6

Simulating Generalized Linear Models

6.1 INTRODUCTION

In the previous chapter, we dug much deeper into simulations, choosing to focus on the standard linear model for all the reasons we discussed. However, most social scientists study processes that do not conform to the assumptions of OLS. Thus, most social scientists must use a wide array of other models in their research. Still, they could benefit greatly from conducting simulations with them. In this chapter, we discuss several models that are extensions of the linear model called generalized linear models (GLMs). We focus on how to simulate their DGPs and evaluate the performance of estimators designed to recover those DGPs. Specifically, we examine binary, ordered, unordered, count, and duration GLMs. These models are not quite as straightforward as OLS because the dependent variables are not continuous and unbounded. As we show below, this means we need to use a slightly different strategy to combine our systematic and stochastic components when simulating the data. None of these examples are meant to be complete accounts of the models; readers should have some familiarity beforehand.[1]

After showing several GLM examples, we close this chapter with a brief discussion of general computational issues that arise when conducting simulations. You will find that several of the simulations we illustrate in this chapter take much longer to run than did those from the previous chapter. As the DGPs get more demanding to simulate, the statistical estimators become more computationally demanding, and the simulation studies we want to conduct increase in complexity, we can quickly find ourselves doing projects that require additional resources. We consider when it is advantageous to use research computing clusters and/or parallel processing. We also show a basic example of parallel processing. We look at how to distribute the workload of a simulation across multiple cores of a computer (this can be done on most modern desktops or laptops). As we will see, this can considerably reduce the time it takes to complete a simulation.

[1]For lengthier treatments of these models see Faraway (2006), Gelman and Hill (2007), Greene (2011), King (1998), or Long (1997).

6.2 SIMULATING OLS AS A PROBABILITY MODEL

Although OLS is typically taught through its analytic solution using matrix algebra, it can also be estimated by ML, which makes more clear the connections between OLS, GLMs, and probability theory. ML is a general method of estimating the unknown parameters of a statistical model using a sample of data.[2] We assume that the outcome we are interested in studying follows some probability distribution that has one or more unknown parameters that we want to estimate. In the OLS case, we assume a normal distribution with parameters μ and σ. Instead of thinking of the error term as taking on a normal distribution, we now think of the dependent variable as being normally distributed, and we think of the independent variables informing estimates of μ and/or σ.

To see this more clearly, consider the OLS model as we have seen it to this point. The dependent variable (Y) is a function of an intercept (β_0), coefficients operating on independent variables (β and X, respectively), and an error term that follows a normal distribution by assumption (ε).

$$\begin{aligned} Y &= \beta_0 + \beta X_1 + \beta_2 X_2 + \cdots + \varepsilon \\ & \quad \varepsilon \sim \mathcal{N}(0, \sigma) \end{aligned} \tag{6.1}$$

Another way to define this same model is to write Y as following a normal distribution.

$$\begin{aligned} Y &\sim \mathcal{N}(\mu, \sigma) \\ & \quad \mu = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \end{aligned} \tag{6.2}$$

In this representation, we see that Y is normally distributed with the mean (μ) and a constant standard deviation (σ). The second line of the model expresses μ as equal to the sum of the coefficients multiplied by the independent variables (often called the "linear predictor"). In other words, the dependent variable is explicitly written as following a probability distribution with its mean set to a linear combination of coefficients and independent variables rather than having such a distribution "attached" to the end of the equation via an error term.

This is important to point out because the dependent variables in other types of models are assumed to follow other distributions. Thus, creating a DGP for these kinds of dependent variables is not as straightforward as adding random noise to the end of an equation. For example, to simulate a binary dependent variable model (e.g., logistic regression), we need to produce a dependent variable that takes on only the values of 0 or 1. We cannot do this by simply adding `rnorm()` to the end of the equation. Instead, we will use a function to link the systematic portion of the model to the probability parameter of the Bernoulli distribution, which we then use to generate a series of Bernoulli trials (e.g., "coin flips"), producing the 0s and 1s for our observed dependent variable. In the case of OLS, this

[2]In fact, ML can be thought of as a general theory of inference (see King, 1998).

function is called the "identity," which means we leave the model's linear predictor unchanged to produce the normal distribution's μ parameter.

The main consequence of making this change for simulation studies is that we will generate the DGP in R in a slightly different way. Recall that our simulations to this point have included some variant of the following lines, used to produce the dependent variable, Y.[3]

```
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

Y <- b0 + b1*X + rnorm(n, 0, 1) # The true DGP, with N(0, 1) error
```

We can produce this exact same result by writing the code for the last line as follows. This produces Y using only the rnorm() command with the systematic component of the model substituted for μ.

```
Y <- rnorm(n, b0 + b1*X, 1) # The true DGP, Y ~ N(mu, sigma)
```

You can substitute this second line in the first version in the basic OLS simulation from Chapter 1 to check their equivalence. We did so. Below is the first six rows of the results matrix produced using the original code to generate Y (the first block of code above). Recall that the first and second columns are the estimates of β_0 and β_1, respectively, and the third and fourth columns are the standard errors of β_0 and β_1, respectively. Recall also that we defined b0 = 0.2, b1 = 0.5, and X is drawn from a uniform distribution bounded by – 1 and 1.

```
head(par.est)
          [,1]      [,2]       [,3]       [,4]
[1,] 0.2025988 0.4388826 0.03167166 0.05497156
[2,] 0.2109310 0.4665079 0.03173455 0.05508072
[3,] 0.2218581 0.5508824 0.03124873 0.05423750
[4,] 0.2417893 0.5583468 0.03227737 0.05602289
[5,] 0.1927056 0.5097159 0.03146897 0.05461976
[6,] 0.2133593 0.5549790 0.03166222 0.05495519
```

The results using the new approach from the code above are identical. We encourage you to try this to see it for yourself.

```
head(par.est)
          [,1]      [,2]       [,3]       [,4]
[1,] 0.2025988 0.4388826 0.03167166 0.05497156
[2,] 0.2109310 0.4665079 0.03173455 0.05508072
```

[3]Note that this is NOT the complete code needed for the simulation.

```
[3,] 0.2218581 0.5508824 0.03124873 0.05423750
[4,] 0.2417893 0.5583468 0.03227737 0.05602289
[5,] 0.1927056 0.5097159 0.03146897 0.05461976
[6,] 0.2133593 0.5549790 0.03166222 0.05495519
```

In the rest of the simulations in this book, we will use this new approach to generate the dependent variable explicitly from a given probability distribution.

6.3 SIMULATING GLMS

Having seen the DGP assumed by OLS constructed explicitly from its assumed probability distribution, we now turn to simulating other types of DGPs and evaluating different statistical models designed to recover them. In each one, we use the same basic strategy. We link the systematic component of the model—which we keep the same as in Chapter 5—to the mean of a probability distribution, then draw our dependent variable from that distribution. We can then evaluate the results as we have done in previous chapters.

6.3.1 Binary Models

We start with binary models, where the dependent variable can take on two outcomes, usually coded 1 or 0. For example, a large literature in political science examines the determinants of voter turnout. Individual-level turnout involves one of two options: a person voted (1) or did not vote (0), which researchers then model as a function of several independent variables. Two of the most common models are the logistic regression (logit) and probit models. We simulate both below.

To create the DGP for a binary model, we need the systematic component of the DGP to influence whether an observation is a 1 or 0 on the dependent variable, but we still want to incorporate an element of randomness (e.g., the stochastic component). To do this, we use the `rbinom()` function to produce each observation's value for the dependent variable as a single Bernoulli trial (e.g., a coin flip). The systematic component of the model affects the probability of observing a 1. Importantly, this probability is different for each observation, based on its values of the independent variables. The stochastic component of the model comes from the fact that the systematic component only effects the probability of observing a 1, but does not determine it entirely. Even if the probability of an observation getting a 1 is 0.99, there is still a chance that a single realization will come up 0. The difference between the logit and probit models lies in exactly how the probabilities are computed.

Logit

The logit model we simulate is written as follows:

$$\Pr(Y = 1) = \text{logit}^{-1}(\beta_0 + \beta_1 X) \tag{6.3}$$

To compute the probability of observing a 1 (p) with logit, we use the inverse logit function (logit^{-1}), which is

$$\frac{\exp(p)}{1+\exp(p)} \tag{6.4}$$

Putting this all together, we can rewrite Equation 6.3 as

$$\Pr(Y=1) = \frac{\exp(\beta_0+\beta_1 X)}{1+\exp(\beta_0+\beta_1 X)} \tag{6.5}$$

Several packages in `R` have functions to do this computation, or we can create our own.

```
inv.logit <- function(p){
return(exp(p)/(1 + exp(p)))
}
```

This function takes the systematic component of the model and translates it into probabilities varying between 0 and 1. Each observation gets its own probability because each observation has different values on the independent variable(s). Here is an example for six observations. First, we define the true parameters `b0` and `b1`, and the independent variable, `X`.

```
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X
```

The first line below reports the first six observed values of `X`. The second line reports the expected probability that `Y` = 1 for those six values, given `b0` = 0.2 and `b1` = 0.5 in the population DGP.

```
head(X)
[1] 0.39876948 -0.17150942 -0.49228411 0.19624512 -0.03102461 -0.30160264

head(inv.logit(b0 + b1*X))
[1] 0.5985398 0.5285303 0.4884665 0.5739835 0.5459916 0.5122972
```

The next step is putting these probabilities into the `rbinom()` function to produce the dependent variable. The first argument is the number of random draws, which we set to the sample size because we want a draw of either 1 or 0 for each observation. The second argument is the number of trials. We set this number to 1 because each of the n observations in our data set is one single Bernoulli trial (one coin flip). Finally, the third argument is the probability of observing a 1 for each observation; we place our systematic component here. Again, this influences

the chances of each observation coming up 1, but does not determine its value entirely.

```
Y <- rbinom(n, 1, inv.logit(b0 + b1*X)) # The true DGP Bernoulli trials
```

As a check to see that `Y` only takes on values of 0 or 1, you can use the `table()` function in `R` to produce a frequency table, like this:

```
table(Y)
Y
  0   1
477 523
```

In this particular example, where `n` was set equal to 1,000, we ended up with 477 values of 0 and 523 values of 1. If you repeat the code (without setting the seed), you would expect to get a slightly different number of 0s and 1s just due to random chance.

How we generate the dependent variable is the only major difference so far between this simulation and the OLS simulations we have done in the earlier chapters. To evaluate the logit model as an estimator of our parameters, we tell `R` to estimate a logit model instead of OLS using the `glm()` function rather than the `lm()` function. The full simulation code is below, including the inverse logit function and coverage probability function (see Chapter 5).

```
# Logit
# Inverse Logit Function
inv.logit <- function(p){
return(exp(p)/(1 + exp(p)))
}

# CP Function
coverage <- function(b, se, true, level = .95, df = Inf){ # Estimate,
                                                          # standard error,
                                                          # true parameter,
                                                          # confidence level,
                                                          # and df
qtile <- level + (1 - level)/2 # Compute the proper quantile
lower.bound <- b - qt(qtile, df = df)*se # Lower bound
upper.bound <- b + qt(qtile, df = df)*se # Upper bound
# Is the true parameter in the confidence interval? (yes = 1)
true.in.ci <- ifelse(true >= lower.bound & true <= upper.bound, 1, 0)
cp <- mean(true.in.ci) # The coverage probability
mc.lower.bound <- cp - 1.96*sqrt((cp*(1 - cp))/length(b)) # Monte Carlo error
mc.upper.bound <- cp + 1.96*sqrt((cp*(1 - cp))/length(b))
return(list(coverage.probability = cp, # Return results
         true.in.ci = true.in.ci,
         ci = cbind(lower.bound, upper.bound),
         mc.eb = c(mc.lower.bound, mc.upper.bound)))
}
```

```
set.seed(32945) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.logit <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store
                                                   # the estimates
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

for(i in 1:reps){ # Start the loop
Y <- rbinom(n, 1, inv.logit(b0 + b1*X)) # The true DGP, Bernoulli trials
model <- glm(Y ~ X, family = binomial (link = logit)) # Estimate logit model
vcv <- vcov(model) # Variance-covariance matrix
par.est.logit[i, 1] <- model$coef[1]  # Put the estimate for the
                                      # intercept in the first column
par.est.logit[i, 2] <- model$coef[2]   # Put the estimate for the coefficient
                                      # on X in the second column
par.est.logit[i, 3] <- sqrt(diag(vcv)[1]) # SE of the intercept
par.est.logit[i, 4] <- sqrt(diag(vcv)[2]) # SE of the coefficient on X
} # End the loop
```

We can then assess results of this logit simulation in the same way that we did for the OLS simulation we conducted in Chapter 5 (e.g., plot histograms or density curves of the estimates, compute AB, MSE, or coverage probabilities). Figure 6.1 plots the distributions of the coefficient estimates. We can verify that our simulation worked because both sets of estimates are centered at the true parameter values. Additionally, the coverage probabilities show no problems: 0.956 (β_0) and 0.958 (β_1). Of course, now you could return to the DGP for this simulation and explore how any alterations to it affect the logit estimator's ability to accurately and efficiently recover the parameters of the underlying DGP.

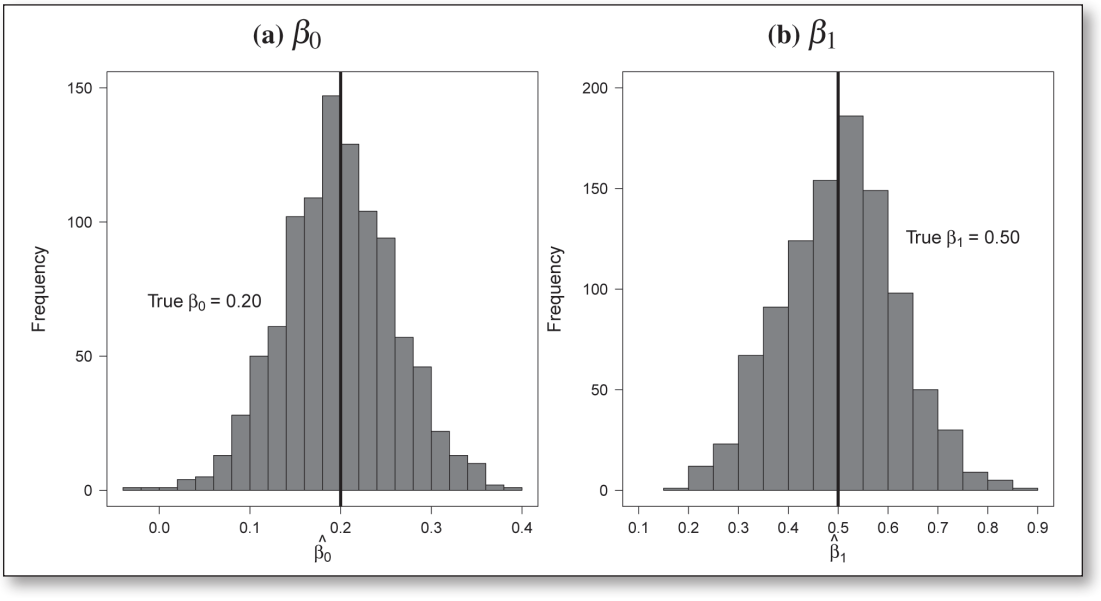
Probit

In the probit model, the probability of observing a 1 is assumed to come from the normal CDF instead of the inverse logit. To simulate a probit model, we simply replace the `inv.logit()` function with `pnorm()` function (recall that this computes the normal CDF) in our DGP. Then, we must set the link function to probit in the `glm()` function to estimate the probit model rather than the logit model.

```
# Probit
set.seed(3295255) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.probit <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store
                                                    # the estimates
```

```
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

for(i in 1:reps){ # Start the loop
Y <- rbinom(n, 1, pnorm(b0 + b1*X)) # The true DGP, Bernoulli trials
model <- glm(Y ~ X, family = binomial (link = probit)) # Estimate probit model
vcv <- vcov(model) # Variance-covariance matrix
par.est.probit[i, 1] <- model$coef[1]  # Put the estimate for the
                                       # intercept in the first column
par.est.probit[i, 2] <- model$coef[2]  # Put the estimate for the coefficient
                                       # on X in the second column
par.est.probit[i, 3] <- sqrt(diag(vcv)[1]) # SE of the intercept
par.est.probit[i, 4] <- sqrt(diag(vcv)[2]) # SE of the coefficient on X
} # End the loop
```

Figure 6.1 Histograms of 1,000 Simulated Logit β_0 and β_1 Estimates



We can then assess results as usual. Again, we see that the means of the estimates are very close to the true values, and the coverage probabilities are near 0.95. Now that you know how to simulate a proper DGP for the probit model, you can explore how changes to that DGP affect the results produced by your simulation.

```
mean(par.est.probit[ , 1]) # Mean of intercept estimates
[1] 0.2007092
mean(par.est.probit[ , 2]) # Mean of coefficient on X estimates
[1] 0.4996981
# Coverage probability for the intercept
coverage(par.est.probit[ , 1], par.est.probit[ , 3], b0,
 df = n - model$rank)$coverage.probability
[1] 0.957
# Coverage probability for the coefficient on X
coverage(par.est.probit[ , 2], par.est.probit[ , 4], b1,
 df = n - model$rank)$coverage.probability
 [1] 0.942
```

6.3.2 Ordered Models

The ordered models we consider here represent extensions of the binary models we presented in the last section. Ordered models allow for more than two categories on the dependent variable to be observed, but with a rank ordering to those categories. An example could be a survey question that asks respondents how often they feel angry about their job, with answer choices "never," "seldom," "often," or "every day." This variable has four categories with a clear ordering. We can use an ordered logit or ordered probit model to estimate the effect of an independent variable on the probability of a respondent falling into the available categories.

To simulate an ordered model, we use a *latent variable* interpretation of the dependent variable.[4] The idea behind this interpretation is that we would prefer to measure the dependent variable of interest on a continuous scale and then model it as a linear function of one or more independent variables. However, the measure we have of the dependent variable only records whether respondents fall into one of several categories. We call the unobserved continuous measure Y^*. The observed dependent variable, Y, represents categories that each cover a range of this unobserved latent variable. Formally, labeling the cutpoints between the k categories τ_k, Y is defined as follows.[5]

$$Y = \begin{cases} 1 & \text{if } Y^* < \tau_1 \\ 2 & \text{if } \tau_1 \leq Y^* < \tau_2 \\ 3 & \text{if } \tau_2 \leq Y^* < \tau_3 \\ 4 & \text{if } \tau_3 \leq Y^* < \tau_4 \\ . & \\ . & \\ . & \\ k & \text{if } Y^* \geq \tau_K \end{cases} \tag{6.6}$$

[4]This same interpretation could also apply to the logit and probit models that we described in the previous section. See Long (1997) for an excellent presentation of the latent variable approach.

[5]Different texts use different Greek letters or other symbols to refer to these cutpoints, including μ, ς, and c. We follow Long's (1997) use of τ below. The `R` command `polr()` that we use in estimation refers to the cutpoints as `zeta`.

Figure 6.2 Illustration of the Dependent Variable in an Ordered Model

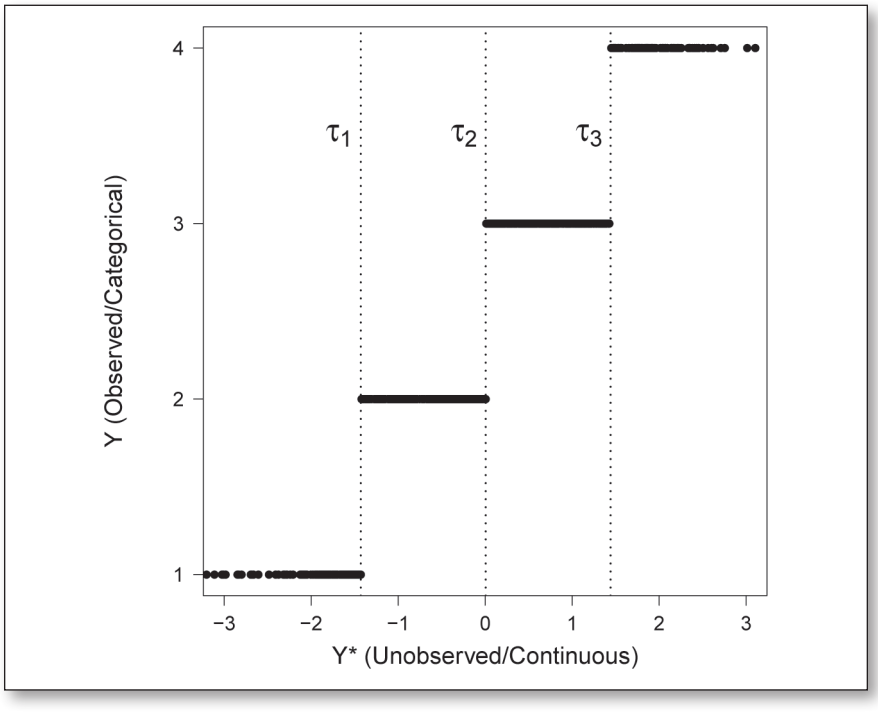


Figure 6.2 provides an example with simulated data. We plot Y^* on the x-axis and the four categories of Y on the y-axis. The three vertical dashed lines mark the three cutpoints that divide the values of Y^* into the four categories of Y. Notice that as Y^* increases, Y also increases, but there is some loss in precision—observations within the same category of Y have different values for Y^*.

In this example, we will simulate an ordered probit model, which assumes that the unobserved Y^* is normally distributed.[6] In this example, we will simulate one common systematic component of the model across all categories, which means that the effect of a change in X on the probability of a respondent moving from one category on Y to the next remains the same across all categories.[7] Just like the simulations for binary logit and probit, our task is to construct a DGP that

[6]We could use this same procedure to simulate ordered logit using the logistic distribution.

[7]By "same," we do not mean that the probability of being in any of the categories must be equal to each other—they do not. Rather, the best way to think about this assumption is that Y^* is assumed to be a linear function of X. This fulfills the proportional odds assumption (also called the "parallel regressions" assumption) of standard ordered models (see Long, 1997).

captures the probability of Y^* falling in a particular category given the location defined by the systematic component of a model plus its stochastic component. For the latent variable framework we present, what we need to do is generate random samples of Y^* and then assign those simulated values of Y^* to different categories of Y based on where those Y^* values are relative to the cutpoints. That means we need a method of defining those cutpoints.

Because Y^* is a latent variable—a variable we do not directly observe or measure—it has no defined scale. Thus, before we can estimate the model, we need to establish a scale for it. Specifically, we need to define a location and a variance for $Y.^*$ To define the variance, we first need to define the probability distribution we assume for it, which for ordered models is typically either the normal or logistic distribution. We know by definition that the variance for the standard normal is equal to 1, while the variance for the standard logistic is equal to $\frac{\pi^2}{3}$. Thus, the variance of the stochastic portion Y^* will be either 1 (for a normal) or $\frac{\pi^2}{3}$ (for a logistic), which could be multiplied by some constant if you wanted to introduce more or less variance in the stochastic portion of Y^*. In fact, we were making these assumptions about the variance of the stochastic term when we estimated binary logit and probit models in the previous sections.

The location of Y^* can be set in a number of ways, two of which are most common. First, we could set one of the cutpoints equal to zero and then estimate the model's intercept term (β_0) and the rest of the cutpoints relative to the cutpoint fixed at zero. In fact, this is the default built into most binary logit and probit estimation routines in statistical software, including those we used above. In binary logit and probit, there is only one cutpoint anyway, so setting it to zero and estimating the intercept is common because that looks familiar to users of OLS and many other GLMs.

The second common alternative that many software programs use, including the `R` function we will use below, is to fix the intercept term at zero and then produce estimates of all of the cutpoints relative to that intercept. There is no right or wrong answer regarding which strategy to use—not even a better or worse. The intercept and the cutpoints cannot be estimated in absolute terms—they can only be estimated relative to each other. In other words, the model is not identified in a statistical sense without restricting one of these to zero. Importantly, the choice of which one to set at zero will not affect the *relative* estimates of the others. Most important, the choice of whether we set β_0 to zero or one of the τ parameters to zero has absolutely no effect on the estimate of our slope parameters, and it has absolutely no effect on our estimates of the probability of an observation falling into a given category of Y. In this simulation, we will set the intercept to zero and estimate the different cutpoints.

Because our statistical estimator is going to fix β_0 to zero and produce estimates of the cutpoints, we need a way to define those cutpoints in our DGP so that we can check to see if our estimator accurately recovers them. To determine the values of the cutpoints in the DGP, we need to define them relative to the probability distribution that Y^* follows. In other words, we need to know the expected population mean and variance of Y^* before we simulate the DGP for Y. We take advantage of a couple of shortcuts here to make the problem more tractable.

Remember that Y^* is the sum of a systematic and a stochastic component. As such, both the mean and the variance of Y^* will come from the means and variances of these two components. For this simulation, we will consider the ordered probit model, which means we know the stochastic component of the model will follow a normal distribution with a mean of zero and a constant variance. For this simulation, we are going to draw our values for X from a normal distribution as well (rather than the uniform distribution we have been using). Because the values of the β coefficients in the DGP are fixed, the systematic component of our DGP will also be normally distributed with a mean of $\beta_0 + \beta_1 X$ and a constant variance. If two random variables are each normally distributed, their sum will also be normally distributed, with the mean and variance defined as follows:

$$
\begin{aligned}
A &\sim N(\mu_A, \sigma_A^2) \\
B &\sim N(\mu_B, \sigma_B^2) \\
AB &= A + B \\
AB &\sim N(\mu_A + \mu_B, \sigma_A^2 + \sigma_B^2)
\end{aligned}
$$

Thus, in this simulation, Y^* will be normally distributed with an expected mean equal to the mean of the systematic portion of the model, $\beta_0 + \beta_1 X$ plus zero (because the expected mean of the error is zero) and a variance equal to the variance of $\beta_0 + \beta_1 X$ plus whatever we set for the variance of the stochastic component of the model.

This allows us to use the quantiles associated with a normally distributed variable with a defined mean and variance to generate cutpoints for our DGP. Specifically, the `qnorm()` function in `R` will produce a quantile from a normal distribution for a defined probability, mean, and variance. For example, if you want to know the cutpoint on a normal distribution that separates the lower 25% of the distribution from the upper 75% of the distribution, and that normal distribution has a mean of 5 and a standard deviation of 7, you would type

```
qnorm(.25, mean = 5, sd = 7)
[1] 0.2785717
```

This returns a value of just over 0.278. Now, we can finally do a simulation of an ordered probit model. In this example, we generate a dependent variable that will have four categories. Our DGP will assume that 10% of the observations fall in the first category, 40% fall in the second category, another 40% fall in the third category, and the last 10% fall in the fourth category.[8] We will use a single independent variable again, and we will set the intercept equal to zero in the DGP because the statistical estimator we use below will fix the intercept to zero and estimate the cutpoints. We will evaluate our simulation based on its ability to

[8]This distribution is defined for the DGP. Of course, the observed distribution of observations across the four categories will vary from sample to sample due to randomness.

recover the slope coefficient and the cutpoints. This first block of code defines several values and variables, including β_0, β_1, X, and our three cutpoints.

```
# Ordered Models
library(MASS)
set.seed(8732) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.oprobit <- matrix(NA, nrow = reps, ncol = 2)# Empty matrices to store
taus.oprobit <- matrix(NA, nrow = reps, ncol = 3) # the estimates
b0 <- 0 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- rnorm(n, 0, 1) # Create a sample of n observations on the
                    # independent variable X

XB <- b0 + b1*X # Systematic component
sd.error <- 1 # SD of the error of the unobserved Y*
# Define the true cutpoints
tau1 <- qnorm(.1, mean = mean(XB), sd = sqrt(var(XB) + sd.error^2))
tau2 <- qnorm(.5, mean = mean(XB), sd = sqrt(var(XB) + sd.error^2))
tau3 <- qnorm(.9, mean = mean(XB), sd = sqrt(var(XB) + sd.error^2))
```

Next, inside the `for` loop we generate the object `Y.star`, the unobserved dependent variable. We create this using the `rnorm()` function in the same way as if we were simulating an OLS model.[9] Then, we generate the observed dependent variable, `Y`, such that observations of Y^* falling below the first cutpoint, τ_1, get coded as 1, those between τ_1 and τ_2 get coded as 2, those between τ_2 and τ_3 get coded as 3, and those above τ_3 get coded as 4. We do this in `R` by using the square brackets and the logical operators from Table 4.2. For example, the code `Y[Y.star < tau1] <- 1` means "for observations in which `Y.star` is less than the object `tau1`, code `Y` as a 1."

```
for(i in 1:reps){ # Start the loop
Y.star <- rnorm(n, XB, sd.error) # The unobserved Y*
Y <- rep(NA, n) # Define Y as a vector of NAs with length n
Y[Y.star < tau1] <- 1 # Set Y equal to a value according to Y.star
Y[Y.star >= tau1 & Y.star < tau2] <- 2
Y[Y.star >= tau2 & Y.star < tau3] <- 3
Y[Y.star >= tau3] <- 4
```

Then, we need to tell `R` to estimate an ordered probit model, which we can do with the `polr()` function from the `MASS` package.

```
model <- polr(as.ordered(Y) ~ X, method = "probit", Hess = TRUE)
```

[9] If we wanted to simulate ordered logit, we would use the `rlogis()` function.

We add a few options to this code. First, we wrap the `as.ordered()` function around the dependent variable to tell `R` that it should treat it as an ordered variable. Additionally, notice that the `polr()` function takes the argument `method`, which we set to "`probit`" for an ordered probit model (type `?polr` for other options). Finally, `Hess = TRUE` tells `R` that we want to produce the standard errors. The complete simulation code is below. Note that we also create an empty matrix called `par.est.oprobit` to store the estimates of β_1 and its standard error, and a second empty matrix called `taus.oprobit` to store the estimates of the cutpoints from the simulation.

```
# Ordered Models
library(MASS)
set.seed(8732) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.oprobit <- matrix(NA, nrow = reps, ncol = 2) # Empty matrices to store
taus.oprobit <- matrix(NA, nrow = reps, ncol = 3)  # the estimates
b0 <- 0 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- rnorm(n, 0, 1) # Create a sample of n observations on the
                    # independent variable X

XB <- b0 + b1*X # Systematic component
sd.error <- 1 # SD of the error of the unobserved Y*
# Define the true cutpoints
tau1 <- qnorm(.1, mean = mean(XB), sd = sqrt(var(XB) + sd.error^2))
tau2 <- qnorm(.5, mean = mean(XB), sd = sqrt(var(XB) + sd.error^2))
tau3 <- qnorm(.9, mean = mean(XB), sd = sqrt(var(XB) + sd.error^2))

for(i in 1:reps){ # Start the loop
Y.star <- rnorm(n, XB, sd.error) # The unobserved Y*
Y <- rep(NA, n) # Define Y as a vector of NAs with length n
Y[Y.star < tau1] <- 1 # Set Y equal to a value according to Y.star
Y[Y.star >= tau1 & Y.star < tau2] <- 2
Y[Y.star >= tau2 & Y.star < tau3] <- 3
Y[Y.star >= tau3] <- 4
# Estimate ordered model
model <- polr(as.ordered(Y) ~ X, method = "probit", Hess = TRUE)
vcv <- vcov(model) # Variance-covariance matrix
par.est.oprobit[i, 1] <- model$coef[1]  # Put the estimate for the coefficient
                                        # on X in the second column
par.est.oprobit[i, 2] <- sqrt(diag(vcv)[1]) # SE of the coefficient on X
taus.oprobit[i, ] <- model$zeta
cat("Just completed iteration", i, "\n")
} # End the loop
```

We can check the results by computing the mean of the vector of simulated estimates of β_1, the means of the estimated cutpoints (τ_1, τ_2, and τ_3), and the coverage probability for the standard error of β_1. All of these quantities are very close to their true values.

```
mean(par.est.oprobit[ , 1]) # Mean of coefficient on X estimates
[1] 0.5020573

# Compare the actual taus to the means of the tau estimates
data.frame(True = c(tau1, tau2, tau3),
 Estimated = apply(taus.oprobit, 2, mean))
          True    Estimated
1 -1.429516141 -1.430006071
2  0.006193689  0.009096949
3  1.441903520  1.445353416

# Coverage probability for the coefficient on X
coverage(par.est.oprobit[ , 1], par.est.oprobit[ , 2], b1,
 df = n - length(c(coef(model), model$zeta)))$coverage.probability
[1] 0.944
```

6.3.3 Multinomial Models

Multinomial models also have dependent variables that consist of more than two categories, but there is no inherent rank–order relationship between the categories. For example, a prospective homeowner may receive financial assistance for a down payment on the home from a family member, a community grant program, or an employer. There is no ordering from "less" to "more" among these outcomes as there are with ordered variables. A model such as multinomial logit (MNL) or multinomial probit (MNP) can be used to estimate the probability of observing one of the outcomes given the independent variables.[10] In this section, we will focus on MNL.

A distinguishing feature of MNL is that for K possible outcomes, it estimates $K - 1$ sets of coefficients on the independent variable(s)—treating one category as a baseline and estimating sets of coefficients comparing each of the other categories with that baseline category. One way to conceptualize MNL is as a series of logit models connected together, one for each outcome (hence the multiple sets of coefficients).[11] Recall that our inverse logit function produced expected probabilities of observing a 1 with the logit model by computing $\frac{\exp(\beta_0+\beta_1 X)}{1+\exp(\beta_0+\beta_1 X)}$.

[10]See Kropko (2008) for a simulation study that compares MNL to MNP.

[11]A distinguishing feature of this model is the IIA assumption, which stands for "independence of irrelevant alternatives." Briefly, this means that the relative probabilities between two outcome categories do not change if another outcome category is introduced. Consider a group of people deciding on which restaurant to choose for lunch. If the chance of selecting Restaurant *A* over Restaurant *B* changes when Restaurant *C* becomes a possibility, IIA is violated.

We can compute expected probabilities for multiple logit models by adding multiple systematic components to that formula. For example, consider a dependent variable, *Y*, with three outcomes (*A*, *B* , or *C*) and Category C as the baseline category. The probability of Outcome *A* can be written as follows.[12]

$$\Pr(Y = A) = \frac{\exp(\beta_{0A} + \beta_{1A} X)}{1 + \exp(\beta_{0A} + \beta_{1A} X) + \exp(\beta_{0B} + \beta_{1B} X)} \tag{6.7}$$

Notice that there are different coefficients for Outcome *A* (β_{0A}, β_{1A}) and Outcome *B* (β_{0B}, β_{1B}). The probability of Outcome *B* is defined similarly.

$$\Pr(Y = B) = \frac{\exp(\beta_{0B} + \beta_{1B} X)}{1 + \exp(\beta_{0A} + \beta_{1A} X) + \exp(\beta_{0B} + \beta_{1B} X)} \tag{6.8}$$

Finally, we take advantage of the fact that the probabilities of the three outcomes must sum to 1 (i.e., the three outcomes are mutually exclusive and exhaustive). That means the probability of Outcome *C* is relatively straightforward.

$$\Pr(Y = C) = 1 - \Pr(Y = A) - \Pr(Y = B) \tag{6.9}$$

To simulate this type of dependent variable, we need to make a few changes to our code. First, we need to set true values of the coefficients for two of the outcomes (remember, one outcome is used as a baseline category). Instead of just `b0` and `b1`, we create `b0A` and `b1A` for Outcome *A* and `b0B` and `b1B` for Outcome *B*.

```
# Unordered Models
library(Zelig)
set.seed(45262) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.mnl <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store
                                                 # the estimates
b0A <- .2 # True values for the intercepts
b0B <- -.2
b1A <- .5 # True values for the slopes
b1B <- .75
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X
```

Next, outside the `for` loop we compute the probabilities from these coefficients using Equations 6.7 to 6.9. This can be done outside the `for` loop because these

[12]The "1" appears in the denominator because the linear predictor of the baseline category (*C*) is set to zero, and exp(0) = 1.

probabilities only represent the systematic component of the model (which is the same in every repetition of the simulation), not the stochastic component.

```
# Compute the probabilities of each outcome based on the DGP
pA <- exp(b0A + b1A*X)/(1 + exp(b0A + b1A*X) + exp(b0B + b1B*X))
pB <- exp(b0B + b1B*X)/(1 + exp(b0A + b1A*X) + exp(b0B + b1B*X))
pC <- 1 - pA - pB
```

The next step is to combine the systematic and stochastic components inside the `for` loop to create the dependent variable. To do this, we use the `sample()` function inside a separate `for` loop that iterates through every observation in the sample. For each observation, we draw a "sample" of length 1 from the letters A, B, and C with replacement (i.e., each observation gets an *A* , *B* , or *C* for the dependent variable). Importantly, we make use of the `prob` argument within the `sample()` function to define the probabilities of getting an *A* , *B* , or *C* for each observation. We use the `for` loop to iterate through each observation because these probabilities are unique to each observation based on the independent variable(s) from the systematic portion of the model.

```
for(i in 1:reps){ # Start the loop
Y <- rep(NA, n) # Define Y as a vector of NAs with length n
for(j in 1:n){ # Create the dependent variable in another loop
Y[j] <- sample(c("A", "B", "C"), 1, replace = TRUE,
prob = c(pA[j], pB[j], pC[j]))
}
```

For example, the code below shows the result of `sample()` for the first observation and the 983rd observation in the last data set in the simulation below. The probabilities of each outcome are about 0.41 (Outcome *A*), 0.30 (Outcome *B*), and 0.29 (Outcome *C*) for Observation #1. In this case, Outcome *A* was drawn. The probabilities of each outcome for Observation #983 are 0.39, 0.25, and 0.36, respectively. In that case, Outcome *B* was chosen even though it had the lowest probability of the three options.

```
# Observation 1
c(pA[1], pB[1], pC[1])
[1] 0.4127436 0.3001694 0.2870870
Y[1]
[1] "A"

# Observation 983
c(pA[983], pB[983], pC[983])
[1] 0.3931114 0.2503725 0.3565161
Y[983]
[1] "B"
```

The next step is to estimate the model and store the results as usual. The complete simulation code is below. Several packages will estimate MNL, including

`Zelig`. Similar to ordered models, it is helpful to wrap `as.factor()` around the dependent variable name to tell `R` that it is a factor variable (not ordered). Additionally, notice the use of the `coefficients()` function to store the coefficient estimates from the model.

```
# Unordered Models
library(Zelig)
set.seed(45262) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.mnl <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store the
                                                 # estimates
b0A <- .2 # True values for the intercepts
b0B <- -.2
b1A <- .5 # True values for the slopes
b1B <- .75
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X
# Compute the probabilities of each outcome based on the DGP
pA <- exp(b0A + b1A*X)/(1 + exp(b0A + b1A*X) + exp(b0B + b1B*X))
pB <- exp(b0B + b1B*X)/(1 + exp(b0A + b1A*X) + exp(b0B + b1B*X))
pC <- 1 - pA - pB

for(i in 1:reps){ # Start the loop
Y <- rep(NA, n) # Define Y as a vector of NAs with length n
for(j in 1:n){ # Create the dependent variable in another loop
Y[j] <- sample(c("A", "B", "C"), 1, replace = TRUE,
prob = c(pA[j], pB[j], pC[j]))
 }

# Estimate a MNL model
model <- zelig(as.factor(Y) ~ X, model = "mlogit",
  data = data.frame(Y, X), cite = FALSE)
vcv <- vcov(model) # Variance-covariance matrix
par.est.mnl[i, 1] <- coefficients(model)[3] # Coefficient on X, outcome 1
par.est.mnl[i, 2] <- coefficients(model)[4] # Coefficient on X, outcome 2
par.est.mnl[i, 3] <- sqrt(diag(vcv)[3]) # SE of coefficient on X, outcome 1
par.est.mnl[i, 4] <- sqrt(diag(vcv)[4]) # SE of coefficient on X, outcome 2
cat("Just completed iteration", i, "\n")
} # End the loop
```

As before, we can then examine the results to check that our estimates come close to the true DGP. Generally, the results look good, though the mean of the estimate on β_{1B} is a bit off the true value of 0.75 (mean of 0.757). Of course, this is due to random chance. We ran the simulation again with the number of repetitions set to 10,000 and came up with a mean of 0.75.

```
# Mean of coefficients on X estimates
mean(par.est.mnl[ , 1]) # Outcome 1
[1] 0.5011012
mean(par.est.mnl[ , 2]) # Outcome 2
[1] 0.7565751

# Coverage probabilities for the coefficients on X
# Outcome 1
coverage(par.est.mnl[ , 1], par.est.mnl[ , 3], b1A,
 df =n - length(coef(model)))$coverage.probability
[1] 0.949
# Outcome 2
coverage(par.est.mnl[ , 2], par.est.mnl[ , 4], b1B,
 df = n - length(coef(model)))$coverage.probability
[1] 0.958
```

6.4 EXTENDED EXAMPLES

The examples we have shown so far in this chapter have been relatively similar to each other and to the OLS examples from the previous chapters. We create the DGP of a particular estimator, generate data from that DGP many times, estimate the model on each simulated data set, then check to see if we accurately recovered the true DGP through the estimator. We kept things relatively simple in each simulation by not violating any of the assumptions of the models we explored. We encourage interested readers to expand and refine these simulations for other analyses, possibly exploring changes to the DGPs and observing how the various models perform. However, simulation is also a valuable tool for better understanding problems that arise in real data and comparing the ability of competing estimators to handle them. We show examples of this below.

6.4.1 Ordered or Multinomial?

Ordered and multinomial models are similar in the sense that the dependent variables are each composed of a finite number of categories. The difference is that ordered models assume there truly is an ordered structure to those categories, while the DGP of the multinomial model is assumed to have no inherent ordering. This leads to the question of consequences for violating these assumptions. What would happen if we estimated a multinomial model on a dependent variable that is truly ordered or an ordered model on a dependent variable that is truly unordered?

In his book *Regression Models for Categorical and Limited Dependent Variables*, Long (1997) provides the following answer to these questions:

> If a dependent variable is ordinal and a model for nominal variables [e.g., multinomial logit] is used, there is a loss of efficiency since information is being ignored. On the other hand, when a method for ordinal variables [e.g., ordered logit] is applied to a nominal dependent variable, the resulting estimates are biased or even nonsensical (p. 149).

We can validate Long's assertions with a simulation. We will create two different DGPs with one independent variable—one DGP in which the dependent variable is ordered and one in which it is unordered—then estimate an ordered logit and a multinomial logit on each of these dependent variables (four models in total). For each of the four models, we will compute the change in expected probability for each category associated with moving from the minimum to the maximum value of the independent variable (e.g., the marginal effect of X).[13]

The first part of the code is given below. It is adapted from the last two examples, but in this case, both the ordered and unordered dependent variables are created with three numeric categories (1, 2, and 3). We create both with numeric categories so that we can estimate an ordered model on each. If we generated the unordered dependent variable as "*A*," "*B*," and "*C*," the ordered model would produce an error. Also, we create the object `d.pp`, which is a group of matrices (called an array), to store changes in expected probabilities estimated from the models (see below).

```
# Ordered vs. MNL
library(Zelig)
set.seed(99999)

reps <- 1000 # Set the number of repetitions at the top of the script
d.pp <- array(NA, c(4, 3, reps)) # Empty array to store
                                 # simulated change in probabilities

# Ordered logit model DGP
b0 <- 0 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

# MNL model DGP
b0A <- .2 # True values for the intercepts
b0B <- -.2
b1A <- .5 # True values for the slopes
b1B <- .75
n <- 1000 # Sample size

# Compute the probabilities of each outcome based on the DGP
pA <- exp(b0A + b1A*X)/(1 + exp(b0A + b1A*X) + exp(b0B + b1B*X))
pB <- exp(b0B + b1B*X)/(1 + exp(b0A + b1A*X) + exp(b0B + b1B*X))
pC <- 1 - pA - pB
```

[13]We cannot directly compare coefficients between ordered and unordered models because the former produces one set of coefficients and the other produces multiple sets of coefficients (one for each category minus the baseline category).

```
for(i in 1:reps){
# Ordered dependent variable
Y.star <- rlogis(n, b0 + b1*X, 1) # The unobserved Y*
# Define the true cutpoints
tau1 <- quantile(Y.star, .25)
tau2 <- quantile(Y.star, .75)
Y.o <- rep(NA, n) # Define Y as a vector of NAs with length n
Y.o[Y.star < tau1] <- 1 # Set Y equal to a value according to Y.star
Y.o[Y.star >= tau1 & Y.star < tau2] <- 2
Y.o[Y.star >= tau2] <- 3
# Ordered data
o.data <- data.frame(Y.o, X) # Put the data in a data frame

# Unordered dependent variable
Y.m <- rep(NA, n) # Define Y as a vector of NAs with length n
for(j in 1:n){ # Create the dependent variable in another loop
Y.m[j] <- sample(1:3, 1, replace = TRUE, prob = c(pA[j], pB[j], pC[j]))
}
# Unordered data
m.data <- data.frame(Y.m, X) # Put the data in a data frame
```

Next, we need to estimate an ordered logit and an MNL on *both* dependent variables. We use the `zelig()` command for both types of models.[14] The object names reflect the type of model and whether that model makes the "correct" assumption about the DGP. For example, `o.correct` is an ordered logit model of the dependent variable that is truly ordered. In contrast, `m.incorrect` is an MNL model of the dependent variable that is truly ordered.

```
# Estimate the models with the ordered dependent variable
o.correct <- zelig(as.ordered(Y.o) ~ X, model = "ologit",
 data = o.data, cite = FALSE)
m.incorrect <- zelig(as.factor(Y.o) ~ X, model = "mlogit",
 data = o.data, cite = FALSE)

# Estimate the models with the multinomial dependent variable
m.correct <- zelig(as.factor(Y.m) ~ X, model = "mlogit",
 data = m.data, cite = FALSE)
o.incorrect <- zelig(as.ordered(Y.m) ~ X, model = "ologit",
 data = m.data, cite = FALSE)
```

Recall that ordered models assume that one set of coefficients characterizes the effect of the independent variables on the dependent variable, regardless of the category (the proportional odds or parallel regressions assumption). In contrast, multinomial models estimate a different set of coefficients for all but one of the categories. As a result, the coefficients from the two types of models are not directly comparable to each other.

[14]The `zelig()` command uses `polr()` to conduct the estimation, but makes computing changes in expected probabilities easier.

However, we can compare changes in the expected probability of each category computed from each model. The `Zelig` package allows this to be done with the `setx()` and `sim()` commands. We discuss these commands in more detail in Chapter 9, but for now the basic idea is that we can set the independent variable to some values (in this case, the minimum and maximum), and simulate the change in expected probability for each category moving between those two values. We store these values in the array `d.pp`. Recall from above that an array is a group of matrices. In this case, the object contains 1,000 matrices (one for each repetition in the simulation) with four rows and three columns each. In each iteration of the `for` loop, a matrix in `d.pp` is filled with the change in expected probability of each of the three categories (columns) for each of the four models (rows).[15]

```
# Set X to its minimum and maximum for each model
x.oc <- setx(o.correct, X = min(X)) # For o.correct
x.oc1 <- setx(o.correct, X = max(X))

x.mi <- setx(m.incorrect, X = min(X)) # For m.incorrect
x.mi1 <- setx(m.incorrect, X = max(X))

x.mc <- setx(m.correct, X = min(X)) # For m.correct
x.mc1 <- setx(m.correct, X = max(X))

x.oi <- setx(o.incorrect, X = min(X)) # For o.incorrect
x.oi1 <- setx(o.incorrect, X = max(X))

# Compute the change in expected probabilities of falling in each category
# when moving from the minimum to the maximum of X
sim.oc <- sim(o.correct, x = x.oc, x1 = x.oc1)$qi$fd
sim.mi <- sim(m.incorrect, x = x.mi, x1 = x.mi1)$qi$fd
sim.mc <- sim(m.correct, x = x.mc, x1 = x.mc1)$qi$fd
sim.oi <- sim(o.incorrect, x = x.oi, x1 = x.oi1)$qi$fd
d.pp[1, , i] <- apply(sim.oc, 2, mean)
d.pp[2, , i] <- apply(sim.mi, 2, mean)
d.pp[3, , i] <- apply(sim.mc, 2, mean)
d.pp[4, , i] <- apply(sim.oi, 2, mean)
cat("Just completed iteration", i, "of", reps, "\n")
}
```

The final step is to compare the changes in expected probabilities. We know that those computed from the models `o.correct` and `m.correct` are, in fact, correct, because in each one the estimator's assumptions about the DGP match the truth. To assess Long's (1997) claim, we need to compare the results from `o.incorrect` and `m.incorrect` to `o.correct` and `m.correct`,

[15]Specifically, the objects `sim.oc`, `sim.mi`, `sim.mc`, and `sim.oi` hold 100 simulated changes in the expected probability for each category. We use the `apply()` command to compute the mean change in each of the three categories. See Chapter 9 for more details on this method, which we call "QI (quantities of interest) simulation."

respectively. In the last section of the code, we compute the means and standard deviations of the 1,000 sets of changes in the expected probability of each category (remember the `for` loop ran for 1,000 iterations). These quantities allow us to assess bias and efficiency, respectively.

```
# Compute the average change in probability
# for each of the four models
dpp.means <- rbind(apply(d.pp[1, , ], 1, mean),
 apply(d.pp[2, , ], 1, mean), apply(d.pp[3, , ], 1, mean),
 apply(d.pp[4, , ], 1, mean))

# Compute the SD of the change in probability
# for each of the four models
dpp.sds <- rbind(apply(d.pp[1, , ], 1, sd),
 apply(d.pp[2, , ], 1, sd), apply(d.pp[3, , ], 1, sd),
 apply(d.pp[4, , ], 1, sd))
```

We present these results in Table 6.1. The top half of the table gives the means of the 1,000 simulated changes in expected probability for each category computed by each model/dependent variable combination. The bottom half of the table reports the standard deviations of those simulated changes.

Table 6.1 Means and Standard Deviations of the Simulated Changes in Expected Probability for Each Category With Ordered Logit and MNL Models

Category:	**1**	**2**	**3**
Means			
Ordered DV, Ordered Model	–0.1820	–0.0015	0.1835
Ordered DV, MNL Model	–0.1800	–0.0040	0.1840
Unordered DV, MNL Model	0.0718	0.1852	–0.2570
Unordered DV, Ordered Model	0.1668	–0.0107	–0.1561
Standard Deviations			
Ordered DV, Ordered Model	0.0364	0.0013	0.0370
Ordered DV, MNL Model	0.0442	0.0522	0.0453
Unordered DV, MNL Model	0.0508	0.0472	0.0459
Unordered DV, Ordered Model	0.0443	0.0056	0.0421

Note: The top half of the table gives the means of the 1,000 simulated changes in expected probability for each category computed by each model/dependent variable combination. The bottom half of the table reports the standard deviations of those simulated changes.

Beginning with the means, notice that when the dependent variable is truly ordered, the ordered model and MNL model produce changes in the expected probability of each category that are nearly identical. In contrast, when the dependent variable is truly unordered, the two models diverge considerably; the incorrect ordered model does not return the same changes in expected probabilities that the correct MNL model does. Moving to the standard deviations, notice that they are larger with the MNL model under both dependent variable scenarios. This is not surprising given that the MNL model estimates more parameters than does the ordered model, and thus uses more degrees of freedom.

In short, the results validate Long's (1997) claim. When the dependent variable is truly ordered, but a multinomial model is incorrectly used, no bias results (similar means), but there is a loss of efficiency (larger standard deviations for MNL). This is because the multinomial model is simply not using all of the available information (i.e., the ordered structure of Y), and so it estimates parameters that it does not need (coefficients for each category). In contrast, when an ordered model is applied to a truly unordered dependent variable, the corresponding changes in expected probabilities are biased because the estimator tries to impose an ordered structure that is not really there.

6.4.2 Count Models

We next move to a few examples of simulations with count models. Count models are used for dependent variables that take on positive integer values. Examples include the number of births at a hospital in a day or the number of patents awarded during some period. Count models allow the analyst to model such variables as a function of independent variables. A typical starting point for estimating the parameters of a count model is Poisson regression.

The Poisson Model

Recall from Chapter 2 that the Poisson distribution has one parameter, λ, which is both the mean and variance of the distribution. Poisson regression links the systematic portion of the model to a Poisson distribution through λ. It assumes the natural log of the dependent variable's expected value can be modeled by the sum of the independent variables multiplied by their coefficients.

$$\log[E(Y|X)] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \tag{6.10}$$

Exponentiating both sides yields an expected mean of the dependent variable of

$$E(Y|X) = \exp(\beta_0 + \beta_1 X_1 + \beta_2 X_2) \tag{6.11}$$

As with other examples from this chapter, the coefficients can be estimated by ML (see King, 1998). We can simulate such a model using the `rpois()` function. To produce the dependent variable, we set λ to the exponentiated systematic component of the model (because the dependent variable cannot be negative), as

in `rpois(n, lambda = exp(b0 + b1*X))`. Everything else is exactly as we have seen before.

```
# Poisson
set.seed(3759) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.pois <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store the
                                                  # estimates
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

for(i in 1:reps){ # Start the loop
Y <- rpois(n, exp(b0 + b1*X)) # The true DGP
model <- glm(Y ~ X, family = "poisson") # Estimate Poisson model
vcv <- vcov(model) # Variance-covariance matrix
par.est.pois[i, 1] <- model$coef[1] # Put the estimate for the intercept
                                    # in the first column
par.est.pois[i, 2] <- model$coef[2] # Put the estimate for the coefficient on
                                    # X in the second column
par.est.pois[i, 3] <- sqrt(diag(vcv)[1]) # SE of the intercept
par.est.pois[i, 4] <- sqrt(diag(vcv)[2]) # SE of the coefficient on X
} # End the loop

# Means of the coefficient estimates
mean(par.est.pois[ , 1]) # Intercept
[1] 0.2002308
mean(par.est.pois[ , 2]) # Coefficient on X
[1] 0.4988037

# Coverage probabilities
# Intercept
coverage(par.est.pois[ , 1], par.est.pois[ , 3], b0,
 df = model$rank)$coverage.probability
[1] 0.952
# Coefficient on X
coverage(par.est.pois[ , 2], par.est.pois[ , 4], b1,
 df = model$rank)$coverage.probability
[1] 0.948
```

The equivalence of the mean and variance in the Poisson distribution is an important limitation of the Poisson regression model because count processes in the social world often unfold such that the variance is larger than the mean, a phenomenon called overdispersion.[16] Several options exist to handle this problem, including the negative binomial model.

[16]Technically, what is most important is the comparison of the *conditional* mean and *conditional* variance. If the independent variables can account for the difference between the mean and variance of *Y*, the Poisson model is appropriate.

Comparing Poisson and Negative Binomial Models

The negative binomial (NB) is a more flexible approach to modeling count data because it can accommodate overdispersion through the estimation of a second parameter that allows the variance to be different from the mean. The NB model can be thought of as a "mixture" model because it combines the Poisson density with another density—the Gamma (see Cameron & Trivedi, 1998). Notice from the last simulation that we perfectly determined λ by setting it equal to the systematic component of the model. With NB, we set λ equal to the systematic portion plus a random component. This gives it a mean, μ, which is equal to the systematic component and a separate variance, υ.

$$\mu = \exp(\beta_0 + \beta_1 X_1 + \beta_2 X_2) \tag{6.12}$$

$$\upsilon = \mu + \frac{\mu^2}{\theta} \tag{6.13}$$

where θ is a dispersion parameter. The main point to keep in mind is that because there is a separate parameter for the variance, the NB is not adversely affected by overdispersion.

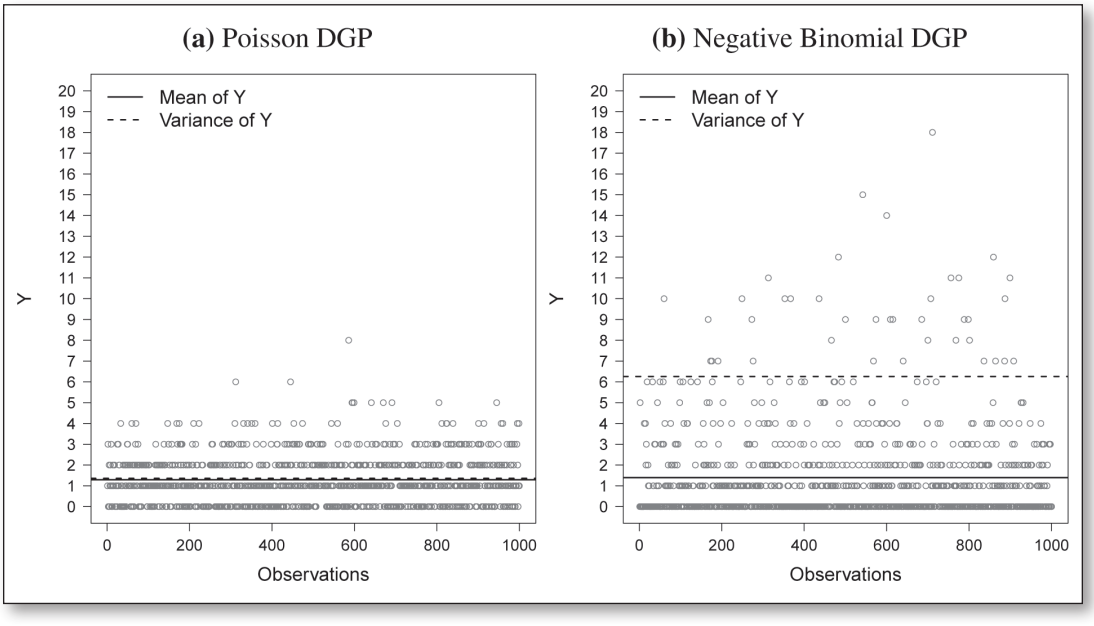
To illustrate the impact of overdispersion, we simulate overdispersed data, estimate Poisson and NB models, then compare the results. To simulate overdispersed count data we use the `rnbinom()` function, which takes the arguments `size`, which is the dispersion parameter in Equation 6.13, and `mu`, which is the mean. We set the dispersion parameter to 0.50 and the mean to the systematic component of the model. To see the difference in the data generated with and without overdispersion, compare the two panels of Figure 6.3. Panel (a) plots one simulated data set from the simulation above, which uses `rpois()` to produce *Y*. Panel (b) comes from data simulated with `rnbinom()`. In each graph, a solid line is placed at the mean of *Y* and a dashed line is placed at its variance. Notice that in Panel (a) the mean and variance are virtually identical, but in Panel (b) the variance is nearly six times the mean. Our next step is to assess the consequences of this difference for the Poisson and NB estimators.

The simulation code is below. We simulate `Y` with `rnbinom()`, then estimate the Poisson and NB models with `glm()` and `glm.nb()`, respectively. The `glm.nb()` function comes from the `MASS` package. We store the coefficients and standard errors for β_1.

```
# Poisson vs. Negative Binomial
library(MASS)
set.seed(763759) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.pnb <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store the
                                                 # estimates
b0 <- .2 # True value for the intercept
```

```
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

for(i in 1:reps){
Y <- rnbinom(n, size = .5, mu = exp(b0 + b1*X)) # Generate data with
                                                # overdispersion
model.p <- glm(Y ~ X, family = "poisson") # Estimate Poisson model
model.nb <- glm.nb(Y ~ X) # Estimate NB model
vcv.p <- vcov(model.p) # Variance-covariance matrices
vcv.nb <- vcov(model.nb)
par.est.pnb[i, 1] <- model.p$coef[2] # Store the results
par.est.pnb[i, 2] <- model.nb$coef[2]
par.est.pnb[i, 3] <- sqrt(diag(vcv.p)[2])
par.est.pnb[i, 4] <- sqrt(diag(vcv.nb)[2])
cat("Completed", i, "of", reps, "\n")
}
```

Figure 6.3 Dependent Variables From Poisson and Negative Binomial DGPs



Textbook discussions of count models tell us that the Poisson model's coefficient estimates are consistent if overdispersion is the only problem, but that the standard errors are too small (Faraway, 2006). We can check this by computing the means of the simulated estimates, MSE, and coverage probabilities.

```
# Means of the coefficient on X estimates
mean(par.est.pnb[ , 1]) # Poisson estimates
[1] 0.4971698
mean(par.est.pnb[ , 2]) # NB estimates
[1] 0.4984487

# MSE
mean((par.est.pnb[ , 1])^2) # Poisson MSE
[1] 0.2560358
mean((par.est.pnb[ , 2])^2) # NB MSE
[1] 0.2572339

# Coverage probabilities
# Poisson SEs
coverage(par.est.pnb[ , 1], par.est.pnb[ , 3], b1,
df = n - model.p$rank)$coverage.probability
[1] 0.724
# NB SEs
coverage(par.est.pnb[ , 2], par.est.pnb[ , 4], b1,
df = n - model.nb$rank)$coverage.probability
[1] 0.946
```

The results show that both estimators produced coefficient estimates with means near the true value of 0.50. Additionally, MSE is very similar for both. However, the coverage probabilities show a big difference. The Poisson standard errors produce a coverage probability of 0.723 while the NB coverage probability is 0.946. Thus, when the data are overdispersed the Poisson model's assumptions produce estimates of coefficient variability (standard errors) that are too small.

Zero-Inflation Processes

Another possible issue with count models is having a large number of cases with the value of zero for *Y*. Many zeros may simply be a part of the true DGP governing all of the other counts (e.g., 1s, 2s, 3s, etc.), or the amount of zeros may be "inflated" by an additional process other than that which influences the counts that are greater than zero. In other words, the DGP may have two separate systematic processes: (1) a process influencing whether an observation produces a zero on the dependent variable or not, and (2) the actual count. One way of accounting for these two processes is with a zero-inflated count model. This type of model is actually two models in one—an equation predicting whether the observation is zero or nonzero (often with a binary model such as logit) and another equation predicting the expected count (done with a count model). Zero-inflation models work with both Poisson and NB; we will focus on the zero-inflated NB (ZINB) throughout this example.[17]

[17]There are also other means of handling zero inflation, such as dichotomizing the data or hurdle models (Cameron & Trivedi, 1998).

Underlying the choice between conventional count regression and zero-inflated modeling is the common tension between overfitting and successfully explaining empirical features of the data. This produces an important theoretical and empirical question: Is there a unique process that inflates the probability of a zero case? Much is at stake in the answer to this question. A "yes" amounts to more than just the addition of an explanatory variable. As mentioned above, an entire process in the form of another equation and several more parameters to be estimated is added to the model. Most important, what happens if a researcher makes the incorrect choice (i.e., choosing the standard model when the zero-inflated model should be used or incorrectly choosing the zero-inflated model)? We can answer that question with simulation.

In the following code, we simulate one dependent variable in which there is a zero-inflation component in the true DGP and a second dependent variable in which the true DGP has no zero-inflation component. We estimate both a standard NB model and ZINB model on both of these dependent variables and compare the results. To simulate the dependent variable, we create a new function called `rzinbinom()`.

```
rzinbinom <- function(n, mu, size, zprob){
ifelse(rbinom(n, 1, zprob) == 1, 0, rnbinom(n, size = size, mu = mu))
}
```

This function uses `ifelse()` to separate the zero inflation and count processes. It takes an argument `zprob` that is the probability that an observation is a 0. It uses `rbinom()` to take one draw from a Bernoulli distribution (i.e., a single coin flip) with the probability of success set to `zprob`. If this draw comes up a success, the function should return a 0. If not, the function should proceed as `rnbinom()`. With this setup, observations that have a high probability of becoming a 0 will be a 0 more often than those with low probability.

We use this function to generate the dependent variables as follows.

```
# Generate data with a zero-inflation component
Y.zi <- rzinbinom(n, mu = exp(b0 + b1*X), size = .5,
zprob = exp(b0z + b1z*Z)/(1 + exp(b0z + b1z*Z)))
# Generate data with no zero-inflation component
Y.nozi <- rzinbinom(n, mu = exp(b0 + b1*X), size = .5, zprob = 0)
```

For the dependent variable that has a zero-inflation component, we set `zprob` to the formula for the inverse logit of a linear combination of an intercept (`b0z`) and a variable `Z` multiplied by a coefficient, `b1z`. This means that we are creating a logit model just like in Section 6.3.1 in the true DGP to determine whether an observation gets a 0 or not. We also set the mean parameter of the NB to the exponentiated linear combination of an intercept, `b0c`, and a coefficient, `b1c` multiplied by an independent variable, `X`. This is the systematic portion of the count model.

For the dependent variable with no zero inflation, we set `zprob` to 0, which means the condition in the `ifelse()` statement will never be true and all of the dependent variable values will be generated with the standard `rnbinom()` function. In other words, the true DGP only has a count equation. Finally, we estimate both a standard NB and a ZINB with both dependent variables (four models in total). We use the `zeroinfl()` function from the `pscl` package to estimate ZINB. The complete code is listed below.

```
# Negative Binomial vs. Zero-Inflated Negative Binomial
library(pscl)
# Zero-inflated negative binomial random number generator
rzinbinom <- function(n, mu, size, zprob){
ifelse(rbinom(n, 1, zprob) == 1, 0, rnbinom(n, size = size, mu = mu))
}

set.seed(2837) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.zinb <- matrix(NA, nrow = reps, ncol = 4) # Empty matrix to store the
                                                  # estimates
b0z <- -.8 # True value for the inflation intercept
b1z <- .3 # True value for the inflation slope
b0c <- .2 # True value for the count intercept
b1c <- .5 # True value for the count slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X
Z <- rnorm(n, X, 1) # Inflation independent variable

for(i in 1:reps){
# Generate data with a zero-inflation component
Y.zi <- rzinbinom(n, mu = exp(b0c + b1c*X), size = .5,
zprob = exp(b0z + b1z*Z)/(1 + exp(b0z + b1z*Z)))

# Generate data with no zero-inflation component
Y.nozi <- rzinbinom(n, mu = exp(b0c + b1c*X), size = .5, zprob = 0)
model.nb1 <- glm.nb(Y.zi ~ X) # Standard negative binomial
model.nb2 <- glm.nb(Y.nozi ~ X)
model.zinb1 <- zeroinfl(Y.zi ~ X | Z, dist = "negbin") # Zero-inflated model
model.zinb2 <- zeroinfl(Y.nozi ~ X | Z, dist = "negbin")

# Store the estimates of the coefficient on X (count equation)
par.est.zinb[i, 1] <- model.nb1$coef[2] # Standard NB, with ZI
par.est.zinb[i, 2] <- model.nb2$coef[2] # Standard NB, no ZI
par.est.zinb[i, 3] <- as.numeric(model.zinb1$coef$count[2])  # ZI NB, with ZI
par.est.zinb[i, 4] <- as.numeric(model.zinb2$coef$count[2]) # ZI NB, no ZI
cat("Completed", i, "of", reps, "\n")
}
```

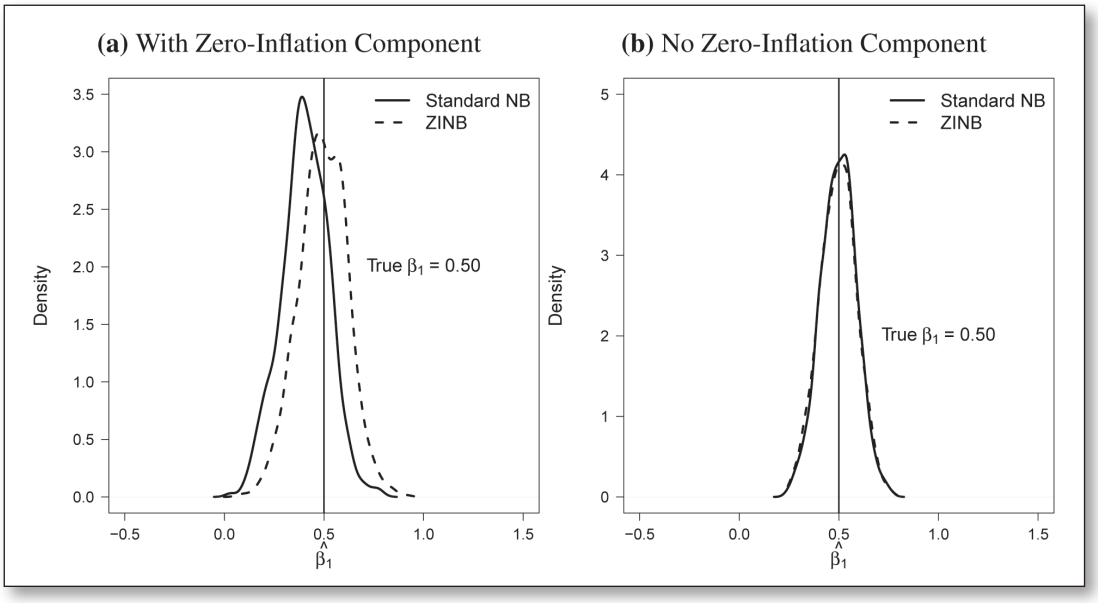
We plot the densities of the simulated estimates from the standard NB (solid lines) and ZINB (dashed lines) in Figure 6.4. Panel (a) shows the results when the true DGP includes a zero-inflation component and Panel (b) shows results without zero inflation in the true DGP. Notice that when there is a zero-inflation component (Panel a), the standard NB shows a downward bias; its estimates are being "pulled down" by the excess 0s. In contrast, the ZINB estimates are centered directly on the true parameter value of 0.50. The picture changes when we look at the DGP with no zero-inflation component (Panel b). In that case, both estimators show unbiasedness—both are centered on the true coefficient value. But notice that the standard NB is slightly more efficient. It has higher density at the peak and smaller spread (standard deviations of 0.09 [NB] and 0.093 [ZINB]).

Overall, we see from these count model simulations that different estimators perform better or worse under different DGP conditions. It is not wise to simply choose one estimator for all count data. Instead, these results show us the importance of carefully thinking about the theoretical and empirical features of the data and using that information to inform estimator selection. We will see this illustrated again in our next extended example on duration models.

6.4.3 Duration Models

Duration models, or event history models, are commonly used in social science to study the time it takes for some process to occur. Examples include how long

Figure 6.4 Distribution of Zero-Inflated and Standard Negative Binomial β_1 Estimates With and Without a Zero-Inflation Component in the DGP



a war lasts, the length of a workers' strike, or the length of time for members of social groups to transition to adulthood. The dependent variable in these examples denotes the number of periods until the transition from one state to the other occurs (e.g., "strike" to "not strike"). There are several estimators available for duration models and many more intricacies associated with them than we can cover here, but we will give a basic introduction in this extended example.[18]

We need to define some key terms before we proceed with this section. The first is the survival function, which is defined as the probability that the transition occurs (e.g., a strike ends) later than some specified time, t .

$$S(t) = \Pr(T > t) \tag{6.14}$$

Here T is a random variable that signifies the time in which the transition occurs. The complement of the survival function is the CDF, $F(t)$, or the cumulative probability of transition.

$$F(t) = \Pr(T \leq t) = 1 - S(t) \tag{6.15}$$

The derivative of $F(t)$, denoted $f(t)$, can be used to produce the hazard rate, $h(t)$:

$$h(t) = \frac{f(t)}{S(t)} \tag{6.16}$$

The hazard rate is the risk of an observation experiencing the transition given that it has not experienced it up to the current period. The hazard rate gets the most attention in applications of duration models because it most naturally comports with research questions, such as "What is the chance a strike will end, given that it has lasted 3 months?" We can include a systematic component (i.e., independent variables) in answering this question, producing estimates of how each one influences the chances of the strike ending.

A key feature of these models is how they handle the baseline hazard rate, which is the risk of the transition occurring when the systematic component of the model is equal to zero. Thus, statements made based on the estimated coefficients of a duration model are in the language of relative risk. At any point in time, there is some chance of a strike ending. The estimators we explore here produce parameter estimates that describe the relative increase in that chance based on a change to an independent variable. Those estimates can depend on the assumed shape of the baseline hazard function. There are several parametric models available, such as the exponential and the Weibull. The former assumes a constant baseline hazard, while the latter can accommodate a baseline hazard that increases or decreases (monotonically) over time. Another possibility is the Cox (1972) Proportional Hazards Model, which has become quite popular because it

[18]We recommend Box-Steffensmeier and Jones (2004) as a good starting point for readers interested in learning more.

leaves the baseline hazard unspecified. We examine the Cox model in detail below.[19]

The Cox Model

The simulation we present here is based on Desmarais and Harden (2012). We seek to demonstrate how simulation can be used to address a methodological problem. Specifically, we examine two different methods for estimating the Cox model: (1) the conventional partial likelihood maximization (PLM) estimator proposed by Cox (1972) and (2) an iteratively reweighted robust (IRR) estimator that is robust to outliers (IRR, see Bednarski, 1989; Bednarski, 1993).

The Cox model is unique in that it only requires assumptions about the independent variables to identify the model. Information on the baseline hazard rate is not needed. This means, for instance, that the effect of some independent variable on the hazard rate can be estimated without considering the complicated dynamics that are common among the observations. The ubiquitous approach to estimating the parameters of the Cox model is to select the parameters that maximize the partial likelihood, a method similar to ML that we will call PLM.[20]

However, measurement error, omitted variables, and functional form misspecification all represent distinct problems for the Cox model. Bednarski (1989) shows that all of these problems result in disproportionately influential right-tail outliers or observations that last significantly longer than they are predicted to last. This produces bias in model results leading to incorrect inferences. Generally, these specification issues represent a failure of the model to reflect the real DGP. Moreover, since there is no error term or auxiliary parameter (e.g., variance term) in the Cox model, it cannot "account" for observations that, due to real-world complexity in the DGP, depart from the estimated failure ratios. This can be seen most clearly in predictions from the model that diverge markedly from the actual outcomes, such as a war that, in reality, lasted twice as long as the median war in the sample, but was predicted to be only half as long as the median.[21]

The IRR method attempts to minimize the impact of outliers (though not necessarily eliminate them completely). The method first creates a measure of "outlyingness" for each observation. Given a certain value of the hazard of event occurrence, a greater outlyingness penalty accrues with each time unit that goes by without the event occurring. For example, if the systematic component of the model predicts that a particular observation should be one of the first in the data to experience the event, but instead it was one of the last, that observation would receive a large

[19]We do not examine any parametric models to avoid complexity. The `R` package `survival` parameterizes those models with a slightly different DGP than we use for the Cox model.

[20]As Cox (1975) shows, the PLM converges, in the sample size, to the ML estimator.

[21]The emphasis on differences between reality and prediction is important. Simply observing a large value on the dependent variable is not enough to label that case as an outlier. Indeed, if that case's covariate values produce a prediction of a long duration, then it is not an outlier.

outlyingness value. IRR then adjusts each observation's influence on the model estimates based on outlyingness, with some percentage of observations having no influence on the estimation (see Bednarski, 1989; Desmarais & Harden, 2012).

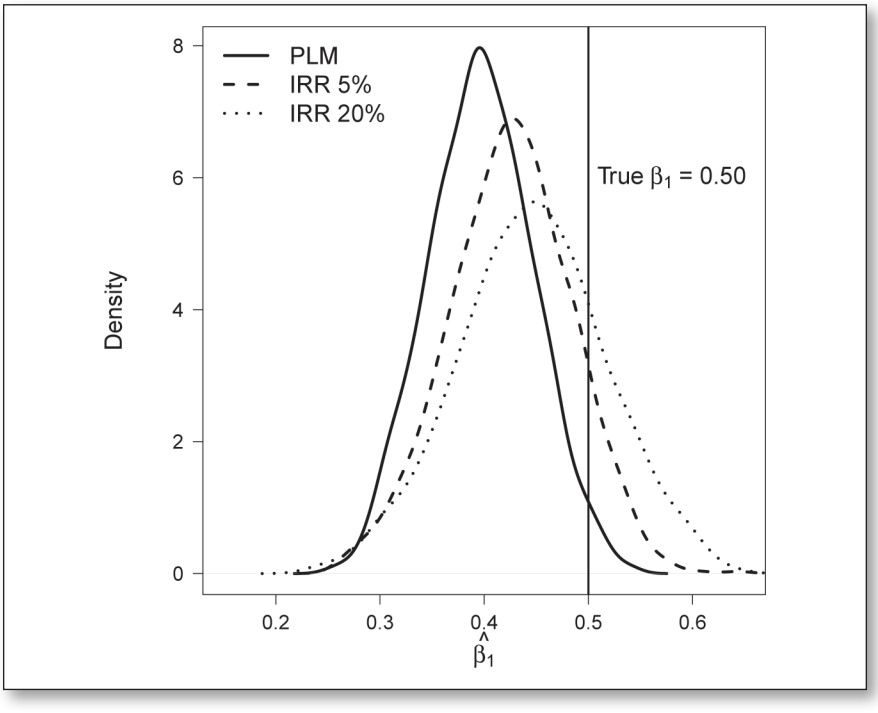
The analyst must choose the level at which outliers are downweighted, which we show below amounts to a tradeoff between bias and efficiency. Downweighting outliers is appropriate if the downweighted observations depart from the DGP of the rest of the data due to measurement or specification errors. However, if all of the observations are consistent with the true DGP, this downweighting reduces the sample size with no apparent benefit. We can illustrate this with simulation using the `R` packages `survival` (for PLM) and `coxrobust` (for IRR).

Our strategy is to generate a model with two independent variables, X and Z, then estimate PLM and IRR only including X in the specification (i.e., an omitted variable problem). Additionally, we estimate two versions of IRR: (1) one in which we set the downweighting such that the top 5% of observations in terms of outlyingness get completely downweighted and (2) one in which the top 20% get completely downweighted. We produce the dependent variable using an exponential distribution with the systematic component of the model as the parameter λ. Again, notice that the variable `Z` is included in the true DGP, but not in the estimation of PLM or either IRR specification. Finally, we store the estimate of the coefficient on X (β_1) from each of the three models.

```
# Duration Models (Cox PH)
library(survival)
library(coxrobust)

set.seed(4679) # Set the seed for reproducible results

reps <- 1000 # Set the number of repetitions at the top of the script
par.est.cox <- matrix(NA, nrow = reps, ncol = 3) # Empty matrix to store the
                                                 # estimates
b1 <- .5 # True value for the slope
n <- 1000 # Sample size
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X
Z <- runif(n, -1, 1) # Create an omitted variable

for(i in 1:reps){ # Start the loop
Y <- rexp(n, exp(b1*X + Z)) # Generate survival times
Ys <- Surv(Y, event = rep(1, n))
model.plm <- coxph(Ys ~ X, method = "breslow") # Standard PLM estimator
model.irr <- coxr(Ys ~ X, trunc = .95) # IRR, downweighting outliers (5%)
model.irr2 <- coxr(Ys ~ X, trunc = .8) # IRR, even more downweighting (20%)
par.est.cox[i, 1] <-model.plm$coef[1] # Store the estimates
par.est.cox[i, 2] <- model.irr$coef[1]
par.est.cox[i, 3] <- model.irr2$coef[1]
cat("Completed", i, "of", reps, "\n")
}
```

Figure 6.5 plots the distribution of the estimates for PLM (solid line), IRR with 5% truncation (dashed line), and IRR with 20% truncation (dotted line). Notice that all three estimators show bias, as none are centered on the true coefficient value of 0.50. However, with means of 0.397 (PLM estimates), 0.425 (IRR 5%), and 0.446 (IRR 20%), the bias gets smaller as the downweighting becomes more aggressive. Figure 6.5 also shows that the variance in the parameter estimates also differs across models; the standard deviations of the estimates are 0.050 (PLM), 0.058 (IRR 5%), and 0.071 (IRR 20%). This shows the tradeoff between bias and efficiency between the two estimators. IRR reduces bias, but is less efficient. PLM is the most biased, but also the most efficient. In this example, the IRR 20% estimator performs best according to MSE (which accounts for both bias and efficiency).[22]

Figure 6.5 The Effect of Omitted Variable Bias on PLM and IRR Estimates



This simulation suggests that it is not wise to simply always use PLM or always use IRR. This leads to the question of how a researcher could decide between PLM and IRR in a sample of data. The relative performance of IRR to

[22]The MSE values are 0.013 (PLM), 0.009 (IRR 5%), and 0.008 (IRR 20%).

PLM depends on properties of the sample that are likely unknown. This presents a clear problem in applied research. Given a sample of data and a specification of the model, it is important to determine which estimator more closely characterizes the DGP of theoretical interest.

Desmarais and Harden (2012) introduce the cross-validated median fit (CVMF) test to allow researchers to determine which method provides a better fit to the majority of their data. When the PLM method provides a better fit to the majority of the observations in the sample, it is clear that a handful of outliers are not driving the results, and IRR is inferior. However, when PLM only fits a minority of the observations better than IRR, this is evidence that the benefits from the downweighting in IRR will be realized. Desmarais and Harden (2012) show in simulations similar to this one that the CVMF test, on average, selects the estimator that produces coefficient estimates closest to the true values.

6.5 COMPUTATIONAL ISSUES FOR SIMULATIONS

In running the code for these examples yourself, you may have noticed that some of them take a few minutes to run. These simulations could easily be expanded in a number of ways (e.g., more parameters or different sample sizes) that would complicate the simulation and extend (potentially dramatically) how much time it takes them to run. In Chapter 4, we briefly discussed ways of making code more efficient, but there are other ways to manage the workload of a simulation project. We close this chapter with some useful strategies for doing this. Specifically, we discuss the use of research computing and parallel processing.

6.5.1 Research Computing

The term *research computing* could refer to many different things, but what we mean is any infrastructure designed for large-scale execution of computing problems. Many universities, for example, maintain clusters of computers to which users can submit `R` script files as "jobs" to be executed. That means you create a script file that performs the simulation and saves the output in some form. You then submit the script file to the computing cluster. Submission procedures will vary depending on the actual computing cluster you are using, but many use Unix systems that use commands such as `bsub R CMD BATCH filename.R`. Once submitted, the cluster performs the simulation and returns the output that you can save to your own computer for analysis. This frees up your computer from doing work that may take a long time.

Using research computing means your script file must be organized to save anything you might want to use later. You will likely not be able to go back to the `R` session on the computing cluster to grab an object, so you should plan accordingly. One possibility is to use the `save.image()` command that we discussed in Chapter 3, which saves every object in the current `R` workspace. If you use this approach, this should be the last line of your code in your script file.

For instance, to save the current R workspace to the file "example.RData" you could type the following code.

```
save.image("example.RData")
```

Recall that once you have "example.RData" on your own computer, you can then access it with the load() command.

```
load("example.RData")
```

The computing cluster may operate faster than your computer, but it also may work at about the same speed. Regardless, using research computing is a good way to execute long jobs in a more convenient place, freeing your own computer for other tasks.

6.5.2 Parallel Processing

Another way to manage simulation projects so that they run faster is to use parallel processing. Parallel processing refers to the completion of multiple computing operations simultaneously—almost like making your computer "walk and chew gum at the same time." Think back to any of the simulations we have done so far. The typical approach is to iterate a for loop for 1,000 repetitions, generate a sample of data, and estimate one or more models each time. As it turns out, there is no real need for the for loop to work sequentially. It is not imperative that the sample generated in Iteration 2 exist before the sample in Iteration 543. Only the logical structure of the for loop makes that the case. Parallel processing speeds up the process by taking parts of the code that can be done at any time and distributes them to different computing nodes to work on. The results are then combined at the end.

This could occur in many different ways. For example, we could use a research computing cluster and divide parts of the simulation among several different computers. On a smaller scale, it is possible to divide code among the cores on a standard multicore desktop or laptop computer. These computers have the capacity to execute different "jobs" at the same time by dividing jobs among their cores. Here we will do a brief example of a simulation in which we divide the 1,000 iterations of the for loop among several cores. To do so, we need the following packages: snow, doSNOW, and foreach.

The example we will use is the simulation on standard NB and ZINB (see Section 6.4.2 of this chapter). That simulation creates two dependent variables and estimates two fairly complex models on each of them 1,000 times, which leads to a lengthy computation time. We will "parallelize" the code to see if we can shorten the completion time by dividing the work among the cores of a quadcore laptop computer.[23]

[23]Specifically, we did this on a Lenovo quadcore ThinkPad T520 with an Intel i5 processor and 8GB of RAM.

First, we need to alter the code slightly by making it a function. This makes it easier for R to store the results from the different cores together.[24] We copy the code from that simulation into a function called zinb.sim().

```
# Simulation Function
zinb.sim <- function(n = 1000){
require(pscl)
par.est.zinb <- matrix(NA, nrow = 1, ncol = 4) # Empty matrix to store the
                                               # estimates
b0 <- .2 # True value for the intercept
b1 <- .5 # True value for the slope
X <- runif(n, -1, 1) # Create a sample of n observations on the
                     # independent variable X

# Generate data with a zero-inflation component
Y.zi <- rzinbinom(n, mu = exp(b0 + b1*X), size = .5,
zprob = exp(b0 + b1*X)/(1 + exp(b0 + b1*X)))
# Generate data with no zero-inflation component
Y.nozi <- rzinbinom(n, mu = exp(b0 + b1*X), size = .5, zprob = 0)
model.nb1 <- glm.nb(Y.zi ~ X) # Standard negative binomial
model.nb2 <- glm.nb(Y.nozi ~ X)
model.zinb1 <- zeroinfl(Y.zi ~ X | X, dist = "negbin") # Zero-inflated model
model.zinb2 <- zeroinfl(Y.nozi ~ X | X, dist = "negbin")
# Store the estimates of the coefficient on X (count equation)
par.est.zinb[ , 1] <- model.nb1$coef[2] # Standard NB, with ZI
par.est.zinb[ , 2] <- model.nb2$coef[2] # Standard NB, no ZI
par.est.zinb[ , 3] <- as.numeric(model.zinb1$coef$count[2]) # ZI NB, with ZI
par.est.zinb[ , 4] <- as.numeric(model.zinb2$coef$count[2]) # ZI NB, no ZI
return(par.est.zinb)
}
```

Notice that this function is set up to return the matrix par.est.zinb, which has one row and four columns. Each cell in the matrix contains an estimate of β_1 from one of the four estimators.[25] Next, we load the packages snow, doSNOW, and foreach. We use the makeCluster() and registerDoSNOW()

[24]There are actually many advantages to simulation with functions rather than the more "procedural" approach (i.e., creating objects then running a for loop) we have focused on up to this point. In particular, functions make "debugging" (finding and fixing errors) easier, can be tested more quickly, and carry more information about the environments in which they were created with them. This last benefit comes up with parallel computing because information about the environment often needs to be placed on different computing nodes. We elected to focus on the procedural approach for the majority of this book because functions and their associated capabilities can be somewhat difficult to understand for inexperienced users. We thank a manuscript reviewer for bringing this point to our attention.

[25]As in the first version of the simulation, these four are (1) the standard NB on the data that include a zero-inflation component, (2) the standard NB on the data with no zero inflation, (3) the ZINB with zero inflation, and (4) the ZINB with no zero inflation.

functions to set up the parallel processing. In this case, we use the number 4 to tell R to use four cores.

```
library(pscl)
library(snow)
library(doSNOW)
library(foreach)

cl.tmp <- makeCluster(4)
registerDoSNOW(cl.tmp)
```

Now, we are ready to run the simulation. Instead of the `for()` function, we will use the `foreach()` function. This takes slightly different syntax than we have seen before. We set the counter as `i = 1:reps`. Then, we set the argument `.combine = rbind`. This tells R to take the results from `zinb.sim()`, which is a vector of four coefficient estimates, and bind them together such that each iteration gets one row in a matrix of output. Finally, the code inside the `%` signs tells R how to execute the code. If we write `%do%`, R will execute the code serially, or one iteration at a time. We will do this first to get a baseline execution time. Notice that the only code we need inside the actual loop is the `zinb.sim()` function.

```
# Serial Processing
set.seed(2837) # Set the seed for reproducible results
reps <- 1000 # Set the number of repetitions

start.time <- Sys.time()
results <- foreach(i = 1:reps, .combine = rbind) %do% {
zinb.sim(n = 1000)
 }

end.time <- Sys.time()
end.time - start.time
```

This serial run produces the results in just under 12 minutes. The output is correct, which can be verified by checking it against the results from the original version of the simulation.[26]

```
Time difference of 11.71957 mins

head(results)
            [,1]       [,2]       [,3]       [,4]
[1,]  0.05882971 0.5623784 0.48865878 0.4683902
[2,] -0.06915600 0.3567442 0.08945868 0.3218285
[3,] -0.01693489 0.6521395 0.20225698 0.5924448
[4,]  0.27497556 0.5041720 0.46120764 0.5041924
[5,]  0.13497402 0.4206432 0.54417701 0.4912774
[6,]  0.13088511 0.4539706 0.33806668 0.4135637
```

[26]Using the `apply()` function on the results of the original simulation produces: `[1] 0.2147594 0.4963802 0.4941624 0.4864211`.

```
apply(results, 2, mean)
[1] 0.2150614 0.4975114 0.4810489 0.4860895
```

The next step is to run the exact same code in parallel and check the improvement in time. We already set up the "back end" of our code with the `makeCluster()` and `registerDoSNOW()` functions, so all we need to do is tell `R` to actually execute this code in parallel. We do that by changing `%do%` to `%dopar%`.

```
# Parallel Processing
set.seed(2837) # Set the seed for reproducible results
reps <- 1000 # Set the number of repetitions

start.time <- Sys.time()
results <- foreach(i = 1:reps, .combine = rbind) %dopar% {
zinb.sim(n = 1000)
 }

end.time <- Sys.time()
end.time - start.time
```

This produces the following results:

```
Time difference of 4.8734 mins

head(results)
             [,1]      [,2]      [,3]      [,4]
[1,]  0.15051572 0.4517511 0.5860100 0.4691888
[2,]  0.19717087 0.6604850 0.5470129 0.7190622
[3,]  0.43322114 0.4927281 0.8087086 0.4126702
[4,]  0.16557821 0.4871358 0.3542323 0.5073770
[5,]  0.46028678 0.6873374 0.7443504 0.5591115
[6,] -0.03127193 0.4493201 0.3148359 0.4799454

apply(results, 2, mean)
[1] 0.2206657 0.4998477 0.4880605 0.4924071
```

By dividing the code across four cores, the computer took less than 5 minutes to complete the exact same code—a nearly 60% reduction in time. Notice that the results also look correct (though they are not identical). This example shows that parallel processing is quite feasible in `R` and can greatly reduce the time it takes to do simulations. We recommend using this procedure or a similar one for simulations that take several minutes to run in a standard `for` loop.

Finally, we should note again a point we made in Chapter 4: replicating results from simulations executed through parallel processing may not be accomplished in some instances simply by setting the seed. This is a complicated issue that extends beyond the scope of this book, but readers should be

aware of this potential problem. Again, test your code with a limited number of iterations first to make sure it performs as expected before you launch a time-consuming simulation.

6.6 CONCLUSIONS

This chapter provided an overview of simulation with GLMs. We began with the process of simulating the DGP of a GLM, then recovering that DGP through estimation. Doing so illustrated that the basic idea of a GLM is to link the systematic component of the model to a probability distribution that produces the type of values the dependent variable can take on. We also illustrated how simulation can be used to compare competing estimators in extended examples with categorical, count, and duration models. We saw how knowing the true DGP in a simulation gives us considerable analytic leverage in assessing how estimators perform under different data conditions.

We closed this chapter with a discussion of computational issues. Many of the examples from this chapter are computationally intensive and take a relatively long time to complete. Simulations beyond those shown here could easily take hours or days to estimate on a standard computer. We addressed how research computing can make managing such computations less of a burden. We also showed a basic example of parallel processing in which we reduced run time by more than half through dividing the simulation among four cores of a computer. It is worth noting that many research computing systems available at universities consist of hundreds or even thousands of cores, making the potential time savings of parallel processing in that environment very appealing.

After two chapters of nothing but statistical simulation, it is time for a change. Social scientists use statistical methods to better understand social processes. Using simulation to illustrate and evaluate these methods is certainly a beneficial exercise. However, it is also possible to use simulation to directly evaluate substantive problems, theories, and questions. We show several examples of this in the next chapter.