`clarify`: Simulation-Based Inference for Regression Models

by Noah Greifer, Steven Worthington, Stefano Iacus, and Gary King

Abstract Simulation-based inference is an alternative to the delta method for computing the uncertainty around regression post-estimation (i.e., derived) quantities such as average marginal effects and adjusted predictions. It avoids several assumptions required by the delta method that are commonly violated, especially in nonlinear models, but does not require models to be refit as when using the bootstrap. It works by drawing model parameters from their joint distribution and estimating quantities of interest from each set of simulated values, which form the sampling distribution of the quantity. `clarify` provides a simple, unified interface for performing simulation-based inference for any user-specified derived quantities as well as wrappers for common quantities of interest. `clarify` supports a large and growing number of models through its interface with the `marginaleffects` package and provides native support for multiply imputed data.

Introduction

Although regression models are frequently used in empirical research to study relationships among variables, often the quantity of substantive interest is not one of the coefficients of the model, but rather a quantity derived from the coefficients, such as predicted values or average marginal effects. The usual method for estimating the uncertainty of the derived quantities is known as the "delta method", which involves two approximations: 1) that the variance of the derived quantity can be represented as a first-order Taylor series, and 2) that the estimate of the derived quantity is normally distributed[1]. In many cases, especially with nonlinear models, these approximation can fail badly. For example, predicted probabilities close to 0 or 1 or ratios of coefficients or probabilities typically do not have normal (or even symmetrical) distributions in finite samples, and the usual Wald-type confidence interval limits produced from delta method standard errors can include values outside the domain of the quantity of interest.

clarify implements an alternative to the delta method—simulation-based inference—which involves simulating the sampling distributions of the derived quantities. Simulation-based inference is not only often more accurate than using the delta method, it is also simpler to understand and implement, as it does not require understanding Taylor series or the calculus that underlies it. This makes it more palatable to nontechnical audiences and easier to learn for students without sacrificing statistical performance.

The methodology **clarify** relies on is described in King, Tomz, and Wittenberg (2000) and Rainey (2023). Simulation-based inference involves taking draws from a specified joint distribution of model parameters, computing derived quantities from these draws, and collecting the derived quantities in a "posterior" distribution, which is taken to be the sampling distribution of the derived quantity, and from which uncertainty measures (standard errors and confidence intervals) can be computed. This method assumes the model parameters are drawn from a multivariate Normal (or T) distribution with means at the estimated values and covariance equal to the asymptotic covariance matrix of the estimated values, a standard assumption motivated by the central limit theorem that underlies usual inference on the original model parameters. Arriving at the posterior distribution does not require taking any derivatives or making any approximations beyond those usually used for inference on model parameter estimates. The delta method makes these same assumptions in addition to the approximations about the variance and the shape of the sampling distribution of the estimates of the derived quantities.

The nonparametric bootstrap is another alternative to the delta method for inference that does not require these approximations; it typically involves re-sampling individuals from the sample, fitting the model in each sample, and computing the quantity of interest from each model. Although bootstrapping tends to work well in practice, especially for complex and non-Normal estimators, refitting the model repeatedly can be prohibitively time-consuming and computationally expensive, especially for complicated models or large datasets. Simulation-based inference only requires the model to be fit once, and the simulations involve taking draws from a distribution produced from the single set of estimated parameters, making it much quicker in practice and allowing the user to capitalize on the already valid estimation of the model parameters.

More formally, we fit a regression model $y_i = f(x_i; \beta)$, such as a linear or other generalized linear

[1] In addition, most software that uses the delta method uses numerical differentiation, which also involves an approximation.

model with model coefficients β. We assume

$$\hat{\beta} \sim \text{MVN}(\beta, \Sigma_{\hat{\beta}})$$

where $\hat{\beta}$ is the vector of estimates of β and $\Sigma_{\hat{\beta}}$ is their asymptotic covariance matrix. We define a function $\tau(\beta)$ that represents a quantity of interest derived from the model parameters, and compute its estimate $\widehat{\tau(\beta)}$ as $\tau(\hat{\beta})$. Next, we take M draws $\tilde{\beta}^{(j)}$ for $j \in (1, \dots, M)$ from a multivariate Normal distribution with mean vector $\mu = \hat{\beta}$ and covariance $\Sigma = \hat{\Sigma}_{\hat{\beta}}$, where $\hat{\Sigma}_{\hat{\beta}}$ is an estimate of the asymptotic covariance matrix of the parameter estimates. We use the distribution of $\tilde{\tau} = \tau(\tilde{\theta})$ as the "posterior" sampling distribution of $\widehat{\tau(\beta)}$, and compute its variance as

$$\sigma^2_{\widehat{\tau(\theta)}} = \frac{1}{M} \sum_{j=1}^{M} (\tilde{\tau}^{(j)} - \bar{\tilde{\tau}})^2$$

and quantile $100(1-\alpha)\%$ confidence interval limits as $\left[\tilde{\tau}_{(\frac{\alpha}{2})}, \tilde{\tau}_{(1-\frac{\alpha}{2})}\right]$ where $\tilde{\tau}_{(k)}$ is the kth value of $\tilde{\tau}$ when arranged in ascending order.

Similar functionality exists in the **CLARIFY** package in Stata[2] (Tomz, Wittenberg, and King 2003) and used to be available in the `Zelig` R package (Imai, King, and Lau 2008), though there are differences in these implementations. **clarify** provides additional flexibility by allowing the user to request any derived quantity, in addition to providing shortcuts for common quantities, including predictions at representative values, average marginal effects, and average dose-response functions (described below). **clarify** relies on and can be seen as a companion to the **marginaleffects** package, which offers similar functionality but primarily uses the delta method for calculating uncertainty (though simulation-based inference is supported in a more limited capacity as well).

Using clarify

There are four steps to using **clarify**:

1. Fit the model to the data using modeling functions in supported packages.
2. Use `sim()` to take draws from the sampling distribution of the estimated model coefficients.
3. Use `sim_apply()` or its wrappers `sim_setx()`, `sim_ame()`, and `sim_adrf()` to compute derived quantities using each simulated set of coefficients.
4. Use `summary()` and `plot()` to summarize and visualize the distribution of the derived quantities and perform inference on them.

In the sections below, we will describe how to implement these steps in detail. First, we will load **clarify** using `library()`.

```
library(clarify)
```

For a running example, we will use the `lalonde` dataset in the **MatchIt** package (Ho et al. 2011), which contains data on 614 participants enrolled in a job training program or sampled from a survey (Dehejia and Wahba 1999). The treatment variable is `treat` and the outcome is `re78`, and all other variables are confounders. Although the original use of this dataset was to estimate the effect of `treat` on `re78`, we will use it more generally to demonstrate all of **clarify**'s capabilities. In addition, we will use a transformation of the outcome variable to demonstrate applications to nonlinear models, for which the benefits of simulation-based inference are more apparent.

```
data("lalonde", package = "MatchIt")

lalonde$re78_0 <- ifelse(lalonde$re78 > 0, 1, 0)

head(lalonde)

#>      treat age educ   race married nodegree re74 re75       re78 re78_0
```

[2]Despite the similar name, the R package **clarify** and the Stata package **CLARIFY** differ in several ways, one of which is that the estimates reported by **clarify** in R are those computed using the original model coefficients, whereas those reported by **CLARIFY** in Stata are those computed as the average of the simulated distribution. The R implementation avoids the "simulation-induced bias" described by Rainey (2023).

```
#> NSW1      1  37   11  black       1         1    0    0  9930.0460      1
#> NSW2      1  22    9 hispan       0         1    0    0  3595.8940      1
#> NSW3      1  30   12  black       0         0    0    0 24909.4500      1
#> NSW4      1  27   11  black       0         1    0    0  7506.1460      1
#> NSW5      1  33    8  black       0         1    0    0   289.7899      1
#> NSW6      1  22    9  black       0         1    0    0  4056.4940      1
```

1. Fitting the model

The first step is to fit the model. **clarify** can operate on a large set of models (those supported by **marginaleffects**), including generalized linear models, multinomial models, multivariate models, and instrumental variable models, many of which are available in other R packages. Even if **clarify** does not offer direct support for a given model, there are ways to use its functionality regardless (explained in more detail below).

Because we are computing derived quantities, it is not critical to parameterize the model in such a way that the coefficients are interpretable, e.g., by using a model with interpretable coefficients or centering predictors. Below, we will fit a probit regression model for the outcome given the treatment and confounders. Coefficients in probit regression do not have a straightforward interpretation, but that does not matter; our quantities of interest can be expressed as derived quantities—functions of the model parameters, such as predictions, counterfactual predictions, and averages and contrasts of them.

```
fit <- glm(re78_0 ~ treat * married + age + educ + race +
             nodegree + re74 + re75, data = lalonde,
           family = binomial("probit"))
```

2. Drawing from the coefficient distribution

After fitting the model, we will use `sim()` to draw coefficients from their sampling distribution. The sampling distribution is assumed to be multivariate Normal or multivariate T with appropriate degrees of freedom, with a mean vector equal to the estimated coefficients and a covariance matrix equal to the asymptotic covariance matrix extracted from the model. The arguments to `sim()` are listed below:

```
sim(fit = , n = , vcov = , coefs = , dist = )
```

- `fit` – the fitted model object, the output of the call to the fitting function (e.g., `glm()`)
- `n` – the number of simulated values to draw; by default, 1000. More values will yield more replicable and precise results at the cost of speed.
- `vcov` – either the covariance matrix of the estimated coefficients, a function used to extract it from the model (e.g., `sandwich::vcovHC()` for the robust covariance matrix), or a string or formula giving a code for extracting the covariance matrix, which is passed to `marginaleffects::get_vcov()`. If left unspecified, the default covariance matrix will be extracted from the model.
- `coefs` – either a vector of coefficients to be sampled or a function to extract them from the fitted model. If left unspecified, the default coefficients will be extracted from the model. Typically this does not need to be specified.
- `dist` – the name of the distribution from which to draw the sampled coefficients. Can be `"normal"` for a Normal distribution or `t(#)` for a t-distribution, where # represents the degrees of freedom. If left unspecified, `sim()` will decide on which distribution makes sense given the characteristics of the model (the decision is made by `insight::get_df()` with `type = "wald"`). Typically this does not need to be specified.

If one's model is not supported by **clarify**, one can omit the `fit` argument and just specify the `vcov` and `coefs` argument, which will draw the coefficients from the distribution named in `dist` (`"normal"` by default).

`sim()` uses a random number generator to draw the sampled coefficients from the sampling distribution, so a seed should be set using `set.seed()` to ensure results are replicable across sessions.

The output of the call to `sim()` is a `clarify_sim` object, which contains the sampled coefficients, the original model fit object if supplied, and the coefficients and covariance matrix used to sample.

```
set.seed(1234)
```

```
# Drawing 1000 simulated coefficients using an HC2 robust
# covariance matrix
s <- sim(fit, n = 1000,
         vcov = "HC2")

s

#> A `clarify_sim` object
#>  - 11 coefficients, 1000 simulated values
#>  - sampled distribution: multivariate normal
#>  - original fitting function call:
#>
#> glm(formula = re78_0 ~ treat * married + age + educ + race +
#>     nodegree + re74 + re75, family = binomial("probit"), data = lalonde)
```

3. Computing derived quantities

After sampling the coefficients, we will compute derived quantities on each set of sampled coefficients and store the result, which represents the "posterior" distribution of the derived quantity, as well as on the original coefficients, which are used as the final estimates. The core functionality is provided by `sim_apply()`, which accepts a `clarify_sim` object from `sim()` and a function to compute and return one or more derived quantities, then applies that function to each set of simulated coefficients. The arguments to `sim_apply()` are below:

```
sim_apply(sim = , FUN = , verbose = , cl = , ...)
```

- `sim` – a `clarify_sim` object; the output of a call to `sim()`.
- `FUN` – a function that takes in either a model fit object or a vector of coefficients and returns one or more derived quantities. The first argument should be named `fit` to take in a model fit object or `coefs` to take in coefficients.
- `verbose` – whether to display a progress bar.
- `cl` – an argument that controls parallel processing, which can be the number of cores to use or a cluster object resulting from `parallel::makeCluster()`.
- `...` - further arguments to `FUN`.

The `FUN` argument can be specified in one of two ways: either as a function that takes in a model fit object (e.g., a `glm` or `lm` object, the output of a call to `glm()` or `lm()`) or a function that takes in a vector of coefficients. The latter will always work but the former only works for supported models. When the function takes in a model fit object, `sim_apply()` will first insert each set of sampled coefficients into the model fit object and then supply the modified model to `FUN`.

For example, we will let our derived quantity of interest be the predicted probability of the outcome for participant PSID1. We would specified our `FUN` function as follows:

```
sim_fun1 <- function(fit) {
  predict(fit, newdata = lalonde["PSID1",], type = "response")
}
```

The `fit` object supplied to this function will be one in which the coefficients have been set to their values in a draw from their sampling distribution as generated by `sim()`. We then supply the function to `sim_apply()` to simulate the sampling distribution of the predicted value of interest:

```
est1 <- sim_apply(s, FUN = sim_fun1, verbose = FALSE)

est1

#> A `clarify_est` object (from `sim_apply()`)
#>  - 1000 simulated values
#>  - 1 quantity estimated:
#>  PSID1 0.9757211
```

The resulting clarify_est object contains the simulated estimates in matrix form as well as the estimate computed on the original coefficients. We will examine the sampling distribution shortly, but first we will demonstrate computing a derived quantity from the coefficients directly.

The race variable is a factor, and the black category is used as the reference level, so it is not immediately clear whether there is a difference between the coefficients racehispan and racewhite, which represent the non-reference categories hispan and white. To compare these two directly, we can use sim_apply() to compute a derived quantity that corresponds to the difference between them.

```
sim_fun2 <- function(coefs) {
  hispan <- unname(coefs["racehispan"])
  white <- unname(coefs["racewhite"])

  c("w - h" = white - hispan)
}

est2 <- sim_apply(s, FUN = sim_fun2, verbose = FALSE)

est2
```

```
#> A `clarify_est` object (from `sim_apply()`)
#>  - 1000 simulated values
#>  - 1 quantity estimated:
#>  w - h -0.09955915
```

The function supplied to FUN can be arbitrarily complicated and return as many derived quantities as one wants, though the slower each run of FUN is, the longer it will take to simulate the derived quantities. Using parallel processing by supplying an argument to cl can sometimes dramatically speed up evaluation.

There are several functions in **clarify** that serve as convenience wrappers for sim_apply() to automate some common derived quantities of interest. These include

- sim_setx() – computing predicted values and first differences at representative or user-specified values of the predictors
- sim_ame() – computing average adjusted predictions, contrasts of average adjusted predictions, and average marginal effects
- sim_adrf() – computing average dose-response functions and average marginal effects functions

These are described in their own sections below. In addition, there are functions that have methods for clarify_est objects, including cbind() for combining two clarify_est objects together and transform() for computing quantities that are derived from the already-computed derived quantities. These are also described in their own sections below.

4. Summarize and visualize the simulated distribution

To examine the uncertainty around and perform inference on our estimated quantities, we can use plot() and summary() on the clarify_est object.

plot() displays a density plot of the resulting estimates across the simulations, with markers identifying the point estimate (computed using the original model coefficients as recommended by Rainey (2023)) and, optionally, uncertainty bounds (which function like confidence or credible interval bounds). The arguments to plot() are below:

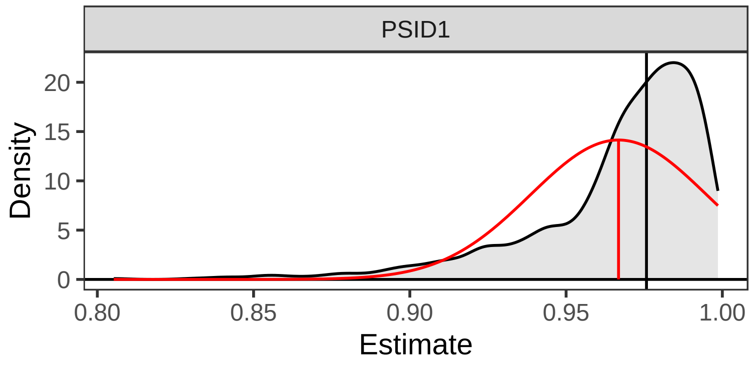
```
plot(x = , parm = , ci = , level = , method = , reference =)
```

- x – the clarify_est object (the output of a call to sim_apply()).
- parm – the names or indices of the quantities to be plotted if more than one was estimated in sim_apply(); if unspecified, all will be plotted.
- ci – whether to display lines at the uncertainty bounds. The default is TRUE to display them.
- level – if ci is TRUE, the desired two-sided confidence level. The default is .95 so that that the bounds are at the .025 and .975 quantiles when method (see below) is "quantile".

- `method` – if `ci` is `TRUE`, the method used to compute the bounds. Allowable methods include a Normal approximation (`"wald"`) or using the quantiles of the resulting distribution (`"quantile"`). The Normal approximation involves multiplying the standard deviation of the estimates (i.e., which functions like the standard error of the sampling distribution) by the critical Z-statistic computed using `(1-level)/2` to create a symmetric margin of error around the point estimate. The default is `"quantile"` to instead use quantile-based bounds, which are more appropriate when the distribution is non-Normal. However, quantile-based bounds may require more simulations to stabilize.
- `reference` – whether to display a normal density over the plot for each estimate. The default is `FALSE` to omit it.

Below, we plot the first estimate we computed above, the predicted probability for participant PSID1:

```
plot(est1, reference = TRUE, ci = FALSE)
```



Overlaid on the plot in red is a Normal distribution with the same mean and standard deviation as the simulated values; this is requested by setting `reference = TRUE`. From the plot, one can see that the distribution of simulated values is non-Normal, asymmetrical, and not centered around the estimate, with no values falling above 1 because the outcome is a predicted probability. Given its non-Normality, the quantile-based bounds are clearly more appropriate than those resulting from the Normal approximation, as the bounds computed from the Normal approximation would be outside the bounds of the estimate. The plot itself is a `ggplot` object that can be modified using `ggplot2` syntax.

We can use `summary()` to display the value of the point estimate, the uncertainty bounds, and other statistics that describe the distribution of estimates. The arguments to `summary()` are below:

```
summary(object = , parm = , level = , method = , null = )
```

- `object` – the `clarify_est` object (the output of a call to `sim_apply()`).
- `parm` – the names or indices of the quantities to be displayed if more than one was estimated in `sim_apply()`; if unspecified, all will be displayed.
- `level` – the desired two-sided confidence level. The default is .95 so that that the bounds are at the .025 and .975 quantiles when method (see below) is `"quantile"`.
- `method` – the method used to compute the uncertainty bounds. Allowable methods include a Normal approximation (`"wald"`) or using the quantiles of the resulting distribution (`"quantile"`). See `plot()` above.
- `null` – an optional argument specifying the null value in a hypothesis test for the estimates. If specified, a p-value will be computed using either a standard Z-test (if `method` is `"wald"`) or an inversion of the uncertainty interval (described below). The default is not to display any p-values.

Inverting the uncertainty interval involves finding the smallest confidence level such that the null value is within the confidence bounds. The p-value for the test is one minus this level. It is only valid as a p-value when the simulated distribution of the estimates differs from its true sampling distribution under the null value by a location shift (which can be violated when the distribution of the estimates is asymmetric). However, because the p-values are invariant to monotonic transformations of the estimates and null value, as long as the distribution of *some* monotonic transformation of the estimate is a location shift from its sampling distribution under the null hypothesis, the p-values will be valid.

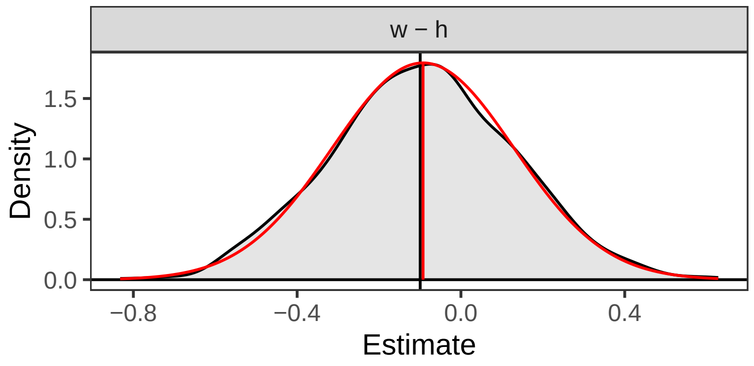
We can use `summary()` with the default arguments on our first `clarify_est` object to view the point estimate and quantile-based uncertainty bounds.

```
summary(est1)

#>       Estimate 2.5 % 97.5 %
#> PSID1    0.976 0.891  0.996
```

A benefit of quantile-based confidence bounds is that the bounds will always be in the domain of the quantity of interest, e.g., bounds for probabilities will not be above 1 or below 0 if the model always generates predicted probabilities between 0 and 1. In contrast, Wald-type symmetric confidence intervals (including those produced by the delta method) can include values outside the domain of the quantity of interest. While it is possible to compute Wald-type confidence interval bounds that respect the domain boundaries, e.g., by computing the confidence interval around the logit of the predicted probability and using the inverse logit of the bounds as the bound on the probability, quantile confidence intervals do not require the user to know which transformation of the quantity of interest will yield appropriate bounds.

Our second estimated quantity, the difference between two regression coefficients, is closer to Normally distributed, as the plot below demonstrates (and would be expected theoretically), so we will use the Normal approximation to test the hypothesis that difference differs from 0.

```
plot(est2, reference = TRUE, ci = FALSE)
```



```
summary(est2, method = "wald", null = 0)

#>       Estimate   2.5 %  97.5 % Std. Error Z value P-value
#> w - h  -0.0996 -0.5352  0.3361     0.2223   -0.45    0.65
```

The uncertainty intervals and p-values in the `summary()` output are computed using the Normal approximation because we set `method = "wald"`, and the p-value for the test that our estimate is equal to 0 is returned because we set `null = 0`. Note that the Normal approximation should be used only when the simulated sampling distribution is both close to Normally distributed and centered around the estimate (i.e., when the mean of the simulated values [red vertical line] coincides with the estimate computed on the original coefficients [black vertical line]).

`sim_apply()` Wrappers: `sim_setx()`, `sim_ame()`, `sim_adrf()`

`sim_apply()` can be used to compute the simulated sampling distribution for an arbitrary derived quantity of interest, but there are some quantities that are common in applied research and may otherwise be somewhat challenging to program by hand, so clarify provides shortcut functions to make computing these quantities simple. These functions include `sim_setx()`, `sim_ame()`, and `sim_adrf()`. Each of these can be used only when regression models compatible with clarify are supplied to the original call to `sim()`.

Like `sim_apply()`, each of these functions is named `sim_*()`, which signifies that they are to be used on an object produced by `sim()` (i.e., a `clarify_sim` object). (Multiple calls to these functions can be applied to the same `clarify_sim` object and combined; see the `cbind()` section below.) These functions are described below.

`sim_setx()`: predictions at representative values

`sim_setx()` provides an interface to compute predictions at representative and user-supplied values of the predictors. For example, we might want to know what the effect of treatment is for a "typical" individual, which corresponds to the contrast between two model-based predictions (i.e., one under treatment and one under control for a unit with "typical" covariate values). This functionality mirrors the `setx()` and `setx1()` functionality of `Zelig` (which is where its name originates) and provides similar functionality to functions in **modelbased**, **emmeans**, **effects**, and **ggeffects**.

For each predictor, the user can specify whether they want predictions at specific values or at "typical" values, which are defined in **clarify** as the mode for unordered categorical and binary variables, the median for ordered categorical variables, and the mean for continuous variables. Predictions for multiple predictor combinations can be requested by specifying values that will be used to create a grid of predictor values, or the grid itself can be supplied as a data frame of desired predictor profiles. In addition, the "first difference", defined here as the difference between predictions for two predictor combinations, can be computed.

The arguments to `sim_setx()` are as follows:

```
sim_setx(sim = , x = , x1 = , outcome = , type = , verbose = , cl = )
```

- `sim` – a `clarify_sim` object; the output of a call to `sim()`.
- `x` – a named list containing the requested values of the predictors, e.g., `list(v1 = 1:4, v2 = "A")`, or a data frame containing the desired profiles. Any predictors not included will be set at their "typical" value as defined above.
- `x1` – an optional named list or data frame similar to `x` except with the value of one predictor changed. When specified, the first difference is computed between the covariate combination defined in `x` (and only one combination is allowed when `x1` is specified) and the covariate combination defined in `x1`.
- `outcome` – a string containing the name of the outcome of interest when a multivariate (multiple outcome) model is supplied to `sim()` or the outcome category of interest when a multinomial model is supplied to `sim()`. For univariate (single outcome) and binary outcomes, this is ignored.
- `type` – a string containing the type of predicted value to return. In most cases, this can be left unspecified to request predictions on the scale of the outcome.
- `verbose` – whether to display a progress bar.
- `cl` – an argument that controls parallel processing, which can be the number of cores to use or a cluster object resulting from `parallel::makeCluster()`.

Here, we will use `sim_setx()` to examine predicted values of the outcome for control and treated units, at `re75` set to 0 and 20000, and `race` set to "black".

```
est3 <- sim_setx(s,
                 x = list(treat = 0:1,
                          re75 = c(0, 20000),
                          race = "black"),
                 verbose = FALSE)
```

When we use `summary()` on the resulting output, we can see the estimates and their uncertainty intervals (calculated using quantiles by default).

```
summary(est3)
```

```
#>                          Estimate 2.5 % 97.5 %
#> treat = 0, re75 = 0         0.667 0.558  0.771
#> treat = 1, re75 = 0         0.712 0.618  0.790
#> treat = 0, re75 = 20000     0.938 0.701  0.993
#> treat = 1, re75 = 20000     0.953 0.751  0.996
```

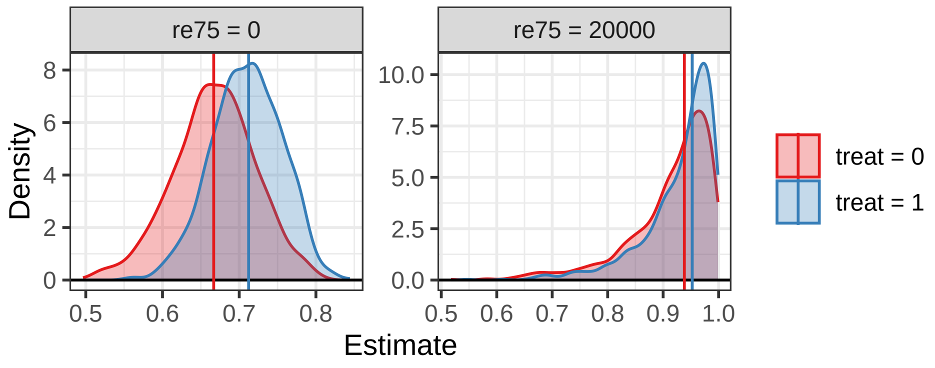
To see the complete grid of the predictor values used in the predictions, which helps to identify the "typical" values of the other predictors, we can access the `"setx"` attribute of the object:

```
attr(est3, "setx")
```

```
#>                           treat married      age     educ  race nodegree     re74
#> treat = 0, re75 = 0           0       0 27.36319 10.26873 black        1 4557.547
#> treat = 1, re75 = 0           1       0 27.36319 10.26873 black        1 4557.547
#> treat = 0, re75 = 20000       0       0 27.36319 10.26873 black        1 4557.547
#> treat = 1, re75 = 20000       1       0 27.36319 10.26873 black        1 4557.547
#>                            re75
#> treat = 0, re75 = 0           0
#> treat = 1, re75 = 0           0
#> treat = 0, re75 = 20000 20000
#> treat = 1, re75 = 20000 20000
```

We can plot the distributions of the simulated values using plot(), which also separates the predictions by the predictor values (it is often clearer without the uncertainty bounds). The var argument controls which variable is used for faceting the plots.

```
plot(est3, var = "re75", ci = FALSE)
```



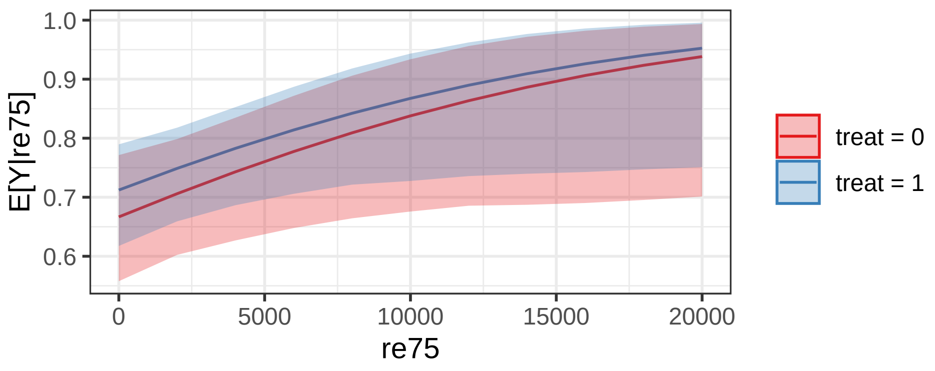
One can see again how a delta method or Normal approximation may not have yielded valid uncertainty intervals given the non-Normality of the distributions.

If a continuous variable with many levels is included in the grid of the predictors, something like a dose-response function for a typical unit can be generated. Below, we set re75 to vary from 0 to 20000 in steps of 2000.

```
est4 <- sim_setx(s,
                 x = list(treat = 0:1,
                          re75 = seq(0, 20000, by = 2000),
                          race = "black"),
                 verbose = FALSE)
```

When we plot the output, we can see how the predictions varies across the levels of re75:

```
plot(est4)
```



We will return to display average dose-response functions using sim_adrf() later.

Finally, we can use `sim_setx()` to compute first differences, the contrast between two covariate combinations. We supply one covariate profile to `x` and another to `x1`, and `sim_setx()` simulates the two predicted values and their difference. Below, we simulate first difference for a treated and control unit who have `re75` of 0 and typical values of all other covariates:

```
est5 <- sim_setx(s,
                 x = list(treat = 0, re75 = 0),
                 x1 = list(treat = 1, re75 = 0),
                 verbose = FALSE)
```

When we use `summary()`, we see the estimates for the predicted values and their first difference ("FD"):

```
summary(est5)
```

```
#>            Estimate   2.5 %  97.5 %
#> treat = 0    0.7856  0.7041  0.8557
#> treat = 1    0.8213  0.7114  0.8989
#> FD           0.0357 -0.0592  0.1187
```

It is possible to compute first differences without using `x1` using `transform()`, which we describe later.

`sim_ame()`: average adjusted predictions and average marginal effects

Using predicted values and effects at representative values is one way to summarize regression models, but another way is to compute average adjusted predictions (AAPs), contrasts of AAPs, and average marginal effects (AMEs). The definitions for these terms may vary and the names for these concepts differ across sources, but here we define AAPs as the average of the predicted values for all units after setting one predictor to a chosen value, and we define AMEs for binary predictors as the contrast of two AAPs and for continuous predictors as the average of instantaneous rate of change in the AAP corresponding to a small change in the predictor from its observed values across all units[3] (Long and Freese 2014).

The arguments to `sim_ame()` are as follows:

```
sim_ame(sim = , var = , subset = , by = , contrast = , outcome = ,
        type = , eps = , verbose = , cl = )
```

- `sim` – a `clarify_sim` object; the output of a call to `sim()`.
- `var` – the name of focal variable over which to compute the AAPs or AMEs, or a list containing the values for which AAPs should be computed.
- `subset` – a logical vector, evaluated in the original dataset used to fit the model, defining a subset of units for which the AAPs or AMEs are to be computed.
- `by` – the name of one or more variables for which AAPs should be computed within subgroups. Can be supplied as a character vector of variable names or a one-sided formula.
- `contrast` – the name of an effect measure used to contrast AAPs. For continuous outcomes, `"diff"` requests the difference in means, but others are available for binary outcomes, including `"rr"` for the risk ratio, `"or"` for the odds ratio, and `"nnt"` for the number needed to treat, among others. If not specified, only AAPs will be computed if the variable named in `var` is categorical or specific values of the focal variable are specified in `var`. Ignored when the variable named in `var` is continuous and no specific values are specified because the AME is the only quantity computed. When `var` names a multi-category categorical variable, `contrast` cannot be used; see the section describing `transform()` for computing contrasts with them.
- `outcome` – a string containing the name of the outcome of interest when a multivariate (multiple outcome) model is supplied to `sim()` or the outcome category of interest when a multinomial model is supplied to `sim()`. For univariate (single outcome) and binary outcomes, this is ignored.

[3]In **marginaleffects**, AAPs are computed using `avg_predictions()`, AMEs for binary variables are computed using `avg_comparisons()`, and AMEs for continuous variables are computed using `avg_slopes()`. AAPs are sometimes known as average "counterfactual" predictions.

- `type` – a string containing the type of predicted value to return. In most cases, this can be left unspecified to request predictions on the scale of the outcome (e.g., probabilities for binary outcomes).
- `eps` – the value by which the observed values of the variable named in `var` are changed when it is continuous to compute the AME. This usually does not need to be specified.
- `verbose` – whether to display a progress bar.
- `cl` – an argument that controls parallel processing, which can be the number of cores to use or a cluster object resulting from `parallel::makeCluster()`.

Here, we will use `sim_ame()` to compute the AME of `treat` just among those who were treated (in causal inference, this is known as the average treatment effect in the treated, or ATT (Greifer and Stuart 2023)). We will request our estimate to be on the risk ratio scale.

```
est6 <- sim_ame(s,
                var = "treat",
                subset = treat == 1,
                contrast = "rr",
                verbose = FALSE)
```

We can use `summary()` to display the estimates and their uncertainty intervals. Here, we will also use `null` to include a test for the null hypothesis that the risk ratio is equal to 1.

```
summary(est6, null = c(`RR` = 1))
```

```
#>         Estimate 2.5 % 97.5 % P-value
#> E[Y(0)]    0.687 0.610  0.759       .
#> E[Y(1)]    0.755 0.686  0.808       .
#> RR         1.100 0.949  1.255    0.21
```

Here we see the estimates for the AAPs, `E[Y(0)]` for the expected value of the outcome setting `treat` to 0 and `E[Y(1)]` for the expected value of the outcome setting `treat` to 1, and the risk ratio `RR`. The p-value on the test for the risk ratio aligns with the uncertainty interval containing 1.

If we instead wanted the risk difference or odds ratio, we would not have to re-compute the AAPs. Instead, we can use `transform()` to compute a new derived quantity from the computed AAPs. The section on `transform()` demonstrates this.

We can compute the AME for a continuous predictor. Here, we will consider `age` (just for demonstration; this analysis does not have a valid interpretation).

```
est7 <- sim_ame(s,
                var = "age",
                verbose = FALSE)
```

We can use `summary()` to display the AME estimate and its uncertainty interval.

```
summary(est7)
```

```
#>             Estimate    2.5 %   97.5 %
#> E[dY/d(age)] -0.00605 -0.00940 -0.00259
```

The AME is named `E[dY/d(age)]`, which signifies that a derivative has been computed (more precisely, the average of the unit-specific derivatives). This estimate can be interpreted like a slope in a linear regression model, but as a single summary of the effect of a predictor it is often too coarse to capture nonlinear relationships. The section below explains how to compute average dose-response functions for continuous predictors, which provide a more complete picture of their effects on an outcome.

Below, we will examine effect modification of the ATT by the predictor `married` using the `by` argument to estimate AAPs and their ratio within levels of `married`:

```
est6b <- sim_ame(s,
                 var = "treat",
                 subset = treat == 1,
                 by = ~married,
                 contrast = "rr",
                 verbose = FALSE)

summary(est6b)

#>           Estimate 2.5 % 97.5 %
#> E[Y(0)|0]    0.691 0.613  0.768
#> E[Y(1)|0]    0.733 0.655  0.796
#> RR[0]        1.061 0.911  1.232
#> E[Y(0)|1]    0.668 0.556  0.768
#> E[Y(1)|1]    0.848 0.677  0.940
#> RR[1]        1.270 0.952  1.579
```

The presence of effect modification can be tested by testing the contrast between the effects computed within each level of the by variable; this demonstrated in the section on `transform()` below.

`sim_adrf()`: average dose-response functions

A dose-response function for an individual is the relationship between the set value of a continuous focal predictor and the expected outcome. The average dose-response function (ADRF) is the average of the dose-response functions across all units. Essentially, it is a function that relates the value of the predictor to the corresponding AAP of the outcome, the average value of the outcome were all units to be set to that level of the predictor. ADRFs can be used to provide additional detail about the effect of a continuous predictor beyond a single AME.

A related quantity is the average marginal effect function (AMEF), which describes the relationship between a continuous focal predictor and the AME at that level of the predictor. That is, rather than describing how the outcome changes as a function of the predictor, it describes how the *effect* of the predictor on the outcome changes as a function of the predictor. It is essentially the derivative of the ADRF and can be used to identify at which points along the ADRF the predictor has an effect.

The ADRF and AMEF can be computed using `sim_adrf()`. The arguments are below:

```
sim_adrf(sim = , var = , subset = , by = , contrast = , at = ,
         n = , outcome = , type = , eps = , verbose = , cl = )
```

- `sim` – a `clarify_sim` object; the output of a call to `sim()`.
- `var` – the name of focal variable over which to compute the ADRF or AMEF.
- `subset` – a logical vector, evaluated in the original dataset used to fit the model, defining a subset of units for which the ARDF or AMEF is to be computed.
- `by` – the name of one or more variables for which the ADRF or AMEF should be computed within subgroups. Can be supplied as a character vector of variable names or a one-sided formula.
- `contrast` – either `"adrf"` or `"amef"` to request the ADRF or AMEF, respectively. The default is to compute the ADRF.
- `at` – the values of the focal predictor at which to compute the ADRF or AMEF. This should be a vector of values that the focal predictor can take on. If unspecified, a vector of `n` (see below) equally-spaced values from the minimum to the maximum value of the predictor will be used. This should typically be used only if quantities are desired over a subset of the values of the focal predictor.
- `n` – if `at` is unspecified, the number of points along the range of the focal predictor at which to compute the ADRF or AMEF. More yields smoother functions, but will take longer and require more memory. The default is 21.
- `outcome` – a string containing the name of the outcome of interest when a multivariate (multiple outcome) model is supplied to `sim()` or the outcome category of interest when a multinomial model is supplied to `sim()`. For univariate (single outcome) and binary outcomes, this is ignored.
- `type` – a string containing the type of predicted value to return. In most cases, this can be left unspecified to request predictions on the scale of the outcome.
- `eps` – the value by which the observed values of the variable named in `var` are changed when it is continuous to compute the AMEF. This usually does not need to be specified.

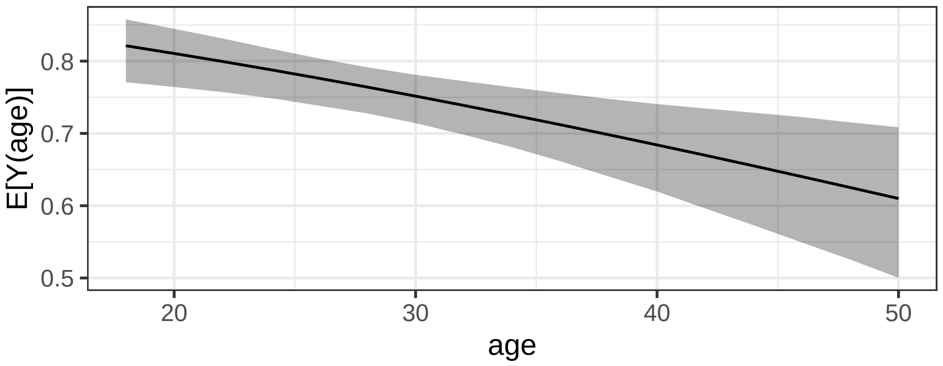
- `verbose` – whether to display a progress bar.
- `cl` – an argument that controls parallel processing, which can be the number of cores to use or a cluster object resulting from `parallel::makeCluster()`.

Here, we will consider age (just for demonstration; this analysis does not have a valid interpretation) and compute the ADRF and AMEF of age on the outcome. We will only examine ages between 18 and 50, even though the range of age goes slightly beyond these values. First, we will compute the ADRF of age, which examines how the outcome would vary on average if one set all unit's value of age to each value between 18 and 50 (here we only use even ages to speed up computation).

```
age_seq <- seq(18, 50, by = 2)

est8 <- sim_adrf(s,
                 var = "age",
                 contrast = "adrf",
                 at = age_seq,
                 verbose = FALSE)
```

We can plot the ADRF using `plot()`.

```
plot(est8)
```



From the plot, we can see that as age increases, the expected outcome decreases.

We can also examine the AAPs at the requested ages using `summary()`, which will display all the estimated AAPs by default, so we will request just the first 4 (ages 18 to 24):

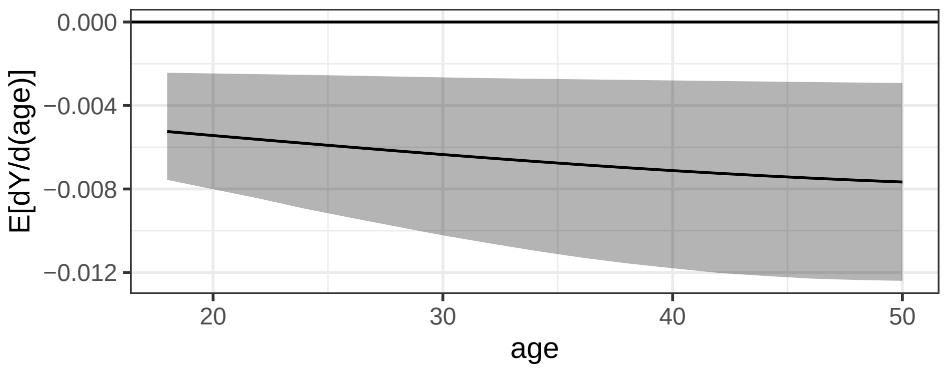
```
summary(est8, parm = 1:4)

#>          Estimate 2.5 % 97.5 %
#> E[Y(18)]    0.821 0.771  0.858
#> E[Y(20)]    0.811 0.764  0.845
#> E[Y(22)]    0.800 0.757  0.831
#> E[Y(24)]    0.788 0.749  0.817
```

Next we will compute the AMEF, the effect of age at each level of age.

```
est9 <- sim_adrf(s,
                 var = "age",
                 contrast = "amef",
                 at = age_seq,
                 verbose = FALSE)
```

We can plot the AMEF using `plot()`:

```
plot(est9)
```



From the plot, we can see the AME of age decreases slightly but is mostly constant across values of age, and the uncertainty intervals for the AMEs consistently exclude 0.

Transforming and Combining Estimates

Often, our quantities of interest are not just the outputs of the functions above, but comparisons between them. For example, to test for moderation of a treatment effect, we may want to compare AMEs in multiple groups defined by the moderator. Or, it might be that we are interested in an effect described using a different effect measure than the one originally produced; for example, we may decide we want the risk difference AME after computing the risk ratio AME. The functions `transform()` and `cbind()` allow users to transform quantities in a single `clarify_est` object and combine two `clarify_est` objects. These are essential for computing quantities that themselves are derived from the derived quantities computed by the `sim_*()` functions.

`transform()`

`transform()` is a generic function in R that is typically used to create a new variable in a data frame that is a function of other columns. For example, to compute the binary outcome we used in our model, we could have run the following[4]:

```
lalonde <- transform(lalonde,
                     re78_0 = ifelse(re78 == 0, 1, 0))
```

Similarly, to compute a derived or transformed quantity from a `clarify_est` object, we can use `transform()`. Here, we will compute the risk difference AME of `treat`; previously, we used `sim_ame()` to compute the AAPs and the risk ratio.

```
est6 <- transform(est6,
                  RD = `E[Y(1)]` - `E[Y(0)]`)
```

Note that we used tics (‘) around the names of the AAPs; this is necessary when they contain special characters like parentheses or brackets.

When we run `summary()` on the output, the new quantity, which we named "RD", will be displayed along with the other estimates. We will also set a null value for this quantity.

```
summary(est6, null = c(`RR` = 1, `RD` = 0))

#>         Estimate   2.5 %  97.5 % P-value
#> E[Y(0)]   0.6866  0.6095  0.7590       .
#> E[Y(1)]   0.7551  0.6859  0.8081       .
#> RR        1.0998  0.9492  1.2553    0.21
#> RD        0.0685 -0.0380  0.1574    0.21
```

One benefit of using simulation-based inference with p-values computed from inverting the confidence intervals is that the p-values for the risk difference and risk ratio (and any other effect

[4]Users familiar with the `tidyverse` will note the similarities between `transform()` and `dplyr::mutate()`; only `transform()` can be used with `clarify_est` objects.

measure for comparing a pair of values) will always exactly align, thereby ensuring inference does not depend on the effect measure used. In contrast, Wald-type inference (based either on the simulation-derived standard error or the delta method) is not invariant to transformations of the quantity of interest.

The same value would be computed if we were to have called `sim_ame()` on the same `clarify_sim` object and requested the risk difference using `contrast = "diff"`; using `transform()` saves time because the AAPs are already computed and stored in the `clarify_est` object.

We can use `transform()` along with the `by` variable in `sim_ame()` to compute the contrast between quantities computed within each subgroup of `married`. Previously we used `by` to compute the risk ratio ATT within levels of `married`; here we will compute the ratio of these risk ratios to assess the presence of effect modification.

```
est6b |>
  transform(RR_ratio = `RR[1]` / `RR[0]`) |>
  summary(parm = c("RR[0]", "RR[1]", "RR_ratio"),
          null = 1)

#>          Estimate 2.5 % 97.5 % P-value
#> RR[0]       1.061 0.911  1.232   0.434
#> RR[1]       1.270 0.952  1.579   0.094 .
#> RR_ratio    1.196 0.910  1.514   0.174
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

`RR_ratio` contains the ratio of the risk ratios for `married = 1` and `married = 0`. Here we also include a test for whether each of the risk ratios and their ratio differ from 1.

cbind()

`cbind()` is another generic R function that is typically used to combine two or more datasets columnwise (i.e., to widen a dataset). In **clarify**, `cbind()` can be used to combine two `clarify_est` objects so that the estimates can be examined jointly and so that it is possible to compare them directly. For example, if we were to compute AMEs in two subgroups using `subset` and wanted to compare them, we would call `sim_ame()` twice, one for each subset (though in practice it is more effective to use `by`; this is just for illustration), as demonstrated below:

```
# AME of treat with race = "black"
est10b <- sim_ame(s, var = "treat", subset = race == "black",
                  contrast = "diff", verbose = FALSE)
summary(est10b)

#>         Estimate   2.5 %  97.5 %
#> E[Y(0)]   0.6677  0.5814  0.7527
#> E[Y(1)]   0.7439  0.6663  0.8007
#> Diff      0.0762 -0.0353  0.1694

# AME of treat with race = "hispan"
est10h <- sim_ame(s, var = "treat", subset = race == "hispan",
                  contrast = "diff", verbose = FALSE)
summary(est10h)

#>         Estimate   2.5 %  97.5 %
#> E[Y(0)]   0.8266  0.7154  0.8972
#> E[Y(1)]   0.8971  0.7892  0.9526
#> Diff      0.0704 -0.0215  0.1382
```

Here, we computed the risk difference for the subgroups `race = "black"` and `race = "hispan"`. If we wanted to compare the risk differences, we could combine them and compute a new quantity equal to their difference. We will do that below.

First, we need to rename to quantities in each object so they do not overlap; we can do so using `names()`, which has a special method for `clarify_est` objects.

```
names(est10b) <- paste(names(est10b), "b", sep = "_")
names(est10h) <- paste(names(est10h), "h", sep = "_")
```

Next, we use cbind() to bind the objects together.

```
est10 <- cbind(est10b, est10h)
summary(est10)

#>            Estimate   2.5 %  97.5 %
#> E[Y(0)]_b    0.6677  0.5814  0.7527
#> E[Y(1)]_b    0.7439  0.6663  0.8007
#> Diff_b       0.0762 -0.0353  0.1694
#> E[Y(0)]_h    0.8266  0.7154  0.8972
#> E[Y(1)]_h    0.8971  0.7892  0.9526
#> Diff_h       0.0704 -0.0215  0.1382
```

Finally, we can use transform() to compute the difference between the risk differences:

```
est10 <- transform(est10,
                   `Dh - Db` = Diff_h - Diff_b)
summary(est10, parm = "Dh - Db")

#>         Estimate    2.5 %   97.5 %
#> Dh - Db -0.00575 -0.06788  0.04102
```

Importantly, cbind() can only be used to join together clarify_est objects computed using the same simulated coefficients (i.e., resulting from the same call to sim()). This preserves the covariance among the estimated quantities, which is critical for valid inference. That is, sim() should only be called once per model, and all derived quantities should be computed using its output.

Using clarify with Multiply Imputed Data

Multiple imputation is a popular method of estimating quantities of interest in the presence of missing data and involves creating multiple versions of the original dataset each with the missing values imputed with estimates from an imputation model. Simulation-based inference in multiply imputed data is relatively straightforward. Simulated coefficients are drawn from the model estimated in each imputed dataset separately, and then the simulated coefficients are pooled into a single set of simulated coefficients. In Bayesian terms, this would be considered "mixing draws" and is the recommended approach for Bayesian analysis with multiply imputed data (Zhou and Reiter 2010).

Using clarify with multiply imputed data is simple. Rather than using sim(), we use the function misim(). misim() functions just like sim() except that it takes in a list of model fits (i.e., containing a model fit to each imputed dataset) or an object containing such a list (e.g., a mira object from mice::with() or a mimira object from MatchThem::with()). misim() simulates coefficient distributions within each imputed dataset and then appends them together to a form a single combined set of coefficient draws.

sim_apply() and its wrappers accept the output of misim() and compute the desired quantity using each set of coefficients. When these functions rely on using a dataset (e.g., sim_ame(), which averages predicted outcomes across all units in the dataset used to fit the model), they automatically know to associate a given coefficient draw with the imputed dataset that was used to fit the model that produced that draw. In user-written functions supplied to the FUN argument of sim_apply(), it is important to correctly extract the dataset from the model fit. This is demonstrated below.

The final estimates of the quantity of interest is computed as the mean of the estimates computed in each imputed dataset (i.e., using the original coefficients, not the simulated ones), which is the same quantity that would be computed using standard pooling rules. This is not always valid for noncollapsible estimates, like ratios, and so care should be taken to ensure the mean of the resulting estimates has a valid interpretation (this is related to the transformation-induced bias described by Rainey (2017)).

The arguments to misim() are as follows:

```
misim(fitlist = , n = , vcov = , coefs = , dist = )
```

- fitlist – a list of model fits or an accepted object containing them (e.g., a mira object from mice::with())

- n – the number of simulations to run *for each imputed dataset*. The default is 1000, but fewer can be used because the total number of simulated quantities will be m * n, where m is the number of imputed datasets.
- vcov, coefs, dist – the same as with sim(), except that a list of such arguments can be supplied to be applied to each imputed dataset.

Below we illustrate using misim() and sim_apply() with multiply imputed data. We will use the africa dataset from the **Amelia** package.

```
library(Amelia)
data("africa", package = "Amelia")

# Multiple imputation
a.out <- amelia(x = africa, m = 10, cs = "country",
                ts = "year", logs = "gdp_pc", p2s = 0)

# Fit model to each dataset
model.list <- with(a.out, lm(gdp_pc ~ infl * trade))

# Simulate coefficients, 100 draws per imputation
si <- misim(model.list, n = 100)

si
```

```
#> A `clarify_misim` object
#>  - 4 coefficients, 10 imputations with 100 simulated values each
#>  - sampled distributions: multivariate t(116)
```

The function we will be applying to each imputed dataset will be one that computes the average marginal effect of infl. (We will run the same analysis afterward using sim_ame().)

```
sim_fun <- function(fit) {
  #Extract the original dataset using get_predictors()
  X <- insight::get_predictors(fit)

  p0 <- predict(fit)

  #Predictions after perturbing infl slightly
  p1 <- predict(fit, newdata = transform(X, infl = infl + 1e-5))

 c(AME = mean((p1 - p0) / 1e-5))
}

est_mi <- sim_apply(si, FUN = sim_fun, verbose = FALSE)

summary(est_mi)
```

```
#>     Estimate 2.5 % 97.5 %
#> AME    -5.75 -8.82  -2.27
```

Note that sim_apply() "knows" which imputation produced each set of simulated coefficients, so using insight::get_predictors() on the fit supplied to sim_fun() will use the right dataset. Care should be taken when analyses restrict each imputed dataset in a different way (e.g. when matching with a caliper in each one), as the resulting imputations may not refer to a specific target population and mixing the draws may be invalid.

Below, we can use sim_ame():

```
est_mi2 <- sim_ame(si, var = "infl", verbose = FALSE)

summary(est_mi2)
```

```
#>               Estimate 2.5 % 97.5 %
#> E[dY/d(infl)]    -5.75 -8.82  -2.27
```

We get the same results, as expected.

Note that `misim()` is compatible with model fit objects from **mice**, **Amelia**, **MatchThem**, and any other package that produces a list of model fit objects with each corresponding to the output of a model fit to an imputed dataset.

Comparison to Other Packages

Several packages offer methods for computing interpretable quantities form regression models, including **emmeans**, **margins**, **modelbased**, and **marginaleffects**. Many of the quantities computed by these packages can also be computed by **clarify**, the primary difference being that **clarify** uses simulation-based inference rather than delta method-based inference.

marginaleffects offers the most similar functionality to **clarify**, and **clarify** depends on functionality provided by **marginaleffects** to accommodate a wide variety of regression models. **marginaleffects** also offers simulation-based inference using `marginaleffects::inferences()` and support for arbitrary user-specified post-estimation functions using `marginaleffects::hypotheses()`. However, **clarify** and `marignalefefcts` differ in several ways. The largest difference is that **clarify** supports iterative building of more and more complex hypotheses through the `transform()` method, which quickly computes new quantities and transformation from the existing computed quantities, whereas **marginaleffects** only supports a single transformation and, as of version 0.13.0, cannot use simulation-based inference for these quantities.

Because of **clarify**'s focus on simulation, it provides functionality directly aimed at improving simulation-based inference, including plots to assess the normality of the distributions of simulated values (important for assessing whether Wald-type confidence intervals and p-values are valid) and support for parallel processing. **clarify** also provides support for simulation-based inference of multiply imputed data, which does not require any special pooling rules and does not require normality of the derived quantities.

There are areas and cases where **marginaleffects** may be the better choice than **clarify** or where the differences between the packages are of little consequence. **marginaleffects** focuses on providing a complete framework for post-estimation using model predictions, whereas **clarify** is primarily focused on supporting user-defined functions, with commonly used estimators offered as a convenience. In cases where the delta method is an acceptable approximation (e.g., for quantities computed from linear models or other quantities known to be approximately normally distributed in finite samples), using the delta method through **marginaleffects** will be much faster, more accurate, and more replicable than the simulation-based inference **clarify** provides. For the quantities easily computed by **marginaleffects** that support simulation-based inference through `marginaleffects::inferences()`, using **marginaleffects** can provide a more familiar and flexible syntax than **clarify** might offer. Ultimately, the user should use the package that supports their desired syntax and mode of inference.

Conclusion

clarify provides functionality to facilitate simulation-based inference of deriving quantities from regression models. This framework avoids some of the approximations of the more common delta method, in particular the first-order approximation to the asymptotic variance and the approximation that the sampling distribution of the derived quantity is Normal. **clarify** supports simulation-based inference of arbitrary derived quantities as well as common ones including average adjusted predictions, average marginal effects, and predictions at representative values. These enhance the usefulness and interpretability of regression models without requiring analysts to sacrifice parameterization for interpretability.

References

Dehejia, Rajeev H., and Sadek Wahba. 1999. "Causal Effects in Nonexperimental Studies: Reevaluating the Evaluation of Training Programs." *Journal of the American Statistical Association* 94 (448): 1053–62. https://doi.org/10.1080/01621459.1999.10473858.

Greifer, Noah, and Elizabeth A. Stuart. 2023. "Choosing the Causal Estimand for Propensity Score Analysis of Observational Studies." https://doi.org/10.48550/arXiv.2106.10577.

Ho, Daniel E., Kosuke Imai, Gary King, and Elizabeth A. Stuart. 2011. "MatchIt: Nonparametric Preprocessing for Parametric Causal Inference." *Journal of Statistical Software, Articles* 42 (8): 128. https://doi.org/10.18637/jss.v042.i08.

Imai, Kosuke, Gary King, and Olivia Lau. 2008. "Toward a Common Framework for Statistical Analysis and Development." *Journal of Computational and Graphical Statistics* 17 (4): 892–913.

https://doi.org/10.1198/106186008X384898.

King, Gary, Michael Tomz, and Jason Wittenberg. 2000. "Making the Most of Statistical Analyses: Improving Interpretation and Presentation." *American Journal of Political Science* 44 (2): 347–61. https://doi.org/10.2307/2669316.

Long, J. Scott, and Jeremy Freese. 2014. *Regression Models for Categorical Dependent Variables Using Stata*. Third edition. College Station, Texas: Stata Press Publication, StataCorp LP.

Rainey, Carlisle. 2017. "Transformation-Induced Bias: Unbiased Coefficients Do Not Imply Unbiased Quantities of Interest." *Political Analysis* 25 (3): 402–9. https://doi.org/10.1017/pan.2017.11.

———. 2023. "A Careful Consideration of CLARIFY: Simulation-Induced Bias in Point Estimates of Quantities of Interest." *Political Science Research and Methods*, April, 1–10. https://doi.org/10.1017/psrm.2023.8.

Tomz, Michael, Jason Wittenberg, and Gary King. 2003. "Clarify: Software for Interpreting and Presenting Statistical Results." *Journal of Statistical Software* 8 (January): 1–30. https://doi.org/10.18637/jss.v008.i01.

Zhou, Xiang, and Jerome P. Reiter. 2010. "A Note on Bayesian Inference After Multiple Imputation." *The American Statistician* 64 (2): 159–63. https://doi.org/10.1198/tast.2010.09109.

Noah Greifer
Harvard University
Institute for Quantitative Social Science
Cambridge, MA
ngreifer.github.io
ORCiD: 0000-0003-3067-7154
ngreifer@iq.harvard.edu

Steven Worthington
Harvard University
Institute for Quantitative Social Science
Cambridge, MA
ORCiD: 0000-0001-9550-5797
sworthington@iq.harvard.edu

Stefano Iacus
Harvard University
Institute for Quantitative Social Science
Cambridge, MA
ORCiD: 0000-0002-4884-0047
siacus@iq.harvard.edu

Gary King
Harvard University
Institute for Quantitative Social Science
Cambridge, MA
ORCiD: 0000-0002-5327-7631
king@harvard.edu