CSSS508, Week 10

Model Results and Reproducibility

Breon Haskett

June 2, 2022

Updated: Jun 2, 2022

Topics for Today

Working with Model Results

- Tidy model output with `broom`
- Visualizing models with `ggeffects`
- Tables with `gt`, `modelsummary`, and `gtsummary`

Reproducible Research

Best Practices

- Organization
- Portability
- Version Control

Wrapping up the course

Working with Model Results

broom

`broom` is a package that "tidies up" the output from models such a `lm()` and `glm()`.

It has a small number of key functions:

- `tidy()` - Creates a dataframe summary of a model.
- `augment()` - Adds columns—such as fitted values—to the data used in the model.
- `glance()` - Provides one row of fit statistics for models.

```
library(broom)
```

Model Output is a List

`lm()` and `summary()` produce lists as output, which cannot go directly into tidyverse functions, particularly those in `ggplot2`.

```
lm_1 <- lm(yn ~ num1 + fac1, data = ex_dat)
summary(lm_1)
```

```
##
## Call:
## lm(formula = yn ~ num1 + fac1, data = ex_dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.7126 -1.8282 -0.0605  1.7395  7.3122
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.14854    0.36194   3.173  0.00175 **
## num1         0.60485    0.09846   6.143 4.43e-09 ***
## fac1B        1.41071    0.49035   2.877  0.00446 **
## fac1C        2.62104    0.50599   5.180 5.49e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.896 on 196 degrees of freedom
## Multiple R-squared:  0.2342,    Adjusted R-squared:  0.2225
## F-statistic: 19.98 on 3 and 196 DF,  p-value: 2.43e-11
....
```

Model Output Varies!

Each type of model also produces somewhat different output, so you can't just reuse the same code to handle output from every model.

```
glm_1 <- glm(yb ~ num1 + fac1, data = ex_dat, family=binomial(link="logit"))
summary(glm_1)
```

```
##
## Call:
## glm(formula = yb ~ num1 + fac1, family = binomial(link = "logit"),
##     data = ex_dat)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.9890  -0.9179   0.4394   0.9831   2.1520
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.27278    0.29970  -4.247 2.17e-05 ***
## num1         0.40477    0.08367   4.838 1.31e-06 ***
## fac1B        0.79416    0.37198   2.135   0.0328 *
## fac1C        1.69175    0.40926   4.134 3.57e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
....
```

broom::tidy()

`tidy()` produces similar output, but as a dataframe.

```
lm_1 %>% tidy()
```

```
## # A tibble: 4 x 5
##   term        estimate std.error statistic       p.value
##   <chr>          <dbl>     <dbl>     <dbl>         <dbl>
## 1 (Intercept)    1.15     0.362       3.17 0.00175
## 2 num1           0.605    0.0985      6.14 0.00000000443
## 3 fac1B          1.41     0.490       2.88 0.00446
## 4 fac1C          2.62     0.506       5.18 0.000000549
```

Each type of model (e.g. `glm`, `lmer`) has a different *method* with its own additional arguments. See `?tidy.lm` for an example.

broom::tidy()

This output is also completely identical between different models.

This can be very useful and important if running models with different test statistics... or just running a lot of models!

```
glm_1 %>% tidy()
```

```
## # A tibble: 4 x 5
##   term        estimate std.error statistic    p.value
##   <chr>          <dbl>     <dbl>     <dbl>      <dbl>
## 1 (Intercept)   -1.27     0.300      -4.25 0.0000217
## 2 num1           0.405    0.0837      4.84 0.00000131
## 3 fac1B          0.794    0.372       2.13 0.0328
## 4 fac1C          1.69     0.409       4.13 0.0000357
```

broom::glance()

`glance()` produces dataframes of fit statistics for models.

If you run many models, you can compare each model row-by-row in each column... or even plot their different fit statistics to allow holistic comparison.

```
glance(lm_1)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl>
## 1     0.234         0.222  2.90      20.0 2.43e-11     3  -494.  999.
## # ... with 4 more variables: BIC <dbl>, deviance <dbl>,
## #   df.residual <int>, nobs <int>
```

broom::augment()

`augment()` takes values generated by a model and adds them back to the original data. This includes fitted values, residuals, and leverage statistics.

```
augment(lm_1) %>% head()
```

```
## # A tibble: 6 x 9
##       yn    num1 fac1  .fitted .resid   .hat .sigma .cooksd .std.resid
##    <dbl>   <dbl> <fct>   <dbl>  <dbl>  <dbl>  <dbl>   <dbl>      <dbl>
## 1  5.85   0.0272 C       3.79   2.06  0.0177   2.90 2.32e-3      0.717
## 2  5.13  -2.32   A      -0.254  5.38  0.0297   2.88 2.72e-2      1.89
## 3  7.50   4.88   C       6.72   0.782 0.0357   2.90 7.01e-4      0.275
## 4 -0.332  1.81   B       3.66  -3.99  0.0154   2.89 7.53e-3     -1.39
## 5 -0.356  0.647  A       1.54  -1.90  0.0139   2.90 1.53e-3     -0.659
## 6  2.71  -1.21   B       1.83   0.884 0.0259   2.90 6.35e-4      0.309
```

See `?augment.lm` for examples of what `augment()` can do.

The Power of `broom`

The real advantage of `broom` becomes apparent when running many models at once. Here we run separate models for each level of `fac1`:

```
ex_dat %>%
  nest_by(fac1) %>%
  mutate(model = list(lm(yn ~  num1 + fac2, data = data))) %>%
  summarize(tidy(model), .groups = "drop")
```

```
## # A tibble: 9 x 6
##   fac1  term        estimate std.error statistic     p.value
##   <fct> <chr>          <dbl>     <dbl>     <dbl>       <dbl>
## 1 A     (Intercept)    0.679     0.510     1.33  0.187
## 2 A     num1           0.619     0.156     3.96  0.000171
## 3 A     fac2No         0.921     0.649     1.42  0.160
## 4 B     (Intercept)    1.90      0.582     3.27  0.00178
## 5 B     num1           0.510     0.172     2.96  0.00439
## 6 B     fac2No         1.64      0.706     2.32  0.0238
## 7 C     (Intercept)    3.43      0.570     6.03  0.000000138
## 8 C     num1           0.667     0.182     3.66  0.000551
## 9 C     fac2No         0.558     0.773     0.721 0.474
```

`nest_by()` nests data into a list column by levels of `fac1`.

Plotting Model Results

`geom_smooth()`

I have used `geom_smooth()` in many past examples.

`geom_smooth()` generates "smoothed conditional means" including loess curves and generalized additive models (GAMs).

Note, however, that most regression models are conditional mean models, such as ordinary least squares and generalized linear models.

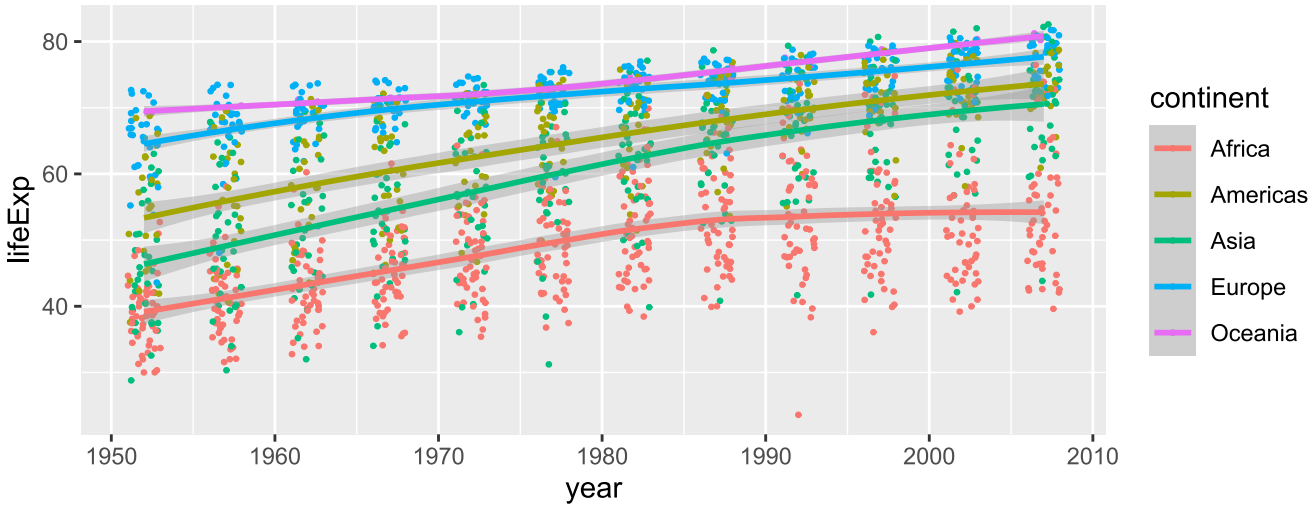
We can use `geom_smooth()` to add a layer depicting common bivariate models.

We'll look at this with the `gapminder` data from Week 2.

```
library(gapminder)
```

Default `geom_smooth()`

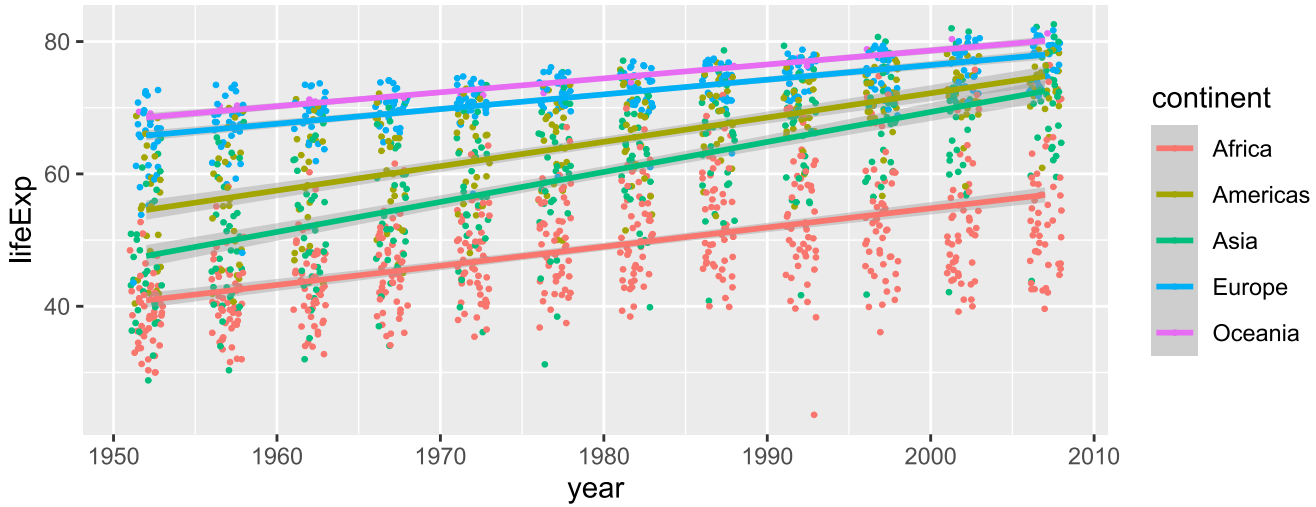
```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp, color = continent)) +
  geom_point(position = position_jitter(1,0), size = 0.5) +
  geom_smooth()
```



By default, `geom_smooth()` chooses either a loess smoother (N < 1000) or a GAM depending on the number of observations.

Linear `glm`

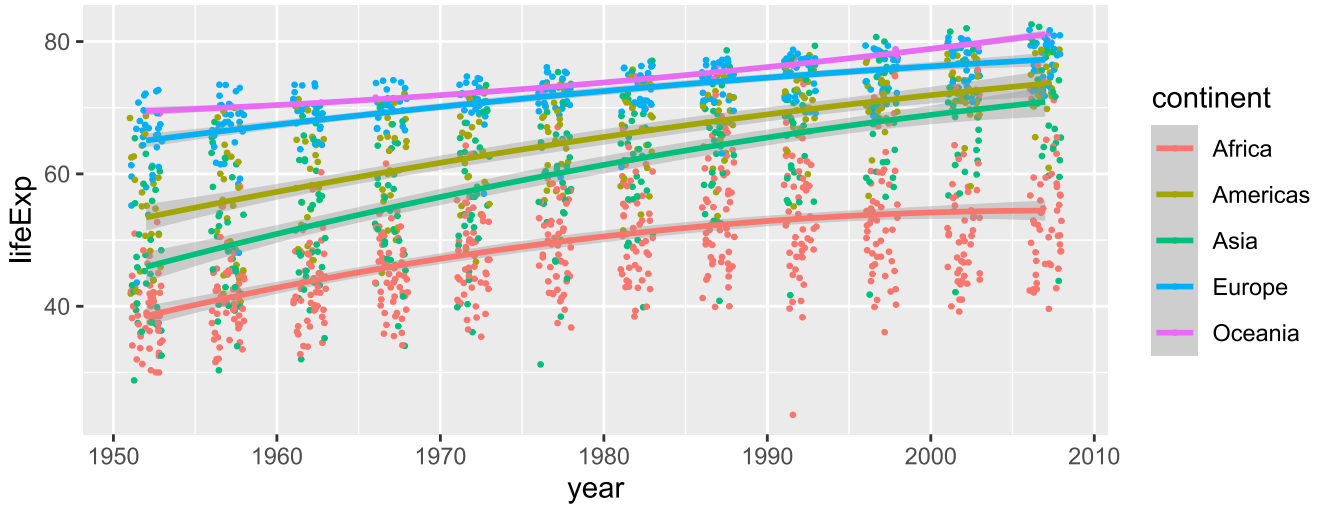
```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp, color = continent)) +
  geom_point(position = position_jitter(1,0), size = 0.5) +
  geom_smooth(method = "glm", formula = y ~ x)
```



We could also fit a standard linear model using either `method = "glm"` or `method = "lm"` and a formula like `y ~ x`.

Polynomial `glm`

```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp, color = continent)) +
  geom_point(position = position_jitter(1,0), size = 0.5) +
  geom_smooth(method = "glm", formula = y ~ poly(x, 2))
```



`poly(x, 2)` produces a quadratic model which contains a linear term (`x`) and a quadratic term (`x^2`).

More Complex Models

What if we want something more complex than a bivariate model?

What if we have a statistically complex model, like nonlinear probability model or multilevel model?

We need to go beyond `geom_smooth()`!

But first, vocab!

We are often interested in what might happen if some variables take particular values, often ones not seen in the actual data.

When we set variables to certain values, we refer to them as **counterfactual values** or just **counterfactuals**.

For example, if we know nothing about a new observation, our prediction for that estimate is often based on assuming every variable is at its mean.

Sometimes, however, we might have very specific questions which require setting (possibly many) combinations of variables to particular values and making an estimate or prediction.

Providing specific estimates, conditional on values of covariates, is a nice way to summarize results, particularly for models with unintuitive parameters (e.g. logit models).

ggeffects

ggeffects

If we want to look at more complex models, we can use `ggeffects` to create and plot tidy *marginal effects*.

That is, tidy dataframes of *ranges* of predicted values that can be fed straight into `ggplot2` for plotting model results.

We will focus on two `ggeffects` functions:

- `ggpredict()` - Computes predicted values for the outcome variable at margins of specific variables.
- `plot.ggeffects()` - A plot method for `ggeffects` objects (like `ggredict()` output)

```
library(ggeffects)
```

Quick Simulated Data

To best show off `ggeffects`, I need a data frame with numeric and categorical variables with strong relationships. It is easiest to just simulate it:

```
ex_dat <- data.frame(num1 = rnorm(200, 1, 2),
                     fac1 = sample(c(1, 2, 3), 200, TRUE),
                     num2 = rnorm(200, 0, 3),
                     fac2 = sample(c(1, 2))) %>%
  mutate(yn = num1 * 0.5 + fac1 * 1.1 + num2 * 0.7 +
              fac2 - 1.5  + rnorm(200, 0, 2)) %>%
  mutate(yb = as.numeric(yn > mean(yn))) %>%
  mutate(fac1 = factor(fac1, labels = c("A", "B", "C")),
         fac2 = factor(fac2, labels = c("Yes", "No")))
```

Now we can get `ggpredict`ing!

`ggpredict()`

When you run `ggpredict()`, it produces a dataframe with a row for every unique value of a supplied predictor ("independent") variable (`term`).

Each row contains an expected (estimated) value for the outcome ("dependent") variable, plus confidence intervals.

```
lm_1 <- lm(yn ~ num1 + fac1, data = ex_dat)
lm_1_est <- ggpredict(lm_1, terms = "num1")
```

If desired, the argument `interval="prediction"` will give predicted intervals instead.

`ggpredict()` output

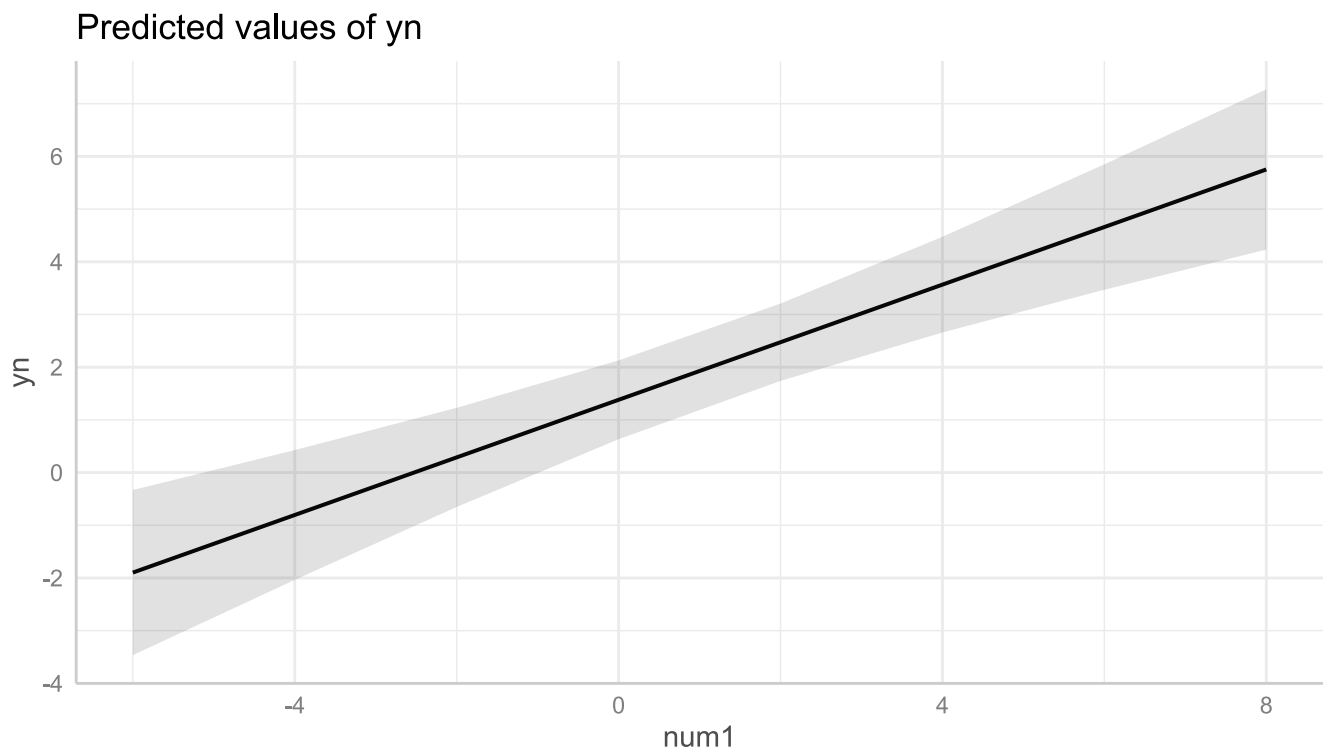
```
lm_1_est
```

```
## # Predicted values of yn
##
## num1 | Predicted |         95% CI
## --------------------------------
##   -6 |     -1.90 | [-3.46, -0.33]
##   -4 |     -0.80 | [-2.04,  0.43]
##   -2 |      0.29 | [-0.65,  1.23]
##    0 |      1.38 | [ 0.63,  2.13]
##    2 |      2.47 | [ 1.74,  3.21]
##    4 |      3.57 | [ 2.66,  4.48]
##    6 |      4.66 | [ 3.47,  5.85]
##    8 |      5.75 | [ 4.23,  7.27]
##
## Adjusted for:
## * fac1 = A
```

`plot()` for `ggpredict()`

`ggeffects` features a `plot()` *method*, `plot.ggeffects()`, which produces a ggplot when you give `plot()` output from `ggpredict()`.

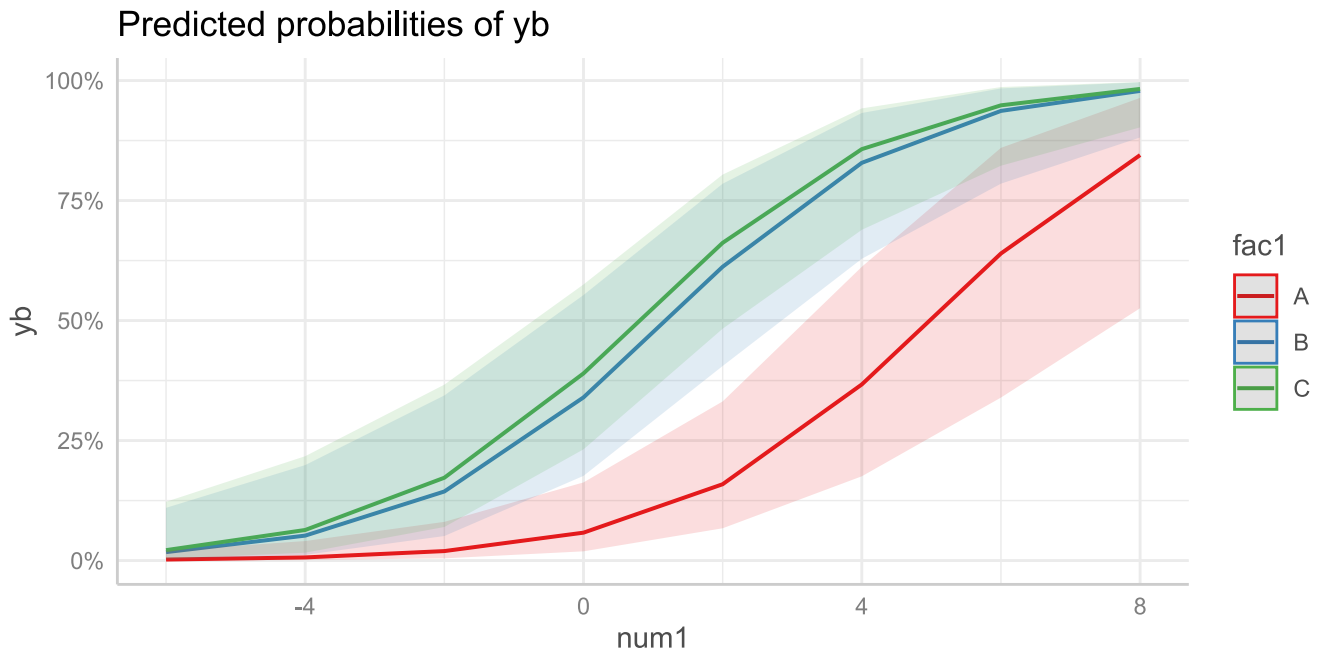
```
plot(lm_1_est)
```



Grouping with `ggpredict()`

When using a vector of `terms`, `ggeffects` will plot the first along the x-axis and use others for *grouping*. Note we can pipe a model into `ggpredict()`!

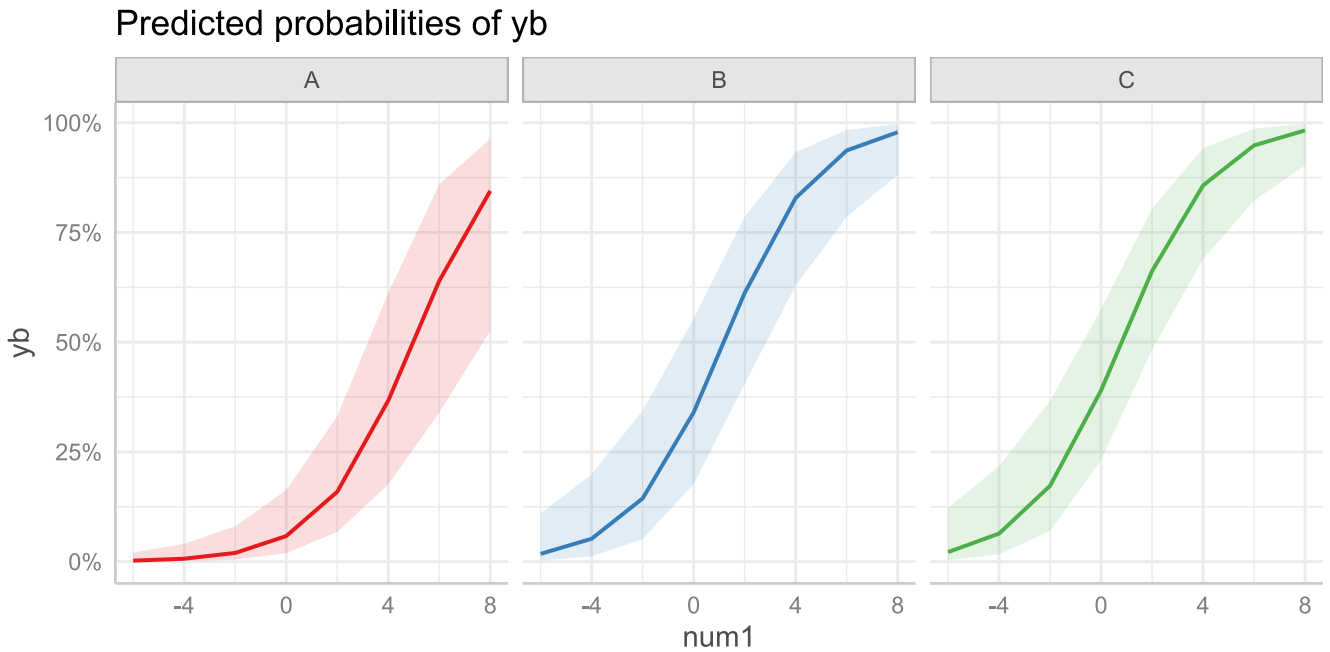
```
glm(yb ~ num1 + fac1 + num2 + fac2, data = ex_dat, family=binomial(link = "logit")) %>%
  ggpredict(terms = c("num1", "fac1")) %>% plot()
```



Faceting with `ggpredict()`

You can add `facet=TRUE` to the `plot()` call to facet over *grouping terms*.

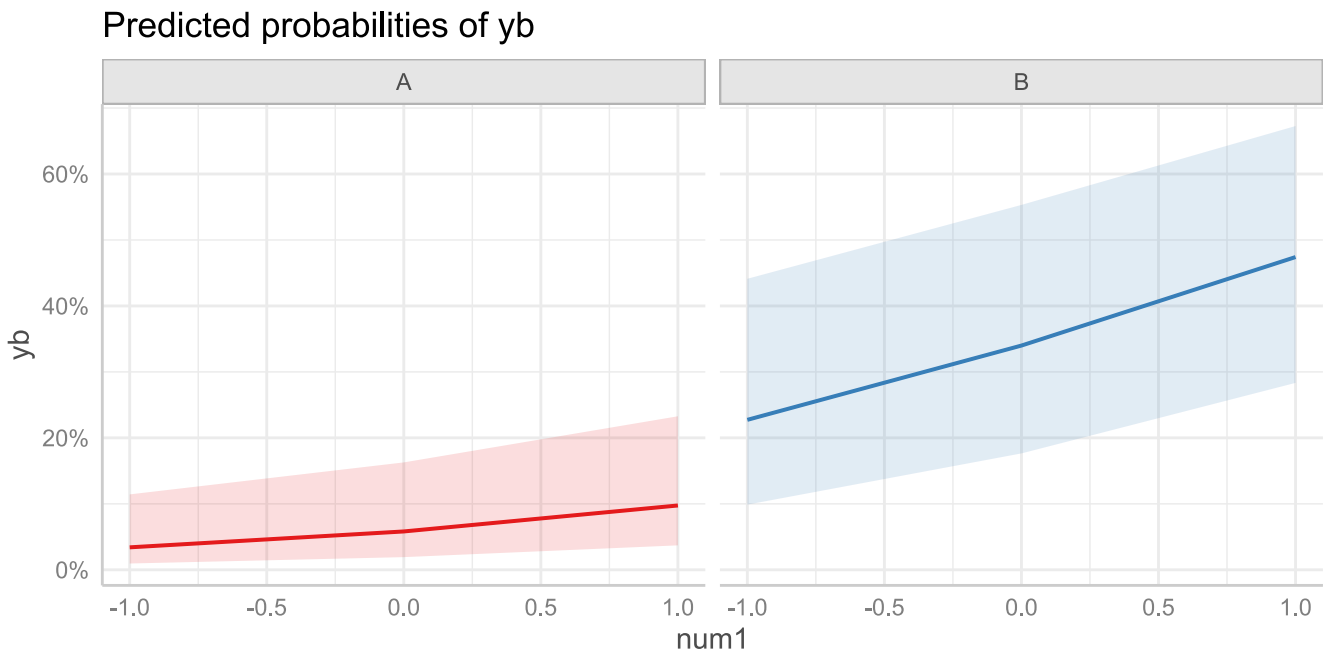
```
glm(yb ~ num1 + fac1 + num2 + fac2, data = ex_dat, family = binomial(link = "logit")) %>%
  ggpredict(terms = c("num1", "fac1")) %>% plot(facet=TRUE)
```



Counterfactual Values

You can add values in square brackets in the `terms=` argument to specify counterfactual values.

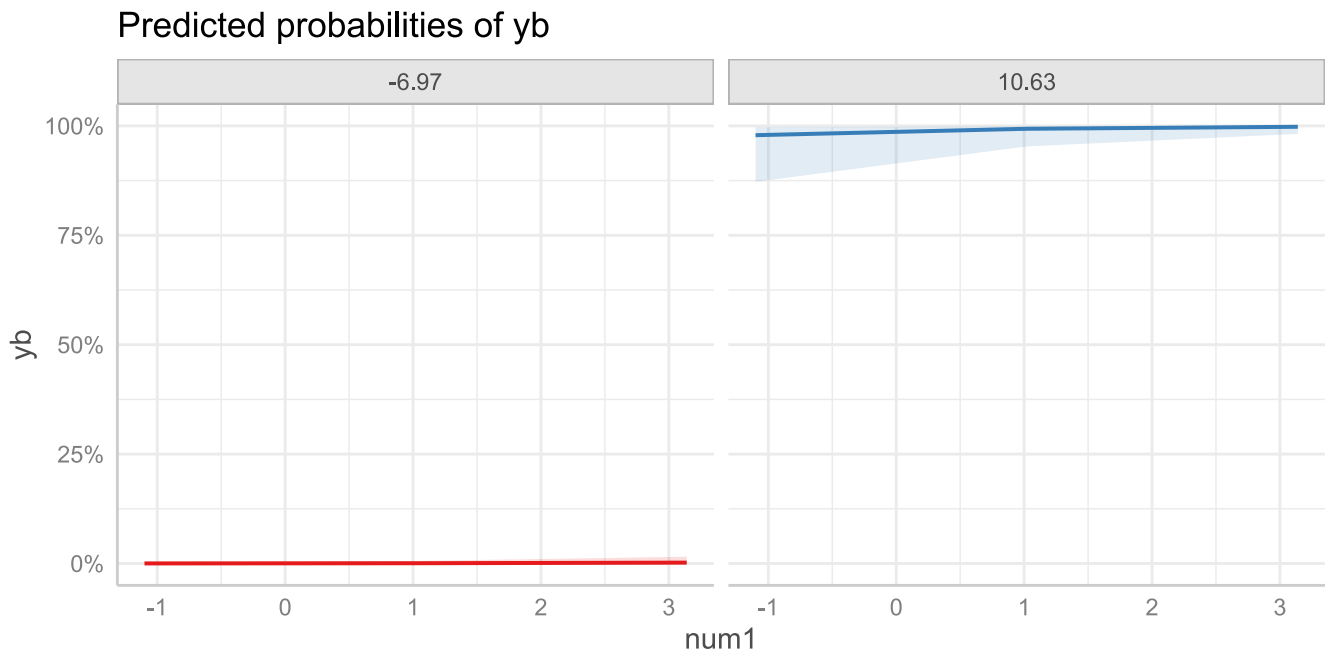
```
glm(yb ~ num1 + fac1 + num2 + fac2, data=ex_dat, family=binomial(link="logit")) %>%
  ggpredict(terms = c("num1 [-1,0,1]", "fac1 [A,B]")) %>% plot(facet=TRUE)
```



Representative Values

You can also use `[meansd]` or `[minmax]` to set representative values.

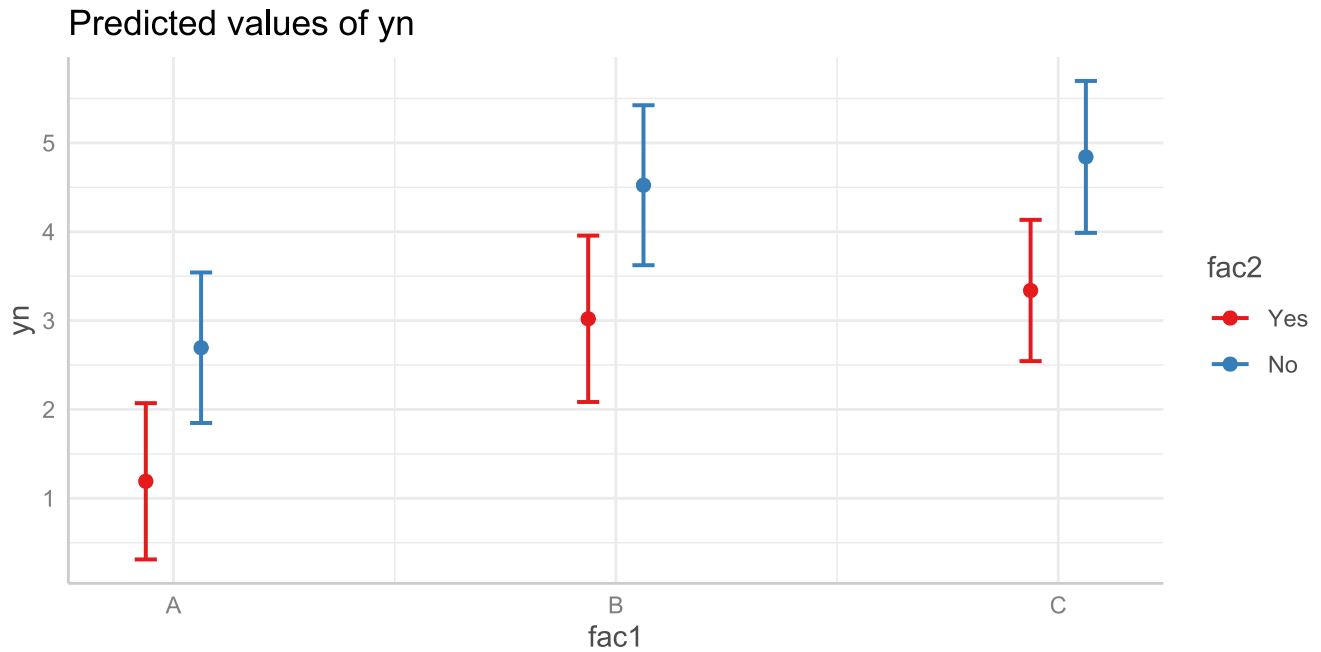
```
glm(yb ~ num1 + fac1 + num2 + fac2, data = ex_dat, family = binomial(link = "logit")) %>%
  ggpredict(terms = c("num1 [meansd]", "num2 [minmax]")) %>% plot(facet=TRUE)
```



Dot plots with `ggpredict()`

`ggpredict` will produce dot plots with error bars for categorical predictors.

```
lm(yn ~ fac1 + fac2, data = ex_dat) %>%
  ggpredict(terms=c("fac1", "fac2")) %>% plot()
```



Notes on `ggeffects`

There is a lot more to the `ggeffects` package that you can see in the package vignette and the github repository. This includes, but is not limited to:

- Predicted values for polynomial and interaction terms
- Getting predictions from models from dozens of other packages
- Sending `ggeffects` objects to `ggplot2` to freely modify plots

Making Tables

`pander` Regression Tables

We've used `pander` to create nice tables for dataframes. But `pander` has *methods* to handle all sort of objects that you might want displayed nicely.

This includes model output, such as from `lm()`, `glm()`, and `summary()`.

```
library(pander)
```

`pander()` and `lm()`

You can send an `lm()` object straight to `pander`:

```
pander(lm_1)
```

	Estimate	Std. Error	t value	Pr(>t)
(Intercept)	37.23	1.599	23.28	2.565e-20
wt	-3.878	0.6327	-6.129	1.12e-06
hp	-0.03177	0.00903	-3.519	0.001451

Table: Fitting linear model: mpg ~ wt + hp

`pander()` and `summary()`

You can do this with `summary()` as well, for added information:

```
pander(summary(lm_1))
```

	Estimate	Std. Error	t value	Pr(>t)
(Intercept)	37.23	1.599	23.28	2.565e-20
wt	-3.878	0.6327	-6.129	1.12e-06
hp	-0.03177	0.00903	-3.519	0.001451

Observations	Residual Std. Error	R^2	Adjusted R^2
32	2.593	0.8268	0.8148

Table: Fitting linear model: mpg ~ wt + hp

Advanced Tables

`pander` tables are great for basic `rmarkdown` documents, but they're not generally publication ready.

We're going to talk about a few different approaches for making nicer tables:

- `gt` from RStudio for general table construction
- `modelsummary` for creating model tables
- `gtsummary` for creating data summaries

gt

If you need to more customizability or different output types, RStudio's `gt` package is a recent and powerful system for creating tables from dataframes. We'll use `dplyr`'s built-in `starwars` data for some examples.

```
library(gt)
tes_chars <- starwars %>%
  unnest(films) %>%
  unnest(starships, keep_empty=TRUE) %>%
  filter(films == "The Empire Strikes Back") %>%
  select(name, species, starships, mass, height) %>%
  distinct(name, .keep_all = TRUE) %>%
  mutate(starships = ifelse(name == "Obi-Wan Kenobi" | is.na(starships),
                            "No Ship", starships))
glimpse(tes_chars)
```

```
## Rows: 16
## Columns: 5
## $ name      <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader",~
## $ species   <chr> "Human", "Droid", "Droid", "Human", "Human", "Huma~
## $ starships <chr> "X-wing", "No Ship", "No Ship", "TIE Advanced x1",~
## $ mass      <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 77.0, 112.0, 80.0, ~
## $ height    <int> 172, 167, 96, 202, 150, 182, 228, 180, 170, 66, 17~
```

Big Improvement!

name	species	starships	mass	height
Luke Skywalker	Human	X-wing	77.0	172
C-3PO	Droid	No Ship	75.0	167
R2-D2	Droid	No Ship	32.0	96
Darth Vader	Human	TIE Advanced x1	136.0	202
Leia Organa	Human	No Ship	49.0	150
Obi-Wan Kenobi	Human	No Ship	77.0	182
Chewbacca	Wookiee	Millennium Falcon	112.0	228
Han Solo	Human	Millennium Falcon	80.0	180
Wedge Antilles	Human	X-wing	77.0	170
Yoda	Yoda's species	No Ship	17.0	66
Palpatine	Human	No Ship	75.0	170
Boba Fett	Human	Slave 1	78.2	183
IG-88	Droid	No Ship	140.0	200
Bossk	Trandoshan	No Ship	113.0	190
Lando Calrissian	Human	Millennium Falcon	79.0	177
Lobot	Human	No Ship	79.0	175

Star Wars Characters

The Empire Strikes Back

		Vitals	
	Species	Mass (kg)	Height (cm)
X-wing			
Luke Skywalker	Human	77	172
Wedge Antilles	Human	77	170
Millennium Falcon			
Chewbacca	Wookiee	112	228
Han Solo	Human	80	180
Lando Calrissian	Human	79	177
No Ship			
C-3PO	Droid	75	167
R2-D2	Droid	32	96
Leia Organa	Human	49	150
Obi-Wan Kenobi	Human	77	182
Yoda	Yoda's species	17	66
Palpatine	Human	75	170
IG-88	Droid	140	200
Bossk	Trandoshan	113	190
Lobot	Human	79	175
TIE Advanced x1			
Darth Vader	Human	136	202
Slave 1			
Boba Fett	Human	78	183

LaTeX Tables

`gt` is a fairly new package and is somewhat finicky when used in `.pdf` documents.

For tables in LaTeX—as is needed for `.pdf` files—I recommend also looking into the `kableExtra`, `huxtable`, or `stargazer` packages.

Like `gt`, `kableExtra` and `huxtable` allow the construction of complex tables in either HTML or LaTeX using additive syntax similar to `ggplot2` and `dplyr`.

`stargazer` produces nicely formatted LaTeX tables but is idiosyncratic and doesn't support some models.

If you want to edit LaTeX documents, you could use R using Sweave documents (.Rnw). Alternatively, you may want to work in a dedicated LaTeX editor. I recommend Overleaf for this purpose.

RMarkdown has support for a fair amount of basic LaTeX syntax if you aren't trying to get too fancy!

modelsummary

The `modelsummary` package combines `broom`, `gt`, and `kableExtra` to produce tabular summaries of almost any model fit in R.

An advantage of this package is that it can produce output in every common format: HTML, Markdown, $\LaTeX$, raw text, and even images (`.png` or `.jpg`).

```
library(modelsummary)
```

Its key function is `msummary()` or `modelsummary()` which creates summary tables of models.

You can then build on it using either `gt` or `kableExtra` functions, depending on the selected output format.

msummary

Like `pander()`, `msummary()` takes a model as an object to make a table.

```
mod_1 <- lm(mpg ~ wt, data = mtcars)
modelsummary(mod_1)
```

Note default `modelsummary` look like `pander` tables because they use Markdown.

	Model 1
(Intercept)	37.285
	(1.878)
wt	-5.344
	(0.559)
Num.Obs.	32
R2	0.753
R2 Adj.	0.745
AIC	166.0
BIC	170.4
Log.Lik.	-80.015
F	91.375
RMSE	3.05

modelsummary

You can present multiple models in `modelsummary` using named lists:

```
mod_1 <- lm(mpg ~ wt, data = mtcars)
mod_2 <- lm(mpg ~ hp + wt, data = mtcars)
mod_3 <- lm(mpg ~ hp + wt + factor(am),
            data = mtcars)
model_list <- list("Model 1" = mod_1,
                   "Model 2" = mod_2,
                   "Model 3" = mod_3)
modelsummary(model_list)
```

This allows you to produce the common (and often bad) journal format where one starts with a nonsensical "naive model" then works up to the "full model" justified by the front end of the paper.

	Model 1	Model 2	Model 3
(Intercept)	37.285	37.227	34.003
	(1.878)	(1.599)	(2.643)
wt	-5.344	-3.878	-2.879
	(0.559)	(0.633)	(0.905)
hp		-0.032	-0.037
		(0.009)	(0.010)
factor(am)1			2.084
			(1.376)
Num.Obs.	32	32	32
R2	0.753	0.827	0.840
R2 Adj.	0.745	0.815	0.823
AIC	166.0	156.7	156.1
BIC	170.4	162.5	163.5
Log.Lik.	-80.015	-74.326	-73.067
F	91.375	69.211	48.960
RMSE	3.05	2.59	2.54

PDF Output

`output = "latex"` produces `kableExtra` based output well-suited to PDFs.[1]

```
modelsummary(model_list, output = "latex")
```

	Model 1	Model 2	Model 3
(Intercept)	37.285	37.227	34.003
	(1.878)	(1.599)	(2.643)
wt	-5.344	-3.878	-2.879
	(0.559)	(0.633)	(0.905)
hp		-0.032	-0.037
		(0.009)	(0.010)
factor(am)1			2.084
			(1.376)
Num.Obs.	32	32	32
R2	0.753	0.827	0.840
Adj.R2	0.745	0.815	0.823
AIC	166.0	156.7	156.1
BIC	170.4	162.5	163.5
Log.Lik.	-80.015	-74.326	-73.067

For customization, I recommend referring to `modelsummary`'s documentation.

Saving a `modelsummary`

```
modelsummary(model_list, output = "ex_table.png")
```

	Model 1	Model 2	Model 3
(Intercept)	37.285	37.227	34.003
	(1.878)	(1.599)	(2.643)
wt	-5.344	-3.878	-2.879
	(0.559)	(0.633)	(0.905)
hp		-0.032	-0.037
		(0.009)	(0.010)
factor(am)1			2.084
			(1.376)
Num.Obs.	32	32	32
R2	0.753	0.827	0.840
R2 Adj.	0.745	0.815	0.823
AIC	166.0	156.7	156.1
BIC	170.4	162.5	163.5
Log.Lik.	-80.015	-74.326	-73.067
F	91.375	69.211	48.960
RMSE	3.05	2.59	2.54

`modelsummary` and `gt`

You can select `gt` output to enable modifying summaries with `gt` functions.

```
msummary(model_list, output = "gt") %>%
  tab_header(
    title = "Table 1. Linear Models",
    subtitle = "DV: Miles per Gallon"
  )
```

Note that `gt`'s `(LaTeX)` support for PDFs is immature--this format is better for HTML or image output.

Table 1. Linear Models

DV: Miles per Gallon

	Model 1	Model 2	Model 3
(Intercept)	37.285	37.227	34.003
	(1.878)	(1.599)	(2.643)
wt	-5.344	-3.878	-2.879
	(0.559)	(0.633)	(0.905)
hp		-0.032	-0.037
		(0.009)	(0.010)
factor(am)1			2.084
			(1.376)
Num.Obs.	32	32	32
R2	0.753	0.827	0.840
R2 Adj.	0.745	0.815	0.823
AIC	166.0	156.7	156.1
BIC	170.4	162.5	163.5
Log.Lik.	-80.015	-74.326	-73.067
F	91.375	69.211	48.960
RMSE	3.05	2.59	2.54

gtsummary

The `gtsummary` package is similar to `modelsummary` in that it takes advantage of `broom`, `gt`, and `kableExtra` to provide a flexible table-making framework.

While `gtsummary` can also produce model tables like `modelsummary`, it also produces descriptive statistic tables for dataframes.[1]

```
library(gtsummary)
```

[1] I prefer `modelsummary`'s syntax most model tables.

tbl_summary()

By default, `gtsummary` tables provide:

- Frequencies for categorical and binary variables
- Quantiles of the form "50% (25%, 75%)" for continuous variables
- Sample size

```
mtcars %>%
  select(1:9) %>%
  tbl_summary()
```

Characteristic	N = 32[1]
mpg	19.2 (15.4, 22.8)
cyl	
4	11 (34%)
6	7 (22%)
8	14 (44%)
disp	196 (121, 326)
hp	123 (96, 180)
drat	3.70 (3.08, 3.92)
wt	3.33 (2.58, 3.61)
qsec	17.71 (16.89, 18.90)
vs	14 (44%)
am	13 (41%)

[1] Median (IQR); n (%)

Grouping

You can provide a `by =` argument to do grouped descriptives.

```
mtcars %>%
  select(1:9) %>%
  tbl_summary(by = "am")
```

Characteristic	**0**, N = 19[1]	**1**, N = 13[1]
mpg	17.3 (14.9, 19.2)	22.8 (21.0, 30.4)
cyl		
4	3 (16%)	8 (62%)
6	4 (21%)	3 (23%)
8	12 (63%)	2 (15%)
disp	276 (196, 360)	120 (79, 160)
hp	175 (116, 192)	109 (66, 113)
drat	3.15 (3.07, 3.70)	4.08 (3.85, 4.22)
wt	3.52 (3.44, 3.84)	2.32 (1.94, 2.78)
qsec	17.82 (17.18, 19.17)	17.02 (16.46, 18.61)
vs	7 (37%)	7 (54%)

[1] Median (IQR); n (%)

Adding `gt`

If you select `gt` output, you can dress it up with `gt` functions.

```
mtcars %>%
  select(1:9) %>%
  tbl_summary(by = "am") %>%
  as_gt() %>%
  tab_spanner(
    label = "Transmission",
    columns = starts_with("stat_")
  ) %>%
  tab_header(
    title = "Motor Trend Cars",
    subtitle = "Descriptive Statistics"
  )
```

`starts_with("stat_")` here selects the statistic columns created by `tbl_summary()`.

Motor Trend Cars

Descriptive Statistics

	Transmission	
Characteristic	**0**, N = 19[1]	**1**, N = 13[1]
mpg	17.3 (14.9, 19.2)	22.8 (21.0, 30.4)
cyl		
4	3 (16%)	8 (62%)
6	4 (21%)	3 (23%)
8	12 (63%)	2 (15%)
disp	276 (196, 360)	120 (79, 160)
hp	175 (116, 192)	109 (66, 113)
drat	3.15 (3.07, 3.70)	4.08 (3.85, 4.22)
wt	3.52 (3.44, 3.84)	2.32 (1.94, 2.78)
qsec	17.82 (17.18, 19.17)	17.02 (16.46, 18.61)
vs	7 (37%)	7 (54%)

[1] Median (IQR); n (%)

Bonus: `corrplot`

The `corrplot` package has functions for displaying correlograms.

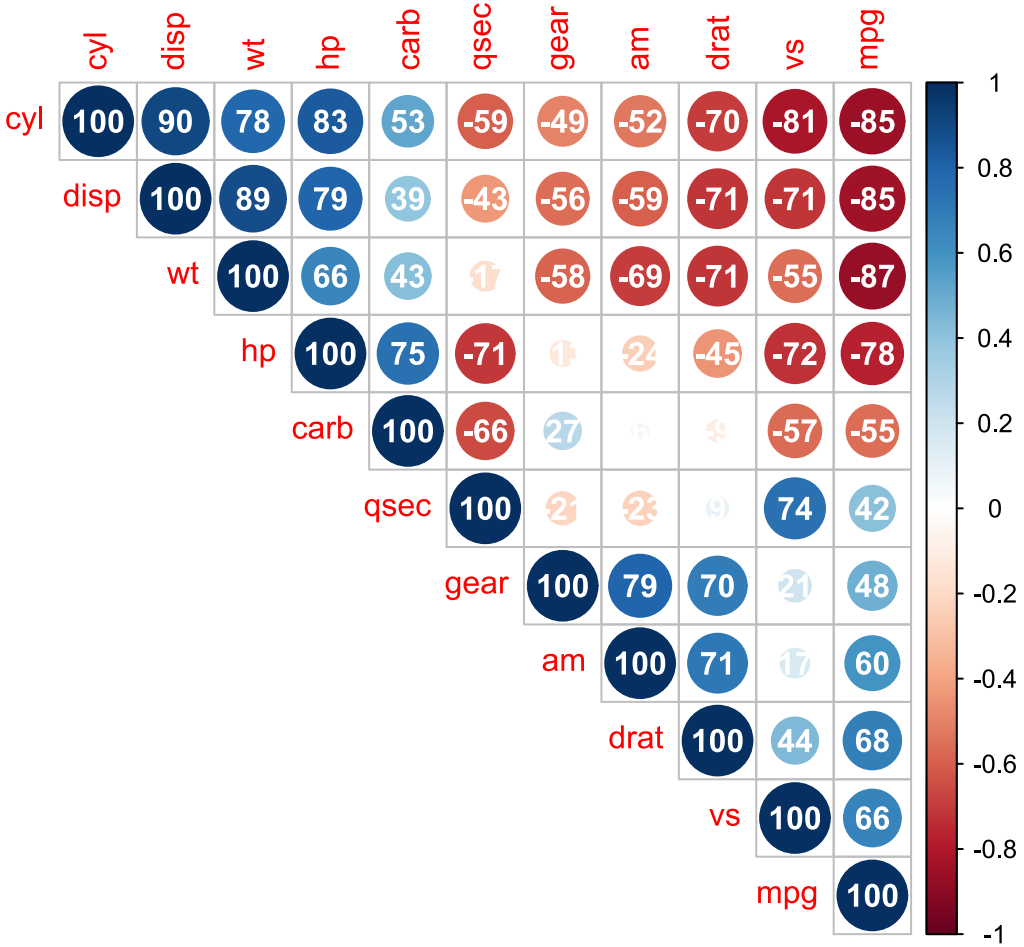
These make interpreting the correlations between variables in a data set easier than conventional correlation tables.

The first argument is a call to `cor()`, the base R function for generating a correlation matrix.

See the vignette for customization options.

```
library(corrplot)
corrplot(
  cor(mtcars),
  addCoef.col = "white",
  addCoefasPercent=T,
  type="upper",
  order="AOE")
```

Correlogram



Reproducible Research

Why Reproducibility?

Reproducibility is not *replication*.

- **Replication** is running a new study to show if and how results of a prior study hold.
- **Reproducibility** is about rerunning *the same study* and getting the *same results*.

Reproducible studies can still be *wrong*... and in fact reproducibility makes proving a study wrong *much easier*.

Reproducibility means:

- Transparent research practices.
- Minimal barriers to verifying your results.

Any study that isn't reproducible can be trusted only on faith.

Reproducibility Definitions

Reproducibility comes in three forms (Stodden 2014):

1. **Empirical:** Repeatability in data collection.
2. **Statistical:** Verification with alternate methods of inference.
3. **Computational:** Reproducibility in cleaning, organizing, and presenting data and results.

R is particularly well suited to enabling **computational reproducibility**.[1]

They will not fix flawed research design, nor offer a remedy for improper application of statistical methods.

Those are the difficult, non-automatable things you want skills in.

[1] Python is equally well suited.

Computational Reproducibility

Elements of computational reproducibility:

- Shared data
  - Researchers need your original data to verify and replicate your work.
- Shared code
  - Your code must be shared to make decisions transparent.
- Documentation
  - The operation of code should be either self-documenting or have written descriptions to make its use clear.
- **Version Control**
  - Documents the research process.
  - Prevents losing work and facilitates sharing.

Levels of Reproducibility

For academic papers, degrees of reproducibility vary:

1. "Read the article"
2. Shared data with documentation
3. Shared data and all code
4. **Interactive document**
5. **Research compendium**
6. Docker compendium: Self-contained ecosystem

Interactive Documents

Interactive documents—like R Markdown docs—combine code and text together into a self-contained document.

- Load and process data
- Run models
- Generate tables and plots in-line with text
- In-text values automatically filled in

Interactive documents allow a reader to examine your computational methods within the document itself; in effect, they are self-documenting.

By re-running the code, they reproduce your results on demand.

Common Platforms:

- **R:** R Markdown (an example from Chuck)
- **Python:** Jupyter Notebooks

Research Compendia

A **research compendium** is a portable, reproducible distribution of an article or other project.

Research compendia feature:

- An interactive document as the foundation
- Files organized in a recognizable structure (e.g. an R package)
- Clear separation of data, method, and output. *Data are read only*.
- A well-documented or even *preserved* computational environment (e.g. Docker)

`rrtools` by UW's Ben Markwick provides a simplified workflow to accomplish this in R.

Another example by Chuck.

Bookdown

`bookdown`—which is integrated into `rrtools`—can generate documents in the proper format for articles, theses, books, or dissertations.

`bookdown` provides an accessible alternative to writing $\LaTeX$ for typesetting and reference management.

You can integrate citations and automate reference page generation using bibtex files (such as produced by Zotero).

`bookdown` supports `.html` output for ease and speed and also renders `.pdf` files through $\LaTeX$ for publication-ready documents.

For University of Washington theses and dissertations, consider Ben Marwick's `huskydown` package which uses Markdown but renders via a UW approved $\LaTeX$ template.

Best Practices

Organization and Portability

Organization Systems

Organizing research projects is something you either do accidentally—and badly—or purposefully with some upfront labor.

Uniform organization makes switching between or revisiting projects easier.

I suggest something like the following:

```
project/
   readme.md
   data/
     derived/
       processed_data.RData
     raw/
       core_data.csv
   docs/
     paper.Rmd
   syntax/
     functions.R
     models.R
```

1. There is a clear hierarchy
   - Written content is in `docs`
   - Code is in `syntax`
   - Data is in `data`
2. Naming is uniform
   - All lower case
   - Words separated by underscores
3. Names are self-descriptive

Workflow versus Project

To summarize Jenny Bryan, one should separate workflow from projects.

Workflow

- The software you use to write your code (e.g. RStudio)
- The location you store a project
- The specific computer you use
- The code you ran earlier or typed into your console

Project

- The raw data
- The code that operates on your raw data
- The packages you use
- The output files or documents

Projects *should not modify anything outside of the project* nor need to be modified by someone else (or future you) to run.

Projects *should be independent of your workflow*.

Portability

For research to be reproducible, it must also be *portable*. Portable software operates *independently of workflow* such as fixed file locations.

Do Not:

- Use `setwd()` in scripts or .Rmd files.
- Use *absolute paths* except for *fixed, immovable sources* (secure data).
  - `read_csv("C:/my_project/data/my_data.csv")`
- Use `install.packages()` in script or .Rmd files.
- Use `rm(list=ls())` anywhere but your console.

Do:

- Use RStudio projects (or the `here` package) to set directories.
- Use *relative paths* to load and save files:
  - `read_csv("./data/my_data.csv")`
- Load all required packages using `library()`.
- Clear your workspace when closing RStudio.
  - Set *Tools* > *Global Options...* > *Save workspace...* to **Never**

Divide and Conquer

Often you do not want to include all code for a project in one `.Rmd` file:

- The code takes too long to knit.
- The file is so long it is difficult to read.

There are two ways to deal with this:

1. Use separate `.R` scripts or `.Rmd` files which save results from complicated parts of a project, then load these results in the main `.Rmd` file.
    - This is good for loading and cleaning large data.
    - Also for running slow models.
2. Use `source()` to run external `.R` scripts when the `.Rmd` knits.
    - This can be used to run large files that aren't impractically slow.
    - Also good for loading project-specific functions.

The Way of Many Files

I find it beneficial to break projects into *many* files:

- Scripts with specialized functions.
- Scripts to load and clean each set of variables.
- Scripts to run each set of models and make tables and plots.
- A main .Rmd that runs some or all of these to reproduce the entire project.

Splitting up a project carries benefits:

- Once a portion of the project is done and in its own file, *it is out of your way.*
- If you need to make changes, you don't need to search through huge files.
- Entire sections of the project can be added or removed quickly (e.g. converted to an appendix of an article)
- **It is the only way to build a proper *pipeline* for a project.**

Pipelines

Professional researchers and teams design projects as a **pipeline**.

A **pipeline** is a series of consecutive processing elements (scripts and functions in R).

Each stage of a pipeline...

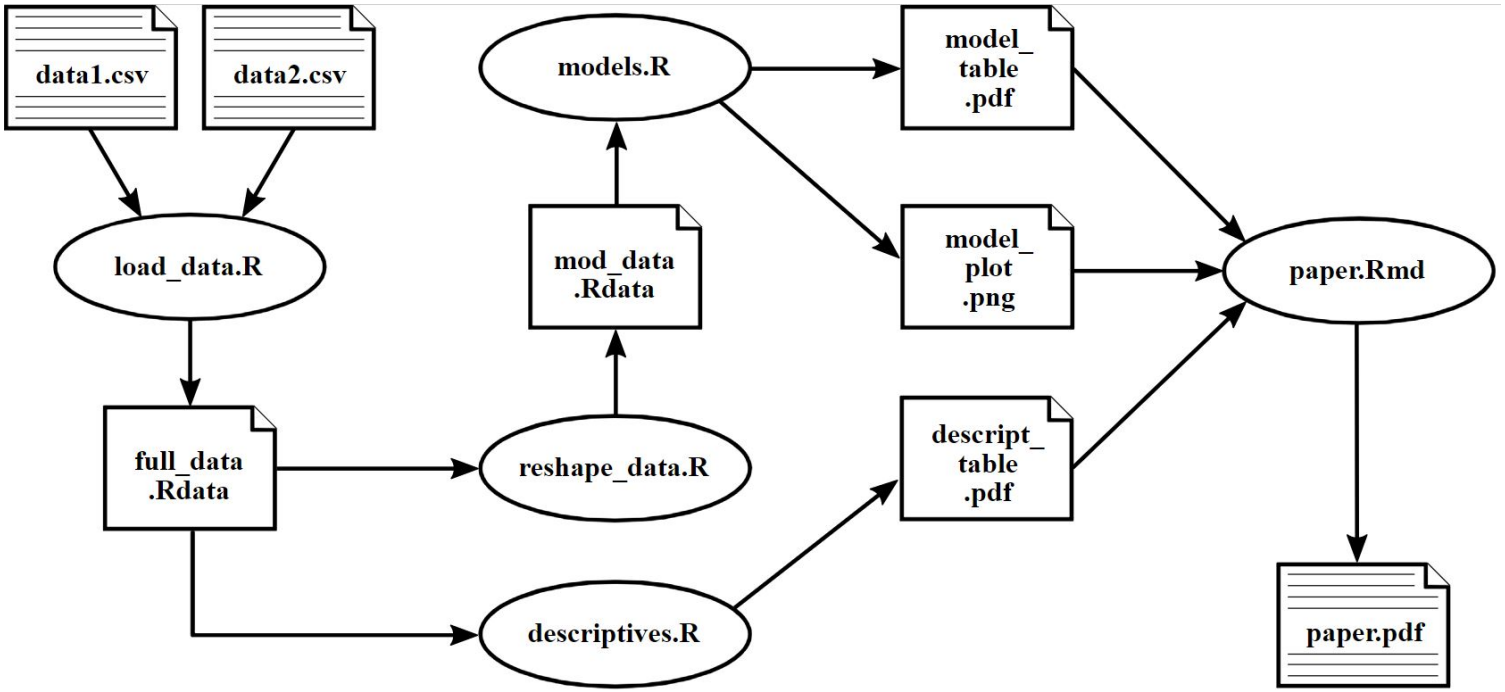
1. Has clearly defined inputs and outputs
2. Does not modify its inputs.
3. Produces the exact same output every time it is re-run.

This means...

1. When you modify one stage, you only need to rerun *subsequent stages*.
2. Different people can work on each stage.
3. Problems are isolated within stages.
4. You can depict your project as a *directed graph* of **dependencies**.

Example Pipeline

Every stage (oval) has an unambiguous input and output. Everything that precedes a given stage is a **dependency**—something required to run it.



Note: `targets` is a terrific package for managing R research pipelines.

Tools

Some opinionated advice

On Formats

Avoid "closed" or commercial software and file formats except where absolutely necessary.

Use open source software and file formats.

- It is always better for *science*:
    - People should be able to explore your research without buying commercial software.
    - You do not want your research to be inaccessible when software is updated.
- It is often just *better*.
    - It is usually updated more quickly
    - It tends to be more secure
    - It is rarely abandoned

The ideal: Use software that reads and writes *raw text*.

Text

Writing and formatting documents are two completely separate jobs.

- Write first
- Format later
- Markdown was made for this

Word processors—like Microsoft Word—try to do both at the same time, usually badly.

They waste time by leading you to format instead of writing.

Find a good modular text editor and learn to use it:

- Atom
- Sublime (Commercial)
- Emacs
- Vim

Version Control

Version Control

Version control originates in collaborative software development.

The Idea: All changes ever made to a piece of software are documented, saved automatically, and revertible.

Version control allows all decisions ever made in a research project to be documented automatically.

Version control can:

1. Protect your work from destructive changes
2. Simplify collaboration by merging changes
3. Document design decisions
4. Make your research process transparent

Git and GitHub

`git` is the dominant platform for version control, and GitHub is a free (and now Microsoft owned) platform for hosting **repositories**.

Repositories are folders on your computer where all changes are tracked by Git.

Once satisfied with changes, you "commit" them then "push" them to a remote repository that stores your project.

Others can copy your project ("pull"), and if you permit, make suggestions for changes.

Constantly committing and pulling changes automatically generates a running "history" that documents the evolution of a project.

`git` is integrated into RStudio under the *Tools* menu. It requires some setup.[1]

[1] You can also use the GitHub desktop application.

GitHub as a CV

Beyond archiving projects and allowing sharing, GitHub also serves as a sort of curriculum vitae for the programmer.

By allowing others to view your projects, you can display competence in programming and research.

If you are planning on working in the private sector, an active GitHub profile will give you a leg up on the competition.

If you are aiming for academia, a GitHub account signals technical competence and an interest in research transparency.

Wrapping up the Course

What You've Learned

A lot!

- How to get data into R from a variety of formats
- How to do "data custodian" work to manipulate and clean data
- How to make pretty visualizations
- How to automate with loops and functions
- How to combine text, calculations, plots, and tables into dynamic R Markdown reports
- How to acquire and work with spatial data

What Comes Next?

- Statistical inference (e.g. more CSSS courses)
  - Functions for hypothesis testing, hierarchical/mixed effect models, machine learning, survey design, etc. are straightforward to use... once data are clean
  - Access output by working with list structures (like from regression models) or using `broom` and `ggeffects`
- Practice, practice, practice!
  - Replicate analyses you've done in Excel, SPSS, or Stata
  - Think about data using `dplyr` verbs, tidy data principles
  - R Markdown for reproducibility
- More advanced projects
  - Using version control (git) in RStudio
  - Interactive Shiny web apps
  - Write your own functions and put them in a package

Neat Things I Didn't Talk About

- The native pipe operator in R 4.0 `|>`.
- The Source Editor
- Keyboard Shortcuts
- The Big Book of R

Course Plugs

If you...

- have no stats background yet - **SOC504: Applied Social Statistics**
- have (only) finished SOC506 - **CSSS510: Maximum Likelihood**
- want to master visualization - **CSSS569: Visualizing Data**
- study events or durations - **CSSS544: Event History Analysis** [1]
- want to use network data - **CSSS567: Social Network Analysis**
- want to work with spatial data - **CSSS554: Spatial Statistics**
- want to work with time series - **CSSS512: Time Series and Panel Data**

[1] Also a great maximum likelihood introduction.

Thank you!