, where row 1 is the confidence intervals for $\theta_1, \ldots,$ and row b is the confidence intervals for θ_b

Return type array

`z_scores`(*null=0*)

Calculate the Z-score using the point estimates and the sandwich variance. The formula for the Z score is

$$\frac{\hat{\theta} - \theta}{\widehat{SE}(\hat{\theta})}$$

where θ is the null. The `.estimate()` function must be called before the Z-scores can be calculated. Note that the default value for the null is zero.

Parameters **null** (*int, float, ndarray, optional*) – Null or reference for the the corresponding P-values. Default is 0.

Returns Array of Z-scores for $\theta_1, ..., \theta_b$, respectively

Return type array

`p_values`(*null=0*)

Calculate two-sided Wald-type P-values using the Z-scores compute using the point and the sandwich variance estimates. Once the Z-scores are computed, the corresponding P-values are obtained by comparing to the standard normal distribution.

The `.estimate()` function must be called before the P-values can be calculated.

Parameters **null** (*int, float, ndarray, optional*) – Null or reference for the the corresponding P-values. Default is 0.

Returns Array of P-values for $\theta_1, ..., \theta_b$, respectively

Return type array

s_values(*null=0*)

Calculate two-sided Wald-type S-values using the point estimates and the sandwich variance. The S-value, or Shannon Information value, is a transformation of the P-values that has been argued to be more easily interpretable as it can be related back to simple coin-flipping scenarios. The transformation from a P-value into a S-value is.

$$S = -\log_2(P)$$

where P is the corresponding P-value. The `.estimate()` function must be called before the S-values can be calculated.

The S-value can be contextualized in terms of coin-flips. Suppose the S-value is s. Then s corresponds to the number of heads in a row with a fair coin (equal chances heads or tails). As s increases, one would be more 'surprised' by the result (e.g., it might not be surprising to have 2 heads in a row, but it would be surprising for 20 in a row).

Parameters **null** (*int, float, ndarray, optional*) – Null or reference for the the corresponding S-values. Default is 0.

Returns Array of S-values for $\theta_1, ..., \theta_b$, respectively

Return type array

References

Cole SR, Edwards JK, & Greenland S. (2021). Surprise! *American Journal of Epidemiology*, 190(2), 191-193.

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

Sandwich Variance Estimator

compute_sandwich(stacked_equations, theta[, ...])	Compute the empirical sandwich variance estimator given a set of estimating equations and parameter estimates.

delicatessen.sandwich.compute_sandwich

`compute_sandwich`(*stacked_equations*, *theta*, *deriv_method='approx'*, *dx=1e-09*, *allow_pinv=True*)

Compute the empirical sandwich variance estimator given a set of estimating equations and parameter estimates. Note that this functionality does not solve for the parameter estimates (unlike `MEstimator`). Instead, it only computes the sandwich for the provided value.

The empirical sandwich variance estimator is defined as

$$V_n(O_i; \theta) = B_n(O_i; \theta)^{-1} F_n(O_i; \theta) \left[B_n(O_i; \theta)^{-1} \right]^T$$

where $\psi(O_i; \theta)$ is the estimating function,

$$B_n(O_i; \theta) = \sum_{i=1}^{n} \frac{\partial}{\partial \theta} \psi(O_i; \theta),$$

and

$$F_n(O_i; \theta) = \sum_{i=1}^{n} \psi(O_i; \theta) \psi(O_i; \theta)^T.$$

To compute the bread matrix, B_n, the matrix of partial derivatives is computed by using either finite difference methods or automatic differentiation. For finite differences, the default is to use SciPy's `approx_fprime` functionality, which uses forward finite differences. However, you can also use the delicatessen homebrew version that allows for forward, backward, and center differences. Automatic differentiation is also supported by a homebrew version.

To compute the meat matrix, F_n, only linear algebra methods, implemented through NumPy, are necessary. The sandwich is then constructed from these pieces using linear algebra methods from NumPy.

Parameters

- **stacked_equations** (*function, callable*) – Function that returns a *v*-by-*n* NumPy array of the estimating equations. See provided examples in the documentation for how to construct a set of estimating equations.
- **theta** (*list, set, array*) – Parameter estimates to compute the empirical sandwich variance estimator at. Note that this function assumes that you have solved for the `theta` that correspond to the root of the input estimating equations.
- **deriv_method** (*str, optional*) – Method to compute the derivative of the estimating equations for the bread matrix. Options include numerical approximation via the forward difference method via SciPy (`'approx'`), forward difference implemented by-hand (*'fapprox'*), backward difference implemented by-hand (*'bapprox'*), central difference implemented by-hand (*'capprox'*), or forward-mode automatic differentiation (`'exact'`). Default is `'approx'`.
- **dx** (*float, optional*) – Spacing to use to numerically approximate the partial derivatives of the bread matrix. Here, a small value for `dx` should be used, since some large values can result in poor approximations. This argument is only used with numerical approximation methods. Default is `1e-9`.
- **allow_pinv** (*bool, optional*) – Whether to allow for the pseudo-inverse (via `numpy.linalg.pinv`) if the bread matrix is determined to be non-invertible. If you want to disallow the pseudo-inverse (i.e., use `numpy.linalg.inv`), set this argument to `False`. Default is `True`, which is more robust to the possible bread matrices.

Returns Returns a *p*-by-*p* NumPy array for the input `theta`, where `p = len(theta)`

Return type array

Examples

Loading necessary functions and building a generic data set for estimation of the mean

```
>>> import numpy as np
>>> from delicatessen import MEstimator
>>> from delicatessen import compute_sandwich
>>> from delicatessen.estimating_equations import ee_mean_variance
```

```
>>> y_dat = [1, 2, 4, 1, 2, 3, 1, 5, 2]
```

The following is an illustration of how to compute sandwich covariance using only an estimating equation and the parameter values. The mean and variance (that correspond to `ee_mean_variance`) can be computed using NumPy by

```
>>> mean = np.mean(y_dat)
>>> var = np.var(y_dat, ddof=0)
```

For the corresponding estimating equation, we can use the built-in functionality as done below

```
>>> def psi(theta):
>>>     return ee_mean_variance(theta=theta, y=y_dat)
```

Calling the sandwich computation procedure

```
>>> sandwich_asymp = compute_sandwich(stacked_equations=psi, theta=[mean, var])
```

The output sandwich is the *asymptotic* variance (or the variance that corresponds to the standard deviation). To get the variance (or the variance that corresponds to the standard error), we rescale `sandwich` by the number of observations

```
>>> sandwich = sandwich_asymp / len(y_dat)
```

The standard errors are then

```
>>> se = np.sqrt(np.diag(sandwich))
```

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Ross RK, Zivich PN, Stringer JSA, & Cole SR. (2024). M-estimation for common epidemiological measures: introduction and applied examples. *International Journal of Epidemiology*, 53(2), dyae030.

Stefanski LA, & Boos DD. (2002). The calculus of M-estimation. The American Statistician, 56(1), 29-38.

3.6.2 Estimating Equations

To ease use, delicatessen comes with a variety of built-in estimating equations, covering a variety of common use cases. Below is reference documentation for currently supported estimating equations.

Basic

ee_mean(theta, y)	Estimating equation for the mean.
ee_mean_robust(theta, y, k[, loss, lower, upper])	Estimating equation for the (unscaled) robust mean.
ee_mean_variance(theta, y)	Estimating equations for the mean and variance. The estimating equations for the mean and
ee_percentile(theta, y, q)	Estimating equation for the q percentile.
ee_positive_mean_deviation(theta, y)	Estimating equations for the positive mean deviation.

delicatessen.estimating_equations.basic.ee_mean

ee_mean(*theta*, *y*)

Estimating equation for the mean. The estimating equation for the mean is

$$\sum_{i=1}^{n} (Y_i - \theta) = 0$$

Parameters

- **theta** (*ndarray, list, vector*) – Theta in the case of the mean consists of a single value. Therefore, an initial value like the form of `[0, ]` should be provided.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.

Returns Returns a 1-by-*n* NumPy array evaluated for the input `theta` and `y`

Return type array

Examples

Construction of a estimating equation(s) with `ee_mean` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_mean
```

Some generic data to estimate the mean for

```
>>> y_dat = [1, 2, 4, 1, 2, 3, 1, 5, 2]
```

Defining psi, or the estimating equation

```
>>> def psi(theta):
>>>     return ee_mean(theta=theta, y=y_dat)
```

Calling the M-estimator

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, ])
>>> estr.estimate()
```

Inspecting the parameter estimates, the variance, and the asymptotic variance

```
>>> estr.theta
>>> estr.variance
>>> estr.asymptotic_variance
```

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

delicatessen.estimating_equations.basic.ee_mean_robust

ee_mean_robust(*theta*, *y*, *k*, *loss='huber'*, *lower=None*, *upper=None*)

Estimating equation for the (unscaled) robust mean. The estimating equation for the robust mean is

$$\sum_{i=1}^{n} f_k(Y_i - \theta) = 0$$

where $f_k(x)$ is the corresponding robust loss function. Options for the loss function include: Huber, Tukey's biweight, Andrew's Sine, and Hampel. See `robust_loss_function` for further details on the loss functions for the robust mean.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in the case of the robust mean consists of a single value. Therefore, an initial value like the form of `[0, ]` is should be provided.
- **y** (*ndarray, vector, list*) – 1-dimensional vector of n observed values.
- **k** (*int, float*) – Tuning or hyperparameter for the chosen loss function. Notice that the choice of hyperparameter depends on the loss function.
- **loss** (*str, optional*) – Robust loss function to use. Default is `'huber'`. Options include `'andrew'`, `'hampel'`, `'tukey'`.
- **lower** (*int, float, None, optional*) – Lower parameter for the Hampel loss function. This parameter does not impact the other loss functions. Default is `None`.
- **upper** (*int, float, None, optional*) – Upper parameter for the Hampel loss function. This parameter does not impact the other loss functions. Default is `None`.

Returns Returns a 1-by-n NumPy array evaluated for the input `theta` and `y`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_mean_robust` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_mean_robust
```

Some generic data to estimate the mean for

```
>>> y_dat = [-10, 1, 2, 4, 1, 2, 3, 1, 5, 2, 33]
```

Defining psi, or the stacked estimating equations for Huber's robust mean

```
>>> def psi(theta):
>>>     return ee_mean_robust(theta=theta, y=y_dat, k=9, loss='huber')
```

Calling the M-estimation procedure

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, ])
>>> estr.estimate()
```

Inspecting the parameter estimates, the variance, and the asymptotic variance

```
>>> estr.theta
>>> estr.variance
>>> estr.asymptotic_variance
```

References

Andrews DF. (1974). A robust method for multiple linear regression. *Technometrics*, 16(4), 523-531.

Beaton AE & Tukey JW (1974). The fitting of power series, meaning polynomials, illustrated on band-spectroscopic data. *Technometrics*, 16(2), 147-185.

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Hampel FR. (1971). A general qualitative definition of robustness. *The Annals of Mathematical Statistics*, 42(6), 1887-1896.

Huber PJ. (1964). Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1), 73–101.

Huber PJ, Ronchetti EM. (2009) Robust Statistics 2nd Edition. Wiley. pgs 98-100

delicatessen.estimating_equations.basic.ee_mean_variance

`ee_mean_variance`(*theta*, *y*)

Estimating equations for the mean and variance. The estimating equations for the mean and variance are

$$\sum_{i=1}^{n} \begin{bmatrix} Y_i - \theta_1 = 0 \\ (Y_i - \theta_1)^2 - \theta_2 \end{bmatrix} = 0$$

Unlike `ee_mean`, `theta` consists of 2 parameters. The output covariance matrix will also provide estimates for each of the `theta` values.

Parameters

- **`theta`** (*`ndarray, list, vector`*) – Theta in this case consists of two values. Therefore, initial values like the form of `[0, 0]` should be provided.

- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values. No missing data should be included (missing data may cause unexpected behavior when attempting to calculate the mean).

Returns Returns a 2-by-*n* NumPy array evaluated for the input `theta` and `y`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_mean_variance` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_mean_variance
```

Some generic data to estimate the mean for

```
>>> y_dat = [1, 2, 4, 1, 2, 3, 1, 5, 2]
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_mean_variance(theta=theta, y=y_dat)
```

Calling the M-estimator (note that *init* has 2 values)

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, 0, ])
>>> estr.estimate()
```

Inspecting the parameter estimates, the variance, and the asymptotic variance

```
>>> estr.theta
>>> estr.variance
>>> estr.asymptotic_variance
```

For this estimating equation, `estr.theta[1]` and `estr.asymptotic_variance[0][0]` are expected to be equal.

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

delicatessen.estimating_equations.basic.ee_percentile

`ee_percentile`(*theta*, *y*, *q*)

Estimating equation for the q percentile. The estimating equation is

$$\sum_{i=1}^{n} \{q - I(Y_i \leq \theta)\} = 0$$

where $0 < q < 1$ is the percentile. Notice that this estimating equation is non-smooth. Therefore, root-finding is difficult.

Note: As the derivative of the estimating equation is not defined at $\hat{\theta}$, the bread (and sandwich) cannot be used to estimate the variance. This estimating equation is offered for completeness, but is not generally recommended for applications.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of one value. Therefore, initial values like the form of `[0, ]` should be provided.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values. No missing data should be included (missing data may cause unexpected behavior when attempting to calculate the mean).
- **q** (*float*) – Percentile to calculate. Must be $(0, 1)$

Returns Returns a 1-by-n NumPy array evaluated for the input `theta` and `y`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_percentile` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_percentile
```

Some generic data to estimate the mean for

```
>>> np.random.seed(89041)
>>> y_dat = np.random.normal(size=100)
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_percentile(theta=theta, y=y_dat, q=0.5)
```

Calling the M-estimation procedure (note that `init` has 2 values now).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, ])
>>> estr.estimate(solver='hybr', tolerance=1e-3, dx=1, order=15)
```

Notice that we use a different solver, tolerance values, and parameters for numerically approximating the derivative here. These changes generally work better for the percentile since the estimating equation is non-smooth. Furthermore, optimization is hard when only a few observations (<100) are available.

```
>>> estr.theta
```

Then displays the estimated percentile / median. In this example, there is a difference between the closed form solution (`-0.07978`) and M-Estimation (`-0.06022`).

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

delicatessen.estimating_equations.basic.ee_positive_mean_deviation

ee_positive_mean_deviation(*theta*, *y*)

Estimating equations for the positive mean deviation. The estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} 2(Y_i - \theta_2) I(Y_i > \theta_2) - \theta_1 \\ 0.5 - I(Y_i \le \theta_2) \end{bmatrix} = 0$$

where the first estimating equation is for the positive mean difference, and the second estimating equation is for the median. Notice that this estimating equation is non-smooth. Therefore, root-finding is difficult.

Note: As the derivative of the estimating equation for the median is not defined at $\hat{\theta}$, the bread (and sandwich) cannot be used to estimate the variance. This estimating equation is offered for completeness, but is not generally recommended for applications.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of two values. Therefore, initial values like the form of `[0, 0]` are recommended.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values. No missing data should be included (missing data may cause unexpected behavior when attempting to calculate the positive mean deviation).

Returns Returns a 2-by-*n* NumPy array evaluated for the input `theta` and `y`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_positive_mean_deviation` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_positive_mean_deviation
```

Some generic data to estimate the mean for

```
>>> y_dat = [1, 2, 4, 1, 2, 3, 1, 5, 2]
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_positive_mean_deviation(theta=theta, y=y_dat)
```

Calling the M-estimation procedure (note that `init` has 2 values now).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0, 0, ])
>>> estr.estimate()
```

Inspecting the parameter estimates

```
>>> estr.theta
```

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Regression

ee_regression(theta, X, y, model[, weights, ...])	Estimating equation for regression.
ee_mlogit(theta, X, y[, weights, offset])	Estimating equation for multinomial logistic regression.
ee_glm(theta, X, y, distribution, link[, ...])	Estimating equation for regression with a generalized linear model.
ee_robust_regression(theta, X, y, model, k)	Estimating equations for (unscaled) robust regression.
ee_ridge_regression(theta, X, y, model, penalty)	Estimating equations for ridge regression.
ee_lasso_regression(theta, X, y, model, penalty)	Estimating equation for an approximate LASSO (least absolute shrinkage and selection operator) regressor.
ee_elasticnet_regression(theta, X, y, model, ...)	Estimating equations for Elastic-Net regression.
ee_bridge_regression(theta, X, y, model, ...)	Estimating equation for bridge penalized regression.
ee_additive_regression(theta, X, y, ...[, ...])	Estimating equation for Generalized Additive Models (GAMs).

delicatessen.estimating_equations.regression.ee_regression

ee_regression(*theta*, *X*, *y*, *model*, *weights=None*, *offset=None*)

Estimating equation for regression. Options include: linear, logistic, and Poisson regression. The general estimating equation is

$$\sum_{i=1}^{n} \left\{ Y_i - g(X_i^T \theta) \right\} X_i = 0$$

where g indicates a transformation function. For linear regression, g is the identity function. Logistic regression uses the inverse-logit function, $\text{expit}(u) = 1/(1 + \exp(u))$. Finally, Poisson regression is $\exp(u)$.

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented by `[0, ] * X.shape[1]`.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.

- **model** (*str*) – Type of regression model to estimate. Options are `'linear'` (linear regression), `'logistic'` (logistic regression), and `'poisson'` (Poisson regression).
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a b-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_regression
```

Some generic data to estimate the regression models

```
>>> n = 500
>>> data = pd.DataFrame()
>>> data['X'] = np.random.normal(size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y1'] = 0.5 + 2*data['X'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
>>> data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(0.5 + 2*data['X'] - 1*data[
↪'Z']), size=n)
>>> data['Y3'] = np.random.poisson(lam=np.exp(0.5 + 2*data['X'] - 1*data['Z']),␣
↪size=n)
>>> data['C'] = 1
```

Note that `C` here is set to all 1's. This will be the intercept in the regression.

To start, we will demonstrate linear regression for the outcome `Y1`. Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_regression(theta=theta, X=data[['C', 'X', 'Z']], y=data['Y1'],␣
↪model='linear')
```

Calling the M-estimator (note that `init` requires 3 values, since `X.shape[1]` is 3).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.,])
>>> estr.estimate()
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Next, we can estimate the parameters for a logistic regression model as follows

```
>>> def psi(theta):
>>>         return ee_regression(theta=theta, X=data[['C', 'X', 'Z']], y=data['Y2'],
↪ model='logistic')
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.,])
>>> estr.estimate()
```

Finally, we can estimate the parameters for a Poisson regression model as follows

```
>>> def psi(theta):
>>>         return ee_regression(theta=theta, X=data[['C', 'X', 'Z']], y=data['Y3'],
↪ model='poisson')
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.,])
>>> estr.estimate()
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

delicatessen.estimating_equations.regression.ee_mlogit

ee_mlogit(*theta*, *X*, *y*, *weights=None*, *offset=None*)

Estimating equation for multinomial logistic regression. This estimating equation functionality supports unranked categorical outcome data, unlike `ee_regression` and `ee_glm`.

Unlike the other regression estimating equations, `ee_mlogit` expects a matrix of indicators for each possible value of `y`, with the first column being used as the referent category. In other words, the outcome variable is a matrix of dummy variables that includes the reference. The estimating equation for column r of the indicator variable Y_r of a Y with k unique categories is

$$\sum_{i=1}^{n} \left\{ Y_{r,i} - \frac{\exp(X_i^T \theta_r)}{1 + \sum_{j=2}^{k} \exp(X_i^T \theta_j)} \right\} X_i = 0$$

where θ_r are the coefficients correspond to the log odds ratio comparing Y_r to all other categories of Y. Here, θ is a 1-by-(b :math`times` (k-1)) array, where b is the distinct covariates included as part of **X**. So, the stack of estimating equations consists of (k-1) estimating equations of the dimension X_i. For example, if X is a 3-by-n matrix and Y has three unique categories, then θ will be a 1-by-6 array.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of $b \times$ (k-1) values. Therefore, initial values should consist of the same number as the number of columns present in the design matrix for each category of the outcome matrix besides the reference.
- **X** (*ndarray, list, vector*) – 2-dimensional design matrix of n observed covariates for b variables.
- **y** (*ndarray, list, vector*) – 2-dimensional indicator matrix of n observed outcomes.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.

- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a $(b \times (k\text{-}1))$-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_mlogit
```

Some generic data to estimate a multinomial logistic regression model

```
>>> d = pd.DataFrame()
>>> d['W'] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
>>> d['Y'] = [1, 1, 1, 1, 2, 2, 3, 3, 3, 1, 2, 2, 3, 3]
>>> d['C'] = 1
```

First, notice that `Y` needs to be pre-processed for use with `ee_mlogit`. To prepare the data, we need to convert `d['Y']` into a matrix of indicator variables. We can do this manually by

```
>>> d['Y1'] = np.where(d['Y'] == 1, 1, 0)
>>> d['Y2'] = np.where(d['Y'] == 2, 1, 0)
>>> d['Y3'] = np.where(d['Y'] == 3, 1, 0)
```

This can also be accomplished with `pd.get_dummies(d['Y'], drop_first=False)`.

For the reference category, we want to have `Y=1` as the reference. Therefore, `Y1` will be the first column in `y`. The pair of matrices are

```
>>> y = d[['Y1', 'Y2', 'Y3']]
>>> X = d[['C', 'W']]
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_mlogit(theta, X=X, y=y)
```

Calling the M-estimator (note that `init` requires 4 values, since `X.shape[1]` is 2 and `y.shape[1]` is 3).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0.])
>>> estr.estimate()
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Here, the first two values of `theta` correspond to `Y2` and the last two values of `theta` correspond to `Y3`.

A weighted multinomial logistic regression can be implemented by specifying the `weights` argument. An offset can be added by specifying the `offset` argument.

References

Kwak C & Clayton-Matthews A. (2002). Multinomial logistic regression. *Nursing Research*, 51(6), 404-410.

delicatessen.estimating_equations.regression.ee_glm

ee_glm(*theta*, *X*, *y*, *distribution*, *link*, *hyperparameter=None*, *weights=None*, *offset=None*)

Estimating equation for regression with a generalized linear model. Unlike `ee_regression`, this functionality supports generic distribution and link specifications. The general estimating equation for the outcome Y_i with the design matrix X_i

$$\sum_{i=1}^{n} \left\{ Y_i - g^{-1}(X_i^T \theta) \right\} \times \frac{D(\theta)}{v(\theta)} X_i = 0$$

where g is the link function, g^{-1} is the inverse link function, $D(\theta)$ is the derivative of the inverse link function by θ, and $v(\theta)$ is the variance function for the specified distribution.

Note: Some distributions (i.e., negative-binomial, gamma) involve additional parameters. These are estimated using additional parameter-specific estimating equations.

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented by `[0, ] * X.shape[1]`.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values.
- **distribution** (*str*) – Distribution for the generalized linear model. Options are: `'normal'` (alias: `gaussian`), `'binomial'` (aliases: `bernoulli`, `bin`), `'poisson'`, `'gamma'`, `'inverse_normal'` (alias: `inverse_gaussian`), `'negative_binomial'` (alias: `nb`), and `'tweedie'`.
- **link** (*str*) – Link function for the generalized linear model. Options are: `identity`, `log`, `logistic` (alias: `logit`), `probit`, `cauchit` (alias: `cauchy`), `loglog`, `cloglog`, `inverse`, and `square_root` (alias: `sqrt`).
- **hyperparameter** (*None, int, float*) – Hyperparameter specification. Default is `None`. This option is only used by the tweedie distribution. It is ignored by all other distributions.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of *n* weights. Default is `None`, which assigns a weight of 1 to all observations.
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Note: Link and distribution combinations are not checked for their validity. Some pairings may not converge or may produce nonsensical results. Please check the distribution-link combination you are using is valid.

Returns Returns a *b*-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_glm
```

Some generic data to estimate the regression models

```
>>> d = pd.DataFrame()
>>> d['X'] = [1, -1, 0, 1, 2, 1, -2, -1, 0, 3, -3, 1, 1, -1, -1, -2, 2, 0, -1, 0]
>>> d['Z'] = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> d['Y1'] = [1, 2, 4, 5, 2, 3, 1, 1, 3, 4, 2, 3, 7, 8, 2, 2, 1, 4, 2, 1]
>>> d['Y2'] = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0]
>>> d['C'] = 1
>>> X = d[['C', 'X', 'Z']]  # design matrix used hereafter
```

To start, we will demonstrate a GLM with a normal distribution and identity link. This GLM is equivalent to linear regression. Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_glm(theta, X=X, y=d['Y1'],
>>>                   distribution='normal', link='identity')
```

Calling the M-estimator (note that `init` requires 3 values, since `X.shape[1]` is 3).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.,])
>>> estr.estimate()
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Next, we will show a GLM with a binomial distribution and log link. This GLM can be used for binary data and estimates log risk ratios. So, one may prefer this model for interpretability over logistic regression (GLM with binomial distribution and logit link).

```
>>> def psi(theta):
>>>     return ee_glm(theta, X=X, y=d['Y2'],
>>>                   distribution='binomial', link='log')
```

Calling the M-estimator (note that `init` requires 3 values, since `X.shape[1]` is 3).

```
>>> estr = MEstimator(stacked_equations=psi, init=[-0.9, 0., 0.,])
>>> estr.estimate()
```

Notice that the root-finding solution may go off to weird places if bad starting values are given for the log-binomial GLM. This is because the log-binomial GLM is not bounded. Providing starting values close to the truth (or changing link functions) can help alleviate these issues. Other variations for binomial distribution link functions that are bounded include: `logit`, `cauchy`, `probit`, or `loglog`.

The negative-binomial and gamma distributions for GLM have an additional parameter that is estimated. Therefore, both of these distribution specifications require `X.shape[1] + 1` input starting values. Here, we illustrate a gamma distribution and log link GLM

```
>>> def psi(theta):
>>>     return ee_glm(theta, X=X, y=d['Y1'],
>>>                   distribution='gamma', link='log')
```

Calling the M-estimator (note that `init` requires 4 values, since `X.shape[1]` is 3 and the gamma distribution has an additional parameter).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0.])
>>> estr.estimate()
```

Note that delicatessen appropriately incorporates the estimation of the additional parameter for the negative-binomial and gamma distributions. This is unlike some statistical software that estimates this parameter but does *not* incorporate the uncertainty in estimation of that parameter. This may explain differences you encounter across software (and the delicatessen implementation is to be preferred, as it is a more honest expression of the uncertainty).

Finally, the tweedie distribution for GLM is a generalization of the Poisson and gamma distributions. Unlike the negative-binomial and gamma distributions, there is a fixed (i.e., not estimated) hyperparameter bounded to be > 0. When the tweedie distribution hyperparameter is set to 1, it is equivalent to the Poisson distribution. When the tweedie distribution hyperparameter is set to 2, it is equivalent to the gamma distribution. When the tweedie distribution hyperparameter is set to 3, it is equivalent to the inverse-normal distribution. However, the tweedie distribution hyperparameter can be specified for any values. Here, we illustrate the tweedie distribution that is between a Poisson and gamma distribution.

```
>>> def psi(theta):
>>>     return ee_glm(theta, X=X, y=d['Y1'],
>>>                   distribution='tweedie', link='log',
>>>                   hyperparameter=1.5)
```

Calling the M-estimator (note that `init` requires 3 values, since `X.shape[1]` is 3).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.])
>>> estr.estimate()
```

Notice that the tweedie distribution does not estimate an additional parameter, unlike the gamma distribution GLM described previously.

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Hilbe JM. (2011). *Negative Binomial Regression*. Cambridge University Press.

Nakashima E. (1997). Some methods for estimation in a Negative-Binomial model. *Annals of the Institute of Statistical Mathematics*, 49, 101-115.

delicatessen.estimating_equations.regression.ee_robust_regression

ee_robust_regression(*theta*, *X*, *y*, *model*, *k*, *loss='huber'*, *weights=None*, *upper=None*, *lower=None*, *offset=None*)

Estimating equations for (unscaled) robust regression. Robust linear regression is robust to outlying observations of the outcome variable. Currently, only linear regression is supported by `ee_robust_regression`. The estimating equation is

$$\sum_{i=1}^{n} f_k(Y_i - X_i^T \theta) X_i = 0$$

where $f_k(x)$ is the corresponding robust loss function. Options for the loss function include: Huber, Tukey's biweight, Andrew's Sine, and Hampel. See `robust_loss_function` for further details on the loss functions for the robust mean.

Note: The estimating-equation is not non-differentiable everywhere for some loss functions. Therefore, it is assumed that no points occur exactly at the non-differentiable points. For truly continuous Y, the probability of that occurring is zero.

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented via `[0, ] * X.shape[1]`.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.
- **model** (*str*) – Type of regression model to estimate. Options include: `'linear'` (linear regression).
- **k** (*int, float*) – Tuning or hyperparameter for the chosen loss function. Notice that the choice of hyperparameter should depend on the chosen loss function.
- **loss** (*str, optional*) – Robust loss function to use. Default is `'huber'`. Options include `'andrew'`, `'hampel'`, `'tukey'`.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.
- **lower** (*int, float, None, optional*) – Lower parameter for the Hampel loss function. This parameter does not impact the other loss functions. Default is `None`.

- **upper** (*int, float, None, optional*) – Upper parameter for the Hampel loss function. This parameter does not impact the other loss functions. Default is `None`.
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a *b*-by-*n* NumPy array evaluated for the input `theta`

Return type array

Examples

Construction of a estimating equation(s) with `ee_robust_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_robust_regression
```

Some generic data to estimate a robust linear regression model

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['X'] = np.random.normal(size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y'] = 0.5 + 2*data['X'] - 1*data['Z'] + np.random.normal(loc=0, scale=3,␣
↪size=n)
>>> data['C'] = 1
```

```
>>> X = data[['C', 'X', 'Z']]
>>> y = data['Y']
```

Note that `C` here is set to all 1's. This will be the intercept in the regression.

Defining psi, or the stacked estimating equations for Huber's robust regression

```
>>> def psi(theta):
>>>         return ee_robust_regression(theta=theta, X=X, y=y, model='linear', k=1.
↪345, loss='huber')
```

Calling the M-estimator procedure (note that `init` has 3 values now, since `X.shape[1]` is 3).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.,])
>>> estr.estimate()
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Andrews DF. (1974). A robust method for multiple linear regression. *Technometrics*, 16(4), 523-531.

Beaton AE & Tukey JW (1974). The fitting of power series, meaning polynomials, illustrated on band-spectroscopic data. *Technometrics*, 16(2), 147-185.

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

Hampel FR. (1971). A general qualitative definition of robustness. *The Annals of Mathematical Statistics*, 42(6), 1887-1896.

Huber PJ. (1964). Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1), 73–101.

Huber PJ, Ronchetti EM. (2009) Robust Statistics 2nd Edition. Wiley. pgs 98-100

delicatessen.estimating_equations.regression.ee_ridge_regression

`ee_ridge_regression`(*theta*, *X*, *y*, *model*, *penalty*, *weights=None*, *center=0.0*, *offset=None*)

Estimating equations for ridge regression. Ridge regression applies an L2-regularization through a squared magnitude penalty. The estimating equation for Ridge linear regression is

$$\sum_{i=1}^{n} \left\{ (Y_i - X_i^T \theta) X_i - \lambda \theta \right\} = 0$$

where λ is the penalty term.

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Note: The 'strength' of the penalty term is indicated by λ, which is the `penalty` argument scaled (or divided by) the number of observations.

Parameters

- **`theta`** (*`ndarray, list, vector`*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented via `[0, ] * X.shape[1]`.
- **`X`** (*`ndarray, list, vector`*) – 2-dimensional vector of n observed values for b variables.
- **`y`** (*`ndarray, list, vector`*) – 1-dimensional vector of n observed values.
- **`model`** (*`str`*) – Type of regression model to estimate. Options are `'linear'` (linear regression), `'logistic'` (logistic regression), and `'poisson'` (Poisson regression).
- **`penalty`** (*`int, float, ndarray, list, vector`*) – Penalty term to apply to all coefficients (if only a integer or float is provided) or the corresponding coefficient (if a list or vector of integers or floats is provided). Note that the penalty term should either consists of a single value or b values (to match the length of `theta`). The penalty is scaled by n.
- **`weights`** (*`ndarray, list, vector, None, optional`*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.

- **center** (*int, float, ndarray, list, vector, optional*) – Center or reference value to penalized estimated coefficients towards. Default is `0`, which penalized coefficients towards the null. Other center values can be specified for all coefficients (by providing an integer or float) or covariate-specific centering values (by providing a vector of values of the same length as X).
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a *b*-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_ridge_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_ridge_regression
```

Some generic data to estimate Ridge regression models

```
>>> n = 500
>>> data = pd.DataFrame()
>>> data['V'] = np.random.normal(size=n)
>>> data['W'] = np.random.normal(size=n)
>>> data['X'] = data['W'] + np.random.normal(scale=0.25, size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y1'] = 0.5 + 2*data['W'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
>>> data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(0.5 + 2*data['W'] - 1*data[
↪'Z']), size=n)
>>> data['Y3'] = np.random.poisson(lam=np.exp(1 + 2*data['W'] - 1*data['Z']),␣
↪size=n)
>>> data['C'] = 1
```

Note that `C` here is set to all 1's. This will be the intercept in the regression.

Defining psi, or the stacked estimating equations. Note that the penalty is a list of values. Here, we are *not* penalizing the intercept (which is generally recommended when the intercept is unlikely to be zero). The remainder of covariates have a penalty of 10 applied.

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y1']
>>>     return ee_ridge_regression(theta=theta, X=x, y=y, model='linear',␣
↪penalty=penalty_vals)
```

Calling the M-estimator (note that `init` has 5 values now, since `X.shape[1]` is 5).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Next, we can estimate the parameters for a logistic regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y2']
>>>     return ee_ridge_regression(theta=theta, X=x, y=y, model='logistic',␣
↪penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Finally, we can estimate the parameters for a Poisson regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y3']
>>>     return ee_ridge_regression(theta=theta, X=x, y=y, model='poisson',␣
↪penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Fu WJ. (1998). Penalized regressions: the Bridge versus the LASSO. *Journal of Computational and Graphical Statistics*, 7(3), 397-416.

Fu WJ. (2003). Penalized estimating equations. *Biometrics*, 59(1), 126-132.

delicatessen.estimating_equations.regression.ee_lasso_regression

`ee_lasso_regression`(*theta*, *X*, *y*, *model*, *penalty*, *epsilon=0.003*, *weights=None*, *center=0.0*, *offset=None*)

Estimating equation for an approximate LASSO (least absolute shrinkage and selection operator) regressor. LASSO regression applies an L1-regularization through a magnitude penalty.

The estimating equation for the approximate LASSO linear regression is

$$\sum_{i=1}^{n} \left\{ (Y_i - X_i^T \theta) X_i - \lambda (1 + \epsilon) |\theta|^{\epsilon} sign(\theta) \right\} = 0$$

where λ is the penalty term.

Note: As the derivative of the estimating equation for LASSO is not defined at $\theta = 0$, the bread (and sandwich) cannot be used to estimate the variance in all settings.

Here, an approximation based on the bridge penalty for the LASSO is used. For the bridge penalty, LASSO is the special case where $\epsilon = 0$. By making $\epsilon > 0$, we can approximate the LASSO. The true LASSO may not be possible to implement due to the existence of multiple solutions

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Note: The 'strength' of the penalty term is indicated by λ, which is the `penalty` argument scaled (or divided by) the number of observations.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented via `[0, ] * X.shape[1]`.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.
- **model** (*str*) – Type of regression model to estimate. Options are `'linear'` (linear regression), `'logistic'` (logistic regression), and `'poisson'` (Poisson regression).
- **penalty** (*int, float, ndarray, list, vector*) – Penalty term to apply to all coefficients (if only a integer or float is provided) or the corresponding coefficient (if a list or vector of integers or floats is provided). Note that the penalty term should either consists of a single value or b values (to match the length of `theta`). The penalty is scaled by n.
- **epsilon** (*float, optional*) – Approximation error to use for the LASSO approximation. Default argument is `0.003`, which results in a bridge penalty of `1.0003`.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.
- **center** (*int, float, ndarray, list, vector, optional*) – Center or reference value to penalized estimated coefficients towards. Default is `0`, which penalized coefficients towards the null. Other center values can be specified for all coefficients (by providing an integer or float) or covariate-specific centering values (by providing a vector of values of the same length as X).
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a b-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_lasso_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_lasso_regression
```

Some generic data to estimate a LASSO regression model

```
>>> n = 500
>>> data = pd.DataFrame()
>>> data['V'] = np.random.normal(size=n)
>>> data['W'] = np.random.normal(size=n)
>>> data['X'] = data['W'] + np.random.normal(scale=0.25, size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y1'] = 0.5 + 2*data['W'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
>>> data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(0.5 + 2*data['W'] - 1*data[
↪'Z']), size=n)
>>> data['Y3'] = np.random.poisson(lam=np.exp(1 + 2*data['W'] - 1*data['Z']),␣
↪size=n)
>>> data['C'] = 1
```

Note that `C` here is set to all 1's. This will be the intercept in the regression.

Defining psi, or the stacked estimating equations. Note that the penalty is a list of values. Here, we are *not* penalizing the intercept (which is generally recommended when the intercept is unlikely to be zero). The remainder of covariates have a penalty of 10 applied.

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y1']
>>>     return ee_lasso_regression(theta=theta, X=x, y=y, model='linear',␣
↪penalty=penalty_vals)
```

Calling the M-estimator (note that `init` has 5 values now, since `X.shape[1]` is 5).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=20000)
```

Inspecting the parameter estimates

```
>>> estr.theta
```

Next, we can estimate the parameters for a logistic regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y2']
>>>     return ee_lasso_regression(theta=theta, X=x, y=y, model='logistic',␣
↪penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=20000)
```

Finally, we can estimate the parameters for a Poisson regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y3']
>>>     return ee_lasso_regression(theta=theta, X=x, y=y, model='poisson',␣
↪penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=20000)
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Fu WJ. (1998). Penalized regressions: the Bridge versus the LASSO. *Journal of Computational and Graphical Statistics*, 7(3), 397-416.

Fu WJ. (2003). Penalized estimating equations. *Biometrics*, 59(1), 126-132.

delicatessen.estimating_equations.regression.ee_elasticnet_regression

ee_elasticnet_regression(*theta*, *X*, *y*, *model*, *penalty*, *ratio*, *epsilon=0.003*, *weights=None*, *center=0.0*, *offset=None*)

Estimating equations for Elastic-Net regression. Elastic-Net applies both L1- and L2-regularization at a pre-specified ratio. Notice that the L1 penalty is based on an approximation. See `ee_lasso_regression` for further details on the approximation for the L1 penalty.

The estimating equation for Elastic-Net linear regression with the approximate L1 penalty is

$$\sum_{i=1}^{n} \left\{ (Y_i - X_i^T \theta) X_i - \lambda r (1+\epsilon) |\theta|^{\epsilon} sign(\theta) - \lambda (1-r) \theta \right\} = 0$$

where λ is the penalty term and r is the ratio for the L1 vs L2 penalty.

Note: As the derivative of the estimating equation for LASSO is not defined at $\theta = 0$, the bread (and sandwich) cannot be used to estimate the variance in all settings.

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Note: The ‘strength’ of the penalty term is indicated by λ, which is the `penalty` argument scaled (or divided by) the number of observations.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented via `[0, ] * X.shape[1]`.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.
- **model** (*str*) – Type of regression model to estimate. Options are `'linear'` (linear regression), `'logistic'` (logistic regression), and `'poisson'` (Poisson regression).
- **penalty** (*int, float, ndarray, list, vector*) – Penalty term to apply to all coefficients (if only a integer or float is provided) or the corresponding coefficient (if a list or vector of integers or floats is provided). Note that the penalty term should either consists of a single value or b values (to match the length of `theta`). The penalty is scaled by n.
- **ratio** (*float*) – Ratio for the L1 vs L2 penalty in Elastic-net. The ratio must be be $0 \leq r \leq 1$. Setting `ratio=1` results in LASSO and `ratio=0` results in ridge regression.
- **epsilon** (*float, optional*) – Approximation error to use for the LASSO approximation. Default argument is `0.003`, which results in a bridge penalty of `1.0003`.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is ``None`, which assigns a weight of 1 to all observations.
- **center** (*int, float, ndarray, list, vector, optional*) – Center or reference value to penalized estimated coefficients towards. Default is `0`, which penalized coefficients towards the null. Other center values can be specified for all coefficients (by providing an integer or float) or covariate-specific centering values (by providing a vector of values of the same length as X).
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a b-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_elasticnet_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_elasticnet_regression
```

Some generic data to estimate a Elastic-Net regression model

```
>>> n = 500
>>> data = pd.DataFrame()
>>> data['V'] = np.random.normal(size=n)
>>> data['W'] = np.random.normal(size=n)
>>> data['X'] = data['W'] + np.random.normal(scale=0.25, size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y1'] = 0.5 + 2*data['W'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
```

(continues on next page)

(continued from previous page)

```
>>> data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(0.5 + 2*data['W'] - 1*data[
↪'Z']), size=n)
>>> data['Y3'] = np.random.poisson(lam=np.exp(1 + 2*data['W'] - 1*data['Z']),␣
↪size=n)
>>> data['C'] = 1
```

Note that `C` here is set to all 1's. This will be the intercept in the regression.

Defining psi, or the stacked estimating equations. Note that the penalty is a list of values. Here, we are *not* penalizing the intercept (which is generally recommended when the intercept is unlikely to be zero). The remainder of covariates have a penalty of 10 applied.

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y1']
>>>     return ee_elasticnet_regression(theta=theta, X=x, y=y, model='linear',␣
↪ratio=0.5, penalty=penalty_vals)
```

Calling the M-estimator (note that `init` has 5 values now, since `X.shape[1]` is 5).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate()
```

Inspecting the parameter estimates

```
>>> estr.theta
```

Next, we can estimate the parameters for a logistic regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y2']
>>>     return ee_elasticnet_regression(theta=theta, X=x, y=y, model='logistic',␣
↪ratio=0.5, penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=20000)
```

Finally, we can estimate the parameters for a Poisson regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y3']
>>>     return ee_elasticnet_regression(theta=theta, X=x, y=y, model='poisson',␣
↪ratio=0.5, penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=20000)
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Fu WJ. (1998). Penalized regressions: the Bridge versus the LASSO. *Journal of Computational and Graphical Statistics*, 7(3), 397-416.

Fu WJ. (2003). Penalized estimating equations. *Biometrics*, 59(1), 126-132.

delicatessen.estimating_equations.regression.ee_bridge_regression

ee_bridge_regression(*theta*, *X*, *y*, *model*, *penalty*, *gamma*, *weights=None*, *center=0.0*, *offset=None*)

Estimating equation for bridge penalized regression. The bridge penalty is a generalization of penalized regression, that includes L1 and L2-regularization as special cases.

Note: While the bridge penalty is defined for $\gamma > 0$, the provided estimating equation only supports $\gamma \geq 1$. Additionally, the derivative of the estimating equation is not defined when $\gamma < 2$. Therefore, the bread (and sandwich) cannot be used to estimate the variance in those settings.

The estimating equation for bridge penalized linear regression is

$$\sum_{i=1}^{n} \left\{ (Y_i - X_i^T \theta) X_i - \lambda \gamma |\theta|^{\gamma - 1} sign(\theta) \right\} = 0$$

where λ is the penalty term and γ is a tuning parameter.

Here, θ is a 1-by-b array, which corresponds to the coefficients in the corresponding regression model and b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then θ will be a 1-by-3 array. The code is general to allow for an arbitrary number of elements in `X`.

Note: The 'strength' of the penalty term is indicated by λ, which is the `penalty` argument scaled (or divided by) the number of observations.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b values. Therefore, initial values should consist of the same number as the number of columns present. This can easily be implemented via `[0, ] * X.shape[1]`.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.
- **model** (*str*) – Type of regression model to estimate. Options are `'linear'` (linear regression), `'logistic'` (logistic regression), and `'poisson'` (Poisson regression).
- **penalty** (*int, float, ndarray, list, vector*) – Penalty term to apply to all coefficients (if only a integer or float is provided) or the corresponding coefficient (if a list or vector of integers or floats is provided). Note that the penalty term should either consists of a single value or b values (to match the length of `theta`). The penalty is scaled by n.
- **gamma** (*float*) – Hyperparameter for the bridge penalty, defined for $\gamma > 0$. However, only $\gamma \geq 1$ are supported.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.

- **center** (*int, float, ndarray, list, vector, optional*) – Center or reference value to penalized estimated coefficients towards. Default is 0, which penalized coefficients towards the null. Other center values can be specified for all coefficients (by providing an integer or float) or covariate-specific centering values (by providing a vector of values of the same length as X).
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is None, which applies no offset term.

Returns Returns a *b*-by-*n* NumPy array evaluated for the input theta.

Return type array

Examples

Construction of a estimating equation(s) with ee_bridge_regression should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_bridge_regression
```

Some generic data to estimate a bridge regression model

```
>>> n = 500
>>> data = pd.DataFrame()
>>> data['V'] = np.random.normal(size=n)
>>> data['W'] = np.random.normal(size=n)
>>> data['X'] = data['W'] + np.random.normal(scale=0.25, size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y1'] = 0.5 + 2*data['W'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
>>> data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(0.5 + 2*data['W'] - 1*data[
↪'Z']), size=n)
>>> data['Y3'] = np.random.poisson(lam=np.exp(1 + 2*data['W'] - 1*data['Z']),␣
↪size=n)
>>> data['C'] = 1
```

Note that C here is set to all 1's. This will be the intercept in the regression.

Defining psi, or the stacked estimating equations. Note that the penalty is a list of values. Here, we are *not* penalizing the intercept (which is generally recommended when the intercept is unlikely to be zero). The remainder of covariates have a penalty of 10 applied.

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y']
>>>     return ee_bridge_regression(theta=theta, X=x, y=y, model='linear', gamma=2.
↪3, penalty=penalty_vals)
```

Calling the M-estimator (note that init has 5 values now, since X.shape[1] is 5).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.])
>>> estr.estimate()
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Next, we can estimate the parameters for a logistic regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y2']
>>>     return ee_bridge_regression(theta=theta, X=x, y=y, model='logistic',␣
↪gamma=2.3, penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=5000)
```

Finally, we can estimate the parameters for a Poisson regression model as follows

```
>>> penalty_vals = [0., 10., 10., 10., 10.]
>>> def psi(theta):
>>>     x, y = data[['C', 'V', 'W', 'X', 'Z']], data['Y3']
>>>     return ee_bridge_regression(theta=theta, X=x, y=y, model='poisson', gamma=2.
↪3, penalty=penalty_vals)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.01, 0.01, 0.01, 0.01, 0.01])
>>> estr.estimate(solver='lm', maxiter=5000)
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Fu WJ. (1998). Penalized regressions: the Bridge versus the LASSO. *Journal of Computational and Graphical Statistics*, 7(3), 397-416.

Fu WJ. (2003). Penalized estimating equations. *Biometrics*, 59(1), 126-132.

delicatessen.estimating_equations.regression.ee_additive_regression

ee_additive_regression(*theta*, *X*, *y*, *specifications*, *model*, *weights=None*, *offset=None*)

Estimating equation for Generalized Additive Models (GAMs). GAMs are an extension of generalized linear models that allow for more flexible specifications of relationships of continuous variables. This flexibility is accomplished via splines. To further control the flexibility, the spline terms are penalized.

Note: The implemented GAM uses L2-penalization. This penalization only applies to the generated spline terms (i.e., penalization decreases the 'wiggliness' of the estimated relationships).

The estimating equation for a generalized additive linear regression model is

$$\sum_{i=1}^{n} \left\{ (Y_i - f(X_i)^T \theta) f(X_i) - \lambda \theta \right\} = 0$$

where λ is the penalty term.

While this looks similar to Ridge regression, there are two important differences: the function $f()$ and how λ is defined. First, the function $f()$ denotes a general vector function. For spline terms, this function defines the basis functions for the splines (set via the `specifications` parameter). For non-spline terms, this is the identity function (i.e., no changes are made to the input). This setup allows for terms to be selectively modeled using splines (e.g., categorical features are not modeled using splines). Next, the penalty term, λ, is only non-zero for θ that correspond to parameters for splines (i.e., only the spline terms are penalized). This is distinction from default Ridge regression, which penalizes all terms in the model.

Note: Originally, GAMs were implemented via splines with a knot at each unique values of X. More recently, GAMs use a more moderate amount of knots to improve computationally efficiency. Both versions can be implemented by `ee_additive_regression` through setting the knot locations.

Here, θ is a 1-by-(b `+ `k) array, where b is the distinct covariates included as part of `X` and the k distinct spline basis functions. For example, if `X` is a 2-by-n matrix with a 10-knot natural spline for the second column in X, then θ will be a 1-by-(2+9) array. The code is general to allow for an arbitrary number of X variables and spline knots.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of b `+ `k values. Number of values should match the number of columns in the additive design matrix.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values.
- **specifications** (*list, dict, None*) – A list of dictionaries that define the hyperparameters for the spline (e.g., number of knots, strength of penalty). For terms that should not have splines, `None` should be specified instead (see examples below). Each dictionary supports the following parameters: "knots", "natural", "power", "penalty" knots (list): controls the position of the knots, with knots are placed at given locations. There is no default, so must be specified by the user. natural (bool): controls whether to generate natural (restricted) or unrestricted splines. Default is `True`, which corresponds to natural splines. power (float): controls the power to raise the spline terms to. Default is 3, which corresponds to cubic splines. penalty (float): penalty term (λ) applied to each corresponding spline basis term. Default is 0, which applies no penalty to the spline basis terms.
- **model** (*str*) – Type of regression model to estimate. Options are `'linear'` (linear regression), `'logistic'` (logistic regression), and `'poisson'` (Poisson regression).
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.
- **offset** (*ndarray, list, vector, None, optional*) – A 1-dimensional offset to be included in the model. Default is `None`, which applies no offset term.

Returns Returns a (b `+ `k)-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_additive_regression` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_additive_regression
>>> from delicatessen.utilities import additive_design_matrix, regression_
↪predictions
```

Some generic data to estimate a generalized additive model

```
>>> n = 2000
>>> data = pd.DataFrame()
>>> x = np.random.uniform(-5, 5, size=n)
>>> data['X'] = x
>>> data['Y1'] = np.exp(x+0.5)) + np.abs(x) + np.random.normal(scale=1.5, size=n)
>>> data['Y2'] = np.random.binomial(n=1, p=logistic.cdf(np.exp(np.sin(x+0.5)) + np.
↪abs(x)), size=n)
>>> data['Y3'] = np.random.poisson(lam=np.exp(np.exp(np.sin(x+0.5)) + np.abs(x)),␣
↪size=n)
>>> data['C'] = 1
```

Note that `C` here is set to all 1's. This will be the intercept in the regression. Further, notice that the relationship between `X` and the various `Y`'s is not linear.

The design matrix for linear regression would be `X = np.asarray(d[['C', 'X']])`. As the intercept is a constant, we only want spline terms to be applied to `'X'` column. To define the spline specifications, we create the following list

```
>>> specs = [None, {"knots": [-4, -3, -2, -1, 0, 1, 2, 3, 4], "penalty": 10}]
```

This tells `ee_additive_regression` to not generate a spline term for the first column in the input design matrix and to generate a default spline with knots at the specified locations and penalty of 10 for the second column in the input design matrix. Interally, the design matrix processing is done by the `additive_design_matrix` utility function. We can see what the output of that function looks like via

```
>>> Xa_design = additive_design_matrix(X=np.asarray(data[['C', 'X']]),␣
↪specifications=specs)
```

That output matrix is the corresponding design matrix. Use of the `additive_design_matrix` utility will be demonstrated later for generating predictions from the estimated parameters.

Now psi, or the stacked estimating equations.

```
>>> def psi(theta):
>>>     x, y = data[['C', 'X']], data['Y']
>>>     return ee_additive_regression(theta=theta, X=x, y=y, model='linear',␣
↪specifications=specs)
```

Calling the M-estimator. Note that the input initial values depends on the number of splines being generated. To easily determine the number of initial values, we can use the shape of the previously generated design matrix (`Xa_design`)

```
>>> n_params = Xa_design.shape[1]
>>> estr = MEstimator(stacked_equations=psi, init=[0., ]*n_params)
>>> estr.estimate()
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

While all these estimates can be easily inspected, interpretting them is not easy. Instead, we can generate predictions from the GAM and plot the estimated regression line. To do this, we will first create a new design matrix where `'X'` is evenly spaced over a range of values

```
>>> p = pd.DataFrame()
>>> p['X'] = np.linspace(-5, 5, 200)
>>> p['C'] = 1
>>> Xa_pred = additive_design_matrix(X=np.asarray(p[['C', 'X']]),␣
↪specifications=specs)
```

To generate the predicted values of Y (and the corresponding confidence intervals), we do the following

```
>>> yhat = regression_predictions(Xa_pred, theta=estr.theta, covariance=estr.
↪variance)
```

For further details, see the `regression_predictions` utility function documentation.

Other optional specifications are available for the spline terms. Here, we will specify an unrestricted quadratic spline with a penalty of 5.5 for the second column of the design matrix.

```
>>> specs = [None, {"knots": [-4, -2, 0, 2, 4], "natural": False, "power": 2,
↪"penalty": 5.5}]
```

See the documentation of `additive_design_matrix` for additional examples of how to specify the additive design matrix and corresponding splines.

Lastly, knots could be placed at each unique observation via

```
>>> specs = [None, {"knots": np.unique(data['X']), "penalty": 500}]
```

Note that the penalty is increased here (as the number of knots has dramatically increased).

A GAM for a binary outcome (i.e., logistic regression) can be implemented as follows

```
>>> specs = [None, {"knots": [-4, -3, -2, -1, 0, 1, 2, 3, 4], "penalty": 10}]
>>> Xa_design = additive_design_matrix(X=np.asarray(data[['C', 'X']]),␣
↪specifications=specs)
>>> n_params = Xa_design.shape[1]
```

```
>>> def psi(theta):
>>>     x, y = data[['C', 'X']], data['Y2']
>>>     return ee_additive_regression(theta=theta, X=x, y=y, model='logistic',␣
↪specifications=specs)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.]*n_params)
>>> estr.estimate(solver='lm', maxiter=5000)
```

A GAM for count outcomes (i.e., Poisson regression) can be implemented as follows

```
>>> specs = [None, {"knots": [-4, -3, -2, -1, 0, 1, 2, 3, 4], "penalty": 10}]
>>> Xa_design = additive_design_matrix(X=np.asarray(data[['C', 'X']]),␣
↪specifications=specs)
>>> n_params = Xa_design.shape[1]
```

```
>>> def psi(theta):
>>>     x, y = data[['C', 'X']], data['Y3']
>>>     return ee_additive_regression(theta=theta, X=x, y=y, model='poisson',␣
↪specifications=specs)
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0.]*n_params)
>>> estr.estimate(solver='lm', maxiter=5000)
```

Weighted models can be estimated by specifying the optional `weights` argument.

References

Fu WJ. (2003). Penalized estimating equations. *Biometrics*, 59(1), 126-132.

Hastie TJ. (2017). Generalized additive models. *In Statistical models in S* (pp. 249-307). Routledge.

Marx BD, & Eilers PH. (1998). Direct generalized additive modeling with penalized likelihood. *Computational Statistics & Data Analysis*, 28(2), 193-209.

Wild CJ, & Yee TW. (1996). Additive extensions to generalized estimating equation methods. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(4), 711-725.

Measurement

ee_rogan_gladen(theta, y, y_star, r[, weights])	Estimating equation for the Rogan-Gladen correction for mismeasured *binary* outcomes.
ee_rogan_gladen_extended(theta, y, y_star, r, X)	Estimating equation for the extended Rogan-Gladen correction for mismeasured *binary* outcomes.

delicatessen.estimating_equations.measurement.ee_rogan_gladen

ee_rogan_gladen(*theta*, *y*, *y_star*, *r*, *weights=None*)

Estimating equation for the Rogan-Gladen correction for mismeasured *binary* outcomes. This estimator uses external data to estimate the sensitivity and specificity, and then uses those external estimates to correct the estimated proportion. The general form of the estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} \mu \times \{\alpha + \beta - 1\} - \{\mu^* + \beta - 1\} \\ R_i(Y_i^* - \mu^*) \\ (1 - R_i) Y_i \{Y_i^* - \beta\} \\ (1 - R_i)(1 - Y_i) \{(1 - Y_i^*) - \alpha\} \end{bmatrix} = 0$$

where Y is the true value of the outcome, Y^* is the mismeasured value of the outcome, R is the indicator for the main study data, μ is the corrected mean, μ^* is the mismeasured mean in the main study data, β is the sensitivity, and α is the specificity. The first estimating equation is the corrected proportion, the second is the naive proportion, the third is for sensitivity, and the fourth for specificity.

Here, `theta` is a 1-by-4 array.

Note: The Rogan-Gladen estimator may provide corrected proportions outside of $[0, 1]$ when $\alpha + \beta \leq 1$.

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of 4 values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n observed values. These are the gold-standard Y measurements in the external sample. All values should be either 0 or 1, and be non-missing among those with $R = 0$.
- **y_star** (*ndarray, list, vector*) – 1-dimensional vector of n observed values. These are the mismeasured Y values. All values should be either 0 or 1, and be non-missing among all observations.
- **r** (*ndarray, list, vector*) – 1-dimensional vector of n indicators regarding whether an observation was part of the external validation data. Indicator should designate if observations are the main data.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations.

Returns Returns a 4-by-n NumPy array evaluated for the input `theta`

Return type array

Examples

Construction of a estimating equation(s) with `ee_rogan_gladen` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_rogan_gladen
```

Replicating the published example from Cole et al. (2023).

```
>>> d = pd.DataFrame()
>>> d['Y_star'] = [0, 1] + [0, 1, 0, 1]
>>> d['Y'] = [np.nan, np.nan] + [0, 0, 1, 1]
>>> d['S'] = [1, 1] + [0, 0, 0, 0]
>>> d['n'] = [270, 680] + [71, 18, 38, 203]
>>> d = pd.DataFrame(np.repeat(d.values, d['n'], axis=0), columns=d.columns)
```

Applying the Rogan-Gladen correction to this example

```
>>> def psi(theta):
>>>     return ee_rogan_gladen(theta=theta, y=d['Y'],
>>>                            y_star=d['Y_star'], r=d['S'])
```

Notice that `y` corresponds to the gold-standard outcomes (only available where R=0), `y_star` corresponds to the mismeasured covariate data (available for R=1 and R=0), and `r` corresponds to the indicator for the main data source. Now we can call the M-Estimator.

```
>>> estr = MEstimator(psi, init=[0.5, 0.5, .75, .75])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

The corrected proportion is

```
>>> estr.theta[0]
```

Inverse probability weights can be used through the `weights` argument. See the applied examples for a demonstration.

References

Cole SR, Edwards JK, Breskin A, Rosin S, Zivich PN, Shook-Sa BE, & Hudgens MG. (2023). Illustration of 2 Fusion Designs and Estimators. *American Journal of Epidemiology*, 192(3), 467-474.

Rogan WJ & Gladen B. (1978). Estimating prevalence from the results of a screening test. *American Journal of Epidemiology*, 107(1), 71-76.

Ross RK, Zivich PN, Stringer JSA, & Cole SR. (2024). M-estimation for common epidemiological measures: introduction and applied examples. *International Journal of Epidemiology*, 53(2), dyae030.

delicatessen.estimating_equations.measurement.ee_rogan_gladen_extended

ee_rogan_gladen_extended(*theta*, *y*, *y_star*, *r*, *X*, *weights=None*)

Estimating equation for the extended Rogan-Gladen correction for mismeasured *binary* outcomes. This estimator uses external data to estimate the sensitivity and specificity conditional on covariates, and then uses those external estimates to correct the estimated proportion. The general form of the estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} R_i \times \left\{ \frac{Y^* + m(X_i;\beta) - 1}{m(X_i;\alpha) + m(X_i;\beta) - 1} - \mu \right\} \\ (1 - R_i) Y_i \left\{ Y_i^* - m(X_i; \beta) \right\} X_i^T \\ (1 - R_i)(1 - Y_i) \left\{ (1 - Y_i^*) - m(X_i; \beta) \right\} X_i^T \end{bmatrix} = 0$$

where Y is the true value of the outcome, Y^* is the mismeasured value of the outcome. The first estimating equation is the corrected proportion, the second is for sensitivity, and the third for specificity.

If X is of dimension p, then `theta` is a 1-by-(1+2`p`) array. Note that the design matrix is shared across the sensitivity and specificity models.

Note: The Rogan-Gladen estimator may provide corrected proportions outside of $[0, 1]$ when $\alpha + \beta \leq 1$, or the addition of sensitivity and specificity is less than or equal to one.

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of 4 values.

- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values. These are the gold-standard Y measurements in the external sample. All values should be either 0 or 1, and be non-missing among those with $R = 0$.
- **y_star** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values. These are the mismeasured Y values. All values should be either 0 or 1, and be non-missing among all observations.
- **r** (*ndarray, list, vector*) – 1-dimensional vector of *n* indicators regarding whether an observation was part of the external validation data. Indicator should designate if observations are the main data.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of a design matrix for the sensitivity and specificity models.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of *n* weights. Default is `None`, which assigns a weight of 1 to all observations.

Returns Returns a 4-by-*n* NumPy array evaluated for the input `theta`

Return type array

Examples

Construction of a estimating equation(s) with `ee_rogan_gladen_extended` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_rogan_gladen_extended
```

Replicating the example from Cole et al. (2023).

```
>>> d = pd.DataFrame()
>>> d['Y_star'] = [0, 1] + [0, 1, 0, 1]
>>> d['Y'] = [np.nan, np.nan] + [0, 0, 1, 1]
>>> d['S'] = [1, 1] + [0, 0, 0, 0]
>>> d['n'] = [270, 680] + [71, 18, 38, 203]
>>> d = pd.DataFrame(np.repeat(d.values, d['n'], axis=0), columns=d.columns)
>>> d['C'] = 1
```

Applying the Rogan-Gladen correction to this example

```
>>> def psi(theta):
>>>     return ee_rogan_gladen_extended(theta=theta, y=d['Y'],
>>>                                     y_star=d['Y_star'],
>>>                                     X=d[['C', ]], r=d['S'])
```

Notice that `y` corresponds to the gold-standard outcomes (only available where R=0), `y_star` corresponds to the mismeasured covariate data (available for R=1 and R=0), and `r` corresponds to the indicator for the main data source. Now we can call the M-Estimator.

```
>>> estr = MEstimator(psi, init=[0.5, 1., 1.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Note: The sensitivity and specificity in `ee_rogan_gladen_extended` correspond to the logit transformations, unlike `ee_rogan_gladen` which returns the sensitivity and specificity directly.

The corrected proportion is

```
>>> estr.theta[0]
```

Inverse probability weights can be used through the `weights` argument. See the applied examples for a demonstration.

References

Cole SR, Edwards JK, Breskin A, Rosin S, Zivich PN, Shook-Sa BE, & Hudgens MG. (2023). Illustration of 2 Fusion Designs and Estimators. *American Journal of Epidemiology*, 192(3), 467-474.

Rogan WJ & Gladen B. (1978). Estimating prevalence from the results of a screening test. *American Journal of Epidemiology*, 107(1), 71-76.

Ross RK, Cole SR, Edwards JK, Zivich PN, Westreich D, Daniels JL, Price JT & Stringer JSA. (2024). Leveraging External Validation Data: The Challenges of Transporting Measurement Error Parameters. *Epidemiology*, 35(2), 196-207.

Survival

`ee_exponential_model`(theta, t, delta)	Estimating equation for an exponential model.
`ee_exponential_measure`(theta, times, n, ...)	Estimating equation to calculate a survival measure (survival, density, risk, hazard, cumulative hazard) from the exponential model.
`ee_weibull_model`(theta, t, delta)	Estimating equation for a two-parameter Weibull model.
`ee_weibull_measure`(theta, times, n, measure, ...)	Estimating equation to calculate a survival measure (survival, density, risk, hazard, cumulative hazard) for the Weibull model.
`ee_aft_weibull`(theta, X, t, delta[, weights])	Estimating equation for accelerated failure time (AFT) model with a Weibull distribution.
`ee_aft_weibull_measure`(theta, times, X, ...)	Estimating equation to calculate a survival measure (survival, density, risk, hazard, cumulative hazard) given a specific covariate pattern and Weibull accelerated failure time (AFT) model.

delicatessen.estimating_equations.survival.ee_exponential_model

ee_exponential_model(*theta*, *t*, *delta*)

Estimating equation for an exponential model. Let T_i indicate the time of the event and C_i indicate the time to right censoring. Therefore, the observable data consists of $t_i = \min(T_i, C_i)$ and $\Delta_i = I(t_i = T_i)$. The estimating equation is

$$\sum_{i=1}^{n} \left\{ \frac{\Delta_i}{\lambda} - t_i \right\} = 0$$

Here, θ is a single parameter that corresponds to the scale parameter for the exponential distribution. The hazard from the exponential model is parameterized as the following

$$h(t) = \lambda$$

Parameters

- **theta** (*ndarray, list, vector*) – Theta in the case of the exponential model consists of a single value. Furthermore, the parameter will be non-negative. Therefore, an initial value like the `[1, ]` should be provided.
- **t** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed times.
- **delta** (*ndarray, list, vector*) – 1-dimensional vector of *n* event indicators, where 1 indicates an event and 0 indicates right censoring.

Returns Returns a 1-by-*n* NumPy array evaluated for the input `theta`

Return type array

Examples

Construction of a estimating equation(s) with `ee_exponential_model` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_exponential_model
```

Some generic survival data to estimate an exponential survival model

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['C'] = np.random.weibull(a=1, size=n)
>>> data['C'] = np.where(data['C'] > 5, 5, data['C'])
>>> data['T'] = 0.8*np.random.weibull(a=1.0, size=n)
>>> data['delta'] = np.where(data['T'] < data['C'], 1, 0)
>>> data['t'] = np.where(data['delta'] == 1, data['T'], data['C'])
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>         return ee_exponential_model(theta=theta,
>>>                                     t=data['t'], delta=data['delta'])
```

Calling the M-estimator

```
>>> estr = MEstimator(stacked_equations=psi, init=[1.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting parameter the specific parameter estimates

```
>>> estr.theta[0]      # lambda (scale)
```

References

Collett D. (2015). Modelling survival data in medical research. CRC press.

delicatessen.estimating_equations.survival.ee_exponential_measure

ee_exponential_measure(*theta*, *times*, *n*, *measure*, *scale*)

Estimating equation to calculate a survival measure (survival, density, risk, hazard, cumulative hazard) from the exponential model. The estimating equation for the survival function at time t is

$$\sum_{i=1}^{n} \{\exp(-\lambda t) - \theta\} = 0$$

and the estimating equation for the hazard function at time t is

$$\sum_{i=1}^{n} \{\lambda - \theta\} = 0$$

For the other measures, we take advantage of the following transformations

$$\begin{aligned} F(t) &= 1 - S(t) \\ H(t) &= -\log(S(t)) \\ f(t) &= h(t)S(t) \end{aligned}$$

Note: For proper uncertainty estimation, this estimating equation is meant to be stacked with `ee_exponential_model`.

Parameters

- **theta** (*ndarray, list, vector*) – theta consists of t values. The initial values should consist of the same number of elements as provided in the `times` argument.
- **times** (*int, float, ndarray, list, vector*) – A single time or 1-dimensional collection of times to calculate the measure at. The number of provided times should consist of the same number of elements as provided in the `theta` argument.
- **n** (*int*) – Number of observations in the input data. This argument ensures that the dimensions of the estimating equation are correct given the number of observations in the data.

- **measure** (*str*) – Measure to calculate. Options include survival (`'survival'`), density (`'density'`), risk or the cumulative density (`'risk'`), hazard (`'hazard'`), or cumulative hazard (`'cumulative_hazard'`).
- **scale** (*float, int*) – The estimated scale parameter from the Weibull model. From `ee_weibull_model`, will be the first element.

Returns Returns a t-by-n NumPy array evaluated for the input `theta`

Return type array

Examples

Construction of a estimating equation(s) with `ee_exponential_model` and `ee_exponential_measure` should be done similar to the following. First, we will estimate the survival at time 5.

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_exponential_model, ee_
↪exponential_measure
```

Some generic survival data to estimate an exponential model

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['C'] = np.random.weibull(a=1, size=n)
>>> data['C'] = np.where(data['C'] > 5, 5, data['C'])
>>> data['T'] = 0.8*np.random.weibull(a=1.0, size=n)
>>> data['delta'] = np.where(data['T'] < data['C'], 1, 0)
>>> data['t'] = np.where(data['delta'] == 1, data['T'], data['C'])
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     exp = ee_exponential_model(theta=theta[0], t=data['t'],
>>>                                delta=data['delta'])
>>>     pred_surv_t = ee_exponential_measure(theta=theta[1], n=data.shape[0],
>>>                                          times=5, measure='survival',
>>>                                          scale=theta[0])
>>>     return np.vstack((exp, pred_surv_t))
```

Calling the M-estimator (note that *init* has 2 value, one for the scale and the other for $S(t = 5)$).

```
>>> estr = MEstimator(stacked_equations=psi, init=[1., 0.5])
>>> estr.estimate(solver='lm')
```

Inspecting the estimate, variance, and confidence intervals for $S(t = 5)$

```
>>> estr.theta[-1]                      # \hat{S}(t)
>>> estr.variance[-1, -1]               # \hat{Var}(\hat{S}(t))
>>> estr.confidence_intervals()[-1, :]  # 95% CI for S(t)
```

Next, we will consider evaluating the survival function at multiple time points (so we can easily create a plot of the survival function and the corresponding confidence intervals)

Note: When calculate the survival (or other measures) at many time points, it is generally best to pre-wash the coefficients to reduce the number of iterations and total run-time.

To make everything easier, we will generate a list of uniformly spaced values between the start and end points of our desired survival function. We will also generate initial values of the same length (to help the optimizer, we also start our starting values from near one and end near zero).

```
>>> resolution = 50
>>> time_spacing = list(np.linspace(0.01, 5, resolution))
>>> fast_inits = list(np.linspace(0.99, 0.01, resolution))
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     exp = ee_exponential_model(theta=theta[0], t=data['t'],
>>>                                delta=data['delta'])
>>>     pred_surv_t = ee_exponential_measure(theta=theta[1:], n=data.shape[0],
>>>                                          times=time_spacing, measure='survival',
>>>                                          scale=theta[0])
>>>     return np.vstack((exp, pred_surv_t))
```

Calling the M-estimator

```
>>> mestr = MEstimator(psi, init=list(estr.theta[0]) + fast_inits)
>>> mestr.estimate(solver="lm")
```

To plot the survival curves, we could do the following:

```
>>> import matplotlib.pyplot as plt
>>> ci = mestr.confidence_intervals()[1:, :]  # Extracting relevant CI
>>> plt.fill_between(time_spacing, ci[:, 0], ci[:, 1], alpha=0.2)
>>> plt.plot(time_spacing, mestr.theta[1:], '-')
>>> plt.show()
```

References

Collett D. (2015). Modelling survival data in medical research. CRC press.

delicatessen.estimating_equations.survival.ee_weibull_model

ee_weibull_model(*theta*, *t*, *delta*)

Estimating equation for a two-parameter Weibull model. Let T_i indicate the time of the event and C_i indicate the time to right censoring. Therefore, the observable data consists of $t_i = min(T_i, C_i)$ and $\Delta_i = I(t_i = T_i)$. The estimating equations are

$$\sum_{i=1}^{n} = \begin{bmatrix} \frac{\Delta_i}{\lambda} - t_i^\gamma \\ \frac{\Delta_i}{\gamma} + \Delta_i \log(t_i) - \lambda t_i^\gamma \log(t_i) \end{bmatrix} = 0$$

Here, θ consists of two parameters for the Weibull model: the scale (λ) and the shape (γ). The hazard from the Weibull model is parameterized as the following

$$h(t) = \lambda \gamma t^{\gamma - 1}$$

Parameters

- **theta** (*ndarray, list, vector*) – Theta in the case of the Weibull model consists of two values. Furthermore, the parameter will be non-negative. Therefore, an initial value like the `[1, ]` is recommended.
- **t** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed times. No missing data should be included (missing data may cause unexpected behavior).
- **delta** (*ndarray, list, vector*) – 1-dimensional vector of *n* event indicators, where 1 indicates an event and 0 indicates right censoring. No missing data should be included (missing data may cause unexpected behavior).

Returns Returns a 2-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_weibull_model` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_weibull_model
```

Some generic survival data to estimate a Weibull survival model

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['C'] = np.random.weibull(a=1, size=n)
>>> data['C'] = np.where(data['C'] > 5, 5, data['C'])
>>> data['T'] = 0.8*np.random.weibull(a=0.8, size=n)
>>> data['delta'] = np.where(data['T'] < data['C'], 1, 0)
>>> data['t'] = np.where(data['delta'] == 1, data['T'], data['C'])
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>         return ee_weibull_model(theta=theta,
>>>                                 t=data['t'], delta=data['delta'])
```

Calling the M-estimator

```
>>> estr = MEstimator(stacked_equations=psi, init=[1., 1.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting parameter the specific parameter estimates

```
>>> estr.theta[0]     # lambda (scale)
>>> estr.theta[1]     # gamma  (shape)
```

References

Collett D. (2015). Modelling survival data in medical research. CRC press.

delicatessen.estimating_equations.survival.ee_weibull_measure

ee_weibull_measure(*theta*, *times*, *n*, *measure*, *scale*, *shape*)

Estimating equation to calculate a survival measure (survival, density, risk, hazard, cumulative hazard) for the Weibull model. The estimating equation for the survival function at time t is

$$\sum_{i=1}^{n} \{\exp(-\lambda t^\gamma) - \theta\} = 0$$

and the estimating equation for the hazard function at time t is

$$\sum_{i=1}^{n} \left\{\lambda \gamma t^{\gamma - 1} - \theta\right\} = 0$$

For the other measures, we take advantage of the following transformation between survival measures

$$\begin{aligned} F(t) &= 1 - S(t) \\ H(t) &= -\log(S(t)) \\ f(t) &= h(t)S(t) \end{aligned}$$

Note: For proper uncertainty estimation, this estimating equation is meant to be stacked with `ee_weibull_model`.

Parameters

- **theta** (*ndarray, list, vector*) – theta consists of *t* values. The initial values should consist of the same number of elements as provided in the `times` argument.
- **times** (*int, float, ndarray, list, vector*) – A single time or 1-dimensional collection of times to calculate the measure at. The number of provided times should consist of the same number of elements as provided in the `theta` argument.
- **n** (*int*) – Number of observations in the input data. This argument ensures that the dimensions of the estimating equation are correct given the number of observations in the data.
- **measure** (*str*) – Measure to calculate. Options include survival (`'survival'`), density (`'density'`), risk or the cumulative density (`'risk'`), hazard (`'hazard'`), or cumulative hazard (`'cumulative_hazard'`).
- **scale** (*float, int*) – The estimated scale parameter from the Weibull model. From `ee_weibull_model`, will be the first element.
- **shape** – The estimated shape parameter from the Weibull model. From `ee_weibull_model`, will be the second (last) element.

Returns Returns a *t*-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_weibull_model` and `ee_weibull_measure` should be done similar to the following. First, we will estimate the survival at time 5.

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_weibull_model, ee_weibull_
↪measure
```

Some generic survival data to estimate a Weibull model

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['C'] = np.random.weibull(a=1, size=n)
>>> data['C'] = np.where(data['C'] > 5, 5, data['C'])
>>> data['T'] = 0.8*np.random.weibull(a=0.8, size=n)
>>> data['delta'] = np.where(data['T'] < data['C'], 1, 0)
>>> data['t'] = np.where(data['delta'] == 1, data['T'], data['C'])
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     exp = ee_weibull_model(theta=theta[0:2], t=data['t'],
>>>                            delta=data['delta'])
>>>     pred_surv_t = ee_weibull_measure(theta=theta[2], n=data.shape[0],
>>>                                      times=5, measure='survival',
>>>                                      scale=theta[0], shape=theta[1])
>>>     return np.vstack((exp, pred_surv_t))
```

Calling the M-estimator

```
>>> estr = MEstimator(stacked_equations=psi, init=[1., 1., 0.5])
>>> estr.estimate(solver='lm')
```

Inspecting the estimate, variance, and confidence intervals for $S(t = 5)$

```
>>> estr.theta[-1]                      # \hat{S}(t)
>>> estr.variance[-1, -1]               # \hat{Var}(\hat{S}(t))
>>> estr.confidence_intervals()[-1, :]  # 95% CI for S(t)
```

Next, we will consider evaluating the survival function at multiple time points (so we can easily create a plot of the survival function and the corresponding confidence intervals)

Note: When calculate the survival (or other measures) at many time points, it is generally best to pre-wash the coefficients to reduce the number of iterations and total run-time.

To make everything easier, we will generate a list of uniformly spaced values between the start and end points of our desired survival function. We will also generate initial values of the same length (to help the optimizer, we also start our starting values from near one and end near zero).

```
>>> resolution = 50
>>> time_spacing = list(np.linspace(0.01, 5, resolution))
>>> fast_inits = list(np.linspace(0.99, 0.01, resolution))
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     exp = ee_weibull_model(theta=theta[0:2], t=data['t'],
>>>                            delta=data['delta'])
>>>     pred_surv_t = ee_weibull_measure(theta=theta[2:], n=data.shape[0],
>>>                                      times=time_spacing, measure='survival',
>>>                                      scale=theta[0], shape=theta[1])
>>>     return np.vstack((exp, pred_surv_t))
```

Calling the M-estimator

```
>>> mestr = MEstimator(psi, init=list(estr.theta[0:2]) + fast_inits)
>>> mestr.estimate(solver="lm")
```

To plot the survival curves, we could do the following:

```
>>> import matplotlib.pyplot as plt
>>> ci = mestr.confidence_intervals()[2:, :]  # Extracting relevant CI
>>> plt.fill_between(time_spacing, ci[:, 0], ci[:, 1], alpha=0.2)
>>> plt.plot(time_spacing, mestr.theta[2:], '-')
>>> plt.show()
```

References

Collett D. (2015). Modelling survival data in medical research. CRC press.

delicatessen.estimating_equations.survival.ee_aft_weibull

ee_aft_weibull(*theta*, *X*, *t*, *delta*, *weights=None*)

Estimating equation for accelerated failure time (AFT) model with a Weibull distribution. Let T_i indicate the time of the event and C_i indicate the time to right censoring. Therefore, the observable data consists of $t_i = min(T_i, C_i)$ and $\Delta_i = I(t_i = T_i)$. The estimating equations are

$$\sum_{i=1}^{n} = \begin{bmatrix} \frac{\Delta_i}{\lambda} - t_i^{\gamma} \exp(\beta X_i) \\ \Delta_i X_i - (\lambda t_i^{\gamma} \exp(\beta X_i)) X_i \\ \frac{\Delta_i}{\gamma} + \Delta_i \log(t) - \lambda t_i^{\gamma} \exp(\beta X_i) \log(t) \end{bmatrix} = 0$$

The AFT consists of the following parameters: μ, β, σ. The above estimating equations use the proportional hazards form of the Weibull model. For the Weibull AFT, notice the following relation between the coefficients: $\lambda = -\mu\gamma$, $\beta_{PH} = -\beta_{AFT}\gamma$, and $\gamma = \exp(\sigma)$.

Here, θ is a 1-by-(2+`b`) array, where b is the distinct covariates included as part of `X`. For example, if `X` is a 3-by-n matrix, then theta will be a 1-by-5 array. The code is general to allow for an arbitrary dimension of `X`.

Parameters

- **theta** (*ndarray, list, vector*) – theta consists of 1+`b`+1 values. Therefore, initial values should consist of the same number as the number of columns present in `X` plus 2. This can easily be implemented via `[0, ] + [0, ] * X.shape[1] + [0, ]`.

- **X** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables.
- **t** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed times.
- **delta** (*ndarray, list, vector*) – 1-dimensional vector of *n* values indicating whether the time was an event or censoring.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of *n* weights. Default is `None`, which assigns a weight of 1 to all observations.

Returns Returns a 1+`b`+1-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_aft_weibull` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_aft_weibull
```

Some generic survival data to estimate a Weibull AFT regresion model

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['X'] = np.random.binomial(n=1, p=0.5, size=n)
>>> data['W'] = np.random.binomial(n=1, p=0.5, size=n)
>>> data['T'] = (1/1.25 + 1/np.exp(0.5)*data['X'])*np.random.weibull(a=0.75, size=n)
>>> data['C'] = np.random.weibull(a=1, size=n)
>>> data['C'] = np.where(data['C'] > 10, 10, data['C'])
>>> data['delta'] = np.where(data['T'] < data['C'], 1, 0)
>>> data['t'] = np.where(data['delta'] == 1, data['T'], data['C'])
>>> d_obs = data[['X', 'W', 't', 'delta']].copy()
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>         return ee_aft_weibull(theta=theta, X=d_obs[['X', 'W']],
>>>                               t=d_obs['t'], delta=d_obs['delta'])
```

Calling the M-estimator (note that *init* has 4 values now, since `X.shape[1]` is 2).

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0.])
>>> estr.estimate(solver='hybr')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting parameter the specific parameter estimates

```
>>> estr.theta[0]     # log(mu)    (scale)
>>> estr.theta[1:-1]  # log(beta)  (scale coefficients)
>>> estr.theta[-1]    # log(sigma) (shape)
```

References

Collett D. (2015). Parametric proportional hazards models In: Modelling survival data in medical research. CRC press. pg171-220

Collett D. (2015). Accelerated failure time and other parametric models. In: Modelling survival data in medical research. CRC press. pg171-220

delicatessen.estimating_equations.survival.ee_aft_weibull_measure

ee_aft_weibull_measure(*theta*, *times*, *X*, *measure*, *mu*, *beta*, *sigma*)

Estimating equation to calculate a survival measure (survival, density, risk, hazard, cumulative hazard) given a specific covariate pattern and Weibull accelerated failure time (AFT) model. The estimating equation for the survival function at time t is

$$\sum_{i=1}^{n} \{\exp(-1 \lambda_i t^{\gamma}) - \theta\} = 0$$

and the estimating equation for the hazard function at time t is

$$\sum_{i=1}^{n} \left\{\lambda_i \gamma t^{\gamma - 1} - \theta\right\} = 0$$

where

$$\gamma = \exp(\sigma)$$
$$\lambda_i = \exp(-1(\mu + X\beta) * \gamma)$$

For the other measures, we take advantage of the following transformation between survival meaures

$$F(t) = 1 - S(t)$$
$$H(t) = -\log(S(t))$$
$$f(t) = h(t)S(t)$$

Note: For proper uncertainty estimation, this estimating equation is meant to be stacked together with the corresponding Weibull AFT model.

Parameters

- **theta** (*ndarray, list, vector*) – theta consists of t values. The initial values should consist of the same number of elements as provided in the `times` argument.
- **times** (*int, float, ndarray, list, vector*) – A single time or 1-dimensional collection of times to calculate the measure at. The number of provided times should consist of the same number of elements as provided in the `theta` argument.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables.

- **measure** (*str*) – Measure to calculate. Options include survival (`'survival'`), density (`'density'`), risk or the cumulative density (`'risk'`), hazard (`'hazard'`), or cumulative hazard (`'cumulative_hazard'`).
- **mu** (*float*, *int*) – The estimated scale parameter from the Weibull AFT. From `ee_aft_weibull`, will be the first element.
- **beta** – The estimated scale coefficients from the Weibull AFT. From `ee_aft_weibull`, will be the middle element(s).
- **sigma** – The estimated shape parameter from the Weibull AFT. From `ee_aft_weibull`, will be the last element.

Returns Returns a t-by-n NumPy array evaluated for the input theta

Return type array

Examples

Construction of a estimating equations for $S(t = 5)$ with `ee_aft_weibull_measure` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_aft_weibull, ee_aft_weibull_
↪measure
```

For demonstration, we will generated generic survival data

```
>>> n = 100
>>> data = pd.DataFrame()
>>> data['X'] = np.random.binomial(n=1, p=0.5, size=n)
>>> data['W'] = np.random.binomial(n=1, p=0.5, size=n)
>>> data['T'] = (1/1.25 + 1/np.exp(0.5)*data['X'])*np.random.weibull(a=0.75, size=n)
>>> data['C'] = np.random.weibull(a=1, size=n)
>>> data['C'] = np.where(data['C'] > 10, 10, data['C'])
>>> data['delta'] = np.where(data['T'] < data['C'], 1, 0)
>>> data['t'] = np.where(data['delta'] == 1, data['T'], data['C'])
>>> d_obs = data[['X', 'W', 't', 'delta']].copy()
```

Our interest will be in the survival among those with $X = 1, W = 1$. Therefore, we will generate a copy of the data and set the values in that copy (to keep the dimension the same across both estimating equations).

```
>>> d_coef = d_obs.copy()
>>> d_coef['X'] = 1
>>> d_coef['W'] = 1
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     aft = ee_aft_weibull(theta=theta[0:4],
>>>                          t=d_obs['t'], delta=d_obs['delta'], X=d_obs[['X', 'W']])
>>>     pred_surv_t = ee_aft_weibull_measure(theta=theta[4], X=d_coef[['X', 'W']],
>>>                                          times=5, measure='survival',
```

(continues on next page)

(continued from previous page)

```
>>>                                                 mu=theta[0], beta=theta[1:3],␣
↪sigma=theta[3])
>>>     return np.vstack((aft, pred_surv_t))
```

Calling the M-estimator

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0., 0.5])
>>> estr.estimate(solver='lm')
```

Inspecting the estimate, variance, and confidence intervals for $S(t = 5)$

```
>>> estr.theta[-1]                          # \hat{S}(t)
>>> estr.variance[-1, -1]                   # \hat{Var}(\hat{S}(t))
>>> estr.confidence_intervals()[-1, :]  # 95% CI for S(t)
```

Next, we will consider evaluating the survival function at multiple time points (so we can easily create a plot of the survival function and the corresponding confidence intervals)

Note: When calculate the survival (or other measures) at many time points, it is generally best to pre-wash the coefficients to reduce the number of iterations and total run-time.

To make everything easier, we will generate a list of uniformly spaced values between the start and end points of our desired survival function. We will also generate initial values of the same length (to help the optimizer, we also start our starting values from near one and end near zero).

```
>>> resolution = 50
>>> time_spacing = list(np.linspace(0.01, 8, resolution))
>>> fast_inits = list(np.linspace(0.99, 0.01, resolution))
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     aft = ee_aft_weibull(theta=theta[0:4],
>>>                     t=d_obs['t'], delta=d_obs['delta'], X=d_obs[['X', 'W']])
>>>     pred_surv_t = ee_aft_weibull_measure(theta=theta[4:], X=d_coef[['X', 'W']],
>>>                                          times=time_spacing, measure='survival',
>>>                                          mu=theta[0], beta=theta[1:3],␣
↪sigma=theta[3])
>>>     return np.vstack((aft, pred_surv_t))
```

Calling the M-estimator

```
>>> estr = MEstimator(psi, init=list(estr.theta[0:4]) + fast_inits)
>>> estr.estimate(solver="lm")
```

To plot the survival curves, we could do the following:

```
>>> import matplotlib.pyplot as plt
>>> ci = estr.confidence_intervals()[4:, :]  # Extracting relevant CI
>>> plt.fill_between(time_spacing, ci[:, 0], ci[:, 1], alpha=0.2)
>>> plt.plot(time_spacing, estr.theta[4:], '-')
>>> plt.show()
```

References

Collett D. (2015). Accelerated failure time and other parametric models. In: Modelling survival data in medical research. CRC press. pg171-220

Dose Response

ee_4p_logistic(theta, X, y)	Estimating equations for the 4-parameter logistic model (4PL).
ee_3p_logistic(theta, X, y, lower)	Estimating equations for the 3-parameter logistic model (3PL).
ee_2p_logistic(theta, X, y, lower, upper)	Estimating equations for the 2-parameter logistic model (2PL).
ee_effective_dose_delta(theta, y, delta, ...)	Default stacked estimating equation to pair with the 4 parameter logistic model for estimation of the $delta$ effective dose.

delicatessen.estimating_equations.dose_response.ee_4p_logistic

ee_4p_logistic(*theta*, *X*, *y*)

Estimating equations for the 4-parameter logistic model (4PL). The estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} -2(Y_i - \hat{Y}_i)(1 - 1/(1+\rho)) \\ 2(Y_i - \hat{Y}_i)(\theta_3 - \theta_0)\frac{\theta_2}{\theta_1}\frac{\rho}{(1+\rho)^2} \\ 2(Y_i - \hat{Y}_i)(\theta_3 - \theta_0)\log(D_i/\theta_1)\frac{\rho}{(1+\rho)^2} \\ 2(Y_i - \hat{Y}_i)(1/(1+\rho))) \end{bmatrix} = 0$$

where R_i is the response of individual i, D_i is the dose, $\rho = \frac{D_i}{\theta_1}^{\theta_2}$, and $\hat{Y}_i = \theta_0 + \frac{\theta_3 - \theta_0}{1+\rho}$.

Here, theta is a 1-by-4 array. The first theta corresponds to lower limit (θ_0), the second corresponds to the effective dose (ED50) (θ_1), the third corresponds to the steepness of the curve (θ_2), and the fourth corresponds to the upper limit (θ_3).

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of 4 values. In general, starting values > 0 are better choices for the 4PL model
- **X** (*ndarray, list, vector*) – 1-dimensional vector of *n* dose values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* response values.

Returns Returns a 4-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_4p_logistic` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.data import load_inderjit
>>> from delicatessen.estimating_equations import ee_4p_logistic
```

For demonstration, we use dose-response data from Inderjit et al. (2002), which can be loaded from `delicatessen` directly.

```
>>> d = load_inderjit()   # Loading array of data
>>> dose_data = d[:, 1]   # Dose data
>>> resp_data = d[:, 0]   # Response data
```

Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_4p_logistic(theta=theta, X=dose_data, y=resp_data)
```

The 4PL model and others are harder to solve compared to other estimating equations. Namely, the root-finder is not aware of implicit bounds on the parameters. To reduce non-convergence issues, we can give the root-finder good starting values.

For the 4PL, the upper limit should *always* be greater than the lower limit. Second, the ED50 should be between the lower and upper limits. Third, the sign for the steepness depends on whether the response declines (positive) or the response increases (negative). Finally, some solvers may be better suited to the problem, so try a few different options.

Here, we use some general starting values that should perform well in many cases. For the lower-bound, give the minimum response value as the initial. For ED50, give the mid-point between the maximum response and the minimum response. The initial value for steepness is more difficult. Ideally, we would give a starting value of zero, but that will fail in this example. Giving a small positive starting value works in this example. For the upper-bound, give the maximum response value as the initial. Finally, we use the `lm` solver.

Note: To summarize the recommendations, be sure to examine your data (e.g., scatterplot). This will help to determine the initial starting values for the root-finding procedure. Otherwise, you may come across a convergence error.

```
>>> estr = MEstimator(psi, init=[np.min(resp_data),
>>>                              (np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              (np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              np.max(resp_data)])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting the parameter estimates

```
>>> estr.theta[0]    # lower limit
>>> estr.theta[1]    # ED(50)
>>> estr.theta[2]    # steepness
>>> estr.theta[3]    # upper limit
```

References

Ritz C, Baty F, Streibig JC, & Gerhard D. (2015). Dose-response analysis using R. *PloS One*, 10(12), e0146021.

An H, Justin TL, Aubrey GB, Marron JS, & Dittmer DP. (2019). dr4pl: A Stable Convergence Algorithm for the 4 Parameter Logistic Model. *R J.*, 11(2), 171.

Inderjit, Streibig JC, & Olofsdotter M. (2002). Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiologia Plantarum*, 114(3), 422-428.

delicatessen.estimating_equations.dose_response.ee_3p_logistic

ee_3p_logistic(*theta*, *X*, *y*, *lower*)

Estimating equations for the 3-parameter logistic model (3PL). The estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} -2(Y_i - \hat{Y}_i)(1 - 1/(1+\rho)) \\ 2(Y_i - \hat{Y}_i)(\theta_3 - \theta_0)\frac{\theta_2}{\theta_1}\frac{\rho}{(1+\rho)^2} \\ 2(Y_i - \hat{Y}_i)(\theta_3 - \theta_0)\log(D_i/\theta_1)\frac{\rho}{(1+\rho)^2} \end{bmatrix} = 0$$

where R_i is the response of individual i, D_i is the dose, $\rho = \frac{D_i}{\theta_1}^{\theta_2}$, and $\hat{Y}_i = \theta_0 + \frac{\theta_3 - \theta_0}{1+\rho}$.

Here, theta is a 1-by-3 array for the 3PL. The first theta corresponds to the effective dose (ED50) (θ_1), the second corresponds to the steepness of the curve (θ_2), and the third corresponds to the upper limit (θ_3). The lower limit (θ_0, `lower`) is pre-specified by the user (and is no longer estimated)

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of 3 values. In general, starting values > 0 are better choices for the 3PL model
- **X** (*ndarray, list, vector*) – 1-dimensional vector of *n* dose values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* response values.
- **lower** (*int, float*) – Set value for the lower limit.

Returns Returns a 3-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_3p_logistic` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.data import load_inderjit
>>> from delicatessen.estimating_equations import ee_3p_logistic
```

For demonstration, we use dose-response data from Inderjit et al. (2002), which can be loaded from `delicatessen` directly.

```
>>> d = load_inderjit()   # Loading array of data
>>> dose_data = d[:, 1]   # Dose data
>>> resp_data = d[:, 0]   # Response data
```

Since there is a natural lower-bound of 0 for root growth, we set `lower=0`. Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_3p_logistic(theta=theta, X=dose_data, y=resp_data,
>>>                           lower=0)
```

The 3PL model and others are harder to solve compared to other estimating equations. See the advice provided in the `ee_4p_logistic` documentation.

```
>>> estr = MEstimator(psi, init=[(np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              (np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              np.max(resp_data)])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting the parameter estimates

```
>>> estr.theta[0]    # ED(50)
>>> estr.theta[1]    # steepness
>>> estr.theta[2]    # upper limit
```

References

Ritz C, Baty F, Streibig JC, & Gerhard D. (2015). Dose-response analysis using R. *PloS One*, 10(12), e0146021.

An H, Justin TL, Aubrey GB, Marron JS, & Dittmer DP. (2019). dr4pl: A Stable Convergence Algorithm for the 4 Parameter Logistic Model. *R J.*, 11(2), 171.

Inderjit, Streibig JC, & Olofsdotter M. (2002). Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiologia Plantarum*, 114(3), 422-428.

delicatessen.estimating_equations.dose_response.ee_2p_logistic

ee_2p_logistic(*theta*, *X*, *y*, *lower*, *upper*)

Estimating equations for the 2-parameter logistic model (2PL). The estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} 2(Y_i - \hat{Y}_i)(\theta_3 - \theta_0)\frac{\theta_2}{\theta_1}\frac{\rho}{(1+\rho)^2} \\ 2(Y_i - \hat{Y}_i)(\theta_3 - \theta_0)\log(D_i/\theta_1)\frac{\rho}{(1+\rho)^2} \end{bmatrix} = 0$$

where R_i is the response of individual i, D_i is the dose, $\rho = \frac{D_i}{\theta_1}^{\theta_2}$, and $\hat{Y}_i = \theta_0 + \frac{\theta_3 - \theta_0}{1+\rho}$.

Here, theta is a 1-by-2 array for the 2PL. The first theta corresponds to the effective dose (ED50) (θ_1), and the second corresponds to the steepness of the curve (θ_2). The lower limit (θ_0, `lower`) and upper limit (θ_3, `upper`) are pre-specified by the user (and are no longer estimated)

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of 2 values. In general, starting values > 0 are better choices for the 2PL model
- **X** (*ndarray, list, vector*) – 1-dimensional vector of n dose values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of n response values.
- **lower** (*int, float*) – Set value for the lower limit.
- **upper** (*int, float*) – Set value for the upper limit.

Returns Returns a 2-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_2p_logistic` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.data import load_inderjit
>>> from delicatessen.estimating_equations import ee_2p_logistic
```

For demonstration, we use dose-response data from Inderjit et al. (2002), which can be loaded from `delicatessen` directly.

```
>>> d = load_inderjit()    # Loading array of data
>>> dose_data = d[:, 1]    # Dose data
>>> resp_data = d[:, 0]    # Response data
```

Since there is a natural lower-bound of 0 for root growth, we set `lower=0`. While a natural upper bound does not exist for this example, we set `upper=8` for illustrative purposes. Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     return ee_2p_logistic(theta=theta, X=dose_data, y=resp_data,
>>>                           lower=0, upper=8)
```

The 2PL model and others are harder to solve compared to other estimating equations. See the advice provided in the `ee_4p_logistic` documentation.

```
>>> estr = MEstimator(psi, init=[(np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              (np.max(resp_data)+np.min(resp_data)) / 2])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting the parameter estimates

```
>>> estr.theta[0]    # ED(50)
>>> estr.theta[1]    # steepness
```

References

Ritz C, Baty F, Streibig JC, & Gerhard D. (2015). Dose-response analysis using R. *PloS One*, 10(12), e0146021.

An H, Justin TL, Aubrey GB, Marron JS, & Dittmer DP. (2019). dr4pl: A Stable Convergence Algorithm for the 4 Parameter Logistic Model. *R J.*, 11(2), 171.

Inderjit, Streibig JC, & Olofsdotter M. (2002). Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiologia Plantarum*, 114(3), 422-428.

delicatessen.estimating_equations.dose_response.ee_effective_dose_delta

ee_effective_dose_delta(*theta*, *y*, *delta*, *steepness*, *ed50*, *lower*, *upper*)

Default stacked estimating equation to pair with the 4 parameter logistic model for estimation of the $delta$ effective dose. The estimating equation is

$$\sum_{i=1}^{n} \left\{ \theta_1 + \frac{\theta_4 - \theta_1}{1 + (\theta_5 / \theta_2)^{\theta_3}} - \theta_4 (1 - \delta) - \theta_1 \delta \right\} = 0$$

where θ_5 is the $ED(\delta)$, and the other θ are from a PL model (1: lower limit, 2: steepness, 3: ED(50), 4: upper limit). For proper uncertainty estimation, this estimating equation should be stacked with the correspond PL model.

Parameters

- **theta** (*int, float*) – Theta value corresponding to the ED(alpha).
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* response values, used to construct correct shape for output.
- **delta** (*float*) – The effective dose level of interest, ED(alpha).
- **steepness** (*float*) – Estimated parameter for the steepness from the PL.
- **ed50** (*float*) – Estimated parameter for the ED50, or ED(alpha=50), from the PL.
- **lower** (*int, float*) – Estimated parameter or pre-specified constant for the lower limit. This should be a pre-specified constant for both the 3PL and 2PL.
- **upper** (*int, float*) – Estimated parameter or pre-specified constant for the lower limit. This should be a pre-specified constant for the 2PL.

Returns Returns a 1-by-*n* NumPy array evaluated for the input theta

Return type array

Examples

Construction of a estimating equations for ED25 with `ee_3p_logistic` should be done similar to the following

```
>>> from delicatessen import MEstimator
>>> from delicatessen.data import load_inderjit
>>> from delicatessen.estimating_equations import ee_2p_logistic, ee_effective_dose_
↪delta
```

For demonstration, we use dose-response data from Inderjit et al. (2002), which can be loaded from `delicatessen` directly.

```
>>> d = load_inderjit()   # Loading array of data
>>> dose_data = d[:, 1]   # Dose data
>>> resp_data = d[:, 0]   # Response data
```

Since there is a natural lower-bound of 0 for root growth, we set `lower=0`. While a natural upper bound does not exist for this example, we set `upper=8` for illustrative purposes. Defining psi, or the stacked estimating equations

```
>>> def psi(theta):
>>>     pl_model = ee_3p_logistic(theta=theta, X=dose_data, y=resp_data,
>>>                               lower=0)
>>>     ed_25 = ee_effective_dose_delta(theta[3], y=resp_data, delta=0.20,
>>>                                     steepness=theta[0], ed50=theta[1],
>>>                                     lower=0, upper=theta[2])
>>>     # Returning stacked estimating equations
>>>     return np.vstack((pl_model,
>>>                       ed_25,))
```

Notice that the estimating equations are stacked in the order of the parameters in `theta` (the first 3 belong to 3PL and the last belong to ED(25)).

```
>>> estr = MEstimator(psi, init=[(np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              (np.max(resp_data)+np.min(resp_data)) / 2,
>>>                              np.max(resp_data),
>>>                              (np.max(resp_data)+np.min(resp_data)) / 2])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

Inspecting the parameter estimates

```
>>> estr.theta[0]    # ED(50)
>>> estr.theta[1]    # steepness
>>> estr.theta[2]    # upper limit
>>> estr.theta[3]    # ED(25)
```

References

Ritz C, Baty F, Streibig JC, & Gerhard D. (2015). Dose-response analysis using R. *PloS One*, 10(12), e0146021.

An H, Justin TL, Aubrey GB, Marron JS, & Dittmer DP. (2019). dr4pl: A Stable Convergence Algorithm for the 4 Parameter Logistic Model. *R J.*, 11(2), 171.

Inderjit, Streibig JC, & Olofsdotter M. (2002). Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiologia Plantarum*, 114(3), 422-428.

Causal Inference

ee_gformula(theta, y, X, X1[, X0, ...])	Estimating equations for the g-formula (or g-computation).
ee_ipw(theta, y, A, W[, truncate, weights])	Estimating equation for inverse probability weighting (IPW) estimator.
ee_ipw_msm(theta, y, A, W, V, distribution, link)	Estimating equation for parameters of a marginal structural model estimated using inverse probability weighting.
ee_aipw(theta, y, A, W, X, X1, X0[, ...])	Estimating equation for augmented inverse probability weighting (AIPW) estimator.
ee_gestimation_snmm(theta, y, A, W, V[, X, ...])	Estimating equations for g-estimation of structural mean models (SMMs).
ee_mean_sensitivity_analysis(theta, y, ...)	Estimating equation for weighted sensitivity analysis estimator of the mean.

delicatessen.estimating_equations.causal.ee_gformula

ee_gformula(*theta*, *y*, *X*, *X1*, *X0=None*, *force_continuous=False*)

Estimating equations for the g-formula (or g-computation). The parameter of interest can either be the mean under a single policy or plan of action, or the mean difference between two policies. This is accomplished by providing the estimating equation the observed data (`X`, `y`), and the same data under the actions (`X1` and optionally `X0`).

The stack of estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} \left\{ g({X_i^*}^T \beta) - \theta_1 \right\} \\ \left\{ Y_i - g(X_i^T \beta) \right\} X_i \end{bmatrix} = 0$$

where the first is the mean under the specified plan, with the plan setting the values of action A (e.g., exposure, treatment, vaccination, etc.), and the second equation is the outcome regression model. Here, g indicates a transformation function. For linear regression, g is the identity function. Logistic regression uses the inverse-logit function. By default, `ee_gformula` detects whether *y* is all binary (zero or one), and applies logistic regression if that is evaluated to be true.

Note: This variation includes 1+`b` parameters, where the first parameter is the causal mean, and the remainder are the parameters for the regression model.

Alternatively, a causal mean difference is estimated when `X0` is specified. A common example of this would be the average causal effect, where the plans are all-action-one versus all-action-zero. Therefore, the estimating equations consist of the following three equations

$$\sum_{i=1}^{n} \begin{bmatrix} (\theta_1 - \theta_2) - \theta_0 \\ g({X_i^1}^T \beta) - \theta_1 \\ g({X_i^0}^T \beta) - \theta_2 \\ \left\{ Y_i - g(X_i^T \beta) \right\} X_i \end{bmatrix} = 0$$

Note: This variation includes 3+`b` parameters, where the first parameter is the causal mean difference, the second is the causal mean under plan 1, the third is the causal mean under plan 0, and the remainder are the parameters for the regression model.

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of 1+`b` values if `X0` is `None`, and 3+`b` values if `X0` is not `None`.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables.
- **X1** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables under the action plan.
- **X0** (*ndarray, list, vector, None, optional*) – 2-dimensional vector of *n* observed values for *b* variables under the separate action plan. This second argument is optional and should be specified if the causal mean difference between two action plans is of interest.
- **force_continuous** (*bool, optional*) – Option to force the use of linear regression despite detection of a binary variable.

Returns Returns a (1+`b`)-by-*n* NumPy array if `X0=None`, or returns a (3+`b`)-by-*n* NumPy array if `X0!=None`

Return type array

Examples

Construction of a estimating equation(s) with `ee_gformula` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_gformula
```

Some generic confounded data

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['W'] = np.random.binomial(1, p=0.5, size=n)
>>> d['A'] = np.random.binomial(1, p=(0.25 + 0.5*d['W']), size=n)
>>> d['Ya0'] = np.random.binomial(1, p=(0.75 - 0.5*d['W']), size=n)
>>> d['Ya1'] = np.random.binomial(1, p=(0.75 - 0.5*d['W'] - 0.1*1), size=n)
>>> d['Y'] = (1-d['A'])*d['Ya0'] + d['A']*d['Ya1']
>>> d['C'] = 1
```

In the first example, we will estimate the causal mean had everyone been set to `A=1`. Therefore, the optional argument `X0` is left as `None`. Before creating the estimating equation, we need to do some data prep. First, we will create an interaction term between `A` and `W` in the original data. Then we will generate a copy of the data and update the values of `A` to be all `1`.

```
>>> d['AW'] = d['A']*d['W']
>>> d1 = d.copy()
>>> d1['A'] = 1
>>> d1['AW'] = d1['A']*d1['W']
```

Having setup our data, we can now define the psi function.

```
>>> def psi(theta):
>>>     return ee_gformula(theta,
>>>                        y=d['Y'],
>>>                        X=d[['C', 'A', 'W', 'AW']],
>>>                        X1=d1[['C', 'A', 'W', 'AW']])
```

Notice that y corresponds to the observed outcomes, X corresponds to the observed covariate data, and X1 corresponds to the covariate data *under the action plan*.

Now we can call the M-Estimator. Since we are estimating the causal mean, and the regression parameters, the length of the initial values needs to correspond with this. Our linear regression model consists of 4 coefficients, so we need 1+4=5 initial values. When the outcome is binary (like it is in this example), we can be nice to the optimizer and give it a starting value of 0.5 for the causal mean (since 0.5 is in the middle of that distribution).

```
>>> estr = MEstimator(psi, init=[0.5, 0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

The causal mean is

```
>>> estr.theta[0]
```

Continuing from the previous example, let's say we wanted to estimate the average causal effect. Therefore, we want to contrast two plans (all A=1 versus all A=0). As before, we need to create the reference data for X0

```
>>> d0 = d.copy()
>>> d0['A'] = 0
>>> d0['AW'] = d0['A']*d0['W']
```

Having setup our data, we can now define the psi function.

```
>>> def psi(theta):
>>>     return ee_gformula(theta,
>>>                        y=d['Y'],
>>>                        X=d[['C', 'A', 'W', 'AW']],
>>>                        X1=d1[['C', 'A', 'W', 'AW']],
>>>                        X0=d0[['C', 'A', 'W', 'AW']], )
```

Notice that y corresponds to the observed outcomes, X corresponds to the observed covariate data, X1 corresponds to the covariate data under A=1, and X0 corresponds to the covariate data under A=0. Here, we need 3+4=7 starting values, since there are two additional parameters from the previous example. For the difference, a starting value of 0 is generally a good choice. Since Y is binary, we again provide 0.5 as starting values for the causal means

```
>>> estr = MEstimator(psi, init=[0., 0.5, 0.5, 0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates

```
>>> estr.theta[0]    # causal mean difference of 1 versus 0
>>> estr.theta[1]    # causal mean under X1
```

(continues on next page)

(continued from previous page)

```
>>> estr.theta[2]     # causal mean under X0
>>> estr.theta[3:]    # logistic regression coefficients
```

References

Snowden JM, Rose S, & Mortimer KM. (2011). Implementation of G-computation on a simulated data set: demonstration of a causal inference technique. *American Journal of Epidemiology*, 173(7), 731-738.

Hernán MA, & Robins JM. (2006). Estimating causal effects from epidemiological data. *Journal of Epidemiology & Community Health*, 60(7), 578-586.

delicatessen.estimating_equations.causal.ee_ipw

ee_ipw(*theta*, *y*, *A*, *W*, *truncate=None*, *weights=None*)

Estimating equation for inverse probability weighting (IPW) estimator. The average causal effect is estimated by this implementation of the IPW estimator. For estimation of the propensity scores, a logistic model is used.

The stacked estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} (\theta_1 - \theta_2) - \theta_0 \\ \frac{A_i Y_i}{\pi_i} - \theta_1 - \theta_1 \\ \frac{(1-A_i) Y_i}{1-\pi_i} - \theta_2 \\ \left\{ A_i - \text{expit}(W_i^T \alpha) \right\} W_i \end{bmatrix} = 0$$

where A is the action, math:W is the set of confounders, and $\pi_i = expit(W_i^T \alpha)$. The first estimating equation is for the average causal effect, the second is for the mean under $A := 1$, the third is for the mean under $A := 0$, and the last is the logistic regression model for the propensity scores. Here, the length of the theta vector is 3+`b`, where *b* is the number of parameters in the regression model.

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of 3+`b` values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values.
- **A** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values. The A values should all be 0 or 1.
- **W** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables to model the probability of `A` with.
- **truncate** (*None, list, set, ndarray, optional*) – Bounds to truncate the estimated probabilities of `A` at. For example, estimated probabilities above 0.99 or below 0.01 can be set to 0.99 or 0.01, respectively. This is done by specifying `truncate=(0.01, 0.99)`. Note this step is done via `numpy.clip(.., a_min, a_max)`, so order is important. Default is `None`, which applies no truncation.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations. This argument is intended to support the use of missingness weights. The propensity score model is *not* fit using these weights.

Returns Returns a (3+`b`)-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_ipw` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_ipw
```

Some generic data

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['W'] = np.random.binomial(1, p=0.5, size=n)
>>> d['A'] = np.random.binomial(1, p=(0.25 + 0.5*d['W']), size=n)
>>> d['Ya0'] = np.random.binomial(1, p=(0.75 - 0.5*d['W']), size=n)
>>> d['Ya1'] = np.random.binomial(1, p=(0.75 - 0.5*d['W'] - 0.1*1), size=n)
>>> d['Y'] = (1-d['A'])*d['Ya0'] + d['A']*d['Ya1']
>>> d['C'] = 1
```

Defining psi, or the stacked estimating equations. Note that `'A'` is the action.

```
>>> def psi(theta):
>>>     return ee_ipw(theta, y=d['Y'], A=d['A'],
>>>                   W=d[['C', 'W']])
```

Calling the M-estimation procedure. Since `W` is 2-by-n here and IPW has 3 additional parameters, the initial values should be of length 3+2=5. In general, it will be best to start with [0., 0.5, 0.5, . . .] as the initials when `Y` is binary. Otherwise, starting with all 0. as initials is reasonable.

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0.5, 0.5, 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

More specifically, the corresponding parameters are

```
>>> estr.theta[0]    # causal mean difference of 1 versus 0
>>> estr.theta[1]    # causal mean under A=1
>>> estr.theta[2]    # causal mean under A=0
>>> estr.theta[3:]   # logistic regression coefficients
```

If you want to see how truncating the probabilities works, try repeating the above code but specifying `truncate=(0.1, 0.9)` as an optional argument in `ee_ipw`.

References

Hernán MA, & Robins JM. (2006). Estimating causal effects from epidemiological data. *Journal of Epidemiology & Community Health*, 60(7), 578-586.

Cole SR, & Hernán MA. (2008). Constructing inverse probability weights for marginal structural models. *American Journal of Epidemiology*, 168(6), 656-664.

delicatessen.estimating_equations.causal.ee_ipw_msm

ee_ipw_msm(*theta*, *y*, *A*, *W*, *V*, *distribution*, *link*, *hyperparameter=None*, *truncate=None*, *weights=None*)

Estimating equation for parameters of a marginal structural model estimated using inverse probability weighting. For estimation of the propensity scores, a logistic model is used.

The stacked estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} \frac{1}{\pi_i} \left\{ Y_i - g^{-1}(X_i^T \beta) \right\} \times \frac{D(\beta)}{v(\beta)} X_i \\ \left\{ A_i - \text{expit}(W_i^T \alpha) \right\} W_i \end{bmatrix} = 0$$

where A is the action, math:W is the set of confounders, and $\pi_i = \text{expit}(W_i^T \alpha)$. Here, X is the design matrix for the marginal structural model (it includes A, and possibly some covariates from W). The first estimating equation is a weighted generalized linear model is used. See `ee_glm` for details on this estimating equation. The second estimating equation is the logistic model for the propensity scores.

Here, `theta` corresponds to multiple quantities. The *first* set of values correspond to the parameters of the marginal structural model, and the *second* set correspond to the logistic regression model coefficients for the propensity scores.

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of *c`+`b* values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values.
- **A** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values. The A values should all be 0 or 1.
- **W** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables to model the probability of `A` with.
- **V** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *c* variables in the marginal structural model.
- **distribution** (*str*) – Distribution for the generalized linear model. See `ee_glm` for options.
- **link** (*str*) – Link function for the generalized linear model. See `ee_glm` for options.
- **truncate** (*None, list, set, ndarray, optional*) – Bounds to truncate the estimated probabilities of `A` at. For example, estimated probabilities above 0.99 or below 0.01 can be set to 0.99 or 0.01, respectively. This is done by specifying `truncate=(0.01, 0.99)`. Note this step is done via `numpy.clip(.., a_min, a_max)`, so order is important. Default is `None`, which applies no truncation.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is None, which assigns a weight of 1 to all observations. This argument is intended to support the use of missingness weights. The propensity score model is *not* fit using these weights.

Returns Returns a (*c`+`b*)-by-*n* NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_ipw_msm` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_ipw_msm
```

Some generic data

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['W'] = np.random.binomial(1, p=0.5, size=n)
>>> d['A'] = np.random.binomial(1, p=(0.25 + 0.5*d['W']), size=n)
>>> d['Ya0'] = np.random.binomial(1, p=(0.75 - 0.5*d['W']), size=n)
>>> d['Ya1'] = np.random.binomial(1, p=(0.75 - 0.5*d['W'] - 0.1*1), size=n)
>>> d['Y'] = (1-d['A'])*d['Ya0'] + d['A']*d['Ya1']
>>> d['C'] = 1
```

Defining psi, or the stacked estimating equations for a logistic marginal structural model. Note that 'A' is the action.

```
>>> def psi(theta):
>>>     return ee_ipw_msm(theta, y=d['Y'], A=d['A'],
>>>                       W=d[['C', 'W']], V=d[['C', 'A']],
>>>                       link='logit', distribution='binomial')
```

Calling the M-estimation procedure. Since `W` is 2-by-n here and the marginal structural model (`V`) consists of 2 parameters, the initial values should be of length 2+2=4. Starting values for the marginal structural model may need to be adjusted to lie within the domain of the chosen link-distribution functions.

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

More specifically, the corresponding parameters are

```
>>> estr.theta[0:2]   # Marginal structural model paramters
>>> estr.theta[2:]    # Nuisance model parameters
```

If you want to see how truncating the probabilities works, try repeating the above code but specifying `truncate=(0.1, 0.9)` as an optional argument in `ee_ipw_msm`.

References

Hernán MA, & Robins JM. (2006). Estimating causal effects from epidemiological data. *Journal of Epidemiology & Community Health*, 60(7), 578-586.

Cole SR, & Hernán MA. (2008). Constructing inverse probability weights for marginal structural models. *American Journal of Epidemiology*, 168(6), 656-664.

delicatessen.estimating_equations.causal.ee_aipw

`ee_aipw`(*theta*, *y*, *A*, *W*, *X*, *X1*, *X0*, *truncate=None*, *force_continuous=False*)

Estimating equation for augmented inverse probability weighting (AIPW) estimator. AIPW consists of two nuisance models (the propensity score model and the outcome model).

The stacked estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} (\theta_1 - \theta_2) - \theta_0 \\ \frac{A_i Y_i}{\pi_i} - \frac{\hat{Y^1}(A_i - \pi_i}{\pi_i} - \theta_1 \\ \frac{(1-A_i) Y_i}{1-\pi_i} + \frac{\hat{Y^0}(A_i - \pi_i}{1-\pi_i} - \theta_2 \\ \left\{ A_i - \text{expit}(W_i^T \alpha) \right\} W_i \\ \left\{ Y_i - g(X_i^T \beta) \right\} X_i \end{bmatrix} = 0$$

where A is the action and W is the set of confounders, Y is the outcome, and $\pi_i = \text{expit}(W_i^T \alpha)$. The first estimating equation is for the average causal effect, the second is for the mean under $A := 1$, the third is for the mean under $A := 0$, the fourth is the logistic regression model for the propensity scores, and the last is for the outcome model. Here, the length of the theta vector is 3+`b`+`c`, where *b* is the number of parameters in the propensity score model and *c* is the number of parameters in the outcome model.

By default, *ee_aipw* detects whether *y* is all binary (zero or one), and applies logistic regression. Notice that `X` here should consists of both `A` and `W` (with possible interaction terms or other differences in functional forms from the propensity score model).

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of 3+`b`+`c` values.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values.
- **A** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values. The A values should all be 0 or 1.
- **W** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* variables to model the probability of `A` with.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *c* variables to model the outcome `y`.
- **X1** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *c* variables under the action plan where `A=1` for all units.
- **X0** (*ndarray, list, vector, None, optional*) – 2-dimensional vector of *n* observed values for *c* variables under the action plan where `A=0` for all units.
- **truncate** (*None, list, set, ndarray, optional*) – Bounds to truncate the estimated probabilities of `A` at. For example, estimated probabilities above 0.99 or below 0.01 can be set to 0.99 or 0.01, respectively. This is done by specifying `truncate=(0.01, 0.99)`. Note this step is done via `numpy.clip(.., a_min, a_max)`, so order is important. Default is `None`, which applies to no truncation.

- **force_continuous** (*bool, optional*) – Option to force the use of linear regression despite detection of a binary variable.

Returns Returns a (3+`b`+`c`)-by-*n* NumPy array evaluated for the input `theta`

Return type array

Examples

Construction of a estimating equation(s) with `ee_aipw` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_aipw
```

Some generic data

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['W'] = np.random.binomial(1, p=0.5, size=n)
>>> d['A'] = np.random.binomial(1, p=(0.25 + 0.5*d['W']), size=n)
>>> d['Ya0'] = np.random.binomial(1, p=(0.75 - 0.5*d['W']), size=n)
>>> d['Ya1'] = np.random.binomial(1, p=(0.75 - 0.5*d['W'] - 0.1*1), size=n)
>>> d['Y'] = (1-d['A'])*d['Ya0'] + d['A']*d['Ya1']
>>> d['C'] = 1
```

Defining psi, or the stacked estimating equations. Note that `A` is the action of interest. First, we will apply some necessary data processing. We will create an interaction term between `A` and `W` in the original data. Then we will generate a copy of the data and update the values of `A=1`, and then generate another copy but set `A=0` in that copy.

```
>>> d['AW'] = d['A']*d['W']
>>> d1 = d.copy()    # Copy where all A=1
>>> d1['A'] = 1
>>> d1['AW'] = d1['A']*d1['W']
>>> d0 = d.copy()    # Copy where all A=0
>>> d0['A'] = 0
>>> d0['AW'] = d0['A']*d0['W']
```

Having setup our data, we can now define the psi function.

```
>>> def psi(theta):
>>>     return ee_aipw(theta,
>>>                    y=d['Y'],
>>>                    A=d['A'],
>>>                    W=d[['C', 'W']],
>>>                    X=d[['C', 'A', 'W', 'AW']],
>>>                    X1=d1[['C', 'A', 'W', 'AW']],
>>>                    X0=d0[['C', 'A', 'W', 'AW']])
```

Calling the M-estimator. AIPW has 3 parameters with 2 coefficients in the propensity score model, and 4 coefficients in the outcome model, the total number of initial values should be 3+2+4=9. When Y is binary, it will be best to start with `[0., 0.5, 0.5, ...]` followed by all `0.` for the initial values. Otherwise, starting with all 0. as initials is reasonable.

```
>>> estr = MEstimator(psi,
>>>                   init=[0., 0.5, 0.5, 0., 0., 0., 0., 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

More specifically, the corresponding parameters are

```
>>> estr.theta[0]      # causal mean difference of 1 versus 0
>>> estr.theta[1]      # causal mean under A=1
>>> estr.theta[2]      # causal mean under A=0
>>> estr.theta[3:5]    # propensity score regression coefficients
>>> estr.theta[5:]     # outcome regression coefficients
```

References

Hernán MA, & Robins JM. (2006). Estimating causal effects from epidemiological data. *Journal of Epidemiology & Community Health*, 60(7), 578-586.

Funk MJ, Westreich D, Wiesen C, Stürmer T, Brookhart MA, & Davidian M. (2011). Doubly robust estimation of causal effects. *American Journal of Epidemiology*, 173(7), 761-767.

Tsiatis AA. (2006). Semiparametric theory and missing data. Springer, New York, NY.

delicatessen.estimating_equations.causal.ee_gestimation_snmm

ee_gestimation_snmm(*theta*, *y*, *A*, *W*, *V*, *X=None*, *model='linear'*, *weights=None*)

Estimating equations for g-estimation of structural mean models (SMMs). The parameter(s) of interest are the parameter(s) of the corresponding SMM. Rather than estimating the average causal effect, g-estimation of SMM estimates the average causal effect in the acted on within strata of a set of covariates, V. Options for SMM include the linear SMM and the log-linear SMM. The linear SMM is defined as

$$E[Y^a - Y^0 | A = a, V] = \beta_1 a + \beta_2 a V$$

This model corresponds to the average causal effect among those with $A = a$ by V. The log-linear SMM is defined as

$$\frac{E[Y^a | A = a, V]}{E[Y^0 | A = a, V]} = \exp(\beta_1 a + \beta_2 a V)$$

This model corresponds to the causal mean ratio among those with $A = a$ by V. Note that the log-linear SMM is only defined when $Y > 0$. The parameters of either SMM are identified under the assumptions of causal consistency, and exchangeability with positivity.

Two different estimating equations are available for g-estimation. The first set is referred to at the 'inefficient' g-estimator. For the inefficient g-estimator we solve for β in the following estimating equation

$$\sum_{i=1}^{n} \begin{bmatrix} \{H(\beta) \times (A - \pi_i)\} \times V_i \\ \left\{A_i - \text{expit}(W_i^T \alpha)\right\} W_i \end{bmatrix} = 0$$

where $\pi_i = \text{expit}(W_i^T \alpha)$, and $H(\beta) = Y - \beta A \mathbb{V}$ for a linear SMM and $H(\beta) = Y \times \exp(-A \beta \mathbb{V})$ for a log-linear SMM, where . Note that $V \subseteq W$, where W is the set of confounding variables. The length of the parameter vector is *b*`+`*c*, where *b* is the number of columns in `V`, and *c* is the number of columns in `W`.

The second implementation for g-estimation is the 'efficient' g-estimator. For the efficient g-estimator we replace $H(\beta)$ with $\{H(\beta) - E[H(\beta)|W]\}$ in the prior estimating equation and specify a model for $E[H(\beta)|W]$. The corresponding stacked estimating equations are

$$\sum_{i=1}^{n} \begin{bmatrix} \{(H(\beta) - g^{-1}(W_i^T \gamma)) \times (A - \pi_i)\} \times V_i \\ \{A_i - \text{expit}(W_i^T \alpha)\} W_i \\ \{H(\beta) - g^{-1}(W_i^T \gamma)\} W_i \end{bmatrix} = 0$$

where g^{-1} is the inverse transformation for the specified SMM. Therefore, there are b+c+d parameters for the efficient g-estimator, where *d* is the number of parameters in the model for $E[H(\beta)|W]$.

Parameters

- **theta** (*ndarray, list, vector*) – Theta consists of 1+`b` values if `X0` is `None`, and 3+b values if `X0` is not `None`.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values of the outcome.
- **A** (*ndarray, list, vector*) – 1-dimensional vector of *n* observed values of the action. The A values should all be 0 or 1.
- **W** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for b columns of a design matrix to model the expected value of `A`.
- **V** (*ndarray, list, vector*) – 2-dimensional vector of *n* observed values for *b* columns of a design matrix for the structural mean model. Note that the design matrix here is expected to not include the observed values of `A`
- **X** (*ndarray, list, vector, None, optional*) – Default of this argument is `None`, which implements the estimating equation for the inefficient g-estimator. To use the efficient g-estimator, a 2-dimensional vector of n observed values for *b* columns of a design matrix for the $E[H(\beta)|W]$ model should be provided here.
- **model** (*str, optional*) – Type of structural mean model to fit. Options are currently: `linear`, `poisson`. Default is `linear`. The Poisson model specification can be used for positive continuous data, or with binary data in order to estimate causal risk ratios.
- **weights** (*ndarray, list, vector, None, optional*) – 1-dimensional vector of n weights. Default is `None`, which assigns a weight of 1 to all observations. This argument is intended to support the use of sampling or missingness weights.

Returns Returns a (*b*`+`*c*)-by-*n* (inefficient) or (*b*`+`*c*`+`*d*)-by-*n* (efficient) NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_gestimation_snmm` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from scipy.stats import logistic
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_gestimation_snmm
```

Some generic data

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['W'] = np.random.normal(size=n)
>>> d['V'] = np.random.binomial(1, p=0.5, size=n)
>>> d['A'] = np.random.binomial(1, p=logistic.cdf(0.25 + 0.5*d['V'] + d['W']),␣
↪size=n)
>>> d['Ya0'] = 12.75 - 3.5*d['V'] + d['W'] + np.random.normal(size=n)
>>> d['Ya1'] = 10.75 - 0.8*d['V'] + d['W'] + np.random.normal(size=n)
>>> d['Y'] = (1-d['A'])*d['Ya0'] + d['A']*d['Ya1']
>>> d['C'] = 1
```

Defining psi, or the stacked estimating equations. Note that `A` is the action of interest and `Y` is the outcome of interest. Here, we are interested in estimating the following linear SMM

$$E[Y^a - Y^0 | A = a, V] = \beta_1 a + \beta_2 aV$$

```
>>> def psi(theta):
>>>     return ee_gestimation_snmm(theta,
>>>                                y=d['Y'], A=d['A'],
>>>                                W=d[['C', 'V', 'W']],
>>>                                V=d[['C', 'V']])
```

Calling the M-estimator. Since there are 2 coefficients in the SMM and 3 coefficients in the $E[A|W]$ model, the total number of initial values should be 2+3=5:

```
>>> estr = MEstimator(psi,
>>>                   init=[0., ]*5)
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

More specifically, the corresponding parameters are

```
>>> estr.theta[0]     # beta_1 of SMM
>>> estr.theta[1]     # beta_2 of SMM
>>> estr.theta[2:]    # propensity score regression coefficients
```

The efficient g-estimator can be implemented by providing a design matrix to the argument `X`

```
>>> def psi(theta):
>>>     return ee_gestimation_snmm(theta,
>>>                                y=d['Y'], A=d['A'],
>>>                                W=d[['C', 'V', 'W']],
>>>                                V=d[['C', 'V']],
>>>                                X=d[['C', 'V', 'W']])
```

Here, there are 2+3+3=8 parameters to estimate

```
>>> estr = MEstimator(psi,
>>>                   init=[0., ]*8)
>>> estr.estimate(solver='lm')
```

A log-linear SMM for this example can be estimated by specifying `model='poisson'`.

References

Dukes O, & Vansteelandt S (2018). A note on G-estimation of causal risk ratios. *American Journal of Epidemiology*, 187(5), 1079-1084.

Robins JM, Mark SD, Newey WK (1992). Estimating exposure effects by modelling the expectation of exposure conditional on confounders. *Biometrics*, 48(2), 479–495.

Vansteelandt S, & Joffe M (2014). Structural nested models and G-estimation: the partially realized promise. *Statist Sci*, 29(4), 707-731.

Vansteelandt S, & Sjolander A (2016). Revisiting g-estimation of the effect of a time-varying exposure subject to time-varying confounding. *Epidemiologic Methods*, 5(1), 37-56.

delicatessen.estimating_equations.causal.ee_mean_sensitivity_analysis

`ee_mean_sensitivity_analysis`(*theta*, *y*, *delta*, *X*, *q_eval*, *H_function*)

Estimating equation for weighted sensitivity analysis estimator of the mean. This estimator can handle cases of missing completely at random, missing at random, and missing not at random. The sensitivity analysis consists of two sets of estimating equations. The first is for the mean of interest (μ), and the second is for the sensitivity analysis model for the estimable parameters of the missingness model (γ):

$$\sum_{i=1}^{n} \begin{bmatrix} \frac{S_i Y_i}{H[\gamma X_i + q(Y_i, \alpha)]} - \mu \\ \left(\frac{S_i}{H[\gamma X_i + q(Y_i, \alpha)]} - 1 \right) X_i^T \end{bmatrix} = 0$$

where Y_i is the outcome of interest with missing data, X_i is the corresponding design matrix, and $H(b)$ is a known, continuous, and monotone increasing distribution function that is bound by $[0, 1]$. Here, $q(Y_i, \alpha)$ is a user-specified sensitivity function. For example, $q(Y_i, \alpha) = \alpha Y_i$ Importantly, α is treated as known (i.e., this approach is not possible when α needs to be estimated).

Note: This estimator looks like the inverse probability weighting estimator, but the estimating equation for the mean is slightly different. When $q(Y, \alpha) = 0$, the estimates between this estimator and the inverse probability weighting estimator will result in different (but similar) estimates.

The length of the parameter vector, θ, is 1+`b`, where *b* is the number of columns in `X`. The *first* value in the theta vector is the corrected mean of Y. The remainder of the parameters correspond to the regression model coefficients.

Parameters

- **theta** (*ndarray, list, vector*) – Theta in this case consists of 1+`b` values. Therefore, initial values should consist of one plus the number of columns present in `X`. This can easily be accomplished generally by `[0, ] + [0, ] * X.shape[1]`.
- **y** (*ndarray, list, vector*) – 1-dimensional vector of *n* values. Any values of `y` that are missing should be indicated by the `delta` parameter.

- **delta** (*ndarray, list, vector*) – 1-dimensional vector of n observed values indicating whether the observation has a value for `y` observed, where 1 indicates yes and 0 indicated no. This vector should not include any `nan` values.
- **X** (*ndarray, list, vector*) – 2-dimensional vector of n observed values for b variables consider as predictors. At a minimum, a vector of ones (intercept) should be included. This matrix cannot include any `nan` values.
- **q_eval** (*ndarray, list, vector*) – 1-dimensional vector of n values evaluated using the $q(Y; \alpha)$ function.
- **H_function** (*callable*) – Function use to bound the observations between $[0, 1]$. The function must be monotonic increasing and be bounded by $[0, 1]$. For example, the expit (`delicatessen.utilities.inverse_logit`) function meets this criteria.

Returns Returns a (1+`b`)-by-n NumPy array evaluated for the input `theta`.

Return type array

Examples

Construction of a estimating equation(s) with `ee_mean_sensitivity_analysis` should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_mean_sensitivity_analysis
>>> from delicatessen.utilities import inverse_logit
```

Some generic data with missing values for Y

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['W'] = np.random.binomial(1, p=0.5, size=n)
>>> d['Y'] = 200. - 35*d['W'] + np.random.normal(scale=5, size=n)
>>> d['M'] = np.random.binomial(1, p=inverse_logit(2 + d['W'] - 0.01 * d['Y']),␣
↪size=n)
>>> d['Y'] = np.where(d['M'] == 0, np.nan, d['Y'])
>>> d['C'] = 1
```

To apply the sensitivity analysis, we need to specify the corresponding sensitivity analysis function. The following is a simple model where missingness depends on a single parameter and the (possibly unobserved) outcome value. Note that the function replaces missing observations with zero.

```
>>> def q_function(y_vals, alpha):
>>>     # q(Y_i; \alpha) = alpha * Y_i
>>>     y_no_miss = np.where(np.isnan(y_vals), 0, y_vals)
>>>     return alpha * y_no_miss
```

We can now define psi, or the stacked estimating equations.

```
>>> def psi(theta):
>>>     yhat = q_function(d['Y'], alpha=-0.01)
>>>     return ee_mean_sensitivity_analysis(theta=theta,
```

(continues on next page)

(continued from previous page)

```
>>>                                                 y=d['Y'], delta=d['M'], X=d[['C', 'W']],
>>>                                                 q_eval=yhat, H_function=inverse_logit)
```

Calling the M-estimator.

```
>>> estr = MEstimator(psi, init=[200., 0., 0.])
>>> estr.estimate(solver='lm')
```

Inspecting the parameter estimates, variance, and 95% confidence intervals

```
>>> estr.theta
>>> estr.variance
>>> estr.confidence_intervals()
```

More specifically, the corresponding parameters are

```
>>> estr.theta[0]     # mean of interest
>>> estr.theta[1:]    # weighting model parameters given alpha
```

Proportions (binary outcomes) are also natively supported

```
>>> y_bin = np.where(d['Y'] <= 200., 1, 0)    # Binary conversion of outcome
>>> def psi(theta):
>>>     yhat = q_function(d['Y'], alpha=-0.1)
>>>     return ee_mean_sensitivity_analysis(theta=theta,
>>>                                         y=y_bin, delta=d['M'], X=d[['C', 'W']],
>>>                                         q_eval=yhat, H_function=inverse_logit)
```

```
>>> estr = MEstimator(psi, init=[0.5, 0., 0.])
>>> estr.estimate(solver='lm')
```

Often, we will want to conduct the sensitivity analysis for a range of different values of alpha. The following is code to accomplish this.

```
>>> def psi(theta):
>>>     yhat = q_function(d['Y'], alpha=alpha_current)
>>>     return ee_mean_sensitivity_analysis(theta=theta,
>>>                                         y=d['Y'], delta=d['M'], X=d[['C', 'W']],
>>>                                         q_eval=yhat, H_function=inverse_logit)
```

```
>>> alphas = np.linspace(0, 0.5, 40)
>>> est, lcl, ucl = [], [], []
>>> prev_optim = [200., 0., 0.]
>>> for alpha_current in alphas:
>>>     mest = MEstimator(psi, init=prev_optim)
>>>     mest.estimate(solver='lm')
>>>     prev_optim = mest.theta
>>>     est.append(mest.theta[0])
>>>     ci = mest.confidence_intervals()
>>>     lcl.append(ci[0][0])
>>>     ucl.append(ci[0][1])
>>> # plotting
>>> plt.fill_between(alphas, lcl, ucl, color='blue', alpha=0.2)
>>> plt.plot(alphas, est, '-', color='blue')
>>> plt.show()
```

Note: Note that we use the previous iteration as the starting values for the next alpha as a computational trick to speed up the root-finding process and prevent convergence issues.

The corresponding plot provides a visualization of how the estimated mean changes as α changes. This can be useful to help judge the extent of bias for the mean due to data missing not at random for a specific model.

References

Cole SR, Zivich PN, Edwards JK, Shook-Sa BE, & Hudgens MG. (2023). Sensitivity Analyses for Means or Proportions with Missing Outcome Data. *Epidemiology*

Robins JM, Rotnitzky A, & Scharfstein DO. (2000). Sensitivity analysis for selection bias and unmeasured confounding in missing data and causal inference models. In *Statistical models in epidemiology, the environment, and clinical trials* (pp. 1-94). New York, NY: Springer New York.

3.6.3 Utilities

For manipulation of output or inputs, there are several basic utility functionalities for transformation of variables, predicted parameters, or computations. Some are used internally by the estimating equations but are also made available to users.

Data transformations

logit(prob)	Logistic transformation.
inverse_logit(logodds)	Inverse logistic transformation.
identity(value)	Identity transformation.
robust_loss_functions(residual, loss, k[, a, b])	Loss functions for robust mean and robust regression estimating equations.
spline(variable, knots[, power, restricted, ...])	Generate generic polynomial spline terms for a given NumPy array and pre-specified knots.
regression_predictions(X, theta, covariance)	Generate predicted values of the outcome given a design matrix, point estimates, and covariance matrix.
polygamma(n, x)	Polygamma function.
digamma(z)	Digamma function.
standard_normal_cdf(x)	Cumulative distribution function for the standard normal distribution.
standard_normal_pdf(x)	Probability density function for the standard normal distribution.

delicatessen.utilities.logit

`logit`(*prob*)

Logistic transformation. Used to transform probabilities into log-odds.

$$\log\left(\frac{p}{1-p}\right)$$

Parameters `prob` (*float, ndarray*) – A single probability or an array of probabilities

Returns logit-transformed values

Return type array

delicatessen.utilities.inverse_logit

`inverse_logit`(*logodds*)

Inverse logistic transformation. Used to transform log-odds into probabilities.

$$\frac{1}{1+\exp(o)}$$

Parameters `logodds` (*float, ndarray*) – A single log-odd or an array of log-odds

Returns inverse-logit transformed values

Return type array

delicatessen.utilities.identity

`identity`(*value*)

Identity transformation. Used to transform input into itself (i.e., no transformation in applied).

Note: This function doesn't actually apply any transformation. It is used for arbitrary function calls that apply transformations, and this is called when no transformation is to be applied

Parameters `value` (*float, ndarray*) – A single value or an array of values

Returns

Return type value

delicatessen.utilities.robust_loss_functions

`robust_loss_functions`(*residual*, *loss*, *k*, *a=None*, *b=None*)

Loss functions for robust mean and robust regression estimating equations. This function is called internally for `ee_mean_robust` and `ee_robust_regression`. This function can also be accessed, so user's can easily adapt their own regression models into robust regression models using the pre-defined loss functions.

Note: The loss functions here are technically the first-order derivatives of the loss functions you see in the literature.

The following score of the loss functions, $f_k()$, are available.

Andrew's Sine

$$f_k(x) = I(k\pi \le x \le k\pi) \times \sin(x/k)$$

Huber

$$f_k(x) = xI(-k < x < k) + \text{sign}(x)k(1 - I(-k < x < k))$$

Tukey's biweight

$$f_k(x) = xI(-k < x < k) + x\left(1 - (x/k)^2\right)^2$$

Hampel (Hampel's add two additional parameters, a and b)

$$f_{k,a,b}(x) = \begin{bmatrix} I(-a < x < a) \times x \\ +I(a \le |x| < b) \times a \times \text{sign}(x) \\ +I(b \le x < k) \times a\frac{k-x}{k-b} \\ +I(-k \ge x > -b) \times -a\frac{-k+x}{-k+b} \\ +I(|x| \ge k) \times 0 \end{bmatrix}$$

Parameters

- **residual** (*ndarray, vector, list*) – 1-dimensional vector of n observed values. Input should consists of the residuals (the difference between the observed value and the predicted value). For the robust mean, this is $Y_i - \mu$. For robust regression, this is $Y_i - X_i^T \beta$
- **loss** (*str*) – Loss function to use. Options include: 'andrew', 'hampel', 'huber', 'minimax', 'tukey'
- **k** (*int, float*) – Tuning parameter for the corresponding loss function. Note: no default is provided, since each loss function has different recommendations.
- **a** (*int, float, None, optional*) – Lower parameter for the 'hampel' loss function
- **b** (*int, float, None, optional*) – Upper parameter for the 'hampel' loss function

Returns Returns a 1-by-n NumPy array evaluated for the input theta and residual

Return type array

Examples

Using the robust loss function

```
>>> import numpy as np
>>> from delicatessen.utilities import robust_loss_functions
```

Some generic data to stand-in for the residuals

```
>>> residuals = np.random.standard_cauchy(size=20)
```

Huber's loss function

```
>>> robust_loss_functions(residuals, loss='huber', k=1.345)
```

Andrew's Sine

```
>>> robust_loss_functions(residuals, loss='andrew', k=1.339)
```

Tukey's biweight

```
>>> robust_loss_functions(residuals, loss='tukey', k=4.685)
```

Hampel's loss function

```
>>> robust_loss_functions(residuals, loss='hampel', k=8, a=2, b=4)
```

References

Andrews DF. (1974). A robust method for multiple linear regression. *Technometrics*, 16(4), 523-531.

Beaton AE & Tukey JW (1974). The fitting of power series, meaning polynomials, illustrated on band-spectroscopic data. *Technometrics*, 16(2), 147-185.

Hampel FR. (1971). A general qualitative definition of robustness. *The Annals of Mathematical Statistics*, 42(6), 1887-1896.

Huber PJ. (1964). Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1), 73–101.

Huber PJ, Ronchetti EM. (2009) Robust Statistics 2nd Edition. Wiley. pgs 98-100

delicatessen.utilities.spline

spline(*variable*, *knots*, *power=3*, *restricted=True*, *normalized=False*)

Generate generic polynomial spline terms for a given NumPy array and pre-specified knots. Default is restricted cubic splines but unrestricted splines to different polynomial terms can also be specified.

Unrestricted splines for knot k are generated using the following formula

$$s_k(X) = I(X > k) \{X - k\}^a$$

where a is the power ($a = 3$ for cubic splines).

Restricted splines are generated via

$$r_k(X) = I(X > k) \{X - k\}^a - s_K(X)$$

where K is largest knot value.

Splines are normalized by the upper knot minus the lower knot to the corresponding power. Normalizing the splines can be helpful for the root-finding procedure.

Parameters

- **variable** (*ndarray, vector, list*) – 1-dimensional vector of observed values. Input should consists of the variable to generate spline terms for
- **knots** (*ndarray, vector, list*) – 1-dimensional vector of pre-specified knot locations. All knots should be between the minimum and maximum values of the input variable
- **power** (*int, float, optional*) – Power or polynomial term to use for the splines. Default is 3, which corresponds to cubic splines

- **restricted** (*bool, optional*) – Whether to generate restricted or unrestricted splines. Default is True, which corresponds to restricted splines. Restricted splines return one less column than the number of knots, whereas unrestricted splines return the same number of columns as knots
- **normalized** (*bool, optional*) – Whether to normalize, or divide, the spline terms by the difference between the upper and lower knots. Default is `False`, which corresponds to unnormalized splines. The default will update to `True` in the v3.0 release.

Returns A 2-dimensional array of the spline terms in ascending order of the knots.

Return type ndarray

Examples

Construction of spline variables should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen.utilities import spline
```

Some generic data to estimate a generalized additive model

```
>>> x = np.random.normal(size=200)
```

A restricted quadratic spline with 3 knots (at -1, 0, 1) can be generated using the following function call

```
>>> spline(variable=x, knots=[-1, 0, 1], power=2, restricted=True)
```

This function will return a 2 by 200 array here. Other knot specifications, other powers, and unrestricted splines can also be generated by updating the corresponding arguments.

References

Mulla ZD (2007). Spline regression in clinical research. *West Indian Med J*, 56(1), 77.

delicatessen.utilities.regression_predictions

regression_predictions(*X*, *theta*, *covariance*, *offset=None*, *alpha=0.05*)

Generate predicted values of the outcome given a design matrix, point estimates, and covariance matrix. This functionality computes $\hat{Y}$, $\hat{Var}\left(\hat{Y}\right)$, and corresponding Wald-type $(1 - \alpha) \times$ 100% confidence intervals from estimated coefficients and covariance of a regression model given a set of specific covariate values.

This function is a helper function to compute the predictions from a regression model for a set of given X values. Importantly, this method allows for the variance of $\hat{Y}$ to be estimated without having to expand the estimating equations. As such, this functionality is meant to be used after `MEstimator` has been used to estimate the coefficients (i.e., this function is for use after the M-estimator has computed the results for the chosen regression model). Therefore, this function should be viewed as a post-processing functionality for generating plots or tables.

Note: No tranformations are applied by this function. So, input from a logistic model will generate the *log-odds* of the outcome (not probability).

Parameters

- **X** (*ndarray, list, vector*) – 2-dimensional vector of values to generate predicted variances for. The number of columns must match the number of coefficients / parameters in `theta`.
- **theta** (*ndarray*) – Estimated coefficients from `MEstimator.theta`.
- **covariance** (*ndarray*) – Estimated covariance matrix from `MEstimator.variance`.
- **offset** (*ndarray, None, optional*) – A 1-dimensional offset to be included in the model. Default is None, which applies no offset term.
- **alpha** (*float, optional*) – The α level for the corresponding confidence intervals. Default is 0.05, which calculate the 95% confidence intervals. Notice that $0 < \alpha < 1$.

Returns Returns a n-by-4 NumPy array with the 4 columns correspond to the predicted outcome, variance of the predictied outcome, lower confidence limit, and upper confidence limit.

Return type array

Examples

The following is a simple example demonstrating how this function can be used to plot a regression line and corresponding 95% confidence intervals.

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from delicatessen import MEstimator
>>> from delicatessen.estimating_equations import ee_regression
>>> from delicatessen.utilities import regression_predictions
```

Some generic data to estimate the regression model with

```
>>> n = 50
>>> data = pd.DataFrame()
>>> data['X'] = np.random.normal(size=n)
>>> data['Z'] = np.random.normal(size=n)
>>> data['Y'] = 0.5 + 2*data['X'] - 1*data['Z'] + np.random.normal(loc=0, size=n)
>>> data['C'] = 1
```

Estimating the corresponding regression model parameters

```
>>> def psi(theta):
>>>     return ee_regression(theta=theta, X=data[['C', 'X', 'Z']], y=data['Y'],␣
↪model='linear')
```

```
>>> estr = MEstimator(stacked_equations=psi, init=[0., 0., 0.,])
>>> estr.estimate(solver='lm')
```

To create a line plot of our regression line, we need to first create a new set of covariate values that are evenly spaced across the range of the predictor values. Here, we will plot the relationship between `Z` and `Y` while holding `X=0`.

```
>>> pred = pd.DataFrame()
>>> pred['Z'] = np.linspace(np.min(data['Z']), np.max(data['Z']), 200)
>>> pred['X'] = 0
>>> pred['C'] = 1
```

Now the predicted values of the outcome, and confidence intervals to plot

```
>>> Xp = pred[['C', 'X', 'Z']]
>>> yhat = regression_predictions(X=Xp, theta=estr.theta, covariance=estr.variance)
```

Finally, the predicted values can be plotted (using matplotlib)

```
>>> plt.plot(pred['Z'], yhat[:, 0], '-', color='blue')
>>> plt.fill_between(pred['Z'], yhat[:, 2], yhat[:, 3], alpha=0.25, color='blue')
>>> plt.show()
```

For predicting with a Poisson or logistic model, one may want to transform the predicted values and confidence intervals to another measure. For the logistic model, the predicted log-odds can easily be transformed using `delicatessen.utilities.inverse_logit`. For the Poisson model, predictions can easily be transformed using `numpy.exp`.

delicatessen.utilities.polygamma

polygamma(*n*, *x*)

Polygamma function. This is a wrapper function of `scipy.special.polygamma` meant to enable automatic differentation with `delicatessen`. When the input is a `PrimalTangentPairs` object, then an internal function that implements the polygamma function is called. Otherwise, `scipy.special.polygamma` is called for the input object.

Parameters

- **n** (*int*) – Order of the derivative of the digamma function
- **x** (*int, float, ndarray*) – Real valued input

Returns

- Return type depends on the input type (`PrimalTangentPairs` will return `PrimalTangentPairs`, otherwise will
- return `ndarray`).

delicatessen.utilities.digamma

digamma(*z*)

Digamma function. This is a wrapper function of `scipy.special.digamma` meant to enable automatic differentation with `delicatessen`. When the input is a `PrimalTangentPairs` object, then an internal function that implements the digamma function is called. Otherwise, `scipy.special.digamma` is called for the input object.

Parameters **x** (*int, float, ndarray*) – Real valued input

Returns

- Return type depends on the input type (`PrimalTangentPairs` will return `PrimalTangentPairs`, otherwise will

- return `ndarray`).

delicatessen.utilities.standard_normal_cdf

`standard_normal_cdf`(*x*)

Cumulative distribution function for the standard normal distribution. This is a wrapper function of `scipy.stats.norm.cdf` meant to enable automatic differentation with `delicatessen`. When the input is a `PrimalTangentPairs` object, then an internal function that implements the CDF function is called. Otherwise, `scipy.stats.norm.cdf` is called for the input object.

Parameters **x** (*int, float, ndarray*) – Real valued input

Returns

- Return type depends on the input type (`PrimalTangentPairs` will return `PrimalTangentPairs`, otherwise will
- return `ndarray`).

delicatessen.utilities.standard_normal_pdf

`standard_normal_pdf`(*x*)

Probability density function for the standard normal distribution. This is a wrapper function of `scipy.stats.norm.pdf` meant to enable automatic differentation with `delicatessen`. When the input is a `PrimalTangentPairs` object, then an internal function that implements the PDF function is called. Otherwise, `scipy.stats.norm.pdf` is called for the input object.

Parameters **x** (*int, float, ndarray*) – Real valued input

Returns

- Return type depends on the input type (`PrimalTangentPairs` will return `PrimalTangentPairs`, otherwise will
- return `ndarray`).

Design matrices

`additive_design_matrix`(X, specifications[, ...])	Generate an additive design matrix for generalized additive models (GAM) given a set of spline specifications to apply.

delicatessen.utilities.additive_design_matrix

`additive_design_matrix`(*X*, *specifications*, *return_penalty=False*)

Generate an additive design matrix for generalized additive models (GAM) given a set of spline specifications to apply.

Note: This function is interally called by `ee_additive_regression`. This function can also be called to aid in easily generating predicted values.

Parameters

- **X** (*ndarray, vector, list*) – Input independent variable data.
- **specifications** (*ndarray, vector, list*) – A list of dictionaries that define the hyper-parameters for the spline (e.g., number of knots, strength of penalty). For terms that should not have splines, `None` should be specified instead (see examples below). Each dictionary supports the following parameters: "knots", "natural", "power", "penalty" knots (list): controls the position of the knots, with knots are placed at given locations. There is no default, so must be specified by the user. natural (bool): controls whether to generate natural (restricted) or unrestricted splines. Default is `True`, which corresponds to natural splines. power (float): controls the power to raise the spline terms to. Default is 3, which corresponds to cubic splines. penalty (float): penalty term (λ) applied to each corresponding spline basis term. Default is 0, which applies no penalty to the spline basis terms. normalized (bool): whether to normalize the spline terms. Default is `False`, with a default change coming with v3.0 release.
- **return_penalty** (*bool, optional*) – Whether the list of the corresponding penalty terms should also be returned. This functionality is used internally to create the list of penalty terms to provide the Ridge regression model, where only the spline terms are penalized. Default is False.

Returns Returns a (b+k)-by-n design matrix as a NumPy array, where b is the number of columns in the input array and k is determined by the specifications of the spline basis functions.

Return type array

Examples

Construction of a design matrix for an additive model should be done similar to the following

```
>>> import numpy as np
>>> import pandas as pd
>>> from delicatessen.utilities import additive_design_matrix
```

Some generic data to estimate a generalized additive model

```
>>> n = 200
>>> d = pd.DataFrame()
>>> d['X'] = np.random.normal(size=n)
>>> d['Z'] = np.random.normal(size=n)
>>> d['W'] = np.random.binomial(n=1, p=0.5, size=n)
>>> d['C'] = 1
```

To begin, consider the simple input design matrix of `d[['C', 'X']]`. This initial design matrix consists of an intercept term and a continuous term. Here, we will specify a natural spline with 20 knots for the second term only

```
>>> x_knots = np.linspace(np.min(d['X'])+0.1, np.max(d['X'])-0.1, 20)
>>> specs = [None, {"knots": x_knots, "penalty": 10}]
>>> Xa_design = additive_design_matrix(X=d[['C', 'X']], specifications=specs)
```

Other optional specifications are also available. Here, we will specify an unrestricted quadratic spline with a penalty of 5.5 for the second column of the design matrix.

```
>>> specs = [None, {"knots": [-2, -1, 0, 1, 2], "natural": False, "power": 2,
↪"penalty": 5.5}]
>>> Xa_design = additive_design_matrix(X=d[['C', 'X']], specifications=specs)
```

Now consider the input design matrix of `d[['C', 'X', 'Z', 'W']]`. This initial design matrix consists of an intercept, two continuous, and a categorical term. Here, we will specify splines for both continuous terms

```
>>> x_knots = np.linspace(np.min(d['X'])+0.1, np.max(d['X'])-0.1, 20)
>>> z_knots = np.linspace(np.min(d['Z'])+0.1, np.max(d['Z'])-0.1, 10)
>>> specs = [None,                                # Intercept term
>>>          {"knots": x_knots, "penalty": 25}, # X (continuous)
>>>          {"knots": z_knots, "penalty": 15}, # Z (continuous)
>>>          None]                                # W (categorical)
>>> Xa_design = additive_design_matrix(X=d[['C', 'X', 'Z', 'W']],␣
↪specifications=specs)
```

Notice that the two continuous terms have different spline specifications.

Finally, we could opt to only generate a spline basis for one of the continuous variables

```
>>> specs = [None,                                # Intercept term
>>>          {"knots": x_knots, "penalty": 25}, # X (continuous)
>>>          None,                                # Z (continuous)
>>>          None]                                # W (categorical)
>>> Xa_design = additive_design_matrix(X=d[['C', 'X', 'Z', 'W']],␣
↪specifications=specs)
```

Specification of splines can be modified and paired in a variety of ways. These are determined by the object type in the specification list, and the input dictionary for the spline terms.

Differentiation

approx_differentiation(xk, f[, epsilon, method])	Numerical approximation to compute the gradient.
auto_differentiation(xk, f)	Forward-mode automatic differentiation.

delicatessen.derivative.approx_differentiation

approx_differentiation(*xk*, *f*, *epsilon=1e-09*, *method='capprox'*)

Numerical approximation to compute the gradient. This function implements numerical approximation methods for derivatives generally (i.e., it provides the first-order forward, backward, and central difference approximations).

Note: This functionality is only intended for use behind the scenes in `delicatessen`. Numerical approximation is implemented from scratch to offer backward and central difference approximations (SciPy's `approx_fprime` only offers the forward difference).

The forward difference approximation is

$$\frac{f(x+\epsilon) - f(x)}{\epsilon}$$

the backward difference approximation is

$$\frac{f(x) - f(x-\epsilon)}{\epsilon}$$

and the central difference approximation is

$$\frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Here, the numerical approximation is implemented by generating matrices for output from a function evaluated under minor perturbations (determined by `epsilon`) of each input argument. These matrices are then subtracted from each other and then scaled by `epsilon`.

Parameters

- **xk** (*ndarray, list, shape (n,)*) – Point(s) or coordinate vector to evaluate the gradient at.
- **f** (*callable*) – Function of which to estimate the gradient of.
- **epsilon** (*float, optional*) – Increment to perturb the points by to compute the gradient. This should be a small value
- **method** (*str, optional*) – Approximation to use to compute the gradient. Default is *capprox* which uses the central difference method. One can also specify the forward difference (*fapprox*) or backward difference (*bapprox*) methods.

Returns Corresponding array of the pairwise derivatives for all different input x values.

Return type numpy.array

Examples

```
>>> import numpy as np
>>> from delicatessen.derivative import approx_differentiation
```

To illustrate use, we will compute the derivative of the following function

$$f(x) = x^2 - x^1 + sin(x + \sqrt{x})$$

```
>>> def f(x):
>>>     return x**2 - x + np.sin(x + np.sqrt(x))
```

If you work out the deriative by-hand, you will end up with the following

$$2x - 1 + \left(\frac{1}{2\sqrt{x}} + 1\right)\cos(x + \sqrt{x})$$

Instead, we can use the central difference approximation to evaluate the derivative at a specific point. Here, we will evaluate the derivative at $x = 1$

```
>>> dy = approx_differentiation(xk=[1, ], f=f)
```

which returns `0.37578`, which is close to plugging in $x = 1$ into the previous equation.

The derivative of a function with multiple inputs and multiple outputs can also be evaluated. Consider the following example with three inputs and two outputs

```
>>> def f(x):
>>>     return [x[0]**2 - x[1], np.sin(np.sqrt(x[1]) + x[2]) + x[2]*(x[1]**2)]
>>> approx_differentiation(xk=[0.7, 1.2, -0.9], f=f, method='fapprox')
>>> approx_differentiation(xk=[0.7, 1.2, -0.9], f=f, method='bapprox')
>>> approx_differentiation(xk=[0.7, 1.2, -0.9], f=f, method='capprox')
```

which will return a 2-by-3 array of all the x-y pair derivatives at the given values. Here, the rows correspond to the output and the columns correspond to the inputs. The approximation methods are forward, backward, and central.

delicatessen.derivative.auto_differentiation

auto_differentiation(*xk*, *f*)

Forward-mode automatic differentiation. Automatic differentiation offers a way to compute the exact derivative, rather than numerically approximated (i.e., the central difference method). Automatic differentiation iteratively applies the chain rule through recursive calls to evaluate the derivative.

Note: This functionality is only intended for use behind the scenes in `delicatessen`. Automatic differentiation is implemented from scratch to avoid additional dependencies.

This is accomplished by the `PrimalTangentPairs` class, which is a special data type in `delicatessen` that stores pairs of the original evaluation and the corresponding derivative for a variety of different mathematical operations. This is what allows for the exact derivative calculation. The `auto_differentiation` function is a wrapper to access and use this class object as it is intended for derivative computations.

Parameters

- **xk** (*ndarray, list, shape (n,)*) – Point(s) or coordinate vector to evaluate the gradient at.
- **f** (*callable*) – Function of which to estimate the gradient of.

Returns Corresponding array of the pairwise derivatives for all different input x values.

Return type numpy.array

Examples

Loading necessary functions

```
>>> import numpy as np
>>> from delicatessen.derivative import auto_differentiation
```

To illustrate use, we will compute the derivative of the following function

$$f(x) = x^2 - x^1 + sin(x + \sqrt{x})$$

```
>>> def f(x):
>>>     return x**2 - x + np.sin(x + np.sqrt(x))
```

If you work out the deriative by-hand, you will end up with the following

$$2x - 1 + \left(\frac{1}{2\sqrt{x}} + 1\right)\cos(x + \sqrt{x})$$

Instead, we can use automatic differentiation to evaluate the derivative at a specific point. Here, we will evaluate the derivative at $x = 1$

```
>>> dy = auto_differentiation(xk=[1, ], f=f)
```

which returns `0.3757795`. This is the same as if you plugged in $x = 1$ into the previous equation.

Note: If a derivative is not defined, then the function will return a `NaN`.

The derivative of a function with multiple inputs and multiple outputs can also be evaluated. Consider the following example with three inputs and two outputs

```
>>> def f(x):
>>>     return [x[0]**2 - x[1], np.sin(np.sqrt(x[1]) + x[2]) + x[2]*(x[1]**2)]
```

```
>>> dy = auto_differentiation(xk=[0.7, 1.2, -0.9], f=f)
```

which will return a 2-by-3 array of all the x-y pair derivatives at the given values. Here, the rows correspond to the output and the columns correspond to the inputs.

References

Baydin AG, Pearlmutter BA, Radul AA, & Siskind JM. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18, 1-43.

Rall LB & Corliss GF. (1996). An introduction to automatic differentiation. Computational Differentiation: Techniques, Applications, and Tools, 89, 1-18.

CHAPTER

FOUR

CODE AND ISSUE TRACKER

Please report bugs, issues, or feature requests on GitHub at pzivich/Delicatessen.

Otherwise, you may contact me via email (gmail: zivich.5).

CHAPTER

FIVE

REFERENCES

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In *Essential Statistical Inference* (pp. 297-337). Springer, New York, NY.

Saul BC, & Hudgens MG. (2020). The Calculus of M-Estimation in R with geex. *Journal of Statistical Software*, 92(2).

Stefanski LA, & Boos DD. (2002). The calculus of M-estimation. *The American Statistician*, 56(1), 29-38.

Ross RK, Zivich PN, Stringer JSA, & Cole SR. (2024). M-estimation for common epidemiological measures: introduction and applied examples. *International Journal of Epidemiology*, 53(2), dyae030.

Zivich PN, Klose M, Cole SR, Edwards JK, & Shook-Sa BE. (2022). Delicatessen: M-Estimation in Python. *arXiv preprint arXiv:2203.11300*.

INDEX