Build a model

Last modified on November 11, 2025 12:33:25 Eastern Standard Time

> Crediting the materials
>
> The majority of the material in this document is taken from https://www.tidymodels.org/start/models/

INTRODUCTION

How do you create a statistical model using `tidymodels`? In this article, we will walk you through the steps. We start with data for modeling, learn how to specify and train models with different engines using the parsnip package, and understand why these functions are designed this way.

To use code in this article, you will need to install the following packages: `broom.mixed`, `dotwhisker`, `readr`, `rstanarm`, and `tidymodels`.

```
library(tidymodels)  # for the parsnip package, along with the rest of tidymodels

# Helper packages
library(readr)       # for importing data
library(broom.mixed) # for converting bayesian models to tidy tibbles
library(dotwhisker)  # for visualizing regression results
```

> Note
>
> All of the required packages are already **installed** on the mathr server. However, you will need to **load** the packages using the `library()` function to access the data and functions in each package.

THE SEA URCHINS DATA

Let's use the data from Constable (1993) to explore how three different feeding regimes affect the size of sea urchins over time. The initial size of the sea urchins at the beginning of the experiment probably affects how big they grow as they are fed.

To start, let's read our urchins data into R, which we'll do by providing `readr::read_csv()` with a url where our CSV data is located ("https://tidymodels.org/start/models/urchins.csv"):

```
urchins <-
  # Data were assembled for a tutorial
  # at https://www.flutterbys.com.au/stats/tut/tut7.5a.html
  read_csv("https://tidymodels.org/start/models/urchins.csv")  |>
  # Change the names to be a little more verbose
  setNames(c("food_regime", "initial_volume", "width")) |>
  # Factors are very helpful for modeling, so we convert one column
  mutate(food_regime = factor(food_regime, levels = c("Initial", "Low", "High")))
```

Let's take a quick look at the data:

```
urchins
```

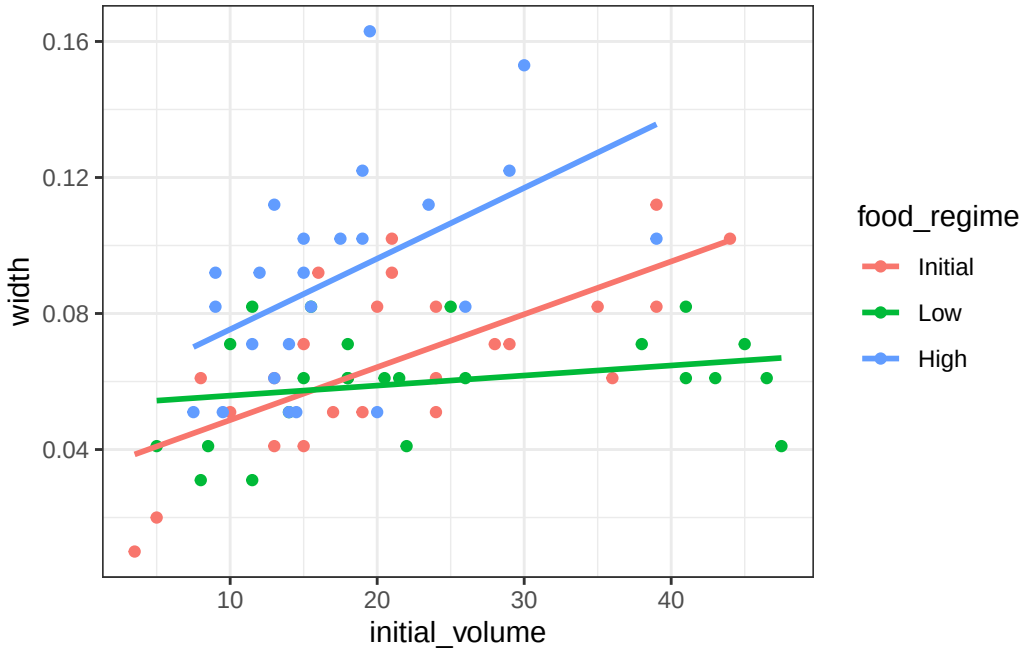
```
# A tibble: 72 x 3
   food_regime initial_volume width
   <fct>                <dbl> <dbl>
 1 Initial                3.5 0.01
 2 Initial                5   0.02
 3 Initial                8   0.061
 4 Initial               10   0.051
 5 Initial               13   0.041
 6 Initial               13   0.061
 7 Initial               15   0.041
 8 Initial               15   0.071
 9 Initial               16   0.092
10 Initial               17   0.051
# i 62 more rows
```

The urchins data is a tibble. If you are new to tibbles, the best place to start is the tibbles chapter in *R for Data Science*. For each of the 72 urchins, we know their:

- experimental feeding regime group (`food_regime`: either `Initial`, `Low`, or `High`),
- size in milliliters at the start of the experiment (`initial_volume`), and
- suture width at the end of the experiment (`width`).

As a first step in modeling, it's always a good idea to plot the data:

```
ggplot(urchins,
       aes(x = initial_volume,
           y = width,
           group = food_regime,
           col = food_regime)) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE) +
  theme_bw()
```



We can see that urchins that were larger in volume at the start of the experiment tended to have wider sutures at the end, but the slopes of the lines look different so this effect may depend on the feeding regime condition.

BUILD AND FIT A MODEL

A standard two-way analysis of variance (ANOVA) model makes sense for this dataset because we have both a continuous predictor and a categorical predictor. Since the slopes appear to be different for at least two of the feeding regimes, let's build a model that allows for two-way interactions. Specifying an R formula with our variables in this way:

```
width ~ initial_volume * food_regime
```

allows our regression model depending on initial volume to have separate slopes and intercepts for each food regime.

For this kind of model, ordinary least squares is a good initial approach. With `tidymodels`, we start by specifying the functional form of the model that we want using the parsnip package.

> parsnip
>
> The goal of parsnip is to provide a tidy, unified interface to models that can be used to try a range of models without getting bogged down in the syntactical minutiae of the underlying packages.

Since there is a numeric outcome and the model should be linear with slopes and intercepts, the model type is "linear regression". We can declare this with:

```
linear_reg()
```

```
Linear Regression Model Specification (regression)

Computational engine: lm
```

That is pretty underwhelming since, on its own, it doesn't really do much. However, now that the type of model has been specified, we can think about a method for fitting or training the model, the model engine. The engine value is often a mash-up of the software that can be used to fit or train the model as well as the estimation method. The default for `linear_reg()` is `"lm"` for ordinary least squares, as you can see above. We could set a non-default option instead:

```
linear_reg() %>%
  set_engine("keras")
```

```
Linear Regression Model Specification (regression)

Computational engine: keras
```

The documentation page for `linear_reg()` lists all the possible engines. We'll save our model object using the default engine as `lm_mod`.

```
lm_mod <- linear_reg()
```

From here, the model can be estimated or trained using the `fit()` function:

```
lm_fit <-
  lm_mod %>%
  fit(width ~ initial_volume * food_regime, data = urchins)
lm_fit
```

```
parsnip model object


Call:
stats::lm(formula = width ~ initial_volume * food_regime, data = data)

Coefficients:
                   (Intercept)                  initial_volume
                     0.0331216                      0.0015546
                food_regimeLow                 food_regimeHigh
                     0.0197824                      0.0214111
 initial_volume:food_regimeLow  initial_volume:food_regimeHigh
                    -0.0012594                      0.0005254
```

> Note
>
> Specifying the model can be done explicitly using the syntax:
>
> ```
> width ~ initial_volume + food_regime +
>   initial_volume:food_regime, data = urchins
> ```

```
lm_fit <-
  lm_mod %>%
  fit(width ~ initial_volume + food_regime +
        initial_volume:food_regime, data = urchins)
lm_fit
```

```
parsnip model object


Call:
```

```
stats::lm(formula = width ~ initial_volume + food_regime + initial_volume:food_regime,
    data = data)

Coefficients:
                  (Intercept)                   initial_volume
                    0.0331216                       0.0015546
                food_regimeLow                 food_regimeHigh
                    0.0197824                       0.0214111
 initial_volume:food_regimeLow  initial_volume:food_regimeHigh
                   -0.0012594                       0.0005254
```

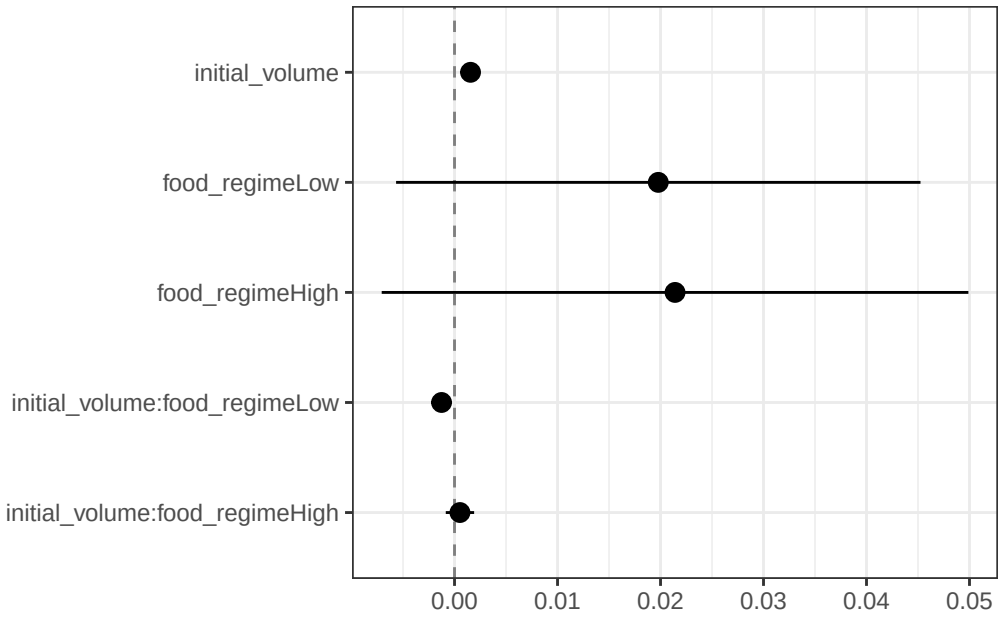
Perhaps our analysis requires a description of the model parameter estimates and their statistical properties. Although the `summary()` function for `lm` objects can provide that, it gives the results back in an unwieldy format. Many models have a `tidy()` method that provides the summary results in a more predictable and useful format (e.g. a data frame with standard column names):

```
lm_fit |>
  tidy() |>
  kable()
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.0331216	0.0096186	3.4434873	0.0010020
initial_volume	0.0015546	0.0003978	3.9077643	0.0002220
food_regimeLow	0.0197824	0.0129883	1.5230864	0.1325145
food_regimeHigh	0.0214111	0.0145318	1.4733993	0.1453970
initial_volume:food_regimeLow	-0.0012594	0.0005102	-2.4685525	0.0161638
initial_volume:food_regimeHigh	0.0005254	0.0007020	0.7484702	0.4568356

This kind of output can be used to generate a dot-and-whisker plot of our regression results using the `dotwhisker` package:

```
lm_fit |>
  tidy() |>
  dwplot(dot_args = list(size = 3, color = "black"),
         whisker_args = list(color = "black"),
         vline = geom_vline(xintercept = 0, colour = "grey50", linetype = 2)) +
  theme_bw()
```



USE A MODEL TO PREDICT

This fitted object `lm_fit` has the `lm` model output built-in, which you can access with `lm_fit$fit`, but there are some benefits to using the fitted parsnip model object when it comes to predicting.

Suppose that, for a publication, it would be particularly interesting to make a plot of the mean body size for urchins that started the experiment with an initial volume of 20ml. To create such a graph, we start with some new example data that we will make predictions for, to show in our graph:

```
new_points <- expand.grid(initial_volume = 20,
                          food_regime = c("Initial", "Low", "High"))
new_points
```

```
  initial_volume food_regime
1             20     Initial
2             20         Low
3             20        High
```

```
new_points |>
  kable()
```

initial_volume	food_regime
20	Initial
20	Low
20	High

To get our predicted results, we can use the `predict()` function to find the mean values at 20ml.

It is also important to communicate the variability, so we also need to find the predicted confidence intervals. If we had used `lm()` to fit the model directly, a few minutes of reading the documentation page for `predict.lm()` would explain how to do this. However, if we decide to use a different model to estimate urchin size (spoiler: we will!), it is likely that a completely different syntax would be required.

Instead, with `tidymodels`, the types of predicted values are standardized so that we can use the same syntax to get these values.

First, let's generate the mean body width values:

```
mean_pred <- predict(lm_fit, new_data = new_points)
mean_pred
```

```
# A tibble: 3 x 1
   .pred
   <dbl>
1 0.0642
2 0.0588
3 0.0961
```

```
mean_pred |>
  kable()
```

.pred
0.0642144
0.0588094
0.0961334

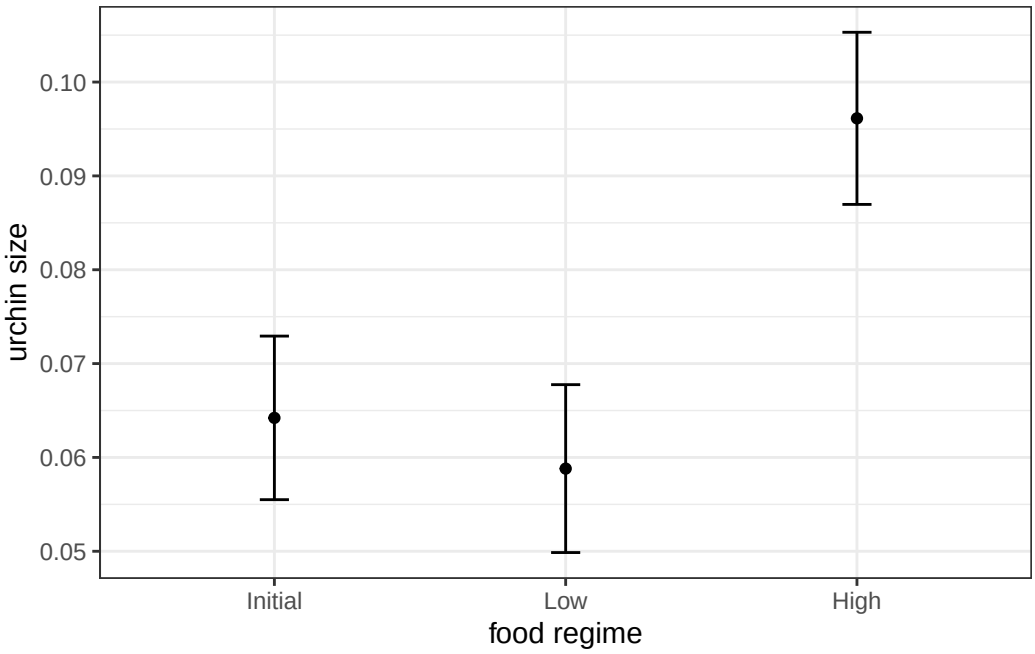
When making predictions, the `tidymodels` convention is to always produce a tibble of results with standardized column names. This makes it easy to combine the original data and the predictions in a usable format:

```r
conf_int_pred <- predict(lm_fit,
                         new_data = new_points,
                         type = "conf_int")
conf_int_pred
```

```
# A tibble: 3 x 2
  .pred_lower .pred_upper
        <dbl>       <dbl>
1      0.0555      0.0729
2      0.0499      0.0678
3      0.0870      0.105
```

```r
# Now combine:
plot_data <-
  new_points %>%
  bind_cols(mean_pred) %>%
  bind_cols(conf_int_pred)

# and plot:
ggplot(plot_data, aes(x = food_regime)) +
  geom_point(aes(y = .pred)) +
  geom_errorbar(aes(ymin = .pred_lower,
                    ymax = .pred_upper),
                width = 0.1) +
  labs(y = "urchin size", x = "food regime") +
  theme_bw()
```



MODEL WITH A DIFFERENT ENGINE

Every one on your team is happy with that plot except that one person who just read their first book on Bayesian analysis. They are interested in knowing if the results would be different if the model were estimated using a Bayesian approach. In such an analysis, a *prior distribution* needs to be declared for each model parameter that represents the possible values of the parameters (before being exposed to the observed data). After some discussion, the group agrees that the priors should be bell-shaped but, since no one has any idea what the range of values should be, to take a conservative approach and make the priors wide using a Cauchy distribution (which is the same as a t-distribution with a single degree of freedom).

The documentation on the `rstanarm` package shows us that the `stan_glm()` function can be used to estimate this model, and that the function arguments that need to be specified are called `prior` and `prior_intercept`. It turns out that `linear_reg()` has a `stan` engine. Since these prior distribution arguments are specific to the Stan software, they are passed as arguments to `parsnip::set_engine()`. After that, the same exact `fit()` call is used:

```
# set the prior distribution
prior_dist <- rstanarm::student_t(df = 1)

set.seed(123)

# make the parsnip model
```

```r
bayes_mod <-
  linear_reg() %>%
  set_engine("stan",
             prior_intercept = prior_dist,
             prior = prior_dist)

# train the model
bayes_fit <-
  bayes_mod %>%
  fit(width ~ initial_volume * food_regime, data = urchins)
bayes_fit
```

```
parsnip model object

stan_glm
 family:       gaussian [identity]
 formula:      width ~ initial_volume * food_regime
 observations: 72
 predictors:   6
------
                              Median MAD_SD
(Intercept)                   0.0    0.0
initial_volume                0.0    0.0
food_regimeLow                0.0    0.0
food_regimeHigh               0.0    0.0
initial_volume:food_regimeLow  0.0    0.0
initial_volume:food_regimeHigh 0.0    0.0

Auxiliary parameter(s):
      Median MAD_SD
sigma 0.0    0.0

------
* For help interpreting the printed output see ?print.stanreg
* For info on the priors used see ?prior_summary.stanreg
```

```r
print(bayes_fit, digits = 5)
```

```
parsnip model object
```

```
stan_glm
 family:       gaussian [identity]
 formula:      width ~ initial_volume * food_regime
 observations: 72
 predictors:   6
------
                              Median   MAD_SD
(Intercept)                    0.03345  0.00959
initial_volume                 0.00154  0.00040
food_regimeLow                 0.01948  0.01303
food_regimeHigh                0.02132  0.01419
initial_volume:food_regimeLow  -0.00125  0.00051
initial_volume:food_regimeHigh  0.00054  0.00068

Auxiliary parameter(s):
      Median  MAD_SD
sigma 0.02124 0.00191

------
* For help interpreting the printed output see ?print.stanreg
* For info on the priors used see ?prior_summary.stanreg
```

This kind of Bayesian analysis (like many models) involves randomly generated numbers in its fitting procedure. We can use `set.seed()` to ensure that the same (pseudo-)random numbers are generated each time we run this code. The number `123` isn't special or related to our data; it is just a "seed" used to choose random numbers.

To update the parameter table, the `tidy()` method is once again used:

```
tidy(bayes_fit, conf.int = TRUE) |>
  kable()
```

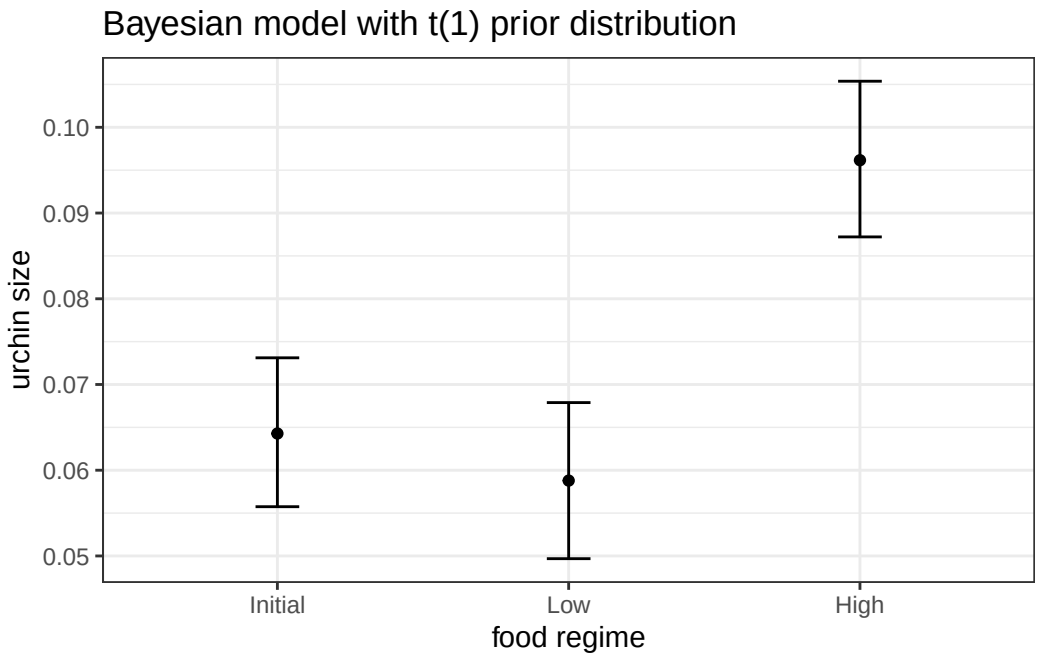
term	estimate	std.error	conf.low	conf.high
(Intercept)	0.0334484	0.0095904	0.0170691	0.0494020
initial_volume	0.0015428	0.0004011	0.0008927	0.0022024
food_regimeLow	0.0194757	0.0130347	-0.0016588	0.0415978
food_regimeHigh	0.0213163	0.0141872	-0.0020526	0.0443661
initial_volume:food_regimeLow	-0.0012499	0.0005138	-0.0020976	-0.0004077
initial_volume:food_regimeHigh	0.0005385	0.0006844	-0.0005875	0.0016587

A goal of the tidymodels packages is that the **interfaces to common tasks are standardized** (as seen in the `tidy()` results above). The same is true for getting predictions; we can

use the same code even though the underlying packages use very different syntax:

```
bayes_plot_data <-
  new_points  |>
  bind_cols(predict(bayes_fit, new_data = new_points))  |>
  bind_cols(predict(bayes_fit, new_data = new_points, type = "conf_int"))

ggplot(bayes_plot_data, aes(x = food_regime)) +
  geom_point(aes(y = .pred)) +
  geom_errorbar(aes(ymin = .pred_lower, ymax = .pred_upper), width = 0.15) +
  labs(y = "urchin size", x= "food regime") +
  ggtitle("Bayesian model with t(1) prior distribution") +
  theme_bw()
```



This isn't very different from the non-Bayesian results (except in interpretation).

> Note
>
> The parsnip package can work with many model types, engines, and arguments. Check out tidymodels.org/find/parsnip to see what is available.

WHY DOES IT WORK THAT WAY?

The extra step of defining the model using a function like `linear_reg()` might seem superfluous

since a call to `lm()` is much more succinct. However, the problem with standard modeling functions is that they don't separate what you want to do from the execution. For example, the process of executing a formula has to happen repeatedly across model calls even when the formula does not change; we can't recycle those computations.

Also, using the `tidymodels` framework, we can do some interesting things by incrementally creating a model (instead of using single function call). Model tuning with `tidymodels` uses the specification of the model to declare what parts of the model should be tuned. That would be very difficult to do if `linear_reg()` immediately fit the model.

If you are familiar with the tidyverse, you may have noticed that our modeling code uses the pipe (`|>`). With dplyr and other tidyverse packages, the pipe works well because all of the functions take the data as the first argument. For example:

```
urchins |>
  group_by(food_regime) |>
  summarize(med_vol = median(initial_volume))
```

```
# A tibble: 3 x 2
  food_regime med_vol
  <fct>         <dbl>
1 Initial        20.5
2 Low            19.2
3 High           15
```

whereas the modeling code uses the pipe to pass around the *model object*:

```
bayes_mod %>%
  fit(width ~ initial_volume * food_regime,
      data = urchins)
```

```
parsnip model object

stan_glm
 family:       gaussian [identity]
 formula:      width ~ initial_volume * food_regime
 observations: 72
 predictors:   6
------
                            Median MAD_SD
(Intercept)                 0.0    0.0
initial_volume              0.0    0.0
```

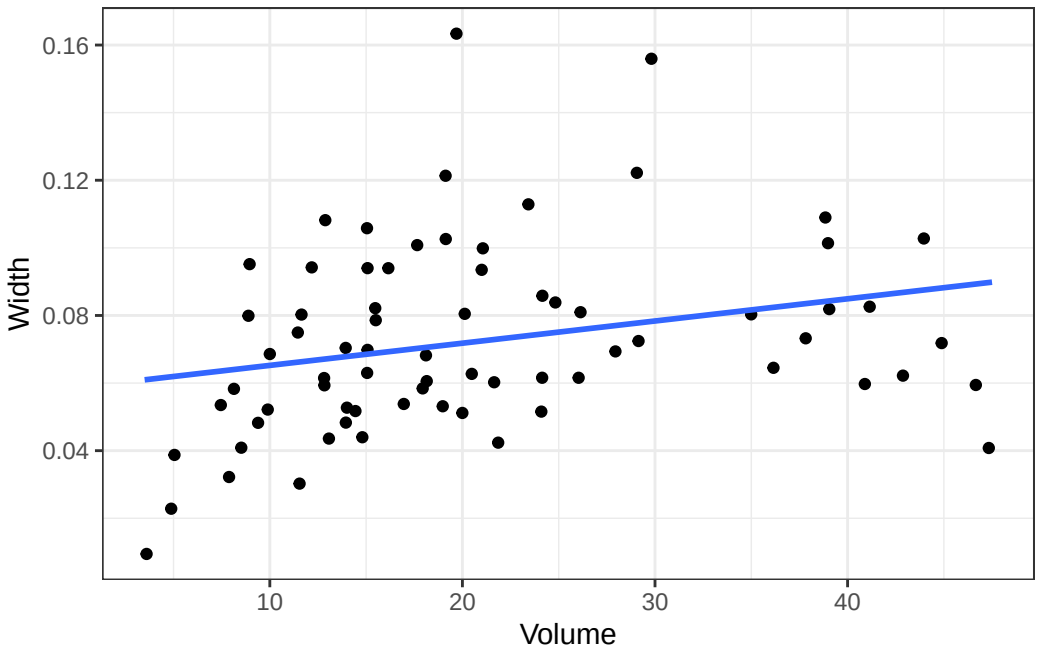
```
food_regimeLow                 0.0    0.0
food_regimeHigh                0.0    0.0
initial_volume:food_regimeLow  0.0    0.0
initial_volume:food_regimeHigh 0.0    0.0

Auxiliary parameter(s):
      Median MAD_SD
sigma 0.0    0.0

------
* For help interpreting the printed output see ?print.stanreg
* For info on the priors used see ?prior_summary.stanreg
```

This may seem jarring if you have used `dplyr` a lot, but it is extremely similar to how `ggplot2` operates:

```
ggplot(urchins,
       aes(initial_volume, width)) +      # returns a ggplot object
  geom_jitter() +                         # same
  geom_smooth(method = lm, se = FALSE) +  # same
  labs(x = "Volume", y = "Width") +       # etc
  theme_bw()
```



SESSION INFORMATION

```
sessionInfo()
```

```
R version 4.5.1 (2025-06-13)
Platform: x86_64-redhat-linux-gnu
Running under: Red Hat Enterprise Linux 9.6 (Plow)

Matrix products: default
BLAS/LAPACK: FlexiBLAS OPENBLAS-OPENMP;  LAPACK version 3.9.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: America/New_York
tzcode source: system (glibc)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] dotwhisker_0.8.4    broom.mixed_0.2.9.6 yardstick_1.3.2
 [4] workflowsets_1.1.1  workflows_1.3.0     tune_2.0.1
 [7] tailor_0.1.0        rsample_1.3.1       recipes_1.3.1
[10] parsnip_1.3.3       modeldata_1.5.1     infer_1.0.9
[13] dials_1.4.2         broom_1.0.10        tidymodels_1.4.1
[16] scales_1.4.0        lubridate_1.9.4     forcats_1.0.1
[19] stringr_1.5.2       dplyr_1.1.4         purrr_1.1.0
[22] readr_2.1.5         tidyr_1.3.1         tibble_3.3.0
[25] ggplot2_4.0.0       tidyverse_2.0.0     knitr_1.50

loaded via a namespace (and not attached):
  [1] RColorBrewer_1.1-3     tensorA_0.36.2.1       rstudioapi_0.17.1
  [4] jsonlite_2.0.0         datawizard_1.3.0       magrittr_2.0.4
  [7] ggstance_0.3.7         nloptr_2.2.1           farver_2.1.2
 [10] rmarkdown_2.30         vctrs_0.6.5            minqa_1.2.8
 [13] base64enc_0.1-3        tinytex_0.57           sparsevctrs_0.3.4
```

```
 [16] htmltools_0.5.8.1      distributional_0.5.0   curl_7.0.0
 [19] StanHeaders_2.32.10    parallelly_1.45.1      htmlwidgets_1.6.4
 [22] plyr_1.8.9             zoo_1.8-14             igraph_2.2.0
 [25] mime_0.13              lifecycle_1.0.4        pkgconfig_2.0.3
 [28] colourpicker_1.3.0     Matrix_1.7-4           R6_2.6.1
 [31] fastmap_1.2.0          rbibutils_2.3          future_1.67.0
 [34] shiny_1.11.1           digest_0.6.37          colorspace_2.1-2
 [37] furrr_0.3.1            patchwork_1.3.2        crosstalk_1.2.2
 [40] labeling_0.4.3         timechange_0.3.0       abind_1.4-8
 [43] mgcv_1.9-3             compiler_4.5.1         bit64_4.6.0-1
 [46] withr_3.0.2            inline_0.3.21          S7_0.2.0
 [49] backports_1.5.0        shinystan_2.6.0        performance_0.15.2
 [52] QuickJSR_1.8.1         pkgbuild_1.4.8         MASS_7.3-65
 [55] lava_1.8.1             loo_2.8.0              gtools_3.9.5
 [58] tools_4.5.1            httpuv_1.6.16          future.apply_1.20.0
 [61] threejs_0.3.4          nnet_7.3-20            glue_1.8.0
 [64] nlme_3.1-168           promises_1.3.3         grid_4.5.1
 [67] checkmate_2.3.3        reshape2_1.4.4         generics_0.1.4
 [70] gtable_0.3.6           tzdb_0.5.0             class_7.3-23
 [73] data.table_1.17.8      hms_1.1.4              utf8_1.2.6
 [76] markdown_2.0           pillar_1.11.1          vroom_1.6.6
 [79] posterior_1.6.1        later_1.4.4            splines_4.5.1
 [82] lhs_1.2.0              lattice_0.22-7         survival_3.8-3
 [85] bit_4.6.0              tidyselect_1.2.1       miniUI_0.1.2
 [88] reformulas_0.4.1       gridExtra_2.3          stats4_4.5.1
 [91] xfun_0.53              rstanarm_2.32.2        hardhat_1.4.2
 [94] matrixStats_1.5.0      timeDate_4051.111      rstan_2.32.7
 [97] DT_0.34.0              stringi_1.8.7          boot_1.3-32
[100] DiceDesign_1.10        yaml_2.3.10            evaluate_1.0.5
[103] codetools_0.2-20       cli_3.6.5              RcppParallel_5.1.11-1
[106] rpart_4.1.24           Rdpack_2.6.4           shinythemes_1.2.0
[109] xtable_1.8-4           parameters_0.28.2      dichromat_2.0-0.1
[112] Rcpp_1.1.0             globals_0.18.0         parallel_4.5.1
[115] rstantools_2.5.0       gower_1.0.2            bayestestR_0.17.0
[118] dygraphs_1.1.1.6       bayesplot_1.14.0       marginaleffects_0.30.0
[121] lme4_1.1-37            GPfit_1.0-9            listenv_0.9.1
[124] ipred_0.9-15           xts_0.14.1             prodlim_2025.04.28
[127] insight_1.4.2          crayon_1.5.3           rlang_1.1.6
[130] shinyjs_2.1.0
```