Distributions in a Regression Context

Ashley I Naimi

Summer 2024

Contents

Often, we use regression models to analyze data, and so they are typically implemented in some way in a simulation setting. Here, we'll explore how to integrate some of the distribution functions in the last chapter into a regression modeling framework. We'll start with some very simple regression modeling frameworks and work our way up in complexity.

1 Regression Models and Distributions for Simulation: Simple Continuous

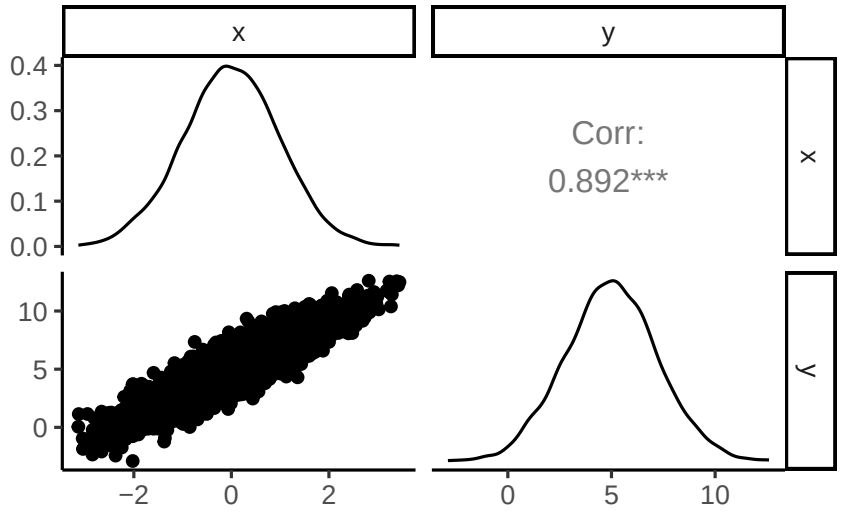
The simplest regression framework we can simulate involves two normally distributed random variables:

```
set.seed(123)

n = 5000

x <- rnorm(n, mean = 0, sd = 1)

y <- 5 + 2*x + rnorm(n, mean = 0, sd = 1)

a <- data.frame(x, y)

head(a)
```

```
##             x        y
## 1 -0.56047565 3.384875
## 2 -0.23017749 5.667238
## 3  1.55870831 6.970467
## 4  0.07050839 6.622035
## 5  0.12928774 6.174767
## 6  1.71506499 8.765261
```

We can explore these data using standard methods:

```
GGally::ggpairs(a)
```




```
summary(a$y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -2.892   3.527   5.007   4.995   6.460  12.599
```

```
summary(a$x)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -3.13739 -0.65514 -0.00757 -0.00057  0.66098  3.44599
```

Because of how we defined the outcome in the simulation, we can analyze these data using a few methods. For example, via `lm()` or `glm()`:

```
mod1 <- lm(y ~ x, data = a)

# mod1 <- glm(y ~ x, data = a, family = gaussian(link = "identity"))

summary(mod1)$coefficients
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 4.995823 0.01418265 352.249        0
## x           1.993971 0.01426149 139.815        0
```

We can see that the coefficients from this model align almost exactly with the values we used to simulate the outcome variable y.

2 Regression Models and Distributions for Simulation: Binary and Continuous

How should we simulate a binary random variable from a regression model? We can use a logistic regression model to do this. We won't explain all of this now, but will have an opportunity to look at each of these elements to get a sense of how this code works and why:

```
# define the inverse logistic function
expit <- function(x){
  1/(1 + exp(-x))
}

set.seed(123)

n = 5000

z <- rnorm(n, mean = 0, sd = 1)

x <- rbinom(n, size = 1, p = expit(-1 + log(2)*z))

y <- 100 + 10*x + 3*z + rnorm(n, mean = 0, sd = 10)

# use these variables to construct a dataset:

a <- data.frame(z, x, y)
```

The above code gives us a dataset of 5000 observations with: one continuous covariate z, one binary exposure x, and one continuous outcome y:

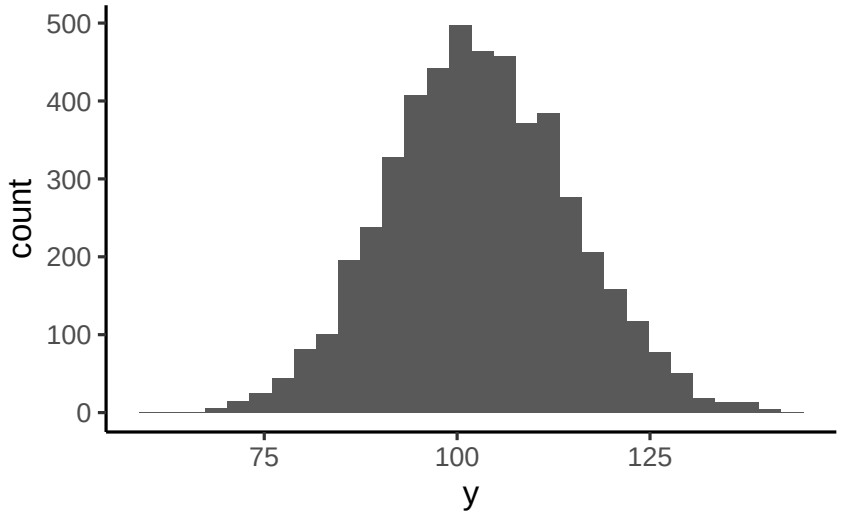
```
head(a)
```

```
##            z x         y
```

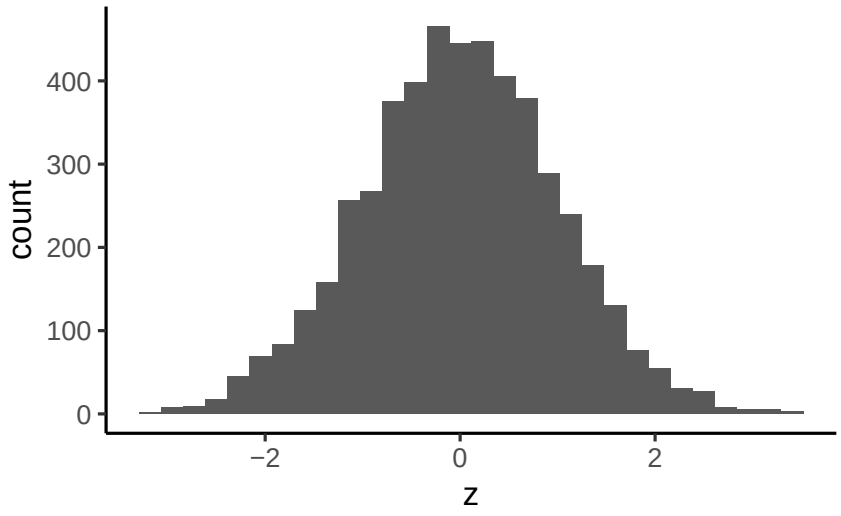
```
## 1 -0.56047565 0 101.81860
## 2 -0.23017749 0 107.45388
## 3  1.55870831 0  99.50946
## 4  0.07050839 0  73.28888
## 5  0.12928774 0  89.41832
## 6  1.71506499 1 102.59044
```

We can do some basic analyses of these data:

```
ggplot(a) + geom_histogram(aes(x = y))
```




```
ggplot(a) + geom_histogram(aes(x = z))
```



```r
table(a$x)
```

```
##
##    0    1
## 3517 1483
```

```r
mean(a$x)
```

```
## [1] 0.2966
```

Again, we can use a simple regression model to analyze these data, such as:

```r
# mod1 <- lm(y ~ x, data = a)

mod1 <- glm(y ~ x + z, data = a, family = gaussian(link = "identity"))

summary(mod1)$coefficients
```

```
##               Estimate Std. Error   t value      Pr(>|t|)
## (Intercept) 99.716726  0.1726271 577.64242  0.000000e+00
## x           10.709244  0.3269832  32.75166 2.603927e-213
## z            2.604083  0.1501827  17.33944  1.888766e-65
```

Let's unpack this model. We can formulate it as a generalized linear model with a Gaussian distribution and identity link function:

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

where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$

Notice that this equation corresponds to the following model code:

```r
mod1 <- glm(y ~ x + z, data = a, family = gaussian(link = "identity"))

summary(mod1)
```

Notice also how this aligned with the code we used to simulate the outcome:

```
y <- 100 + 10 * x + 3 * z + rnorm(n, mean = 0, sd = 10)
```

What about the code that we used to simulate the exposure? This is the propensity score model, and we used logistic regression, defined as:

$$\text{logit}\, P(X = 1 \mid Z) = \alpha_0 + \alpha_1 Z$$

which is equivalent to:

$$P(X = 1 \mid Z) = \text{expit}\{\alpha_0 + \alpha_1 Z\}$$

In our code, the values of α_0, α_1 were set to -1 and $\log(2)$, respectively. Importantly, the "logit" and "expit" functions are defined as:

$$\text{logit}\, P(\bullet) = \frac{P(\bullet)}{1 - P(\bullet)}$$

$$\text{expit}(\bullet) = \frac{1}{[1 + \exp(-\bullet)]}$$

These functions are inversely related, meaning:

$$P(\bullet) = \text{expit}\left(\text{logit}\, P(\bullet)\right)$$

We can fit this model using the GLM routines in R. For example, if we wanted to fit a propensity score model to the `a` data, we might use the following code:

```
a$propensity_score <- glm(x ~ z, data = a, family = binomial("logit"))$fitted.values
```

While this is harder to see, it is aligned with the code we used to simulate the exposure. Specifically:

```
x <- rbinom(n, size = 1, p = expit(-1 + log(2)*z))
```

Note that this code simulates a random variable X from the Bernoulli distribution (`rbinom` with `size = 1`), where `p` is defined as the inverse of the

logit of the regression model. We can write this as:

$$\log\left[\frac{P(X=1 \mid Z)}{1-P(X=1 \mid Z)}\right] = -1 + \log(2) \times Z$$

Or, we can also write this as:

$$P(X=1 \mid Z) = \frac{1}{[1+\exp(-[-1+\log(2)\times Z])]}$$

This $P(X = 1 \mid Z)$ is the propensity score. It's what we used in the `p =` argument of the `rbinom` function, and it's what we estimated when we fit a `glm` regressing our exposure against Z, appended with the `$fitted.values` operator.

Deeper Dive: The $\text{expit}(\bullet)$ Function

Consider the probabilities we get from the expit function above when we have specific values of Z, which is a continuous (Gaussian) random variable with mean = 0 and standard deviation = 1. When $Z = 0$, we have:

$$\frac{1}{1+\exp(-[-1])} \approx \frac{1}{1+2.718282} \approx 0.27$$

In contrast, when $Z = -1$, we have

$$\frac{1}{1+\exp(-[-1+\log(2)\times -1])} \approx \frac{1}{1+0.1839397} \approx 0.15$$

But if $Z = 1$, we have:

$$\frac{1}{1+\exp(-[-1+\log(2)\times 1])} \approx \frac{1}{1+0.7357589} \approx 0.42$$

Each of these gets resolved in the `rbinom` function above, giving us a probability bounded between [0,1] that also depends on Z.

2.1 The Balancing Intercept

Suppose we were interested in simulating data from the following logistic regression model for a binary outcome C, a binary exposure X, and a binary confounder C:

$$\text{logit}\, P(Y=1 \mid X, C) = \alpha_0 + \alpha_1 X + \alpha_2 C$$

Suppose further that we wanted to explore the impact of changing the value of α_2 (the confounder effect) on our ability to estimate the conditionally adjusted odds ratio ($\exp(\alpha_1)$).

We might construct a simulation that generates data under different parameter values for α_2. For example, $\alpha_2 \in \{\log(0.25), \log(0.5), \log(0.75), \log(1.5), \log(2), \log(3)\}$.

If we constructed this simulation, keeping everything else except α_2 the same, and found that it was more difficult to estimate α_1 at the extreme values of α_2, could we conclude that the only reason was due to varying this confounding parameter?

The answer is, not really. Why? Because when we change the value of α_2, we also naturally end up changing the marginal (overall) probability of Y. To see this, we can construct a little simulation example:

```
# define the inverse logistic function
expit <- function(x){
  1/(1 + exp(-x))
}

# set the seed
set.seed(123)

# large sample size
n = 500000

# create variable
z <- rnorm(n, mean = 0, sd = 1)

# simulate exposure
x <- rbinom(n, size = 1, p = expit(-1 + log(2)*z))

# simulate outcome using a range of different parameter values
param_list <- c(.25, .5, .8, 1, 1.5, 2, 2.5, 3)

res <- NULL
for(i in param_list){
```

```
  y0 <- rbinom(n, size = 1,
               p = expit(-1 + log(i)*x + log(2)*z))

  # compute the mean of the outcome under these different parameter values
  res <- rbind(res, mean(y0))
}

res
```

```
##          [,1]
## [1,] 0.224258
## [2,] 0.249944
## [3,] 0.275778
## [4,] 0.287458
## [5,] 0.313316
## [6,] 0.331480
## [7,] 0.346446
## [8,] 0.357740
```

So in a setting similar to this one, how can we tell if the degrading performance of an estimator is due to stronger confounding (coded in the `param_list` object), or to varying prevalence of the outcome (as captured by the `res` object)?

This is where the balancing intercept becomes useful (Rudolph et al., 2021). The basic idea behind the balancing intercept is to balance the impact of a parameter in a model with a special kind of intercept that allows us to hold the probability of the outcome fixed at a value of our choosing.

The specific way in which we do this is to replace a standard intercept (usually a number of our choosing, in the above outcome model this value is -1) with an expanded intercept that has different components to it. Generically, the balancing intercept typically looks like this:

$$-\log(1/\mu - 1) - \log(\alpha) * E(X)$$

This intercept has two parts.

The first allows us to set the desired magnitude of the outcome prevalence: $-\log(1/\mu - 1)$

The second part allows us to offset the impact of any covariates in the model on the outcome prevalence.

To see this in action, let's revisit our simple simulation:

```
# define the inverse logistic function
expit <- function(x){
  1/(1 + exp(-x))
}

set.seed(123)

n = 500000

z <- rnorm(n, mean = 0, sd = 1)

x <- rbinom(n, size = 1, p = expit(-1 + log(2)*z))

param_list <- c(.25, .5, .8, 1, 1.5, 2, 2.5, 3)

res <- NULL
for(i in param_list){
  # set the marginal outcome probability to 0.25
  y25 <- rbinom(n, size = 1,
                p = expit(-log(1/.25 - 1) - log(i)*mean(x) - log(2)*0
                          + log(i)*x + log(2)*z))

  # set the marginal outcome probability to 0.5
  y5 <- rbinom(n, size = 1,
                p = expit(-log(1/.5 - 1) - log(i)*mean(x) - log(2)*0
                          + log(i)*x + log(2)*z))

  # set the marginal outcome probability to 0.75
  y75 <- rbinom(n, size = 1,
```

```
              p = expit(-log(1/.75 - 1) - log(i)*mean(x) - log(2)*0
                        + log(i)*x + log(2)*z))

  # no control of the outcome probability
  y0 <- rbinom(n, size = 1,
               p = expit(-1 + log(i)*x + log(2)*z))

  res <- rbind(res,
               cbind(mean(y25),
                     mean(y5),
                     mean(y75),
                     mean(y0)))
}

res
```

```
##          [,1]     [,2]     [,3]     [,4]
## [1,] 0.275088 0.501214 0.724752 0.224252
## [2,] 0.268972 0.499452 0.730590 0.250594
## [3,] 0.268556 0.500750 0.731766 0.275482
## [4,] 0.269918 0.499560 0.730516 0.287148
## [5,] 0.272252 0.499564 0.726488 0.313954
## [6,] 0.276508 0.499212 0.721918 0.331800
## [7,] 0.280138 0.497270 0.717748 0.346960
## [8,] 0.282182 0.496582 0.713302 0.359108
```

One thing that's important to aknowledge here is that this balancing intercept is not exact, but is approximate.

This approximation results from the fact that, when we offset the effect of a covariate, we use the mean of the covariate to remove this effect. For example, in the code above, we use `- log(i)*mean(x)` to remove the impact of the binary X variable.

A second source of the approximation is how we handle continuous variables, such as `z` in the code above. We're again setting this to the mean of the random variable (in this case zero, included only to be explicit). If we wanted to be exact, we would have to integrate over the density of continuous variables.

3 Log-Linear and Poisson

We can also simulate from a log-linear model where the outcome is distributed following a Poisson distribution. In this case, we could define a log-linear Poisson regression model as:

$$\log E(Y \mid X) = \beta_0 + \beta_1 X$$

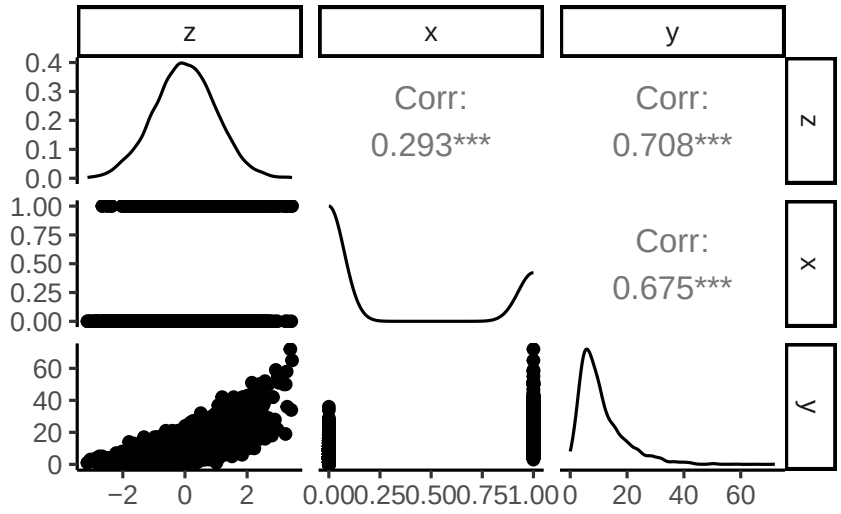
where $Y \mid X \sim Pois(\lambda)$, and where $\lambda = \exp(\beta_0 + \beta_1 X)$. In R, this could look like:

```
set.seed(123)

n = 5000

z <- rnorm(n, mean = 0, sd = 1)

x <- rbinom(n, size = 1, p = expit(-1 + log(2)*z))

y <- rpois(n, lambda = exp(2 + log(2)*x + log(1.5)*z))

# use these variables to construct a dataset:

b <- data.frame(z, x, y)

head(b)
```

```
##             z x  y
## 1 -0.56047565 0  7
## 2 -0.23017749 0  3
## 3  1.55870831 0 16
## 4  0.07050839 0  6
## 5  0.12928774 0  8
## 6  1.71506499 1 36
```

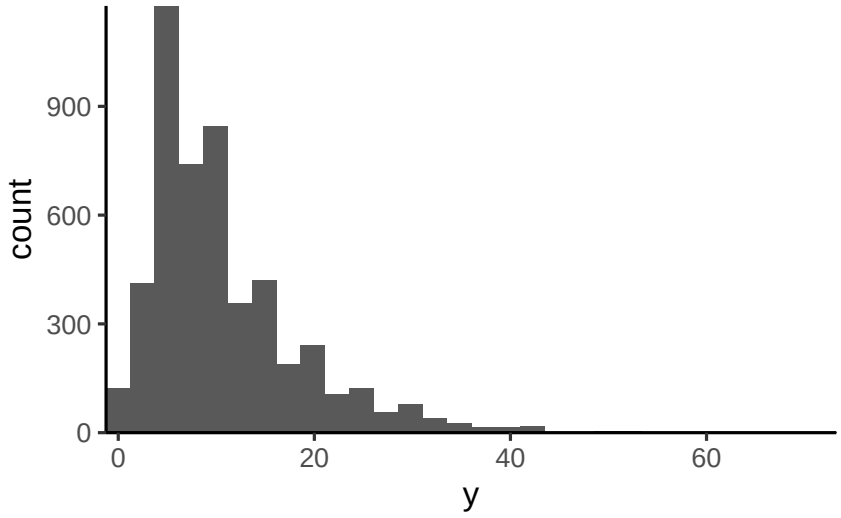
We can explore distributions in this dataset, as we would typically:

```
GGally::ggpairs(b)
```



We could more closely inspect the distribution of the outcome, which follows a Poisson distribution:

```
ggplot(b) +
  geom_histogram(aes(x = y)) +
  scale_x_continuous(expand = c(0,0)) +
  scale_y_continuous(expand = c(0,0))
```



We can then fit a regression model to these simulated data, as follows:

```
mod1 <- glm(y ~ x + z, data = b, family = poisson(link = "log"))

summary(mod1)$coefficients
```

```
##             Estimate  Std. Error   z value Pr(>|z|)
## (Intercept) 1.998889 0.006266520 318.97913        0
## x           0.689279 0.009057425  76.10099        0
## z           0.411776 0.004528943  90.92100        0
```

Again, this model demonstrates that we can recover the true values, which we used to simulate these data.

4 Marginal Standardization

Marginal standardization is equivalent to g computation (aka the parametric g formula) when the exposure is measured at a single time point (Naimi et al., 2017). This process can be implemented by fitting a single regression model, regressing the outcome against the exposure and all confounding variables. But instead of reading the coefficients the model, one can obtain parameter estimates of interest by using this model to generate predicted risks for each individual under "exposed" and "unexposed" scenarios in the dataset. To obtain standard errors, the entire procedure must be bootstrapped.

Here is some code to implement this marginal standardization in the above dataset:

```
library(boot)

#' Regress the outcome against the exposure and covariate
ms_model <- glm(y ~ x + z, data = a, family = gaussian(link = "identity"))

##' Generate predictions for everyone in the sample to obtain
##' unexposed (mu0 predictions) and exposed (mu1 predictions) risks.
mu1 <- predict(ms_model, newdata = transform(a, x=1), type="response")
mu0 <- predict(ms_model, newdata = transform(a, x=0), type="response")
```

```r
#' Mean difference in predicted outcomes
marg_stand_MD <- mean(mu1) - mean(mu0)

#' Using the bootstrap to obtain confidence intervals for the marginally adjusted
#' mean difference.
bootfunc <- function(data,index){
  boot_dat <- data[index,]
  ms_model <- glm(y ~ x + z, data=boot_dat, family = gaussian(link = "identity"))
  mu1 <- predict(ms_model, newdata = transform(boot_dat,x=1), type="response")
  mu0 <- predict(ms_model, newdata = transform(boot_dat,x=0), type="response")

  marg_stand_MD_ <- mean(mu1) - mean(mu0)

  return(marg_stand_MD_)
}

#' Run the boot function. Set a seed to obtain reproducibility
set.seed(123)
boot_res <- boot(a, bootfunc, R=2000)

boot_MD <- boot.ci(boot_res, type = "norm")

marg_stand_MD
```

```
## [1] 10.70924
```

```r
boot_MD
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 2000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = boot_res, type = "norm")
##
## Intervals :
```

```
## Level      Normal
## 95%   (10.07, 11.36 )
## Calculations and Intervals on Original Scale
```

5 Inverse Probability Weighting

We can also estimate the exposure-outcome association using inverse probability weighting. Inverse probability weighting is the commonly employed propensity score adjustment technique. The simple heuristic often used to describe the way IP-weighting works is that, when applied to data, they yield a "pseudo population" where there is no longer an association between the covariate (i.e., confounder) on the exposure.

The weights for each individual needed to create this pseudo-population are defined as the inverse of the probability of receiving their observed exposure. However, simply taking the inverse of the probability of the observed exposure, while valid, is not the usual strategy for implementing inverse probability weights. In practice, one will often use stabilized weights, stabilized normalized weights, potentially with some degree of "truncation" or, more accurately, trimming of the weights.[1]

[1] In contrast to our emphasis of the usage of the word "truncation" which refers to the removal of observations from the dataset, researchers will often refer to "truncating" the weights, which sets the largest value to be equal to the 99th or 95th percentile values. This is more accurately referred to as "trimming" the weights, since no truncation is occurring.

The simplest type of weight used in practice is the stabilized inverse probability weight. These are often defined as:

$$sw = \begin{cases} \dfrac{P(X=1)}{P(X=1 \mid Z)} & \text{if } X = 1 \\ \dfrac{P(X=0)}{P(X=0 \mid Z)} & \text{if } X = 0 \end{cases}$$

Let's use the simulated data again to construct the stabilized weights and apply them to estimate the mean difference. We start by fitting a propensity score model to construct our weights:

```
# create the propensity score in the dataset
a$propensity_score <- glm(x ~ z, data = a, family = binomial("logit"))$fitted.values

# stabilized inverse probability weights
a$sw <- (mean(a$x)/a$propensity_score)*a$x +
  ((1-mean(a$x))/(1-a$propensity_score))*(1-a$x)
```

```
summary(a$sw)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.3638  0.8246  0.9308  1.0002  1.0914  5.3677
```

```
head(a)
```

```
##             z x         y propensity_score        sw
## 1 -0.56047565 0 101.81860        0.2054964 0.8853327
## 2 -0.23017749 0 107.45388        0.2463131 0.9332788
## 3  1.55870831 0  99.50946        0.5370583 1.5194138
## 4  0.07050839 0  73.28888        0.2879360 0.9878326
## 5  0.12928774 0  89.41832        0.2965456 0.9999226
## 6  1.71506499 1 102.59044        0.5644486 0.5254686
```

As we can see from the output above, the stabilized weights are, in fact, well behaved, with a mean of one and a max value that is small relative to the overall sample size.

```
mod_MD_weighted <- glm(y ~ x, data = a, weights=sw, family = gaussian("identity"))

summary(mod_MD_weighted)$coefficients
```

```
##             Estimate Std. Error   t value      Pr(>|t|)
## (Intercept) 99.73286  0.1754109 568.56702  0.000000e+00
## x           10.72026  0.3223223  33.25944 2.693767e-219
```

To get appropriate standard errors for this model, there are a few options we can use. Importantly, the model-based standard errors are no longer valid when weighting is used. One must instead use the robust (sandwich) variance estimators, or the bootstrap.

For example, the robust variance approach could be deployed using the `lmtest` and `sandwich` packages:

```
library(lmtest)
library(sandwich)

coeftest(mod_MD_weighted,
         vcov. = vcovHC(mod_MD_weighted, type = "HC3"))[2,]
```

```
##      Estimate    Std. Error       z value      Pr(>|z|)
##  1.072026e+01  3.545264e-01  3.023825e+01 7.447026e-201
```

```
coefci(mod_MD_weighted,
       level = 0.95,
       vcov. = vcovHC(mod_MD_weighted, type = "HC3"))[2,]
```

```
##    2.5 %   97.5 %
## 10.02540 11.41512
```

One can then construct CIs in the standard way using the estimated standard error in the output above, or using the `coefci` function in the `lmtest` package. Alternatively, we can use the boostrap to get standard errors for IP weighted models. The key to the bootstrap here (as in all cases) is to capture all models within the bootstrap function. In the IP weighting case, this includes the propensity score model and the weighted regression model:

```
#' Using the bootstrap to obtain confidence intervals for the IP weighted
#' mean difference.
bootfunc <- function(data,index){

  boot_dat <- data[index,]

  boot_dat$propensity_score <- glm(x ~ z, data = boot_dat, family = binomial("logit"))$fitted.values

  # stabilized inverse probability weights
  boot_dat$sw <- (mean(boot_dat$x)/boot_dat$propensity_score)*boot_dat$x +
    ((1-mean(boot_dat$x))/(1-boot_dat$propensity_score))*(1-boot_dat$x)
```

```
  mod_MD_weighted_ <- glm(y ~ x, data = boot_dat, weights=sw, family = gaussian("identity"))

  res <- summary(mod_MD_weighted_)$coefficients[2,1]

  return(res)

}

#' Run the boot function. Set a seed to obtain reproducibility
set.seed(123)
boot_res <- boot(a, bootfunc, R = 2000)

boot_IP_weight <- boot.ci(boot_res, type = "norm")

summary(mod_MD_weighted)$coefficients[2,1]
```

```
## [1] 10.72026
```

```
boot_IP_weight
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 2000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = boot_res, type = "norm")
##
## Intervals :
## Level      Normal
## 95%   (10.05, 11.40 )
## Calculations and Intervals on Original Scale
```

Here's a table comparing the results we've obtained so far:

	Estimate	LCL	UCL
marg_stand_res	10.70924	10.07334	11.35882
ip_weighted_res	10.72026	10.02540	11.41512
ip_weighted_boot	10.72026	10.04892	11.39527

References

Ashley I Naimi, Stephen R Cole, and Edward H Kennedy. An Introduction to G Methods. *Int J Epidemiol*, 46(2):756–62, 2017.

Jacqueline E Rudolph, Jessie K Edwards, Ashley I Naimi, and Daniel J Westreich. Simulation in practice: The balancing intercept. *Am J Epidemiol*, 190(8): 1696–1698, 2021.