As data scientists, we are often presented with a dataset and are asked to use it to produce insights. We use R to wrangle, visualize, model, and produce tables and plots for sharing or publication. When we focus on the data in hand in this way, we don’t get to consider where the data came from. The sample size and the set of variables and their scales are fixed. Yet the procedures used to gather or generate them are hugely consequential for how we should analyze the data and also the quality of the insights we can ultimately deliver. Sampling procedures have implications for how the resulting data should be analyzed. For studies that seek to measure causal effects, it matters how some units come to be treated and others left untreated.

Because these processes are so important, we wanted to make a tool that would help data scientists and other researchers **imagine** their data **before** they collect it so that any changes to process can be made before it’s too late.

When the data is already collected, the tool allows you to imagine your data before you **analyze** it. When we make data wrangling and modeling decisions based on the results we find under each procedure, or using model fit statistics. We use the actual data because we don’t have a good substitute: data with the same structure and variables that we have collected.

This post introduces the fabricatr package, whose role in the DeclareDesign suite of packages is to simulate data structure and variables. The philosophy behind it. fabricatr helps you to think about your data before you start analysis or even collection. What are the units? How are they structured? What measurements will you take? What are their ranges and how are they correlated? fabricatr can help you simulate mock data before you collect the real data, and test out different estimation strategies without worrying about biasing your inferences.

**Imagining your data structure**

Most simply, fabricatr will create a single-level data structure given a number of units.

library(fabricatr)

fabricate(N = 100, temp\_fahrenheit = rnorm(N, mean = 80, sd = 20))

## Warning: `is\_lang()` is deprecated as of rlang 0.2.0.

## Please use `is\_call()` instead.

## This warning is displayed once per session.

## Warning: `lang\_name()` is deprecated as of rlang 0.2.0.

## Please use `call\_name()` instead.

## This warning is displayed once per session.

| **ID** | **temp\_fahrenheit** |
| --- | --- |
| 001 | 56.6 |
| 002 | 46.3 |
| 003 | 90.5 |
| 004 | 75.1 |
| 005 | 85.1 |
| 006 | 102.8 |

Social science data is often **hierarchical**. For example, schools have classrooms that have students. fabricatr shines here with the add\_level command. By default, new levels are nested within the levels above them.

library(fabricatr)

fabricate(

# five schools

school = add\_level(N = 5,

n\_classrooms = sample(10:15, N, replace = TRUE)),

# 10 to 15 classrooms per school

classroom = add\_level(N = n\_classrooms),

# 15 students per classroom

student = add\_level(N = 15)

)

## Warning: `lang\_modify()` is deprecated as of rlang 0.2.0.

## Please use `call\_modify()` instead.

## This warning is displayed once per session.

| **school** | **n\_classrooms** | **classroom** | **student** |
| --- | --- | --- | --- |
| 1 | 12 | 01 | 001 |
| 1 | 12 | 01 | 002 |
| 1 | 12 | 01 | 003 |
| 1 | 12 | 01 | 004 |
| 1 | 12 | 01 | 005 |
| 1 | 12 | 01 | 006 |

The real world often produces messy, overlapping hierarchies. For example, student data may be collected from middle school and also high school, in which case students are nested in two different schools, but those schools are not nested within each other. Here’s how to make such “cross-classified” data. The rho parameter governs how correlated primary\_rank and secondary\_rank should be.

dat <-

fabricate(

primary\_schools = add\_level(N = 5, primary\_rank = 1:N),

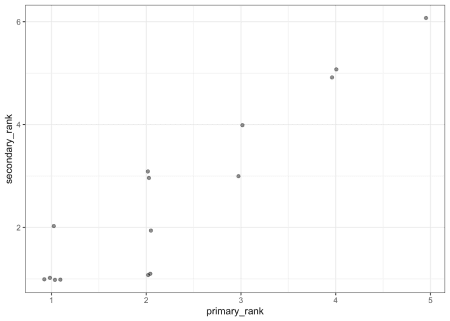
secondary\_schools = add\_level(N = 6, secondary\_rank = 1:N, nest = FALSE),

students = link\_levels(N = 15, by = join(primary\_rank, secondary\_rank, rho = 0.9))

)

## `link\_levels()` calls are faster if the `mvnfast` package is installed.

ggplot(dat, aes(primary\_rank, secondary\_rank)) + geom\_point(position = position\_jitter(width = 0.1, height = 0.1), alpha = 0.5) + theme\_bw()



Similarly, you can create longitudinal data via cross\_levels:

fabricate(

students = add\_level(N = 2),

years = add\_level(N = 20, year = 1981:2000, nest = FALSE),

student\_year = cross\_levels(by = join(students, years))

)

| **students** | **years** | **year** | **student\_year** |
| --- | --- | --- | --- |
| 1 | 01 | 1981 | 01 |
| 2 | 01 | 1981 | 02 |
| 1 | 02 | 1982 | 03 |
| 2 | 02 | 1982 | 04 |
| 1 | 03 | 1983 | 05 |
| 2 | 03 | 1983 | 06 |

**Imagining your variables**

R has lots of great tools for simulating variables. In some cases, though, common kinds of outcome variables are surprisingly tough to simulate. fabricatr collects a small number of functions to create variable types commonly used by social scientists, with simple syntax.

Common Social Science Variables:

Binary outcomes (e.g. turnout)

The simplest possible outcome of interest is a binary outcome: one that can be either true or false. An example of this sort of outcome in a social science research context is whether or not a subject performs an action: Does a person seek medical treatment? Does a person turn out to vote? Does a person write a letter to their congressional representative? Is an infrastructure project completed on time? These sorts of questions are typically modelled as binary outcomes. The “yes” or “no” answer to your question of interest is represented as 1 (“yes”) or 0 (“no”) numerically.

In **fabricatr**, binary data can easily be modeled with the [draw\_binary()](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html) function:

voter\_turnout = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.4, N = 100)

[table](https://rdrr.io/r/base/table.html)(voter\_turnout)

| **voter\_turnout** | **Freq** |
| --- | --- |
| 0 | 58 |
| 1 | 42 |

draw\_binary requires that you supply two pieces of data: prob, which specifies either a single probability or a vector of probabilities (one for each unit), the probability of getting a “1” in your simulated data. For instance, here we are modeling a turnout rate of 40%. Second, it requires N, which specifies how many observations to create.

It is also possible to specify the probability stochastically. Imagine if you wish to model a population where individuals under age 40 have a low turnout rate, while individuals above age 40 have a high turnout rate. This is very simple with **fabricatr**:

population <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 100,

age = [round](https://rdrr.io/r/base/Round.html)([runif](https://rdrr.io/r/stats/Uniform.html)(N, 18, 85)),

turnout = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = [ifelse](https://rdrr.io/r/base/ifelse.html)(age < 40, 0.4, 0.7), N=N)

)

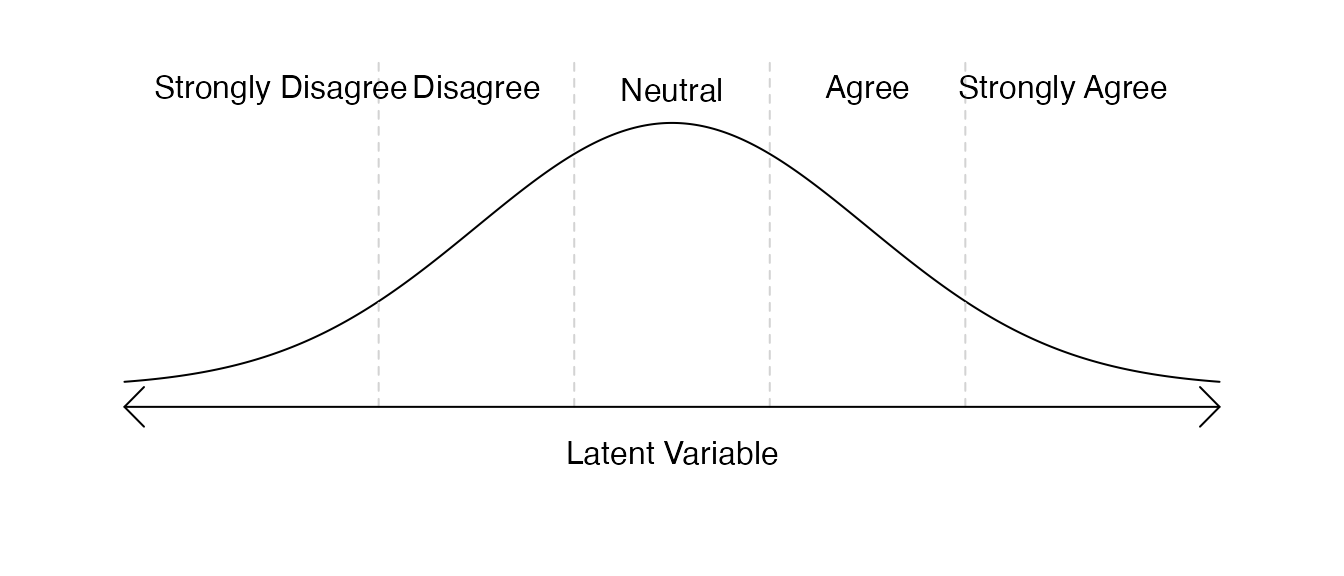
This example introduces you to a few new functions. First, we can wrap multiple variable creation commands in a fabricate call. This will ensure the result is a data frame, and that each variable creation command has access to the previous variables. Second, you see that we can model age using R’s built-in runif function. If you would like more information about modeling statistical distributions in R, see the [?Distributions](https://rdrr.io/r/stats/Distributions.html) manual page. Third, you see that we model turnout probability statistically; if the user has an age below 40, they receive turnout probability 0.4. If not, they receive turnout probability 0.7.

Ordered data (e.g. satisfaction)

The type of data generated for many survey questions is “ordered”. Ordered data includes the “Likert scale”, and is used when respondents to a survey report an outcome which has a logical ordering from lowest to highest but is not necessarily measured in numerically comparable terms. Consider a basic question like “Do you approve of your mayor’s job performance?”. There are many ways you could measure a respondent’s answer to this question, but one of the most common is by offering the respondent the choice to “Strongly Disapprove”, “Disapprove”, feel “Neutral”, “Approve”, or “Strongly Approve”. This five point scale is ordered data.

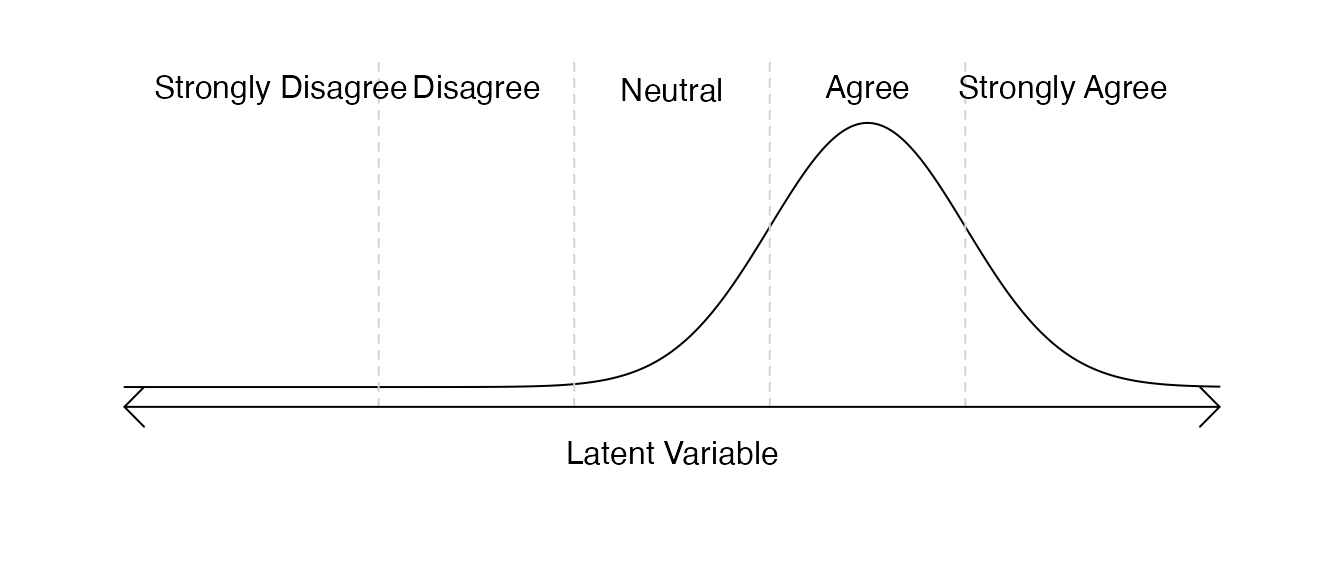
When simulating ordered data, we typically think of the problem differently, as a two-step process. The first step is to assign a respondent a score over some range, and the second step is to translate that score into the categories of the ordered variable. We call the first step a “latent variable”. In the real survey, we never see the latent variable, only the ordered data outcome. But we can rely on the latent variable to help shape our simulated data.

We generate a simulated latent variable and then translate it into a simulated ordered variable, like so:



In the above graph, most respondents (the “density” represented by the height of the curve) are neutral about the mayor, and relatively few hold extreme opinions. We generated a “normal” latent variable for our respondents and then translate this to the ordered outcome. The simplest normal variable is the “standard normal”, a random variable whose mean (center) is 0, and whose standard deviation (spread) is 1. So, although it is not shown in the graph above, the dashed lines which divide the ordered outcomes are located at -1.5, -0.5, 0.5, and 1.5.

Of course it would also be possible to affect the center and spread of this question:



Although thinking about ordered data in this latent context may seem strange at first, actually generating an ordered data outcome in **fabricatr** is quite easy:

mayor\_approval <- [draw\_ordered](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(x = [rnorm](https://rdrr.io/r/stats/Normal.html)(n = 100),

breaks = [c](https://rdrr.io/r/base/c.html)(-1.5, -0.5, 0.5, 1.5),

break\_labels = [c](https://rdrr.io/r/base/c.html)("Strongly Disagree", "Disagree",

"Neutral", "Agree",

"Strongly Agree"))

[table](https://rdrr.io/r/base/table.html)(mayor\_approval)

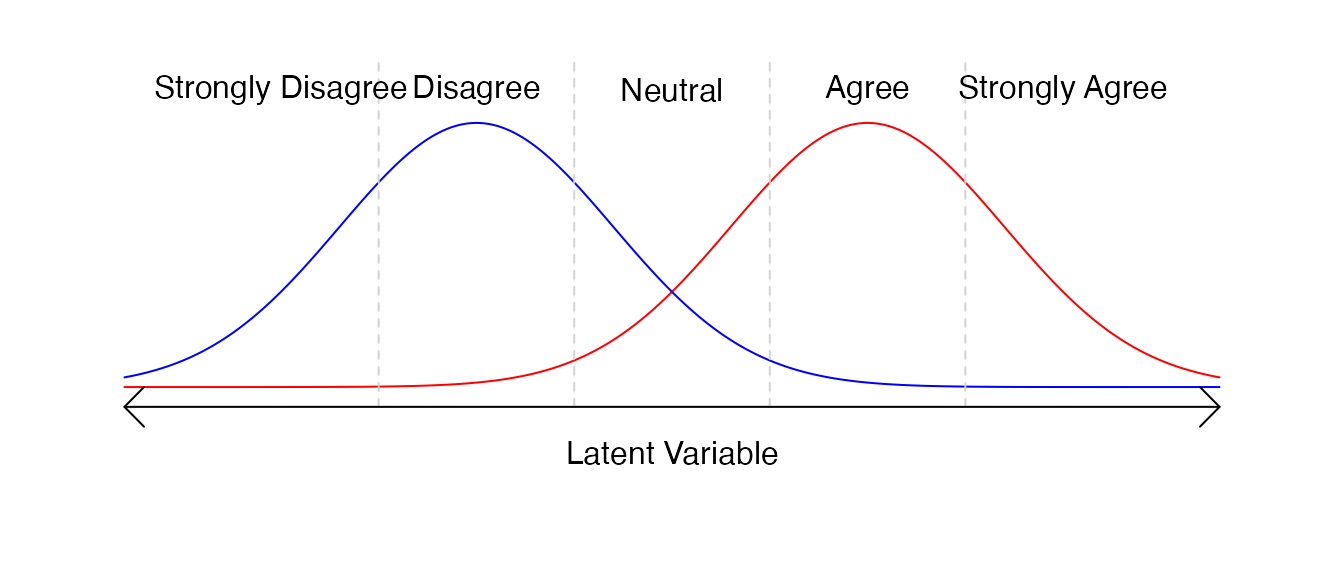
| **mayor\_approval** | **Freq** |
| --- | --- |
| Strongly Disagree | 7 |
| Disagree | 20 |
| Neutral | 42 |
| Agree | 22 |
| Strongly Agree | 9 |

Let’s look at this draw\_ordered example. draw\_ordered requires three pieces of information from us: x (the latent variable), breaks (the places in the latent variable which divide the categories in the ordered outcome), and break\_labels (which label the resulting ordered outcome).

For x, we use the rnorm command to generate 100 standard normal draws representing our respondents. For breaks and break\_labels, we mark and label the breaks as we did above.

The appeal of **fabricatr** is that we can create relationships between variables, so let’s imagine a slightly more complex data where in general the mayor is well liked by members of her political party, but not by members of the opposite political party. To do this, we’ll need to assign respondents to a political party (we’ll mark the mayor’s political party with “1” and the opposition political party with “0”) and then generate our latent variable.

First, let’s visualize the latent variable:



Here, colors indicate the two parties. We see that some members of the red party do disagree with the mayor, but the overwhelming majority do not, and vice versa for the blue party. Now, let’s put our visualization into action:

respondent\_data <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 100,

mayor\_copartisan = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.6, N),

mayor\_approval = [draw\_ordered](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(

x = [rnorm](https://rdrr.io/r/stats/Normal.html)(N, mean = -1 + 2 \* mayor\_copartisan),

breaks = [c](https://rdrr.io/r/base/c.html)(-1.5, -0.5, 0.5, 1.5),

break\_labels = [c](https://rdrr.io/r/base/c.html)("Strongly Disagree", "Disagree", "Neutral",

"Agree", "Strongly Agree")

)

)

[table](https://rdrr.io/r/base/table.html)(respondent\_data$mayor\_approval, respondent\_data$mayor\_copartisan)

|  | **0** | **1** |
| --- | --- | --- |
| Strongly Disagree | 17 | 0 |
| Disagree | 14 | 4 |
| Neutral | 9 | 12 |
| Agree | 1 | 18 |
| Strongly Agree | 1 | 24 |

Additionally, the latent variable x is more complex. Let’s consider the formula for how this variable is specified. We know that the latent variable space from -1.5 to -0.5 indicates “Disagree”. For respondents who have mayor\_copartisan equal to 0 (they are not in the mayor’s political party), their preference will be a standard normal draw centered at −1+(2∗0)=−1−1+(2∗0)=−1. For respondents who have mayor\_copartisan equal to 1 (they are in the mayor’s political party), their preference will be a standard normal draw centered at −1+(2∗1)=1−1+(2∗1)=1 – “Agree”.

It is easy to play with the location of breaks, labels for breaks, and format of the underlying latent variable. Here’s an example where the latent variable is “uniform” – there is an equal likelihood of choosing each of the options:

mayor\_approval <- [draw\_ordered](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(x = [runif](https://rdrr.io/r/stats/Uniform.html)(n = 100, min = -2.5, max = 2.5),

breaks = [c](https://rdrr.io/r/base/c.html)(-1.5, -0.5, 0.5, 1.5),

break\_labels = [c](https://rdrr.io/r/base/c.html)("Strongly Disagree", "Disagree",

"Neutral", "Agree",

"Strongly Agree"))

Users who are new to the R programmming language and want to know more about other statistical distributions beyond the norm and unif distributions used here, should run the command [?Distributions](https://rdrr.io/r/stats/Distributions.html) in the R console.

Likert Data

Many survey responses that focus on labeling agreement, support, or quality evaluation, including the example above, are called “Likert” data after the psychologist Rensis Likert. These scales are typically 4, 5, or 7 point scales of a measure. **fabricatr** includes a simple shortcut to make Likert variables without needing to fill out the breaks and break\_labels each time.

By using draw\_likert, users only need to specify the x latent variable – assumed to be distributed with breaks spaced 1 unit apart and with the data centered on 0 – and the type of Likert they would like (the number of categories). In this example, we examine the same data above, but with a 4-category Likert scale where the options are “Strongly Agree”, “Agree”, “Disagree”, and “Strongly Disagree”, with no neutral category.

mayor\_approval <- [draw\_likert](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(x = [rnorm](https://rdrr.io/r/stats/Normal.html)(n = 100),

type = 4)

[table](https://rdrr.io/r/base/table.html)(mayor\_approval)

| **mayor\_approval** | **Freq** |
| --- | --- |
| Strongly Disagree | 15 |
| Disagree | 32 |
| Agree | 36 |
| Strongly Agree | 17 |

Categorical data (e.g. demographic measures)

Surveys often collect demographic information from respondents, including age, gender, and ethnicity. By this point in our guide, you have begun to see how you might generate a continuous numerical variable like age, or how you traditional binary measures of gender might be specified. But some variables, like inclusive measures of gender, or ethnicity, (or hair color, eye color, city of residence, or many other possible variables of interest) are not “ordered” in any sense.

These data are often called “categorical” data – a given person has a probability to belong to each of the possible categories.

Imagine that researchers are conducting a survey in Kenya. They wish to capture the country’s four largest ethnic groups: Kikuyu, Luhya, Kalenjin, Luo, and are also interested in capturing the smaller Maasai group. Other respondents are identified as “Other”. When preparing their research, the researchers must simulate the ethnicity of their expected respondents, so they first gather the proportions of each group,

## Kikuyu Luhya Kalenjin Luo Maasai Other

## 0.172 0.138 0.129 0.105 0.022 0.435

**fabricatr** makes it easy to generate data based on specifications like these using draw\_categorical:

respondent\_ethnicity <- [draw\_categorical](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(

prob = [c](https://rdrr.io/r/base/c.html)(0.172, 0.138, 0.129, 0.105, 0.022, 0.435),

category\_labels = [c](https://rdrr.io/r/base/c.html)("Kikuyu", "Luhya", "Kalenjin", "Luo", "Maasai", "Other"),

N = 100)

[table](https://rdrr.io/r/base/table.html)(respondent\_ethnicity)

| **respondent\_ethnicity** | **Freq** |
| --- | --- |
| Kalenjin | 14 |
| Kikuyu | 24 |
| Luhya | 9 |
| Luo | 5 |
| Maasai | 4 |
| Other | 44 |

Data with fixed minimum and maximum values

It is often useful to generate data with fixed minimum or maximum values, but unclear how to best do so. In general the approach we would advocate to doing this is to generate a latent variable (described above) which might fall outside the minimum and maximum values, and then truncating the variable to the minimum and maximum values.

Consider an experiment with a pre-post treatment design. A series of respondents are asked about their views on government efficacy, which will be measured by the respondents scoring their government from 0 to 100. Some respondents are then given a treatment intervention: for example, information about government service delivery, or a mechanism to report government fraud or waste. Finally, researchers ask the same respondents the same question about government efficacy.

Researchers might simulate this experiment by choosing a population level Average Treatment Effect (in our example, assume a moderate treatment effect of 15 points), and adding a shock centered at this point to each treated unit’s pre-treatment outcome (in addition to adding noise to the control group). A challenge with this data is that it is naturally truncated: individuals whose scores were already quite high may now be modeled as having post-treatment scores above 100.

[set.seed](https://rdrr.io/r/base/Random.html)(19861108)

efficacy\_experiment <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 1000,

treatment = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(0.5, N),

pre\_outcome = [rnorm](https://rdrr.io/r/stats/Normal.html)(N, mean = 70, sd = 15),

post\_outcome = pre\_outcome + [rnorm](https://rdrr.io/r/stats/Normal.html)(N,

mean = [ifelse](https://rdrr.io/r/base/ifelse.html)(treatment, 15, 0),

sd = 10)

)

In fact, in this example, some respondents have pre-treatment scores above 100! This matters because the real-world data gathered later will be truncated, and so the researchers power to detect effects may be compromised because of a high density of responses near the cutoff. As a result, it is important that researchers model the truncation directly. Doing so is simple using functionality provided by R: pmin and pmax.

When used to truncate variables like this, pmax takes two arguments: a vector of numeric variables, and a minimum value you want to truncate to. Please note that, counter-intuitively, pmax is used to truncate to the **minimum** value and pmin is used to truncate to the **maximum** value. We see this design in action below:

[set.seed](https://rdrr.io/r/base/Random.html)(19861108)

fixed\_efficacy\_experiment <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 1000,

treatment = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(0.5, N),

pre\_outcome = [pmin](https://rdrr.io/r/base/Extremes.html)(

[pmax](https://rdrr.io/r/base/Extremes.html)([rnorm](https://rdrr.io/r/stats/Normal.html)(N, mean = 70, sd = 15), 0),

100),

post\_outcome = [pmin](https://rdrr.io/r/base/Extremes.html)(

[pmax](https://rdrr.io/r/base/Extremes.html)(pre\_outcome + [rnorm](https://rdrr.io/r/stats/Normal.html)(N,

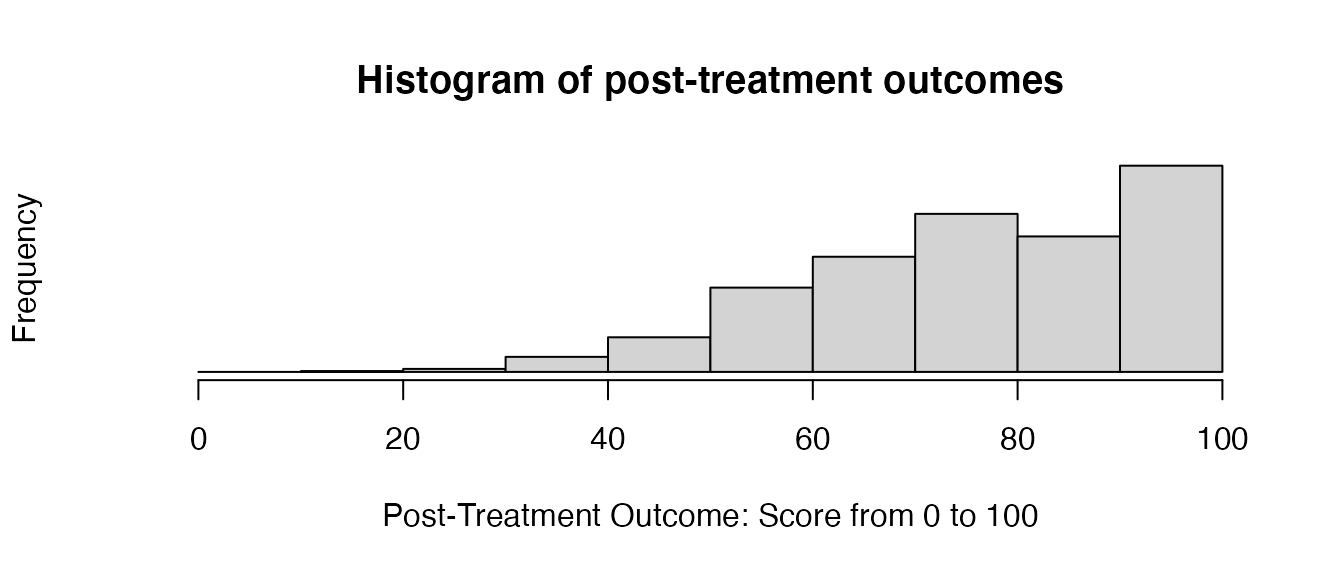
mean = [ifelse](https://rdrr.io/r/base/ifelse.html)(treatment, 15, 0),

sd = 10), 0),

100)

)

Let’s consider how pre\_outcome is created: first, each respondent is assigned a pre\_treatment score with a mean of 70 points and a standard deviation of 15 points. Next, each of those scores is fed to pmax, along with an argument 0. Any scores that are lower than 0 are bumped up and truncated to 0. Finally, these scores are fed to pmin, along with an argument 100. Any scores that are higher than 100 are bumped down and truncated to 100. The result is that our variables display the desired effect:



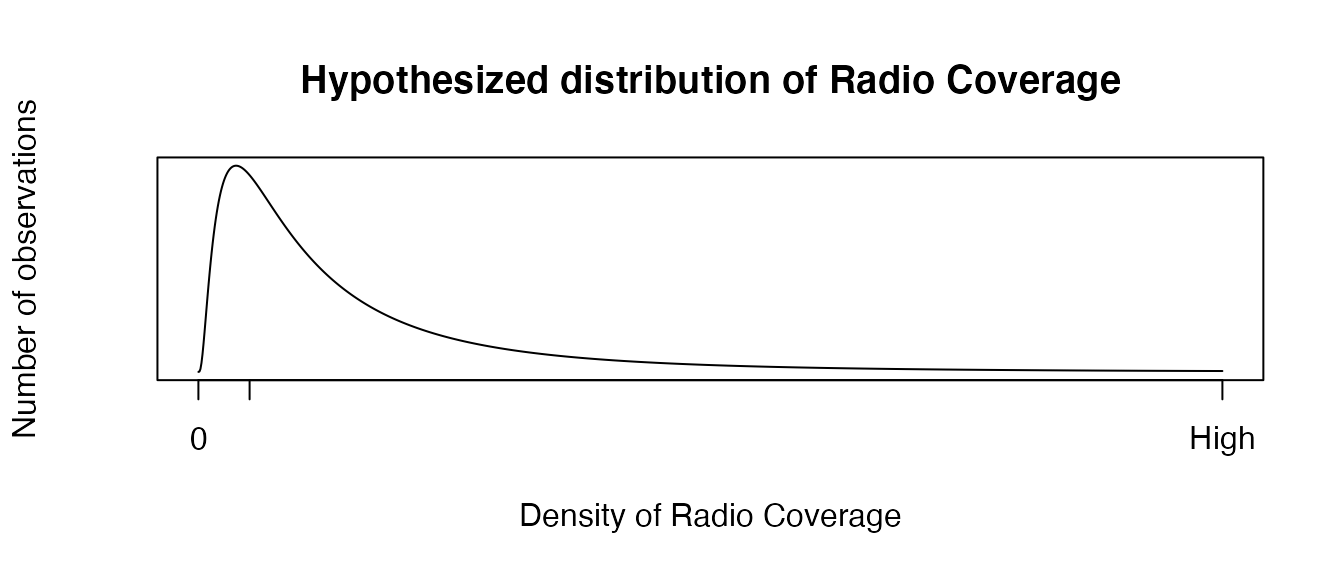
Count outcomes and skewed distributions

Researchers may be interested in “count” outcomes, which describe a count of events of interest during a fixed time period. For our example in this section, we draw from “Propaganda and Conflict: Evidence from the Rwandan Genocide” (Yanagizawa-Drott 2014), an article which uses a quasi-experimental design to establish an association between the broadcasts of Radio Télévision Libre des Mille Collines and local instances of violence during the Rwandan Genocide. Substantively, the paper’s conclusion is that higher density in radio coverage is associated with greater local participation in genocidal violence.

In our version of this data, we have two variables of interest: number of violent incidents, and density of radio coverage. We know a priori that both variables must be at least 0, we are unsure what the maximum number might be, we know that the number of violent incidents must be a “discrete” variable (in this case, a whole number) while radio coverage is “continuous” (can take fractional values).

First, to model the number of violent incidents, we make use of **fabricatr**’s draw\_count function, which is quite simple – it needs two arguments: mean, which describes the mean count, and N, which describes how many units you need count data for. Statistically, this function generates “Poisson-distributed” count data. This is one of the typical distributions used for count data.

Second, to model the radio covarage, we might believe radio coverage is an example of skewed data; in other words, few urban areas might have extremely dense radio coverage, while many rural areas might have very low radio coverage. We, thus, need a data distribution that looks like this:



R makes available a fairly simple function for generating this sort of data, rlnorm, which generates log-normal data. Log-normal data is one of the key distributions used to model data with a skew like the one we hypothesize: for example, income data in a population (where relatively few wealthy people have substantially more income than relatively many working class people). rlnorm takes three arguments: n (the number of observations), meanlog (the mean of the log-transformed data), and sdlog (the standard deviation of the log-transformed data).

rtlme\_model <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

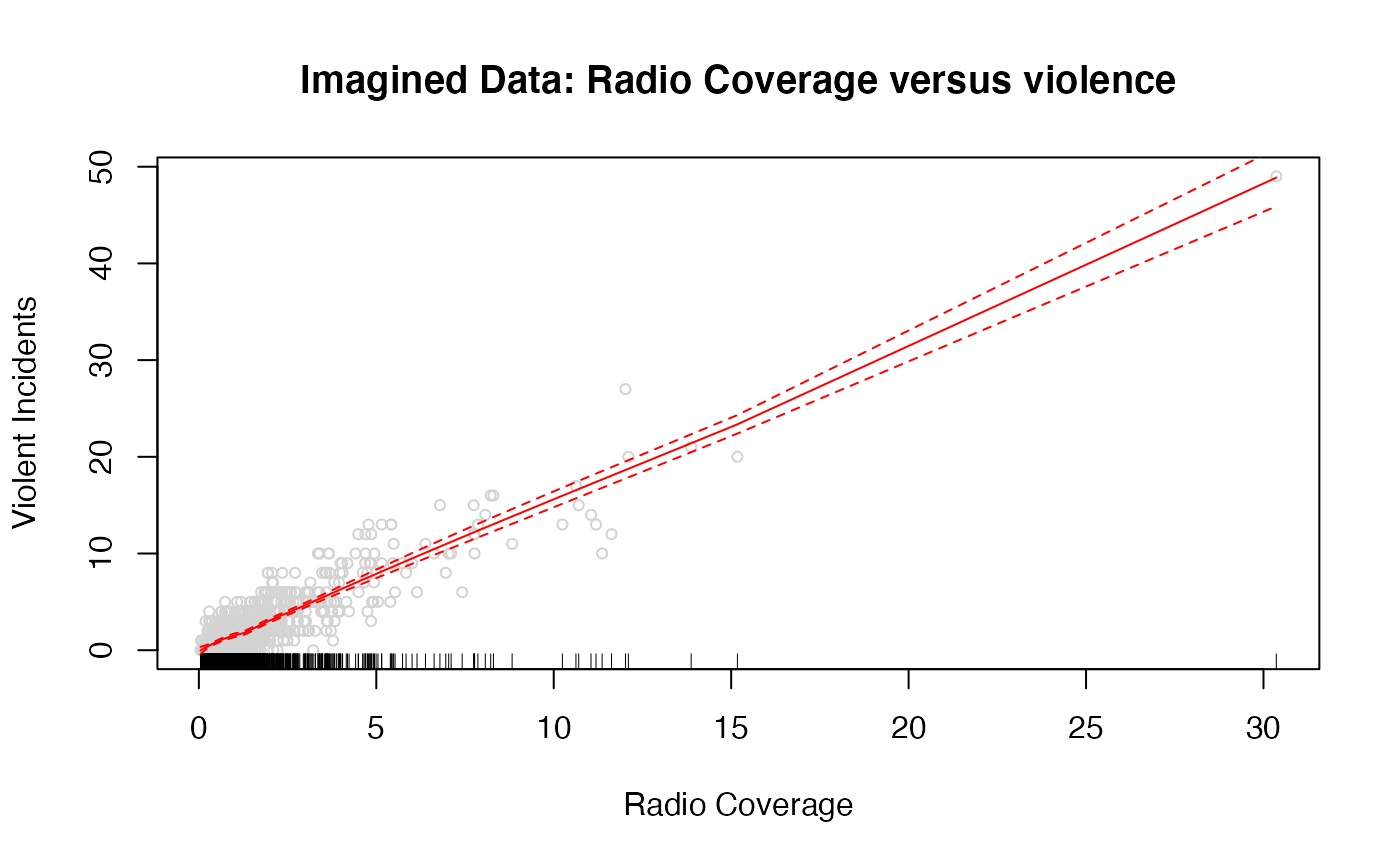
N = 1000,

radio\_coverage = [rlnorm](https://rdrr.io/r/stats/Lognormal.html)(N, meanlog=0, sdlog=1),

violent\_incident\_count = [draw\_count](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(mean = 1.5 \* radio\_coverage, N = N)

)

Replicating real data is important because the inference you draw when modeling results depends on the “support” (range of values observations take) and “density” (how many observations take each value); to the extent that real-world data will likely be dense in some regions of the data and sparse in others, it is important that your synthetic data is as well in order to recover inferences that will be accurate to your future real-world inferences, and particularly your uncertainty about those inferences.



Observe that the uncertainty around our predictions is far wider at the right end of the support, which has relatively little data, than at the left end of the support, where most of the data lives.

Thinking about choosing parameters for meanlog and sdlog is somewhat more mathematically complex in skewed distributions and depending on your use for, you might choose different parameters. Consult R’s documentation on log-normal data by running [?Lognormal](https://rdrr.io/r/stats/Lognormal.html). In addition, you may find it useful to investigate other distributions commonly used for this purpose including [?Beta](https://rdrr.io/r/stats/Beta.html) and [?Gamma](https://rdrr.io/r/stats/family.html).

population\_data <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 1000,

income = 20000 \* [rgamma](https://rdrr.io/r/stats/GammaDist.html)(N, 1.4, 0.65)

)

**Variables with intra-class correlation**

With the data structure tools described above, you can construct data that has  
within-unit and between-unit variation, for example, variation within classrooms  
and variation across classrooms in test scores. However, many times you want to  
set the level of intra-class correlation (ICC) more precisely. We help with  
draw\_normal\_icc and draw\_binary\_icc.

dat <-

fabricate(

N = 1000,

clusters = sample(LETTERS, N, replace = TRUE),

Y1 = draw\_normal\_icc(clusters = clusters, ICC = .2),

Y2 = draw\_binary\_icc(clusters = clusters, ICC = .2)

)

ICC::ICCbare(clusters, Y1, dat)

## [1] 0.09726701

ICC::ICCbare(clusters, Y2, dat)

## [1] 0.176036

**Ordered outcomes**

We provide a set of tools for discrete random variables (including ordered outcomes). We take a latent variable (i.e., test\_ability) and transform it into an ordered variable (test\_score).

dat <-

fabricate(

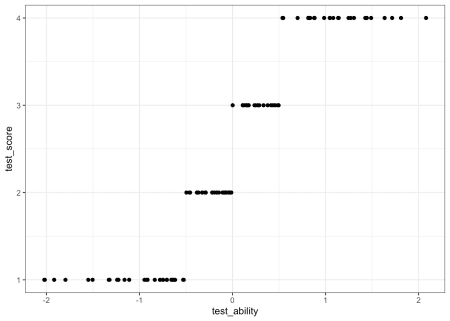
N = 100,

test\_ability = rnorm(N),

test\_score = draw\_ordered(test\_ability, breaks = c(-.5, 0, .5))

)

ggplot(dat, aes(test\_ability, test\_score)) + geom\_point() + theme\_bw()



fabricatr is compatible with almost any R variable creation function. We highlight other terrific R packages that help simulate social science-relevant variables below:

In general, **fabricatr** is going to be compatible with any existing packages you use to generate synthetic data in one of two ways: either using those packages to create variables within a fabricate call, or using those packages to make complete data frames which are then imported into a fabricate. Below we provide examples for some of the most popular packages that serve this purpose.

## ****wakefield****: simulating common demographic features

**wakefield**  is a popular R package for creating synthetic data. **wakefield**’s strength is that it can quickly generate common variables, especially for human demographic features. **wakefield** can easily be integrated into a **fabricatr** workflow in one of two ways: using **wakefield** to create individual variables within a fabricate call, or using **wakefield** to make a data frame and importing that data frame into a fabricate call.

In this example, we create a data-set of participants in a survey experiment, using **wakefield** to generate the demographic variables

[library](https://rdrr.io/r/base/library.html)([wakefield](https://github.com/trinker/wakefield))

survey\_experiment\_df <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 50,

treatment = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.5, N = N),

age = age(n = N),

race = race(n = N),

sex = sex(n = N),

)

In addition to creating variables within a fabricate call, users can import completed **wakefield** data frames into a fabricate call:

survey\_experiment\_df <- r\_data\_frame(

n = 50,

age,

race,

sex)

fabricatr\_df <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

data = survey\_experiment\_df,

treatment = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.5, N = N)

)

## ****randomNames****: Plausible names for human subjects

**randomNames**  is a package that does one thing well: generate random names for human subjects (including with specified genders and ethnicities). The primary use case would be to use this as part of generating a variable within a fabricate call. In the below example, we use fabricate to generate some other demographic data, and then randomNames to generate matching names.

[library](https://rdrr.io/r/base/library.html)([randomNames](https://centerforassessment.github.io/randomNames))

experiment\_data <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 50,

treatment = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.5, N = N),

is\_female = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.5, N = N),

patient\_name = randomNames(N, gender=is\_female)

)

Note that we make use of the existing is\_female variable from the fabricate call to ensure randomNames generates gender-typical names.

## Modeling causality with DAGs and ****simcausal****

Users who are familiar with the DAGS (directed acyclic graphs) model of causal inference may have interest in using the **simcausal** package, which allows users to specify a DAGS model and then sample from it. Integrating this package with **fabricatr** is likely to involve using **simcausal** first to specify a model, simulating data from the model, and then importing data into a fabricate call for further user with **fabricatr**.

Consider this example, common in the literature on educational attainment and school outcomes, where students come from families that have a wealth parameter, assignment to schools is based partially on wealth, and test outcomes (testoutcome) is based on both school quality and wealth.

[library](https://rdrr.io/r/base/library.html)("simcausal")

# Define DAG

D <- DAG.empty() +

node("wealth", distr = "rnorm",

mean = 30000,

sd = 10000) +

node("schoolquality", distr = "runif",

min = 0 + (5 \* (wealth > 50000)),

max = 10) +

node("testoutcome", distr = "runif",

min = 0 + 0.0001 \* wealth + 0.25 \* schoolquality,

max = 10)

# Freeze DAG object

set\_dag <- set.DAG(D)

# Draw data from DAG

df <- sim(set\_dag, n = 100)

# Pass into fabricate call and make new variables as necessary

[fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(df,

passed\_test = testoutcome > 6,

eligible\_for\_snap = wealth < 25000)

## Survival and duration models with ****simsurv****

**simsurv** is a package dedicated to generating panel survival data. The most likely way you might integrate **simsurv** with **fabricatr** would be to use **fabricatr** to generating covariates which can then be imported into **simsurv** to model in a hazard or duration context.

Here, our example will be a clinical trial of a cancer drug. Participants have the expected biographical data: age, gender, whether the patient smokes, the disease stage, assignment to treatment, and a KPS score (commonly used to evaluate overall patient health).

Survival data creates a ragged longitudinal survey; some patients will die during the course of the trial, removing them as observations. Others will continue alive until the end of the trial. We specify a “hazard function”, which tells **simsurv** how the course of patient survival will change over time. Covariates with positive betas increase risk of death, while covariates with negative betas decrease risk of death. We will track patients for 5 years after treatment.

[library](https://rdrr.io/r/base/library.html)(simsurv)

# Simulate patient data in a clinical trial

participant\_data <- [fabricate](https://declaredesign.org/r/fabricatr/reference/fabricate.html)(

N = 100,

age = [runif](https://rdrr.io/r/stats/Uniform.html)(N, min = 18, max = 85),

is\_female = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.5, N = N),

is\_smoker = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.2 + 0.2 \* (age > 50), N = N),

disease\_stage = [round](https://rdrr.io/r/base/Round.html)([runif](https://rdrr.io/r/stats/Uniform.html)(N, min = 1 + 0.5 \* (age > 65), max = 4)),

treatment = [draw\_binary](https://declaredesign.org/r/fabricatr/reference/draw_discrete.html)(prob = 0.5, N = N),

kps = [runif](https://rdrr.io/r/stats/Uniform.html)(N, min = 40, max = 100)

)

# Simulate data in the survival context

survival\_data <- simsurv(

lambdas = 0.1, gammas = 0.5,

x = participant\_data,

betas = [c](https://rdrr.io/r/base/c.html)(is\_female = -0.2, is\_smoker = 1.2,

treatment = -0.4, kps = -0.005,

disease\_stage = 0.2),

maxt = 5)

The generated data from the survival\_data object can then be re-imported into the participant\_data using any data merging tools, including through a fabricate call, and then used for subsequent analyses (e.g. using the **survival** package).

## Time series using ****forecast****

**forecast**, is a package commonly used to analyze time series data which also has functionality capable of generating simulated time series data. **forecast** can use the Arima and simulate functions to create pre-specified ARIMA models, including seasonal time trends.

Below, we provide an example of using **forecast** to generate an ARIMA time series, reshape the data, and import it into **fabricatr** to create new variables of interest.

library(forecast)

arima\_model <- simulate(

Arima(ts(rnorm(100), frequency = 4),

order = c(1, 0, 1))

fabricate(data.frame(arima\_model),

year = rep(1:25, each=4),

quarter = rep(1:4, 25))

Here, ts converts a series of data into a time series, with frequency specifying the number of observations per unit of time (in this case, for example, quarters in a year). Arima ingests this data and fits an ARIMA model with the specified parameters. simulate draws new data from the fit time series, producing a vector of interest. We then import the data into a fabricate call (converting it to a data frame) and add new columns of interest.