Linear Regression with R

Social Media and Web Analytics @ TiSEM

Lachlan Deer

Last updated: 26 April, 2021

Motivation

This note is about the workhose tool of quantitative marketing and data science: regression analysis. The goal is to give you a whirlwind tour of the key functions and packages. I'm going to assume that you already know all of the necessary theoretical background on regression.[1] This will *not* cover any of theoretical concepts or seek to justify a particular statistical model. Most of the models in this document are pretty silly, but they illustrate the main point – how to run a regression in R.

With these disclaimers noted, let's get started..

Software Requirements

R packages

"Base" R already provides all of the tools we need for basic regression analysis. However, we'll be using several additional packages today, because they will make our lives easier and offer increased power for some more sophisticated analyses.

- New (regression): **fixest**, **estimatr**, **broom**, **modelsummary**
- Already used: **tidyverse**

If you haven't installed these packages, now would be a good time to do so:

```
to_install <- c("fixest", "estimatr", "broom")

install.packages(to_install)

# get the latest version of modelsummary - support fixest
library(remotes)
remotes::install_github('vincentarelbundock/modelsummary')
```

Once installed, we can load them:

```
library("tidyverse")
library("fixest")
library("estimatr")
library("broom")
library("modelsummary")
```

I'll try to be as explicit about where a particular function is coming from, whenever I use it below.

[1] See the course website for links to further material if you want to review the key concepts.

Example Data

We'll mostly be working with the `starwars` data frame, which comes with the `dplyr` package. Let's assign it the name `df`:

```
df <- starwars
```

We'll be interested in estimating regressions of the mass of starwars characters with their height.

Regression Basics

The `lm()` function

R's main command for running regression models is the built-in `lm()` function. "**lm**" stands for "**l**inear **m**odels" and the syntax is rather intuitive:

```
lm(y ~ x1 + x2 + x3 + ..., data = my_data)
```

You'll note that the `lm()` call includes a reference to the data source (in this case, a hypothetical data frame called `my_data`). This means we need to be specific about where our regression variables are coming from — even if `my_data` is the only data frame we have loaded at the time.

Let's run a simple bivariate regression of mass on height using our dataset of starwars characters.

```
ols1 <- lm(mass ~ height, data = df)
```

And look at the output:

```
ols1
```

```
##
## Call:
## lm(formula = mass ~ height, data = df)
##
## Coefficients:
## (Intercept)       height
##    -13.8103       0.6386
```

The resulting output is pretty terse. That's only because it buries most of its valuable information — of which there is a lot — within its internal list structure.

If you're in RStudio, you can inspect this structure by typing `View(ols1)` or simply clicking on the "ols1" object in your environment pane. Doing so will prompt an interactive panel to pop up for you to play around with. That approach won't work for this knitted R Markdown document, but I'll explain what you would see if you did this:

The `ols1` object has a bunch of important stuff nested inside it... containing everything from the regression coefficients, to vectors of the residuals and fitted (i.e. predicted) values, to the input data, . To summarise the key pieces of information, we can use the generic `summary()` function.

```
summary(ols1)
```

```
##
## Call:
## lm(formula = mass ~ height, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
##  -61.43  -30.03  -21.13  -17.73 1260.06
##
## Coefficients:
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -13.8103   111.1545  -0.124    0.902
## height        0.6386     0.6261   1.020    0.312
##
## Residual standard error: 169.4 on 57 degrees of freedom
##   (28 observations deleted due to missingness)
## Multiple R-squared:  0.01792,    Adjusted R-squared:  0.0006956
## F-statistic:  1.04 on 1 and 57 DF,  p-value: 0.312
```

Get "tidy" regression coefficients with the `broom` package

While it's easy to extract regression coefficients via the `summary()` function, in practice I've begun using the **broom** package (link) as a simple alternative. **broom** has a bunch of features to convert regression (and other statistical) objects into "tidy" data frames. This is especially useful because regression output is so often used as an input to something else. Let's use `broom::tidy(..., conf.int = TRUE)` to coerce the `ols1` regression object into a tidy data frame of coefficient values and key statistics.

```
# library(broom) ## Already loaded
tidy(ols1, conf.int = TRUE)
```

```
## # A tibble: 2 x 7
##   term        estimate std.error statistic p.value conf.low conf.high
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
## 1 (Intercept)  -13.8     111.       -0.124   0.902 -236.       209.
## 2 height         0.639     0.626     1.02    0.312   -0.615      1.89
```

broom has a couple of other useful functions too. For example, `broom::glance()` summarises the model "meta" data (such as R^2) in a data frame.

```
glance(ols1)
```

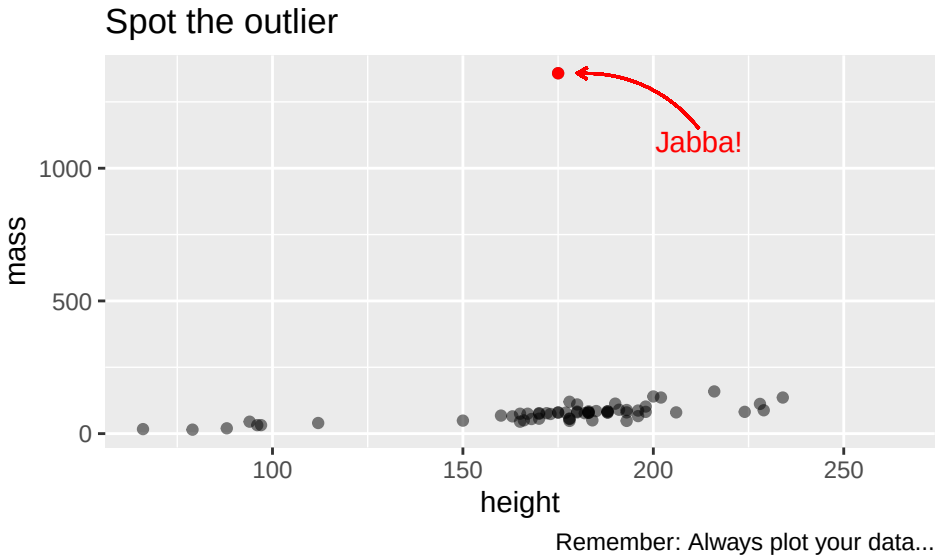
```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1    0.0179      0.000696  169.      1.04   0.312     1  -386.  777.  783.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Note: If you're wondering how to export regression results to other formats (e.g. nice tables that we see in reports and publications), we'll get to that at the end of this note.

Regressing on a Subset of Data

So far we have a single variable regression to explain mass - its all about the height of the character. This seems a little inadequate — we could readily think of other characteristics such as species and homeworld to explain mass in addition to height.

Before we get to that, our data has an extreme outlier:



Maybe we should exclude Jabba from our regression?

You can do this in two ways: 1) Create a new data frame and then regress, or 2) Subset the original data frame directly in the `lm()` call. Let's look at how to do each of these now:

1) Create a new data frame

Recall that we can keep multiple objects in memory in R. So we can easily create a new data frame that excludes Jabba using the `filter()` command from **dplyr** to remove him.

Let's remove him and re-run the regression:

```
starwars2 <-
  df %>%
  filter(name != "Jabba Desilijic Tiure")

ols2 <- lm(mass ~ height,
           data = starwars2)
tidy(ols2, conf.int = TRUE)
```

```
## # A tibble: 2 x 7
##   term        estimate std.error statistic  p.value conf.low conf.high
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
## 1 (Intercept)  -32.5     12.6        -2.59 1.22e- 2  -57.7      -7.38
## 2 height         0.621    0.0707      8.79 4.02e-12    0.480     0.763
```

2) Subset directly in the `lm()` call

Running a regression directly on a subsetted data frame is equally easy.

```
ols2a <- lm(mass ~ height,
            data = starwars %>% filter(name != "Jabba Desilijic Tiure"))

tidy(ols2a, conf.int = TRUE)
```

```
## # A tibble: 2 x 7
##   term        estimate std.error statistic  p.value conf.low conf.high
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
## 1 (Intercept)  -32.5     12.6        -2.59 1.22e- 2  -57.7      -7.38
```

```
## 2 height         0.621    0.0707      8.79 4.02e-12    0.480     0.763
```

```
glance(ols2a)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1     0.580         0.572  19.1      77.2 4.02e-12     1  -252.  511.  517.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

The overall model fit is much improved by the exclusion of this outlier, with R^2 increasing to 0.58. However, we should be quite cautious about throwing out data.[2]

Nonstandard errors

Dealing with statistical irregularities (heteroskedasticity, clustering, etc.) is a fact of life for empirical researchers. The **estimatr** package (link), provides convenient aliases for the standard regression functions, including standard error corrections easily as part of the regression command.[3] Let's see it in action.

Robust standard errors

You can get heteroskedasticity-consistent (HC) "robust" standard errors using `estimatr::lm_robust()`. Let's implementing a robust version of the `ols1` regression that we ran earlier.

```
# library(estimatr) ## Already loaded
ols1_robust <- lm_robust(mass ~ height,
                         data = df)
tidy(ols1_robust, conf.int = TRUE)
```

```
##          term   estimate   std.error  statistic      p.value    conf.low
## 1 (Intercept) -13.810314 23.45557632 -0.5887859 5.583311e-01 -60.7792950
## 2      height   0.638571  0.08791977  7.2631109 1.159161e-09   0.4625147
##    conf.high df outcome
## 1 33.1586678 57    mass
## 2  0.8146273 57    mass
```

The package defaults to using "Eicker-Huber-White" robust standard errors, commonly referred to as "HC2" standard errors. This is not the 'default' robust standard error that some other software packages use, so any difference across software might stem from there.[4]

Clustered standard errors

Clustered standard errors is an issue that most commonly affects data when we observe the same individual multiple times. As such, let's hold off a long discussion of clustering until we get to the panel data section below. But here's a quick example of clustering with `estimatr::lm_robust()` just to illustrate, where we suppose the regression error variance differs by a character's homeworld:

```
ols1_cluster <- lm_robust(mass ~ height,
                          data = starwars,
                          clusters = homeworld)
tidy(ols1_cluster, conf.int = TRUE)
```

[2] Since I am focussed on regression in this note, I'm not going to go in to detail on data cleaning / filtering here. In short, be cautious when you decide to filter out observations and think about the consequences of doing so.

[3] For many years, dealing with standard errors was the domain of the **sandwich** package (link). Lot's of legacy code will use this appraoch, so I thought I should mention it.

[4] You can easily specify alternate methods using the `se_type =` argument.[5] For example, you can specify Stata robust standard errors if you want to replicate code or results from that language.

```
##          term   estimate   std.error  statistic      p.value    conf.low
## 1 (Intercept) -9.3014938 28.84436408 -0.3224718 0.7559158751 -76.6200628
## 2      height  0.6134058  0.09911832  6.1886211 0.0002378887   0.3857824
##    conf.high       df outcome
## 1 58.0170751 7.486034    mass
## 2  0.8410291 8.195141    mass
```

Dummy Variables

For the next couple of sections, it will prove convenient to demonstrate R's regression functionality using a subsample starwars data that uses only the human characters.

```
humans <-
  starwars %>%
  filter(species=="Human")
```

Dummy Variables

Dummy variables are a core component of many regression models. R is "dummy variable friendly" and makes using dummy variables easy for us. Let's suppose we want to add `gender` to our regression model to explain mass. In our `humans` sample, `gender` takes the following values:

```
humans %>%
    select(gender) %>%
    distinct()
```

```
## # A tibble: 2 x 1
##   gender
##   <chr>
## 1 masculine
## 2 feminine
```

Notice here that `gender` is a `character` variable (i.e. a string of text). If we simply include gender in the regression:

```
ols_gender <- lm(mass ~ height + gender,
                 data = humans)
tidy(ols_gender, conf.int = TRUE)
```

```
## # A tibble: 3 x 7
##   term            estimate std.error statistic p.value  conf.low conf.high
##   <chr>              <dbl>     <dbl>     <dbl>   <dbl>     <dbl>     <dbl>
## 1 (Intercept)      -84.3      65.8      -1.28   0.216  -222.         53.4
## 2 height             0.879     0.407     2.16   0.0441    0.0258      1.73
## 3 gendermasculine   10.7      13.2       0.814  0.426   -16.9        38.4
```

R knows what to do with it. Specifically, it knows `gender` takes on two values, and thus knows to include only one of them in the regression (here it chose masculine). R knows this because it understands that the only way to use a set of character strings in a regression sensibly is to include them as dummy variables *and* it knows it must include all but one of the possible set of string values. This is quite nice![6]

Interaction effects

R also provides a convenient syntax for specifying interaction terms directly in the regression model. The following expansion operators are what we need:

[6] OK, it may seem obvious now we've explained it ... but many other statistical softwares don't allow you to do this so easily.

- `x1:x2` "crosses" the variables (equivalent to including only the x1 × x2 interaction term)
- `x1/x2` "nests" the second variable within the first (equivalent to `x1 + x1:x2`;
- `x1*x2` includes all parent and interaction terms (equivalent to `x1 + x2 + x1:x2`)

As a rule of thumb, it is generally advisable to include all of the parent terms alongside their interactions. This makes the `*` option a good default.[7]

Let's see this in action! We might think that the relationship between a person's body mass and their height is modulated by their gender, i.e. the the effect of height on mass differs by gender. Then we would want to run a regression of the form,

$$Mass = \beta_0 + \beta_1 D_{Male} + \beta_2 Height + \beta_3 D_{Male} \times Height$$

To implement this in R, we run the following,

```
ols_ie <- lm(mass ~ gender * height, data = humans)
tidy(ols_ie, conf.int = TRUE)
```

```
## # A tibble: 4 x 7
##   term                   estimate std.error statistic p.value conf.low conf.high
##   <chr>                     <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
## 1 (Intercept)              -61.0      204.     -0.299    0.768  -490.      368.
## 2 gendermasculine          -15.7      220.     -0.0716   0.944  -477.      446.
## 3 height                     0.733      1.27    0.576    0.572    -1.94      3.41
## 4 gendermasculine:height     0.163      1.35    0.121    0.905    -2.67      3.00
```

Fixed Effect Models

Fixed effects with the fixest package

fixest is relatively new on the scene and has become a bit of a mainstay for regression modellers in economics and marketing.
We won't be able to cover all of **fixest**'s features in depth here, but I hope to give you everything you need to get up and running comfortably.

Simple FE model

The package's main function is `fixest::feols()`, which is used for estimating linear fixed effects models. The syntax is such that you first specify the regression model as per normal, and then list the fixed effect(s) after a `|`.

Let's use an example to illustrate. We want to run our simple regression of mass on height, but this time control for species-level fixed effects:

```
# library(fixest) ## Already loaded
ols_fe <- feols(mass ~ height
                | ## Fixed effect(s) go after the "|"
                  ## I think writing over multiple lines
                  ## helps me be explicit
                species,
                data = starwars)

tidy(ols_fe, conf.int = TRUE)
```

```
## # A tibble: 1 x 7
##   term   estimate std.error statistic  p.value conf.low conf.high
```

[7] Though I personally have a habit of using ":" and then manually including the `x1+x2`.

```
##   <chr>     <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
## 1 height    0.975    0.0443      22.0 4.56e-20    0.888      1.06
```

Note that the estimated model `ols_fe` has automatically clustered the standard errors by the fixed effect variable (i.e. species). If we want to do something else, we need to be explicit:

We'll explore some more options for adjusting standard errors in **fixest** objects shortly, but you can specify which standard errors you want reported as follows:

```
tidy(ols_fe, se = 'standard', conf.int = TRUE)
```

```
## # A tibble: 1 x 7
##   term   estimate std.error statistic     p.value conf.low conf.high
##   <chr>     <dbl>     <dbl>     <dbl>       <dbl>    <dbl>     <dbl>
## 1 height    0.975     0.136      7.14 0.000000138    0.707      1.24
```

Which reports the 'standard' standard errors (i.e. no clustering, and no heteroskedasticity adjustment).

Before continuing, let's quickly save a "tidied" data frame of the coefficients for later use.

```
coefs_fe <- tidy(ols_fe, se = 'standard', conf.int = TRUE)
```

High dimensional FEs and multiway clustering

fixest supports (arbitrarily) high-dimensional fixed effects - which means we can have as many fixed effects are we like. It also supports multiway clustering, so we can have clustered standard errors of relatively sophisticated form if we need to. For the purpose of this class, the large number of fixed effects is most important, so let's see it in action. To do this, we'll add "homeworld as a fixed effect".

```
ols_hdfe <- feols(mass ~ height
                  |
                species + homeworld,
                data = starwars)
tidy(ols_hdfe, conf.int = TRUE)
```

```
## # A tibble: 1 x 7
##   term   estimate std.error statistic p.value conf.low conf.high
##   <chr>     <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
## 1 height    0.756     0.333      2.27  0.0308    0.103      1.41
```

That seems easy enough to do, but there's a slight wrinkle. The standard errors are automatically clustered by the first fixed effect variable, which for us is `species`. If we wanted to cluster them instead by `homeworld`, we'd need to be explicit:

```
ols_hdfe2 <- feols(mass ~ height
                   |
                 species + homeworld,
                 cluster = ~homeworld,
                 data = starwars)
```

```
## NOTE: 32 observations removed because of NA values (LHS: 28, RHS: 6, Fixed-effects: 13).
```

```
tidy(ols_hdfe2, conf.int = TRUE)
```

```
## # A tibble: 1 x 7
##   term   estimate std.error statistic  p.value conf.low conf.high
##   <chr>     <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
## 1 height    0.756    0.0457      16.5 1.16e-18    0.666     0.846
```

And if we wanted to cluster by both `species` and homeworld:

```
ols_hdfe3 <- feols(mass ~ height
                     |
                   species + homeworld,
                   cluster = ~species + homeworld,
                   data = starwars)
```

```
## NOTE: 32 observations removed because of NA values (LHS: 28, RHS: 6, Fixed-effects: 13).
```

```
tidy(ols_hdfe3, conf.int = TRUE)
```

```
## # A tibble: 1 x 7
##   term   estimate std.error statistic     p.value conf.low conf.high
##   <chr>     <dbl>     <dbl>     <dbl>       <dbl>    <dbl>     <dbl>
## 1 height    0.756     0.116      6.49 0.000000416    0.528     0.984
```

Presentating Regression Estimates

I'm going to discuss two ways to present regression estimates: as a regression table, and as a coefficient plot.

Regression Tables

There are loads of different packages to help package up regressions into tables. I prefer **modelsummary** package (link) for creating and exporting regression tables.[8] It is extremely flexible and handles all manner of models and output formats. **modelsummary** also supports automated coefficient plots and data summary tables (which we'll want to get get back to shortly).

To see how easily we can get started, lets produce a summary table of a collection of the estimates we produced so far:

```
# library(modelsummary) ## Already loaded
## Note: msummary() is an alias for modelsummary()
msummary(list(ols1, ols_ie, ols_fe, ols_hdfe))
```

Immediately we see two great features (table appears on next page due to space constraints):

1. The default table already looks relatively appealing
2. The table came out in the same format as our document
   - You might not have even noticed this
   - **modelsummary** will automatically coerce your tables to the format that matches your document output: HTML, LaTeX/PDF, RTF, etc. Of course, you can also specify the output type

I'm not going to go further into formatting this table, if you're interested check out the documentation – you can do *a lot*.

[8] The documentation is outstanding and you should read it, but here is a bare-boned example just to demonstrate.

	Model 1	Model 2	Model 3	Model 4
(Intercept)	-13.810	-61.000		
	(111.155)	(204.057)		
height	0.639	0.733	0.975	0.756
	(0.626)	(1.274)	(0.044)	(0.333)
gendermasculine		-15.722		
		(219.544)		
gendermasculine × height		0.163		
		(1.349)		
Num.Obs.	59	22	58	55
R2	0.018	0.444	0.997	0.998
R2 Adj.	0.001	0.352	0.993	1.008
R2 Within			0.662	0.487
R2 Pseudo				
AIC	777.0	188.9	492.1	513.1
BIC	783.2	194.4	558.0	649.6
Log.Lik.	-385.503	-89.456	-214.026	-188.552
F	1.040	4.801		
Std. Errors			Clustered (species)	Clustered (species)
FE: homeworld				X
FE: species			X	X

		feminine (N=9)		masculine (N=26)			
		Mean	Std. Dev.	Mean	Std. Dev.	Diff. in Means	Std. Error
height		160.2	7.0	182.3	8.2	22.1	3.0
mass		56.3	16.3	87.0	16.5	30.6	10.1
birth_year		46.4	18.8	55.2	26.0	8.8	10.2
		N	%	N	%		
eye_color	blue	3	33.3	9	34.6		
	blue-gray	0	0.0	1	3.8		
	brown	5	55.6	12	46.2		
	dark	0	0.0	1	3.8		
	hazel	1	11.1	1	3.8		
	yellow	0	0.0	2	7.7		

Aside: Data Summary Tables

A variety of summary tables can be produced by the set of `datasummary*()` functions that live inside **modelsummary**. Again, read the documentation to see all of the options.

To show an example, let's get a smaller set of columns from our humans data:

```
humans2 <- humans %>%
           select(gender, height, mass, birth_year, eye_color)
```

Now we can look at how summary statistics of `height`, `mass`, `birth_year` and `eye_color` vary across gender (output appears above as the second table):

```
datasummary_balance(~ gender,
                    data = humans2)
```

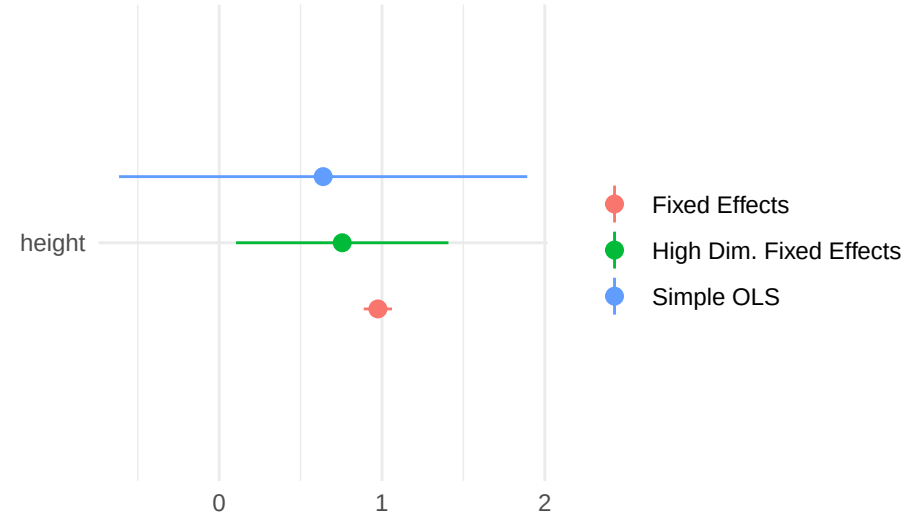
Note that for the continuous variables `height`, `mass`, `birth_year` we get the difference in means between the two groups.

Coefficient Plots

Sometimes instead of reporting regression results as a table, we would like to visually see the difference in an estimated coefficient and the standard error in a plot. The `modelplot()` function from the **modelsummary** package can do this for us.

For example, let's try and visualize the different regression coefficients and the standard errors from the simple OLS regression, the fixed effect regression and the high dimensional fixed effect regression:

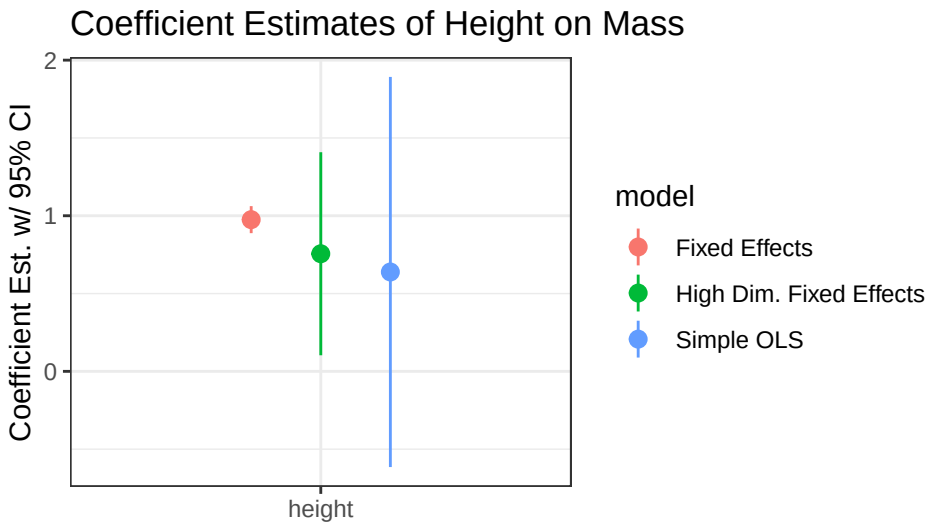
```
# library(modelsummary) ## Already loaded

# create a list of the models we want to present
mods = list('Simple OLS' = ols1,
            'Fixed Effects' = ols_fe,
            'High Dim. Fixed Effects' = ols_hdfe
                )

modelplot(mods,
          coef_omit = "Inter")
```



Coefficient estimates and 95% confidence intervals

Again, the default plot looks quite presentable. `modelplot()` creates figures that are the `ggplot2` objects, which means that you can do further post-processing of the plot using functions we already know from `ggplot2`. For example:

```
modelplot(mods,
          coef_omit = "Inter") +
    coord_flip() +
    ggtitle('Coefficient Estimates of Height on Mass') +
    # xlab because it only appears on y-axis by flipping,
    # need to use the normal location
    xlab('Coefficient Est. w/ 95% CI') +
    theme_bw()
```



Acknowledgements

This guide borrows quite liberally (occasionally maybe even too much so at times) from an **excellent** lecture from Grant McDermott at U Oregon "Regression Analysis in R" as part of his class "Data Science for Economists". I've tried to trim down the content to fit what is needed for this class, and add some of my own flair here and there.

- Any errors are purely my own!