Decisions and Computation

Bill Foote

2025-03-10

We have made the decision to quantify decisions. Yes, logic, counting, arithmetic, algebra, especially of the linear variety, even a bit of calculus all matter to our analysis. Truth is coded as a `1`, not-a-truth as a `0`, and so we have binary data. The binomial distribution, with its many decisions in the form of assumptions, provides a short-hand approach to counting the ways sampled successes systematically relate to hypotheses about the proportion of successes compatible with the sampled data. We will need more tools, and a heart to boot, to decide which hypothetical proportion is more plausible than another. But with @Jeffreys1966 Rules (one through eight) and @Lonergan1957 guidance on the six canons of empirical method, we arrive at a quantification of the plausibility of our beliefs about hypothetical proportions when confronted by data. Probability is a continuous number between 0 (non-truth; no success) and 1 (truth; success). We go further, certainly (probability 1) beyond the statistical model, to a judgment that the most likely, most probable, in this system, is the most plausible answer. Better yet we might compute the upper and lower bounds on quantified hypotheses between which there is a plausibility, nay, a probability mass of 89%, to side with @McElreath2020.

Computing is central to our practice of conceiving models, especially, and perhaps only, in the space we call efficient causality, the *how* we get from point any A to a point B, where the points might be hypotheses, data, results. In this endeavor we use simulation and graphics to explore data with incipient, and often very naive models of bounds, fences, accumulations (sums and averages), all models. We move to the more sophistical models of data and hypotheses fit for purpose. Binary data is fit with binomial models; counts of data with Poisson (often craftily mixed with the Gamma distribution to gain further clarity); convergent continuous data (as with averages and sums; exponential and logarithmic accumulations) with the Gaussian distribution. We have already decided what inference looks like: plausibility measured with probability.

In this chapter, we explain how to get started with R and Stan to model the decisions, and underlying hypotheses and data, we will discuss throughout this book. Every quantitative text which promises a hands-on, live data, experience with the computing bridges across sometimes level-4 flooded rivers, lakes, and oceans of concepts, will offer its own brief, hopefully gentle, introduction, here to the general-purpose statistical programming language R.[1]

There are many R tutorials online that can provide additional information; the introduction here is focused on some of the data processing, analysis, and programming tasks that arise in applied regression. We also provide pointers to useful packages for data analysis in R and Stan.

Downloading and installing R and Stan

We do our computing in the open-source package R, a command-based statistical software environment that you can download and operate on your own computer, using RStudio, a free graphical user interface for R. We fit Bayesian regressions in rstanarm, an R package that calls Stan, an open-source program written in C++. The **rethinking@slim** and full-fledged **rethinking** packages can get us started with building probabilistic programming models (Bayesian) built on top of the more general Stan language. We check out <mc-stan.org> for downloads, interfaces with R and Python (Julia too).

[1] We should always have at our disposal more expansive references. Two on-line resources immediately come to mind: https://rc2e.com/ and https://r4ds.had.co.nz/

R and RStudio

To set up **R**, go to <www.r-project.org> and click on the **download R** link. This will take you to a list of download sites called mirrors. We may choose any of these: we click on the link under **Download and Install R** at the top of the page for your operating system (Linux, Mac, or Windows) and download the binaries corresponding to your system. Follow all the instructions. Default settings should be fine. Go to www.rstudio.com, click on **Download RStudio**, click on **Download RStudio Desktop**, and click on the installer for your platform (Windows, Mac, etc.). Then a download will occur. When the download is done, click on the RStudio icon to start an **R** session.

To check that **R** is working, we go to the Console window within RStudio. There should be a "$>$" prompt. Type

```
> 6 * 7
```

and hit Enter. We should get

```
[1] 42
```

The instruction to **type** will forever more mean literally to type a string into the R console, then, most importantly, hit, not to hard for the sake of your keyboard, **Enter**.

We need to extend base R with packages to help us in our question to quantify, analyze, visualize decisions. Here is a list you can type into RStudio **Tools** $>$ **Install Packages**, type in the package names, click **Install**:

- `knitr` and `rmarkdown` to render documents;
- `tidyverse` for data wrangling and visualization,
- `rstan`, `rethinking`, `rstanarm`, `bayesplot`, `tidybayes`, `loo`, `ROSExamples` for estimation and inference
- `goalp` (just one `l`) for linear goal programming, yes, optimization is very important for us!
- `gtree` and `gtreeExamples` for game theory
- `tidydag` for causal inference
- `AER` for data sets to numerous to name, just type `library(AER); data()` to display a list in a separate pane of RStudio; in fact, all of the packages have examples with associated data sets

Basic training shall commence

If an R practitioner is in the house, please ignore this basics section, or, better, help someone which is a novice.

In R, the expressions `==, <, >` are relationship dyadic operators whichreturn a logical value, TRUE or FALSE as appropriate. Other relationships include `<=` (less than or equal), `>=` (greater than or equal), and `!=` (not equal). We can test relationships with the `ifelse` function, whose syntax is the same as in most spreadsheets, for example `=IF()`. The first argument takes a logical statement, the second argument is an expression to be evaluated if the statement is true, and the third argument is evaluated if the statement is false.

Suppose we want to pick a random number between 0 and 100 and then choose the color red if the number is below 30 or blue otherwise:

```
number <- runif(1, 0, 100)
number
```

```
## [1] 75.12196
```

```
color <- ifelse(number<30, "red", "blue")
color
```

```
## [1] "blue"
```

Yes, we just programmed a computer.

Here we go loopty-loop

A key aspect of computer programming is looping—that is, setting up a series of commands to be performed over and over. Start by trying out the simplest possible loop:

```
for (i in 1:10){
  print("hi everyone!") # instead of hello world
}
```

```
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
## [1] "hi everyone!"
```

We can try this too.

```
for (i in 1:10){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Then use the `paste0` function to concatenate the two loops together, something of a higher viewpoint.

```
for (i in 1:10){
  print( paste("ave servi", i) )
}
```

```
## [1] "ave servi 1"
## [1] "ave servi 2"
## [1] "ave servi 3"
## [1] "ave servi 4"
## [1] "ave servi 5"
## [1] "ave servi 6"
## [1] "ave servi 7"
## [1] "ave servi 8"
## [1] "ave servi 9"
## [1] "ave servi 10"
```

The curly braces define what is repeated in the loop. We notice that the indentations, spacing around parentheses, and the line breaks make this code a bit easier to read than this one-liner:

```
for (i in 1:10){print(paste("ave servi",i))}
```

```
## [1] "ave servi 1"
## [1] "ave servi 2"
## [1] "ave servi 3"
## [1] "ave servi 4"
## [1] "ave servi 5"
## [1] "ave servi 6"
## [1] "ave servi 7"
## [1] "ave servi 8"
## [1] "ave servi 9"
## [1] "ave servi 10"
```

We can make this loop conditional in any number of ways. Here is one of those ways.

```
threshold <- 0.5
for (i in 1:10){
  rando <- runif(1)
  ifelse( rando > threshold,
          print( paste0("ave servi", i) ),
          print( paste0("ave servi, morituri te salutamus", i) ))
  threshold <- rando
}
```

```
## [1] "ave servi1"
## [1] "ave servi, morituri te salutamus2"
## [1] "ave servi3"
## [1] "ave servi4"
## [1] "ave servi, morituri te salutamus5"
## [1] "ave servi, morituri te salutamus6"
## [1] "ave servi7"
## [1] "ave servi8"
## [1] "ave servi, morituri te salutamus9"
## [1] "ave servi10"
```

```
for (i in 1:10){
number <- runif(1, 0, 100)
color <- ifelse(number<30, "red", "blue")
print(color)
}
```

```
## [1] "blue"
## [1] "red"
## [1] "blue"
## [1] "blue"
## [1] "blue"
## [1] "blue"
## [1] "blue"
## [1] "blue"
## [1] "blue"
## [1] "blue"
```

Vectors for fun

A vector is, effectively, an ordered list of items. Each item is associated with an integer index. This allows us to locate specific items by referring to the index. The items in a vector can be continuous, called `numerics`, strings, called `characters`, or truth values, called `logicals`. A scalar is, in the R system, a one item vector. Let's make some vectors in R.

```
x <- 1:5
y <- c(3, 4, 1, 1, 1)
z <- c("A", "B", "C")
```

Those familiar with Microsoft's Excel might recognize how to sample a random integer between a minimum, say 5, and maximum, say 10, number as `=RANDBETWEEN(5, 10)` In R we also specify how many random numbers we want to generate between a lower and an upper bound. And the result is a vector, not of integers, but of continuous data.

```
u_between <- runif(5, 5, 10)
u_between
```

```
## [1] 9.512224 8.389762 5.428332 8.932950 5.537626
```

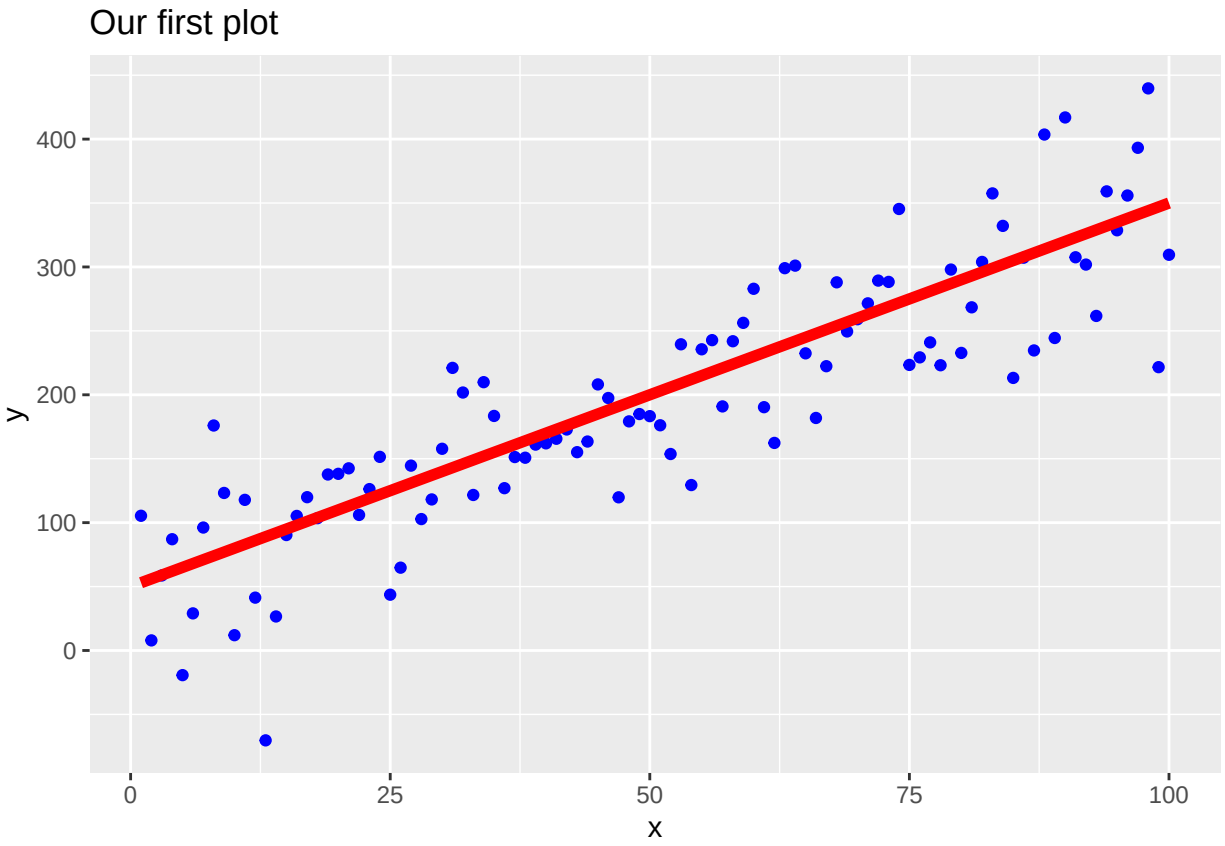
We can operate on vectors with multiplication and division first and addition and subtraction next, in order of computation. Otherwise we use parentheses. Here are some computations deposited into a data frame with the `tibble` function from the **tidyverse** package and the gift of a simple scatterplot using the **ggplot2** package.

```
n <- 100
mu <- 0
sigma <- 50
u <- rnorm(n, mu, sigma)
x <- 1:n
y <- 50 + 3*x + u
df <- tibble(
  y = y,
  x = x,
  u = u
) |>
  mutate(
    pred = 50 + 3*x
  )
p <- df |>
```

```
  ggplot( aes( x=x, y=y )) +
  geom_point( color = "blue") +
  geom_line( aes( x=x, y=pred), color = "red", size = 2.0 ) +
  ggtitle( "Our first plot")
```

```
## Warning: Using ‘size‘ aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use ‘linewidth‘ instead.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```

```
p
```



1. We set up some uncertainty with `u`. Foremost we note that parameters are named, not hard-coded, into the functions. WE also use `#` comments to segment the work flow into three modules.

2. The second module wrangles the data into a table, a frame `df`, called a `tibble`. We pipe the base data into a `mutate` function to create another vector called `pred` (for predict) from the base vectors. We can view the first six rows with `head`. If we want to see a single row, one of a few ways of performing this query is to use the `$` operator.

```
head( df )
```

```
## # A tibble: 6 x 4
```

```
##        y     x       u  pred
##    <dbl> <int>   <dbl> <dbl>
## 1 105.       1  52.4      53
## 2   7.90     2 -48.1      56
## 3  58.7      3  -0.332    59
## 4  87.0      4  25.0      62
## 5 -19.3      5 -84.3      65
## 6  29.0      6 -39.0      68
```

```
head( df$u )
```

```
## [1]  52.3817543 -48.0953439  -0.3323402  25.0376649 -84.2902046 -38.9741632
```

3. The third module plots the data with `ggplot`. We pipe `|>` the data frame `df` into the `ggplot` function to set up a blank canvas of x and y coordinates based on the data in the data frame with the `aes` function. We then layer `+` a mapping of points on to the blank canvas. We want to see `blue` dots. We also draw the `red` line for `pred` with `geom_line`. We view the plot by firing up the graphics object simply by typing in `p`.

We can summarize vectors in various ways, including the sum and the average (called the "mean" in statistics jargon):

```
sum(x)
```

```
## [1] 5050
```

```
mean(x)
```

```
## [1] 50.5
```

```
sd(x)
```

```
## [1] 29.01149
```

We can also compute weighted averages if we know the weights. We illustrate with a vector of three elements, and suppose we have this.

```
x <- c(100, 200, 600)
w1 <- c(1/3, 1/3, 1/3)
w2 <- c(0.5, 0.2, 0.3)
```

In the above code, the vector of weights w1 has the effect of counting each of the three items equally; vector w2 counts the first item more. Here are the weighted averages:

```
sum(w1*x)
```

```
## [1] 300
```

```
sum(w2*x)
```

```
## [1] 270
```

Or suppose we want to weight in proportion to population, `N`.

```
N <- c(310e6, 112e6, 34e6)
sum(N*x)/sum(N)
```

```
## [1] 161.8421
```

Or, equivalently,

```
N <- c(310e6, 112e6, 34e6)
w <- N/sum(N)
sum(w*x)
```

```
## [1] 161.8421
```

The cumsum function does the cumulative sum. Try this:

```
a <- c(1, 1, 1, 1, 1)
cumsum(a)
```

```
## [1] 1 2 3 4 5
```

```
a <- c(2, 4, 6, 8, 10)
cumsum(a)
```

```
## [1]  2  6 12 20 30
```

Indexing vectors

Vectors can be indexed by using brackets, "[ ]". Within the brackets we can put in a vector of elements we are interested in either as a vector of numbers or a logical vector. When using a vector of numbers, the vector can be of arbitrary length, but when indexing using a logical vector, the length of the vector must match the length of the vector you are indexing. Try these:

```
a <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J")
a[1]
```

```
## [1] "A"
```

```
a[2]
```

```
## [1] "B"
```

```
a[4:6]
```

```
## [1] "D" "E" "F"
```

```
a[c(1,3,5)]
```

```
## [1] "A" "C" "E"
```

```
a[c(8,1:3,2)]
```

```
## [1] "H" "A" "B" "C" "B"
```

```
a[c(FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE)]
```

```
## [1] "D" "E" "F"
```

As we have seen in some of the previous examples, we can perform mathematical operations on vectors. These vectors have to be the same length, however. If the vectors are not the same length, we can subset the vectors so they are compatible. Try these:

```
x <- c(1, 1, 1, 2, 2)
y <- c(2, 4, 6)
x[1:3] + y
```

```
## [1] 3 5 7
```

```
x[3:5] * y
```

```
## [1]  2  8 12
```

```
y[3]^x[4]
```

```
## [1] 36
```

```
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
## [1] 3 5 7 4 6
```

The last line runs, but produces a warning. These warnings should not be ignored since it isn't guaranteed that R would carry out the operation as you intended.

We tibble away

We'll be making extensive use of the many packages from the **tidyverse** for data wrangling and plotting.

```
library(tidyverse)
```

The **tidyverse**-style syntax pipes `|>` nouns (data) into verbs (functions). A good practice is to use piping as if they are analytical layers and thus press enter after each `|>`to create a new row of analysis. The beauty of this approach is that we can follow the workflow as each piping literally creates new data from analytical step to analytical step in the workflow.[2]

We will extensively use the `ggplot2` [@R-ggplot2; @wickhamGgplot2ElegantGraphics2016] system in the tidyverse. The `gg` stands for the ***grammar of graphics***. In this framework, similar to the way Adobe Illustrator and other image processors work, we layer graphical elements onto a blank canvas. These elements start with a data frame, here a `tibble`. From the tibble we project axes on a canvas along with groupings of data using factors that are categorical variables in the tibble. We then put geometrical objects on top of this growing canvas. Objects include lines, points, text all with size, shape, and color, among other attributes. We can also make graphics interactive using the `plotly` package with tools such as brushing, zooming, and visually driven queries. The plotly site has much more information and many gallery examples of which we might avail ourselves. As interesting, and where plotly is going is it calls itself *the front-end of ML [machine learning] and data science.*.

Tibbles [@R-tibble] are the backbone of the management of the data we use in all of the functional workings of our model. First of all, a tibble is a data frame and has the two dimensions of any matrix or table, rows and columns. So, whenever we talk about data frames, we're usually talking about tibbles. For more on the topic, check out *R4SD*, Chapter 10.

It is often best to learn of tibbles by doing. Doing what? Why, it is piping data from a raw tibble to aggregations of the data, transformations of the raw data and aggregations. Here is a very simple example in the tidyverse that makes use of the `dplyr` package.

Let's make some toy data to play with. We generate 100 variates normally distributed with mean 10 and standard deviation 5. We then transform this series into another and display the first 10 rows. To reproduce results we set the random seed.

```
library(tidyverse)
set.seed(42)
n_sim <- 100
x <- abs(rnorm(n_sim, 10, 5))
y <- 3 + 0.5*x
xy_tbl <- tibble(
  y = y,
  x = x
  )
xy_tbl
```

```
## # A tibble: 100 x 2
##        y     x
##    <dbl> <dbl>
##  1 11.4  16.9
##  2  6.59  7.18
##  3  8.91 11.8
##  4  9.58 13.2
##  5  9.01 12.0
##  6  7.73  9.47
##  7 11.8  17.6
```

[2]We can read chapter 5 of Grolemund and Wickham's *R for Data Science* available online in Section 5.6.1 https://r4ds.had.co.nz/transform.html#combining-multiple-operations-with-the-pipe.

```
##  8  7.76  9.53
##  9 13.0  20.1
## 10  7.84  9.69
## # i 90 more rows
```

With this tibble we can transform the variables by adding more through a pipe to the `mutate()` verb and assign the results to another tibble object.

```
xy_tbl <- xy_tbl |>
  mutate(
    log_y = log(y),
    log_x = log(x)
    )
head( xy_tbl )
```

```
## # A tibble: 6 x 4
##       y     x log_y log_x
##   <dbl> <dbl> <dbl> <dbl>
## 1 11.4  16.9   2.44  2.82
## 2  6.59  7.18  1.89  1.97
## 3  8.91 11.8   2.19  2.47
## 4  9.58 13.2   2.26  2.58
## 5  9.01 12.0   2.20  2.49
## 6  7.73  9.47  2.05  2.25
```

We notice in this code that we overwrote the first version of the `xy_tbl` tibble.

Next we aggregate this transformed tibble into a custom summary of the data.

```
options(digits = 4, scipen = 99999)
xy_summary <- xy_tbl |>
  gather(key = "variable", value = "value") |>
  group_by(variable) |>
  summarize(
    min = min(value),
    pct_4_5 = quantile(value, 0.045),
    pct_50 = quantile(value, 0.50),
    pct_94_5 = quantile(value, 0.945),
    max = max(value)
  )
xy_summary
```

```
## # A tibble: 4 x 6
##   variable    min pct_4_5 pct_50 pct_94_5   max
##   <chr>     <dbl>   <dbl>  <dbl>    <dbl> <dbl>
## 1 log_x    0.0893   0.883   2.35     2.86  3.06
## 2 log_y    1.27     1.44    2.11     2.46  2.62
## 3 x        1.09     2.43   10.4     17.4  21.4
## 4 y        3.55     4.22    8.22    11.7  13.7
```

Lots of things are happening to us here.

- `options` sets the rest of the calculations to 4 decimal places with sufficient penalties to avoid scientific notation.
- `gather()` puts the data into a long format, also known as a flat table.
- This allows us to `group_by` the subsequent aggregations in `summarize()` by `variable`, the data key.
- The transformed tibble has `summarized()`d each variable's vital statistics in new tibble colums.

The inverse of `gather()` is `arrange()` which awaits us in future work.

This is a great start to exploring our data. My goodness, we could even write a helper function to wrap all of this into an easy to use replacement for the base R `summary()` function. Again this lovely activity awaits us in future work.