Data Science for Economists

Lecture 8: Regression analysis in R

Grant R. McDermott

University of Oregon | EC 607

Contents

Today's lecture is about the bread-and-butter tool of applied econometrics and data science: regression analysis. My 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 causal inference, asymptotics, etc. This lecture will *not* cover any of theoretical concepts or seek to justify a particular statistical model. Indeed, most of the models that we're going to run today are pretty silly. We also won't be able to cover some important topics. For example, I'll only provide the briefest example of a Bayesian regression model and I won't touch times series analysis at all. (Although, I will provide links for further reading at the bottom of this document.) These disclaimers aside, let's proceed…

Software requirements

R packages

It's important to note that "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: **fixest**, **estimatr**, **ivreg**, **sandwich**, **lmtest**, **mfx**, **margins**, **broom**, **modelsummary**, **vtable**
- Already used: **tidyverse**, **hrbrthemes**, **listviewer**

Note that I'm using **ivreg** 0.6.0, which must be installed as the development version as of the time of writing. A convenient way to install (if necessary) and load everything is by running the below code chunk.

```
## Install development version of ivreg if necessary
if (numeric_version(packageVersion("ivreg")) < numeric_version("0.6.0")) {
  remotes::install_github("john-d-fox/ivreg")
}
## Load and install the packages that we'll be using today
if (!require("pacman")) install.packages("pacman")
pacman::p_load(mfx, tidyverse, hrbrthemes, estimatr, ivreg, fixest, sandwich, lmtest, margins, vtable, broom, mo
```

```
## My preferred ggplot2 plotting theme (optional)
theme_set(hrbrthemes::theme_ipsum())
```

While we've already loaded all of the required packages for today, I'll try to be as explicit about where a particular function is coming from, whenever I use it below.

Something else that I want to mention up front is that we'll mostly be working with the `starwars` data frame that we've already seen from previous lectures. Here's a quick reminder of what it looks like to refresh your memory.

```
starwars
```

```
## # A tibble: 87 x 14
##    name  height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke~    172    77 blond      fair       blue            19   male  mascu~
##  2 C-3PO    167    75 <NA>       gold       yellow         112   none  mascu~
##  3 R2-D2     96    32 <NA>       white, bl~ red             33   none  mascu~
##  4 Dart~    202   136 none       white      yellow          41.9 male  mascu~
##  5 Leia~    150    49 brown      light      brown           19   fema~ femin~
##  6 Owen~    178   120 brown, gr~ light      blue            52   male  mascu~
##  7 Beru~    165    75 brown      light      blue            47   fema~ femin~
##  8 R5-D4     97    32 <NA>       white, red red             NA   none  mascu~
##  9 Bigg~    183    84 black      light      brown           24   male  mascu~
## 10 Obi-~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
## # ... with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Regression basics

The `lm()` function

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

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

You'll note that the `lm()` call includes a reference to the data source (in this case, a hypothetical data frame called `df`). We covered this in our earlier lecture on R language basics and object-orientated programming, but the reason is that many objects (e.g. data frames) can exist in your R environment at the same time. So we need to be specific about where our regression variables are coming from — even if `df` is the only data frame in our global environment at the time. Another option would be to use indexing, but I find it a bit verbose:

```
lm(df$y ~ df$x1 + df$x2 + df$x3 + ...)
```

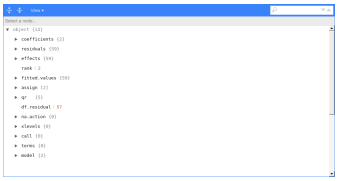
Let's run a simple bivariate regression of mass on height using our dataset of starwars characters.

```
ols1 = lm(mass ~ height, data = starwars)
# ols1 = lm(starwars$mass ~ starwars$height) ## Also works
ols1
```

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

The resulting object is pretty terse, but 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, however, so I'll use the `listviewer::jsonedit()` function that we saw in the previous lecture instead.

```
# View(ols1) ## Run this instead if you're in a live session
listviewer::jsonedit(ols1, mode="view") ## Better for R Markdown
```



As we can see, this `ols1` object has a bunch of important slots… containing everything from the regression coefficients, to vectors of the residuals and fitted (i.e. predicted) values, to the rank of the design matrix, to the input data, etc. etc. To summarise the key pieces of information, we can use the — *wait for it* — generic `summary()` function. This will look pretty similar to the default regression output from Stata that many of you will be used to.

```
summary(ols1)
```

```
##
## Call:
## lm(formula = mass ~ height, data = starwars)
##
## 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
```

We can then dig down further by extracting a summary of the regression coefficients:

```
summary(ols1)$coefficients
```

```
##               Estimate  Std. Error    t value  Pr(>|t|)
## (Intercept) -13.810314 111.1545260 -0.1242443 0.9015590
## height        0.638571   0.6260583  1.0199865 0.3120447
```

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

While it's easy to extract regression coefficients via the `summary()` function, in practice I always use the **broom** package (link) to do so. **broom** has a bunch of neat 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, e.g. a plot of coefficients or marginal effects. Here, I'll 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
```

Again, I could now pipe this tidied coefficients data frame to a **ggplot2** call, using saying `geom_pointrange()` to plot the error bars. Feel free to practice doing this yourself now, but we'll get to some explicit examples further below.

broom has a couple of other useful functions too. For example, `broom::glance()` summarises the model "meta" data (R2, AIC, etc.) in a data frame.

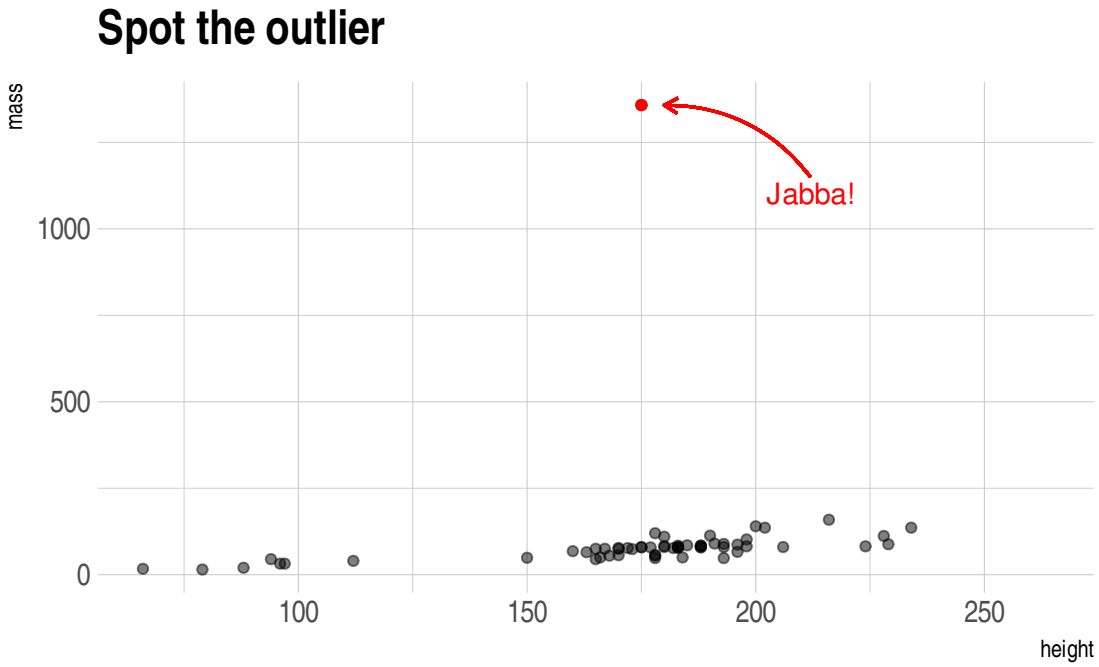
```
glance(ols1)
```

```
## # 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>
```

By the way, if you're wondering how to export regression results to other formats (e.g. LaTeX tables), don't worry: We'll get to that at the end of the lecture.

Regressing on subsetted data

Our simple model isn't particularly good; the R2 is only 0.018. Different species and homeworlds aside, we may have an extreme outlier in our midst…



Remember: Always plot your data...

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.

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, say, **dplyr** (lecture) or **data.table** (lecture). For these lecture notes, I'll stick with **dplyr** commands since that's where our starwars dataset is coming from. But it would be trivial to switch to **data.table** if you prefer.

```
starwars2 =
  starwars %>%
  filter(name ≠ "Jabba Desilijic Tiure")
  # filter(!(grepl("Jabba", name))) ## Regular expressions also work

ols2 = lm(mass ~ height, data = starwars2)
summary(ols2)
```

```
##
## Call:
## lm(formula = mass ~ height, data = starwars2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -39.382  -8.212   0.211   3.846  57.327
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -32.54076   12.56053  -2.591   0.0122 *
## height        0.62136    0.07073   8.785 4.02e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 19.14 on 56 degrees of freedom
##   (28 observations deleted due to missingness)
## Multiple R-squared:  0.5795, Adjusted R-squared:  0.572
## F-statistic: 77.18 on 1 and 56 DF,  p-value: 4.018e-12
```

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(!(grepl("Jabba", name))))
summary(ols2a)
```

```
##
## Call:
## lm(formula = mass ~ height, data = starwars %>% filter(!(grepl("Jabba",
##     name))))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -39.382  -8.212   0.211   3.846  57.327
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -32.54076   12.56053  -2.591   0.0122 *
## height        0.62136    0.07073   8.785 4.02e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 19.14 on 56 degrees of freedom
```

```
##   (28 observations deleted due to missingness)
## Multiple R-squared:  0.5795, Adjusted R-squared:  0.572
## F-statistic: 77.18 on 1 and 56 DF,  p-value: 4.018e-12
```

The overall model fit is much improved by the exclusion of this outlier, with R2 increasing to 0.58. Still, we should be cautious about throwing out data. Another approach is to handle or account for outliers with statistical methods. Which provides a nice segue to nonstandard errors.

Nonstandard errors

Dealing with statistical irregularities (heteroskedasticity, clustering, etc.) is a fact of life for empirical researchers. However, it says something about the economics profession that a random stranger could walk uninvited into a live seminar and ask, "How did you cluster your standard errors?", and it would likely draw approving nods from audience members.

The good news is that there are *lots* of ways to get nonstandard errors in R. For many years, these have been based on the excellent **sandwich** package (link). However, my preferred way these days is to use the **estimatr** package (link), which is both fast and provides convenient aliases for the standard regression functions. Some examples follow below.

Robust standard errors

You can obtain heteroskedasticity-consistent (HC) "robust" standard errors using `estimatr::lm_robust()`. Let's illustrate by implementing a robust version of the `ols1` regression that we ran earlier. Note that **estimatr** models automatically print in pleasing tidied/summary format, although you can certainly pipe them to `tidy()` too.

```
# library(estimatr) ## Already loaded

ols1_robust = lm_robust(mass ~ height, data = starwars)
# tidy(ols1_robust, conf.int = TRUE) ## Could tidy too
ols1_robust
```

```
##               Estimate  Std. Error    t value     Pr(>|t|)    CI Lower
## (Intercept) -13.810314 23.45557632 -0.5887859 5.583311e-01 -60.7792950
## height        0.638571  0.08791977  7.2631109 1.159161e-09   0.4625147
##               CI Upper DF
## (Intercept) 33.1586678 57
## height       0.8146273 57
```

The package defaults to using Eicker-Huber-White robust standard errors, commonly referred to as "HC2" standard errors. You can easily specify alternate methods using the `se_type =` argument.[1] For example, you can specify Stata robust standard errors if you want to replicate code or results from that language. (See here for more details on why this isn't the default and why Stata's robust standard errors differ from those in R and Python.)

```
lm_robust(mass ~ height, data = starwars, se_type = "stata")
```

```
##               Estimate  Std. Error    t value     Pr(>|t|)    CI Lower
## (Intercept) -13.810314 23.36219608 -0.5911394 5.567641e-01 -60.5923043
## height        0.638571  0.08616105  7.4113649 6.561046e-10   0.4660365
##               CI Upper DF
## (Intercept) 32.9716771 57
## height       0.8111055 57
```

estimatr also supports robust instrumental variable (IV) regression. However, I'm going to hold off discussing these until we get to the IV section below.

Aside on HAC (Newey-West) standard errors One thing I want to flag is that **estimatr** does not yet offer support for HAC (i.e. heteroskedasticity *and* autocorrelation consistent) standard errors *a la* Newey-West. I've submitted a feature

[1] See the package documentation for a full list of options.

request on GitHub — vote up if you would like to see it added sooner! — but you can still obtain these pretty easily using the aforementioned **sandwich** package. For example, we can use `sandwich::NeweyWest()` on our existing `ols1` object to obtain HAC SEs for it.

```
# library(sandwich) ## Already loaded

# NeweyWest(ols1) ## Print the HAC VCOV
sqrt(diag(NeweyWest(ols1))) ## Print the HAC SEs
```

```
## (Intercept)      height
##  21.2694130   0.0774265
```

If you plan to use HAC SEs for inference, then I recommend converting the model object with `lmtest::coeftest()`. This function builds on **sandwich** and provides a convenient way to do on-the-fly hypothesis testing with your model, swapping out a wide variety of alternate variance-covariance (VCOV) matrices. These alternate VCOV matrices could extended way beyond HAC — including HC, clustered, bootsrapped, etc. — but here's how it would work for the present case:

```
# library(lmtest) ## Already loaded

ols1_hac = lmtest::coeftest(ols1, vcov = NeweyWest)
ols1_hac
```

```
##
## t test of coefficients:
##
##               Estimate Std. Error t value  Pr(>|t|)
## (Intercept) -13.810314  21.269413 -0.6493    0.5187
## height        0.638571   0.077427  8.2474 2.672e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that its easy to convert `coeftest()`-adjusted models to tidied **broom** objects too.

```
tidy(ols1_hac, 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     21.3       -0.649 5.19e- 1  -56.4      28.8
## 2 height         0.639    0.0774     8.25  2.67e-11    0.484     0.794
```

Clustered standard errors

Clustered standard errors is an issue that most commonly affects panel data. As such, I'm going to hold off discussing 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:

```
lm_robust(mass ~ height, data = starwars, clusters = homeworld)
```

```
##               Estimate  Std. Error    t value     Pr(>|t|)    CI Lower
## (Intercept) -9.3014938 28.84436408 -0.3224718 0.7559158751 -76.6200628
## height       0.6134058  0.09911832  6.1886211 0.0002378887   0.3857824
##               CI Upper       DF
## (Intercept) 58.0170751 7.486034
## height       0.8410291 8.195141
```

Dummy variables and interaction terms

For the next few sections, it will prove convenient to demonstrate using a subsample of the starwars data that comprises only the human characters. Let's quickly create this new dataset before continuing. Note that I'm creating a factored (i.e. ordered) version of the "gender" variable, since I want to demonstrate some general principles about factors in the paragraphs that follow.

```
humans =
  starwars %>%
  filter(species=="Human") %>%
  mutate(gender_factored = as.factor(gender)) %>% ## create factored version of "gender"
  select(contains("gender"), everything())
humans
```

```
## # A tibble: 35 x 15
##    gender gender_factored name  height  mass hair_color skin_color eye_color
##    <chr>  <fct>           <chr>  <int> <dbl> <chr>      <chr>      <chr>
##  1 mascu~ masculine       Luke~    172    77 blond      fair       blue
##  2 mascu~ masculine       Dart~    202   136 none       white      yellow
##  3 femin~ feminine        Leia~    150    49 brown      light      brown
##  4 mascu~ masculine       Owen~    178   120 brown, gr~ light      blue
##  5 femin~ feminine        Beru~    165    75 brown      light      blue
##  6 mascu~ masculine       Bigg~    183    84 black      light      brown
##  7 mascu~ masculine       Obi-~    182    77 auburn, w~ fair       blue-gray
##  8 mascu~ masculine       Anak~    188    84 blond      fair       blue
##  9 mascu~ masculine       Wilh~    180    NA auburn, g~ fair       blue
## 10 mascu~ masculine       Han ~    180    80 brown      fair       brown
## # ... with 25 more rows, and 7 more variables: birth_year <dbl>, sex <chr>,
## #   homeworld <chr>, species <chr>, films <list>, vehicles <list>,
## #   starships <list>
```

Dummy variables as *factors*

Dummy variables are a core component of many regression models. However, these can be a pain to create in some statistical languages, since you first have to tabulate a whole new matrix of binary variables and then append it to the original data frame. In contrast, R has a very convenient framework for creating and evaluating dummy variables in a regression: Simply specify the variable of interest as a factor.[2]

Here's an example using the "gendered_factored" variable that we explicitly created earlier. Since I don't plan on reusing this model, I'm just going to print the results to screen rather than saving it to my global environment.

```
summary(lm(mass ~ height + gender_factored, data = humans))
```

```
##
## Call:
## lm(formula = mass ~ height + gender_factored, data = humans)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -16.068  -8.130  -3.660   0.702  37.112
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)              -84.2520    65.7856  -1.281   0.2157
## height                     0.8787     0.4075   2.156   0.0441 *
## gender_factoredmasculine  10.7391    13.1968   0.814   0.4259
```

[2] Factors are variables that have distinct qualitative levels, e.g. "male", "female", "hermaphrodite", etc.

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.19 on 19 degrees of freedom
##   (13 observations deleted due to missingness)
## Multiple R-squared:  0.444,  Adjusted R-squared:  0.3855
## F-statistic: 7.587 on 2 and 19 DF,  p-value: 0.003784
```

Okay, I should tell you that I'm actually making things more complicated than they need to be with the heavy-handed emphasis on factors. R is "friendly" and tries to help whenever it thinks you have misspecified a function or variable. While this is something to be aware of, normally It Just WorksTM. A case in point is that we don't actually *need* to specify a string (i.e. character) variable as a factor in a regression. R will automatically do this for you regardless, since it's the only sensible way to include string variables in a regression.

```
## Use the non-factored version of "gender" instead; R knows it must be ordered
## for it to be included as a regression variable
summary(lm(mass ~ height + gender, data = humans))
```

```
##
## Call:
## lm(formula = mass ~ height + gender, data = humans)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -16.068  -8.130  -3.660   0.702  37.112
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     -84.2520    65.7856  -1.281   0.2157
## height            0.8787     0.4075   2.156   0.0441 *
## gendermasculine  10.7391    13.1968   0.814   0.4259
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.19 on 19 degrees of freedom
##   (13 observations deleted due to missingness)
## Multiple R-squared:  0.444,  Adjusted R-squared:  0.3855
## F-statistic: 7.587 on 2 and 19 DF,  p-value: 0.003784
```

Interaction effects

Like dummy variables, R provides a convenient syntax for specifying interaction terms directly in the regression model without having to create them manually beforehand.[3] You can use any of the following expansion operators:

- `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`; more on this later)
- `x1*x2` includes all parent and interaction terms (equivalent to `x1 + x2 + x1:x2`)

As a rule of thumb, albeit not always, it is generally advisable to include all of the parent terms alongside their interactions. This makes the `*` option a good default.

For example, we might wonder whether the relationship between a person's body mass and their height is modulated by their gender. That is, we want to run a regression of the form,

[3] Although there are very good reasons that you might want to modify your parent variables before doing so (e.g. centering them). As it happens, I'm on record as stating that interaction effects are most widely misunderstood and misapplied concept in econometrics. However, that's a topic for another day. (Read the paper in the link!)

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

To implement this in R, we simply run the following,

```
ols_ie = lm(mass ~ gender * height, data = humans)
summary(ols_ie)
```

```
##
## Call:
## lm(formula = mass ~ gender * height, data = humans)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -16.250  -8.158  -3.684  -0.107  37.193
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)            -61.0000   204.0565  -0.299    0.768
## gendermasculine        -15.7224   219.5440  -0.072    0.944
## height                   0.7333     1.2741   0.576    0.572
## gendermasculine:height   0.1629     1.3489   0.121    0.905
##
## Residual standard error: 15.6 on 18 degrees of freedom
##   (13 observations deleted due to missingness)
## Multiple R-squared:  0.4445, Adjusted R-squared:  0.3519
## F-statistic: 4.801 on 3 and 18 DF,  p-value: 0.01254
```

Panel models

Fixed effects with the fixest package

The simplest (and least efficient) way to include fixed effects in a regression model is, of course, to use dummy variables. However, it isn't very efficient or scalable. What's the point learning all that stuff about the Frisch-Waugh-Lovell, within-group transformations, etc. etc. if we can't use them in our software routines? Again, there are several options to choose from here. For example, many of you are probably familiar with the excellent **lfe** package (link), which offers near-identical functionality to the popular Stata library, **reghdfe** (link). However, for fixed effects models in R, I am going to advocate that you take a look at the **fixest** package (link).

fixest is relatively new on the scene and has quickly become one of my packages in the entire R catalogue. It has a boatload of functionality built in to it: support for nonlinear models, high-dimensional fixed effects, multiway clustering, etc. It is also insanely fast… as in, up to *orders of magnitude* faster than **lfe** or **reghdfe**. I won't be able to cover all of **fixest**'s features in depth here — see the introductory vignette for a thorough walkthrough — but I hope to least give you a sense of why I am so enthusiastic about it. Let's start off with a simple example before moving on to something a little more demanding.

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 `|`. An example may help to illustrate. Let's say that we again want to run our simple regression of mass on height, but this time control for species-level fixed effects.[4]

```
# library(fixest) ## Already loaded

ols_fe = feols(mass ~ height | species, data = starwars) ## Fixed effect(s) go after the "|"
ols_fe
```

[4]Since we specify "species" in the fixed effects slot below, `feols()` will automatically coerce it to a factor variable even though we didn't explicitly tell it to.

```
## OLS estimation, Dep. Var.: mass
## Observations: 58
## Fixed-effects: species: 31
## Standard-errors: Clustered (species)
##        Estimate Std. Error t value  Pr(>|t|)
## height 0.974876   0.044291   22.01 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 9.6906     Adj. R2: 0.99282
##                Within R2: 0.662493
```

Note that the resulting model object has automatically clustered the standard errors by the fixed effect variable (i.e. species). We'll explore some more options for adjusting standard errors in **fixest** objects shortly, but you can specify vanilla standard errors simply by calling the `se` argument in `summary.fixest()` as follows.

```
summary(ols_fe, se = 'standard')
```

```
## OLS estimation, Dep. Var.: mass
## Observations: 58
## Fixed-effects: species: 31
## Standard-errors: Standard
##        Estimate Std. Error t value Pr(>|t|)
## height 0.974876   0.136463  7.1439 1.38e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 9.6906     Adj. R2: 0.99282
##                Within R2: 0.662493
```

Before continuing, let's quickly save a "tidied" data frame of the coefficients for later use. I'll use vanilla standard errors again, if only to show you that the `broom::tidy()` method for `fixest` objects also accepts an `se` argument. This basically just provides another convenient way for you to adjust standard errors for your models on the fly.

```
# coefs_fe = tidy(summary(ols_fe, se = 'standard'), conf.int = TRUE) ## same as below
coefs_fe = tidy(ols_fe, se = 'standard', conf.int = TRUE)
```

High dimensional FEs and multiway clustering As I already mentioned above, **fixest** supports (arbitrarily) high-dimensional fixed effects and (up to fourway) multiway clustering. To see this in action, let's add "homeworld" as an additional fixed effect to the model.

```
## We now have two fixed effects: species and homeworld
ols_hdfe = feols(mass ~ height |  species + homeworld, data = starwars)
ols_hdfe
```

```
## OLS estimation, Dep. Var.: mass
## Observations: 55
## Fixed-effects: species: 30,  homeworld: 38
## Standard-errors: Clustered (species)
##        Estimate Std. Error t value Pr(>|t|)
## height 0.755844   0.332888  2.2706  0.03078 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 7.4579     Adj. R2: 1.0077
##                Within R2: 0.487231
```

Easy enough, but the standard errors of the above model are automatically clustered by species, i.e. the first fixed effect variable. Let's go a step further and cluster by both "species" and "homeworld". [5] We can do this using either the `se` or

[5] I most definitely am not claiming that this is a particularly good or sensible clustering strategy, but just go with it.

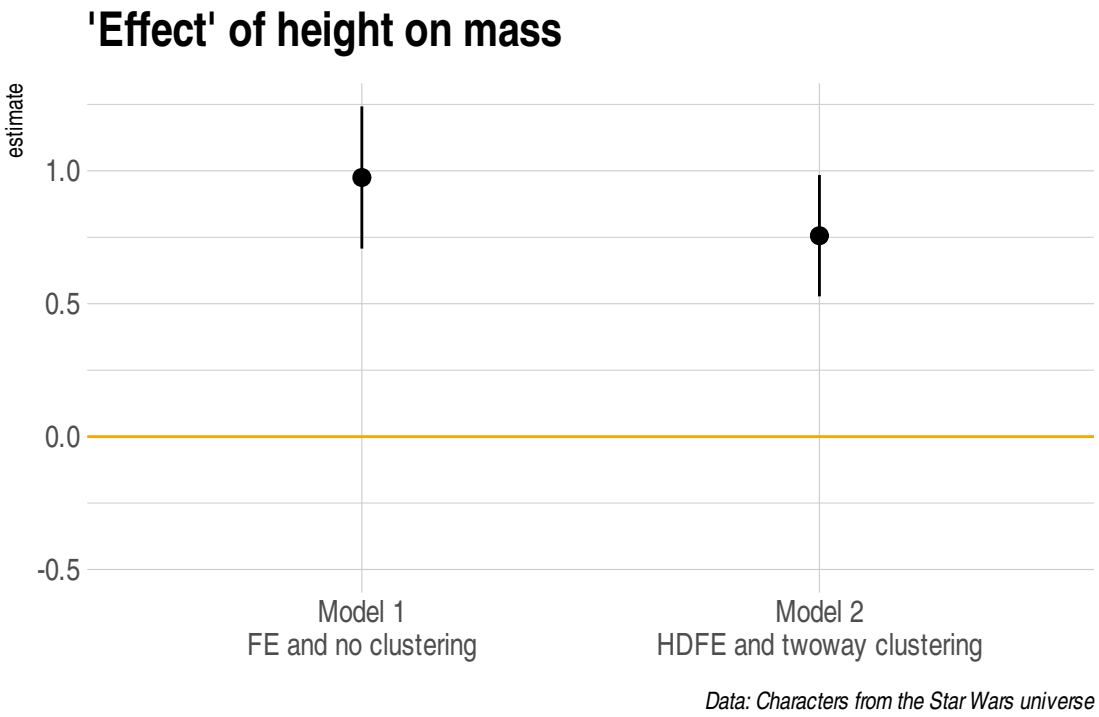
`cluster` arguments of `summary.fixest()`. I'll (re)assign the model to the same `ols_hdfe` object, but you could, of course, create a new object if you so wished.

```
## Cluster by both species and homeworld
# ols_hdfe = summary(ols_hdfe, se = 'twoway') ## Same effect as the next line
ols_hdfe = summary(ols_hdfe, cluster = c('species', 'homeworld'))
ols_hdfe
```

```
## OLS estimation, Dep. Var.: mass
## Observations: 55
## Fixed-effects: species: 30,  homeworld: 38
## Standard-errors: Two-way (species & homeworld)
##        Estimate Std. Error t value Pr(>|t|)
## height 0.755844   0.116416  6.4926 4.16e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 7.4579     Adj. R2: 1.0077
##                Within R2: 0.487231
```

Comparing our model coefficients **fixest** provides an inbuilt `coefplot()` function for plotting estimation results. This is especially useful for tracing the evolution of treatment effects over time. (Take a look here.) When it comes to comparing coefficients across models, however, I often like to do this "manually" with **ggplot2**. Consider the below example, which leverages the fact that we have saved (or can save) regression models as data frames with `broom::tidy()`. As I suggested earlier, this makes it very easy to construct our own bespoke coefficient plots.

```
# library(ggplot2) ## Already loaded

## First get tidied output of the ols_hdfe object
coefs_hdfe = tidy(ols_hdfe, conf.int = TRUE)

bind_rows(
  coefs_fe %>% mutate(reg = "Model 1\nFE and no clustering"),
  coefs_hdfe %>% mutate(reg = "Model 2\nHDFE and twoway clustering")
  ) %>%
  ggplot(aes(x=reg, y=estimate, ymin=conf.low, ymax=conf.high)) +
  geom_pointrange() +
  labs(Title = "Marginal effect of height on mass") +
  geom_hline(yintercept = 0, col = "orange") +
  ylim(-0.5, NA) + ## Added a bit more bottom space to emphasize the zero line
  labs(
    title = "'Effect' of height on mass",
    caption = "Data: Characters from the Star Wars universe"
    ) +
  theme(axis.title.x = element_blank())
```



FWIW, we'd normally expect our standard errors to blow up with clustering. Here that effect appears to be outweighed by the increased precision brought on by additional fixed effects. Still, I wouldn't put too much thought into it. Our clustering choice doesn't make much sense and I really just trying to demonstrate the package syntax.

Aside on standard errors We've now seen some of the different options that **fixest** has for specifying different error structures. In short, run your model and then use either the `se` or `cluster` arguments in `summary.fixest()` (or `broom::tidy()`) if you aren't happy with the default clustering choice. There are two additional points that I want to draw your attention to.

First, if you're coming from another statistical language or package, adjusting the standard errors after the fact rather than in the original model call may seem slightly odd. But this behaviour is actually extremely powerful, because it allows us to analyse the effect of different error structures *on-the-fly* without having to rerun the entire model again. **fixest** is already the fastest game in town, but just think about the implied timesavings for really large models.[6]

Second, reconciling standard errors across different software is a much more complicated process than you may realise. There are a number of unresolved theoretical issues to consider — especially when it comes to multiway clustering — and package maintainers have to make a number of arbitrary decisions about the best way to account for these. See here for a detailed discussion. Luckily, Laurent (the package author) has taken the time to write out a detailed vignette about how to replicate standard errors from other methods and software packages.[7]

Random and mixed effects

Fixed effects models are more common than random or mixed effects models in economics (in my experience, anyway). I'd also advocate for Bayesian hierachical models if we're going down the whole random effects path. However, it's still good to know that R has you covered for random effects models through the **plm** (link) and **nlme** (link) packages.[8] I won't go into detail , but click on those links if you would like to see some examples.

[6] To be clear, adjusting the standard errors via, say, `summary.fixest()` completes instantaneously. It's a testament to how well the package is put together and the novel estimation method that Laurent (the package author) has derived.

[7] If you want a deep dive into the theory with even more simulations, then this paper by the authors of the **sandwich** paper is another excellent resource.

[8] As I mentioned above, **plm** also handles fixed effects (and pooling) models. However, I prefer **fixest** and **lfe** for the reasons already discussed.

Instrumental variables

As you would have guessed by now, there are a number of ways to run instrumental variable (IV) regressions in R. I'll walk through three different options using the `ivreg::ivreg()`, `estimatr::iv_robust()`, and `fixest::feols()` functions, respectively. These are all going to follow a similar syntax, where the IV first-stage regression is specified in a multi-part formula (i.e. where formula parts are separated by one or more pipes, `|`). However, there are also some subtle and important differences, which is why I want to go through each of them. After that, I'll let you decide which of the three options is your favourite.

The dataset that we'll be using for this section describes cigarette demand for the 48 continental US states in 1995, and is taken from the **ivreg** package. Here's a quick a peek:

```
data("CigaretteDemand", package = "ivreg")
head(CigaretteDemand)
```

```
##        packs   rprice  rincome  salestax   cigtax  packsdiff  pricediff
## AL 101.08543 103.9182 12.91535 0.9216975 26.57481 -0.1418075 0.09010222
## AR 111.04297 115.1854 12.16907 5.4850193 36.41732 -0.1462808 0.19998082
## AZ  71.95417 130.3199 13.53964 6.2057067 42.86964 -0.3733741 0.25576681
## CA  56.85931 138.1264 16.07359 9.0363074 40.02625 -0.5682141 0.32079587
## CO  82.58292 109.8097 16.31556 0.0000000 28.87139 -0.3132622 0.22587189
## CT  79.47219 143.2287 20.96236 8.1072834 48.55643 -0.3184911 0.18546746
##    incomediff salestaxdiff  cigtaxdiff
## AL 0.18222144    0.1332853 -3.62965832
## AR 0.15055894    5.4850193  2.03070663
## AZ 0.05379983    1.4004856 14.05923036
## CA 0.02266877    3.3634447 15.86267924
## CO 0.13002974    0.0000000  0.06098283
## CT 0.18404197   -0.7062239  9.52297455
```

Now, assume that we are interested in regressing the number of cigarettes packs consumed per capita on their average price and people's real incomes. The problem is that the price is endogenous, because it is simultaneously determined by demand and supply. So we need to instrument for it using cigarette sales tax. That is, we want to run the following two-stage IV regression.

$$Price_i = \pi_0 + \pi_1 SalesTax_i + v_i \qquad \text{(First stage)}$$

$$Packs_i = \beta_0 + \beta_2 \widehat{Price}_i + \beta_1 RealIncome_i + u_i \qquad \text{(Second stage)}$$

Option 1: `ivreg::ivreg()`

I'll start with `ivreg()` from the **ivreg** package (link).[9] The `ivreg()` function supports several syntax options for specifying the IV component. I'm going to use the syntax that I find most natural, namely a formula with a three-part RHS: `y ~ ex | en | in`. Implementing our two-stage regression from above may help to illustrate.

```
# library(ivreg) ## Already loaded

## Run the IV regression. Note the three-part formula RHS.
iv =
  ivreg(
    log(packs) ~            ## LHS: Dependent variable
      log(rincome) |        ## 1st part RHS: Exogenous variable(s)
      log(rprice) |         ## 2nd part RHS: Endogenous variable(s)
      salestax,             ## 3rd part RHS: Instruments
    data = CigaretteDemand
```

[9] Some of you may wondering, but **ivreg** is a dedicated IV-focused package that splits off (and updates) functionality that used to be bundled with the **AER** package.

```
    )
summary(iv)
```

```
##
## Call:
## ivreg(formula = log(packs) ~ log(rincome) | log(rprice) | salestax,
##     data = CigaretteDemand)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.611000 -0.086072  0.009423  0.106912  0.393159
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    9.4307     1.3584   6.943 1.24e-08 ***
## log(rprice)   -1.1434     0.3595  -3.181  0.00266 **
## log(rincome)   0.2145     0.2686   0.799  0.42867
##
## Diagnostic tests:
##                  df1 df2 statistic  p-value
## Weak instruments   1  45    45.158 2.65e-08 ***
## Wu-Hausman         1  44     1.102      0.3
## Sargan             0  NA        NA       NA
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1896 on 45 degrees of freedom
## Multiple R-Squared: 0.4189,  Adjusted R-squared: 0.3931
## Wald test: 6.534 on 2 and 45 DF,  p-value: 0.003227
```

ivreg has lot of functionality bundled into it, including cool diagnostic tools and full integration with **sandwich** and co. for swapping in different standard errors on the fly. See the introductory vignette for more.

The only other thing I want to mention briefly is that you may see a number `ivreg()` tutorials using an alternative formula representation. (Remember me saying that the package allows different formula syntax, right?) Specifically, you'll probably see examples that use an older two-part RHS formula like: `y ~ ex + en | ex + in`. Note that here we are writing the `ex` variables on both sides of the `|` separator. The equivalent for our cigarette example would be as follows. Run this yourself to confirm the same output as above.

```
## Alternative two-part formula RHS (which I like less but YMMV)
iv2 =
  ivreg(
    log(packs) ~                    ## LHS: Dependent var
      log(rincome) + log(rprice) | ## 1st part RHS: Exogenous vars + endogenous vars
      log(rincome) + salestax,     ## 2nd part RHS: Exogenous vars (again) + Instruments
    data = CigaretteDemand
    )
summary(iv2)
```

This two-part syntax is closely linked to the manual implementation of IV, since it requires explicitly stating *all* of your exogenous variables (including instruments) in one slot. However, it requires duplicate typing of the exogenous variables and I personally find it less intuitive to write.[10] But different strokes for different folks.

[10] Note that we didn't specify the endogenous variable (i.e. `log(rprice)`) directly. Rather, we told R what the *exogenous* variables were. It then figured out which variables were endogenous and needed to be instrumented in the first-stage.

Option 2: `estimatr::iv_robust()`

Our second IV option comes from the **estimatr** package that we saw earlier. This will default to using HC2 robust standard errors although, as before, we could specify other options if we so wished (including clustering). Currently, the function doesn't accept the three-part RHS formula. But the two-part version works exactly the same as it did for `ivreg()`. All we need to do is change the function call to `estimatr::iv_robust()`.

```
# library(estimatr) ## Already loaded

## Run the IV regression with robust SEs. Note the two-part formula RHS.
iv_reg_robust =
  iv_robust( ## Only need to change the function call. Everything else stays the same.
    log(packs) ~
      log(rincome) + log(rprice) |
      log(rincome) + salestax,
    data = CigaretteDemand
    )
summary(iv_reg_robust, diagnostics = TRUE)
```

```
##
## Call:
## iv_robust(formula = log(packs) ~ log(rincome) + log(rprice) |
##     log(rincome) + salestax, data = CigaretteDemand)
##
## Standard error type:  HC2
##
## Coefficients:
##              Estimate Std. Error t value  Pr(>|t|) CI Lower CI Upper DF
## (Intercept)    9.4307     1.2845   7.342 3.179e-09   6.8436  12.0177 45
## log(rincome)   0.2145     0.3164   0.678 5.012e-01  -0.4227   0.8518 45
## log(rprice)   -1.1434     0.3811  -3.000 4.389e-03  -1.9110  -0.3758 45
##
## Multiple R-squared:  0.4189 ,    Adjusted R-squared:  0.3931
## F-statistic: 7.966 on 2 and 45 DF,  p-value: 0.001092
```

Option 3: `fixest::feols()`

Finally, we get back to the `fixest::feols()` function that we've already seen above. Truth be told, this is the IV option that I use most often in my own work. In part, this statement reflects the fact that I work mostly with panel data and will invariably be using **fixest** anyway. But I also happen to like its IV syntax a lot. The key thing is to specify the IV first-stage as a separate formula in the *final* slot of the model call.[11] For example, if we had `fe` fixed effects, then the model call would be `y ~ ex | fe | en ~ in`. Since we don't have any fixed effects in our current cigarette demand example, the first-stage will come directly after the exogenous variables:

```
# library(fixest) ## Already loaded

iv_feols =
  feols(
    log(packs) ~ log(rincome) | ## y ~ ex
      log(rprice) ~ salestax,   ## en ~ in (IV first-stage; must be the final slot)
    data = CigaretteDemand
    )
# summary(iv_feols, stage = 1) ## Show the 1st stage in detail
iv_feols
```

[11] This closely resembles Stata's approach to writing out the IV first-stage, where you specify the endogenous variable(s) and the instruments together in a slot.

```
## TSLS estimation, Dep. Var.: log(packs), Endo.: log(rprice), Instr.: salestax
## Second stage: Dep. Var.: log(packs)
## Observations: 48
## Standard-errors: Standard
##                 Estimate Std. Error   t value    Pr(>|t|)
## (Intercept)     9.430700   1.358400  6.942600 1.24000e-08 ***
## fit_log(rprice) -1.143400   0.359486 -3.180600 2.66200e-03 **
## log(rincome)     0.214515   0.268585  0.798687 4.28667e-01
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 0.183555   Adj. R2: 0.393109
## F-test (1st stage): stat = 45.2  , p = 2.655e-8, on 1 and 45 DoF.
##         Wu-Hausman: stat =  1.102, p = 0.299559, on 1 and 44 DoF.
```

Again, I emphasise that the IV first-stage must always come last in the `feols()` model call. Just to be pedantic — but also to demonstrate how easy **fixest**'s IV functionality extends to panel settings — here's a final `feols()` example. This time, I'll use a panel version of the same US cigarette demand data that includes entries from both 1985 and 1995. The dataset originally comes from the **AER** package, but we can download it from the web as follows. Note that I'm going to modify some variables to make it better comparable to our previous examples.

```
## Get the data
url = 'https://vincentarelbundock.github.io/Rdatasets/csv/AER/CigarettesSW.csv'
cigs_panel =
  read.csv(url, row.names = 1) %>%
  mutate(
    rprice = price/cpi,
    rincome = income/population/cpi
    )
head(cigs_panel)
```

```
##   state year   cpi population    packs    income  tax     price     taxs
## 1    AL 1985 1.076    3973000 116.4863  46014968 32.5 102.18167 33.34834
## 2    AR 1985 1.076    2327000 128.5346  26210736 37.0 101.47500 37.00000
## 3    AZ 1985 1.076    3184000 104.5226  43956936 31.0 108.57875 36.17042
## 4    CA 1985 1.076   26444000 100.3630 447102816 26.0 107.83734 32.10400
## 5    CO 1985 1.076    3209000 112.9635  49466672 31.0  94.26666 31.00000
## 6    CT 1985 1.076    3201000 109.2784  60063368 42.0 128.02499 51.48333
##      rprice  rincome
## 1  94.96438 10.76387
## 2  94.30762 10.46817
## 3 100.90962 12.83046
## 4 100.22058 15.71332
## 5  87.60842 14.32619
## 6 118.98234 17.43861
```

Let's run a panel IV now, where we'll explicitly account for year and state fixed effects.

```
iv_feols_panel =
  feols(
    log(packs) ~ log(rincome) |
      year + state |            ## Now include FEs slot
      log(rprice) ~ taxs,       ## IV first-stage still comes last
    data = cigs_panel
  )
# summary(iv_feols_panel, stage = 1) ## Show the 1st stage in detail
iv_feols_panel
```

```
## TSLS estimation, Dep. Var.: log(packs), Endo.: log(rprice), Instr.: taxs
## Second stage: Dep. Var.: log(packs)
## Observations: 96
## Fixed-effects: year: 2,  state: 48
## Standard-errors: Clustered (year)
##                  Estimate Std. Error       t value Pr(>|t|)
## fit_log(rprice) -1.279300   2.29e-15 -5.578376e+14 1.14e-15 ***
## log(rincome)     0.443422   1.45e-14  3.056237e+13 2.08e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 0.044789     Adj. R2: 0.92791
##                  Within R2: 0.533965
## F-test (1st stage): stat = 108.6   , p = 1.407e-13, on 1 and 45 DoF.
##         Wu-Hausman: stat =   6.0215, p = 0.018161 , on 1 and 44 DoF.
```

Good news, our coefficients are around the same magnitude. But the increased precision of the panel model has yielded gains in statistical significance.

Other models

Generalised linear models (logit, etc.)

To run a generalised linear model (GLM), we use the in-built `glm()` function and simply assign an appropriate family (which describes the error distribution and corresponding link function). For example, here's a simple logit model.

```
glm_logit = glm(am ~ cyl + hp + wt, data = mtcars, family = binomial)
tidy(glm_logit, 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)  19.7       8.12       2.43   0.0152   8.56      44.3
## 2 cyl           0.488     1.07       0.455  0.649   -1.53       3.12
## 3 hp            0.0326    0.0189     1.73   0.0840   0.00332    0.0884
## 4 wt           -9.15      4.15      -2.20   0.0276 -21.4       -3.48
```

Remember that the estimates above simply reflect the naive coefficient values, which enter multiplicatively via the link function. We'll get a dedicated section on extracting marginal effects from non-linear models in a moment. But I do want to quickly flag the **mfx** package (link), which provides convenient aliases for obtaining marginal effects from a variety of GLMs. For example,

```
# library(mfx) ## Already loaded
## Be careful: mfx loads the MASS package, which produces a namespace conflict
## with dplyr for select(). You probably want to be explicit about which one you
## want, e.g. `select = dplyr::select`

## Get marginal effects for the above logit model
glm_logitmfx = logitmfx(glm_logit, atmean = TRUE, data = mtcars)
## Could also plug in the original formula directly
# glm_logitmfx = logitmfx(am ~ cyl + hp + wt, atmean = TRUE, data = mtcars)
tidy(glm_logitmfx, conf.int = TRUE)
```

```
## # A tibble: 3 x 8
##   term  atmean estimate std.error statistic p.value conf.low conf.high
##   <chr> <lgl>     <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
## 1 cyl   TRUE    0.0538    0.113       0.475   0.635 -0.178     0.286
## 2 hp    TRUE    0.00359   0.00290     1.24    0.216 -0.00236   0.00954
```

```
## 3 wt    TRUE   -1.01      0.668      -1.51    0.131 -2.38      0.359
```

Bayesian regression

We could spend a whole course on Bayesian models. The very, very short version is that R offers outstanding support for Bayesian models and data analysis. You will find convenient interfaces to all of the major MCMC and Bayesian software engines: Stan, JAGS, TensorFlow (via Greta), etc. Here follows a *super* simple example using the **rstanarm** package (link). Note that we did not install this package with the others above, as it can take fairly long and involve some minor troubleshooting.[12]

```
# install.packages("rstanarm") ## Run this first if you want to try yourself
library(rstanarm)

bayes_reg =
  stan_glm(
    mass ~ gender * height,
    data = humans,
    family = gaussian(), prior = cauchy(), prior_intercept = cauchy()
    )
```

```
summary(bayes_reg)
```

```
##
## Model Info:
##  function:     stan_glm
##  family:       gaussian [identity]
##  formula:      mass ~ gender * height
##  algorithm:    sampling
##  sample:       4000 (posterior sample size)
##  priors:       see help('prior_summary')
##  observations: 22
##  predictors:   4
##
## Estimates:
##                           mean   sd     10%    50%    90%
## (Intercept)             -67.1   74.9 -161.8  -67.6   27.5
## gendermasculine          -0.2    9.2   -7.1    0.0    6.9
## height                    0.8    0.5    0.2    0.8    1.4
## gendermasculine:height    0.1    0.1    0.0    0.1    0.2
## sigma                    15.9    2.7   12.8   15.6   19.4
##
## Fit Diagnostics:
##            mean   sd   10%   50%   90%
## mean_PPD 82.4    4.8 76.3  82.5  88.3
##
## The mean_ppd is the sample average posterior predictive distribution of the outcome variable (for details see help(
##
## MCMC diagnostics
##                         mcse Rhat n_eff
## (Intercept)             1.7  1.0  1942
## gendermasculine         0.3  1.0  1301
## height                  0.0  1.0  1873
## gendermasculine:height  0.0  1.0  1435
```

[12] FWIW, on my machine (running Arch Linux) I had to install `stan` (and thus `rstanarm`) by running R through the shell. For some reason, RStudio kept closing midway through the installation process.

```
## sigma                 0.1  1.0  2628
## mean_PPD              0.1  1.0  2932
## log-posterior         0.0  1.0  1531
##
## For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effective sample size, and Rhat
```

Even more models

Of course, there are simply too many other models and other estimation procedures to cover in this lecture. A lot of these other models that you might be thinking of come bundled with the base R installation. But just to highlight a few, mostly new packages that I like a lot for specific estimation procedures:

- Difference-in-differences (with variable timing, etc.): **did** (link) and **DRDID** (link)
- Synthetic control: **tidysynth** (link), **gsynth** (link) and **scul** (link)
- Count data (hurdle models, etc.): **pscl** (link)
- Lasso: **biglasso** (link)
- Causal forests: **grf** (link)
- etc.

Finally, just a reminder to take a look at the Further Resources links at the bottom of this document to get a sense of where to go for full-length econometrics courses and textbooks.

Marginal effects

Calculating marginal effects in a linear regression model like OLS is perfectly straightforward… just look at the coefficient values. But that quickly goes out the window when you have interaction terms or non-linear models like probit, logit, etc. Luckily, there are various ways to obtain these from R models. For example, we already saw the **mfx** package above for obtaining marginal effects from GLM models. I want to briefly focus on two of my favourite methods for obtaining marginal effects across different model classes: 1) The **margins** package and 2) a shortcut that works particularly well for models with interaction terms.

The margins package

The **margins** package (link), which is modeled on its namesake in Stata, is great for obtaining marginal effects across an entire range of models.[13] You can read more in the package vignette, but here's a very simple example to illustrate.

Consider our interaction effects regression from earlier, where we were interested in how people's mass varied by height and gender. To get the average marginal effect (AME) of these dependent variables, we can just use the `margins::margins()` function.

```
# library(margins) ## Already loaded

ols_ie_marg = margins(ols_ie)
```

Like a normal regression object, we can get a nice print-out display of the above object by summarising or tidying it.

```
# summary(ols_ie_marg) ## Same effect
tidy(ols_ie_marg, 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 gendermasculine   13.5      26.8       0.505  0.613  -38.9        66.0
## 2 height             0.874     0.420     2.08   0.0376   0.0503      1.70
```

If we want to compare marginal effects at specific values — e.g. how the AME of height on mass differs across genders — then that's easily done too.

[13] I do, however, want to flag that it does not yet support **fixest** (or **lfe**) models. But there are workarounds in the meantime.

```
ols_ie %>%
  margins(
    variables = "height", ## The main variable we're interested in
    at = list(gender = c("masculine", "feminine")) ## How the main variable is modulated by at specific values o
    ) %>%
  tidy(conf.int = TRUE) ## Tidy it (optional)
```

```
## # A tibble: 2 x 9
##   term   at.variable at.value  estimate std.error statistic p.value conf.low
##   <chr>  <chr>       <fct>        <dbl>     <dbl>     <dbl>   <dbl>    <dbl>
## 1 height gender      masculine    0.896     0.443     2.02   0.0431   0.0279
## 2 height gender      feminine     0.733     1.27      0.576  0.565   -1.76
## # ... with 1 more variable: conf.high <dbl>
```

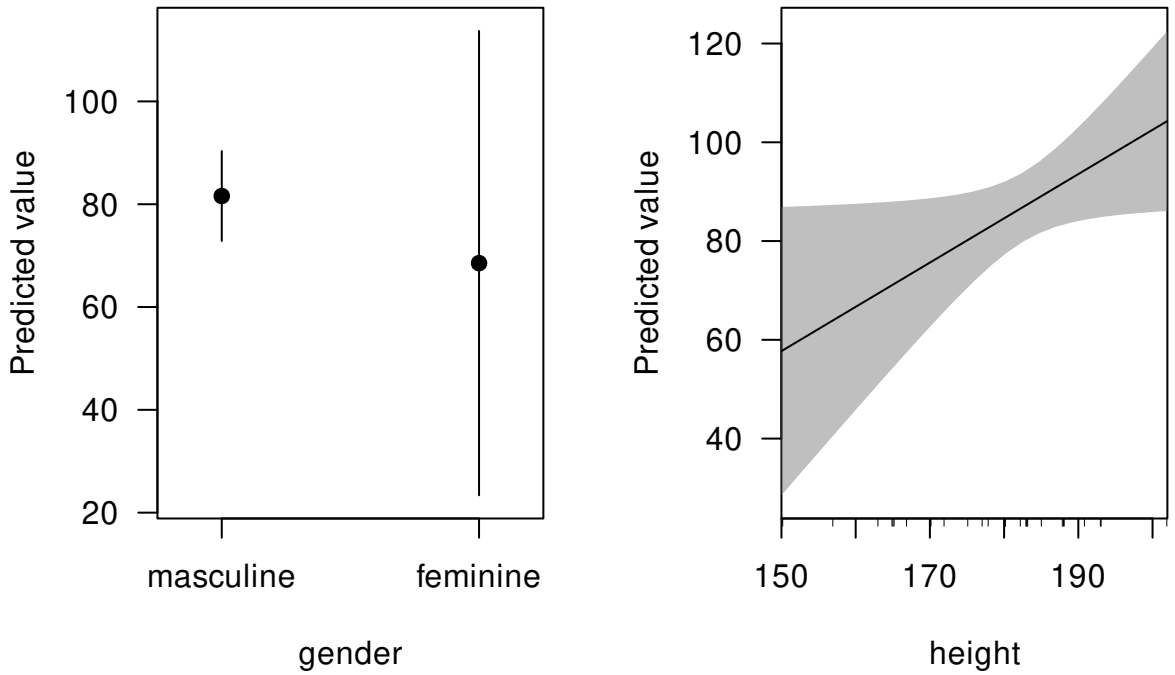
If you're the type of person who prefers visualizations (like me), then you should consider `margins::cplot()`, which is the package's in-built method for constructing *conditional* effect plots.

```
cplot(ols_ie, x = "gender", dx = "height", what = "effect",
      data = humans)
```

In this case,it doesn't make much sense to read a lot into the larger standard errors on the female group; that's being driven by a very small sub-sample size.

Finally, you can also use `cplot()` to plot the predicted values of your outcome variable (here: "mass"), conditional on one of your dependent variables. For example:

```
par(mfrow=c(1, 2)) ## Just to plot these next two (base) figures side-by-side
cplot(ols_ie, x = "gender", what = "prediction", data = humans)
cplot(ols_ie, x = "height", what = "prediction", data = humans)
```




```
par(mfrow=c(1, 1)) ## Reset plot defaults
```

Note that `cplot()` uses the base R plotting method. If you'd prefer **ggplot2** equivalents, take a look at the **marginsplot** package (link).

Finally, I also want to draw your attention to the **emmeans** package (link), which provides very similar functionality to **margins**. I'm not as familiar with it myself, but I know that it has many fans.

Special case: / shortcut for interaction terms

I'll keep this one brief, but I wanted to mention one of my favourite R shortcuts: Obtaining the full marginal effects for interaction terms by using the / expansion operator. I've tweeted about this and even wrote an whole blog post about it too (which you should totally read). But the very short version is that you can switch out the normal `f1 * x2` interaction terms syntax for `f1 / x2` and it automatically returns the full marginal effects. (The formal way to describe it is that the model variables have been "nested".)

Here's a super simple example, using the same interaction effects model from before.

```
# ols_ie = lm(mass ~ gender * height, data = humans) ## Original model
ols_ie_marg2 = lm(mass ~ gender / height, data = humans)
tidy(ols_ie_marg2, 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.      204.      -0.299   0.768  -4.90e+2    368.
## 2 gendermasculine         -15.7     220.      -0.0716  0.944  -4.77e+2    446.
## 3 genderfeminine:height     0.733     1.27      0.576   0.572  -1.94e+0      3.41
## 4 gendermasculine:height    0.896     0.443     2.02    0.0582 -3.46e-2      1.83
```

Note that the marginal effects on the two gender × height interactions (i.e. 0.733 and 0.896) are the same as we got with the `margins::margins()` function above.

Where this approach really shines is when you are estimating interaction terms in large models. The **margins** package relies on a numerical delta method which can be very computationally intensive, whereas using / adds no additional

overhead beyond calculating the model itself. Still, that's about as much as say it here. Read my aforementioned blog post if you'd like to learn more.

Presentation

Tables

Regression tables There are loads of different options here.[14] These days, however, I find myself using the **modelsummary** package (link) for creating and exporting regression tables. It is extremely flexible and handles all manner of models and output formats. **modelsummary** also supports automated coefficient plots and data summary tables, which I'll get back to in a moment. The documentation is outstanding and you should read it, but here is a bare-boned example just to demonstrate.

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

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

	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)	(0.116)
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		
FE: homeworld				X
FE: species			X	X
Std. errors			Clustered (species)	Two-way (species & homeworld)

One nice thing about **modelsummary** is that it plays very well with R Markdown and 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 if you aren't using R Markdown and want to export a table for later use. Finally, you can even specify special table formats like *threepartable* for LaTeX and, provided that you have called the necessary packages in your preamble, it will render correctly (see example here).

Summary tables A variety of summary tables — balance, correlation, etc. — can be produced by the companion set of `modelsummary::datasummary*()` functions. Again, you should read the documentation to see all of the options. But here's an example of a very simple balance table using a subset of our "humans" data frame.

[14] FWIW, the **fixest** package also provides its own dedicated function for exporting regression models, namely `etable()`. This function is highly optimised and is capable of producing great looking tables with minimal effort, but is limited to `fixest` model objects only. More here and here.

```
datasummary_balance(~ gender,
                    data = select(humans, gender, height, mass, birth_year, eye_color))
```

		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		

Another package that I like a lot in this regard is **vtable** (link). Not only can it be used to construct descriptive labels like you'd find in Stata's "Variables" pane, but it is also very good at producing the type of "out of the box" summary tables that economists like. For example, here's the equivalent version of the above balance table.

```
# library(vtable) ## Already loaded

## An additional argument just for formatting across different output types of
## this .Rmd document
otype = ifelse(knitr::is_latex_output(), 'return', 'kable')

## st() is an alias for sumtable()
st(select(humans, gender, height, mass, birth_year, eye_color),
   group = 'gender',
   out = otype)
```

```
##         Variable        N   Mean     SD         N    Mean     SD
## 1         gender feminine               masculine
## 2         height        8 160.25  6.985        23 182.348  8.189
## 3           mass        3 56.333 16.289        19  86.958 16.549
## 4     birth_year        5   46.4  18.77        20  55.165  26.02
## 5      eye_color        9                      26
## 6        ... blue        3  33.3%                9   34.6%
## 7   ... blue-gray        0     0%                1    3.8%
## 8       ... brown        5  55.6%               12   46.2%
## 9        ... dark        0     0%                1    3.8%
## 10      ... hazel        1  11.1%                1    3.8%
## 11     ... yellow        0     0%                2    7.7%
```

In case you were wondering, `vtable::st()` does a clever job of automatically picking defaults and dropping "unreasonable" variables (e.g. list variables or factors with too many levels). Here's what we get if we just ask it to produce a summary table of the main "starwars" data frame.

```
st(starwars, out = otype)
```

```
         Variable  N    Mean Std. Dev. Min Pctl. 25 Pctl. 75  Max
```

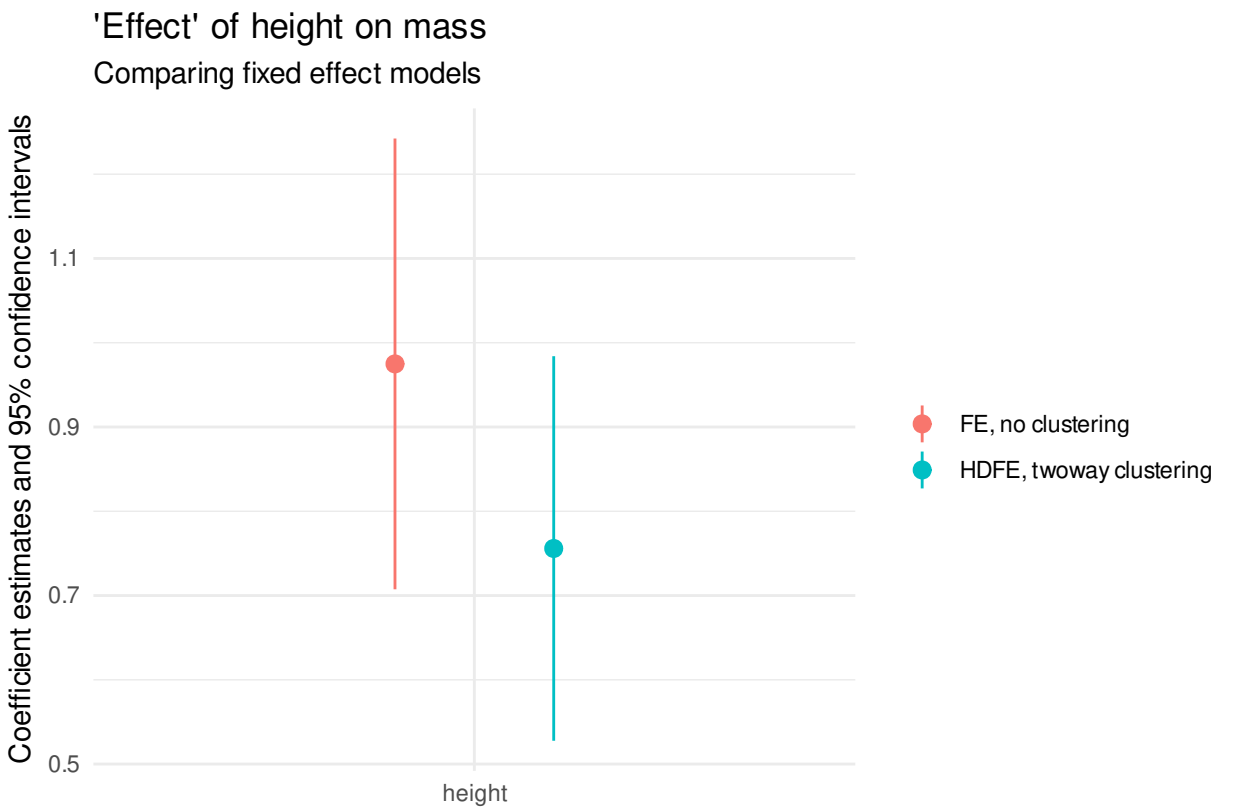
1 height 81 174.358 34.77 66 167 191 264 2 mass 59 97.312 169.457 15 55.6 84.5 1358 3 birth_year 43 87.565 154.691 8 35 72 896 4 sex 83

5 … female 16 19.3%
6 … hermaphroditic 1 1.2%
7 … male 60 72.3%
8 … none 6 7.2%
9 gender 83
10 … feminine 17 20.5%
11 … masculine 66 79.5%

Figures

Coefficient plots We've already worked through an example of how to extract and compare model coefficients here. I use this "manual" approach to visualizing coefficient estimates all the time. However, our focus on **modelsummary** in the preceding section provides a nice segue to another one of the package's features: `modelplot()`. Consider the following, which shows both the degree to which `modelplot()` automates everything and the fact that it readily accepts regular **ggplot2** syntax.

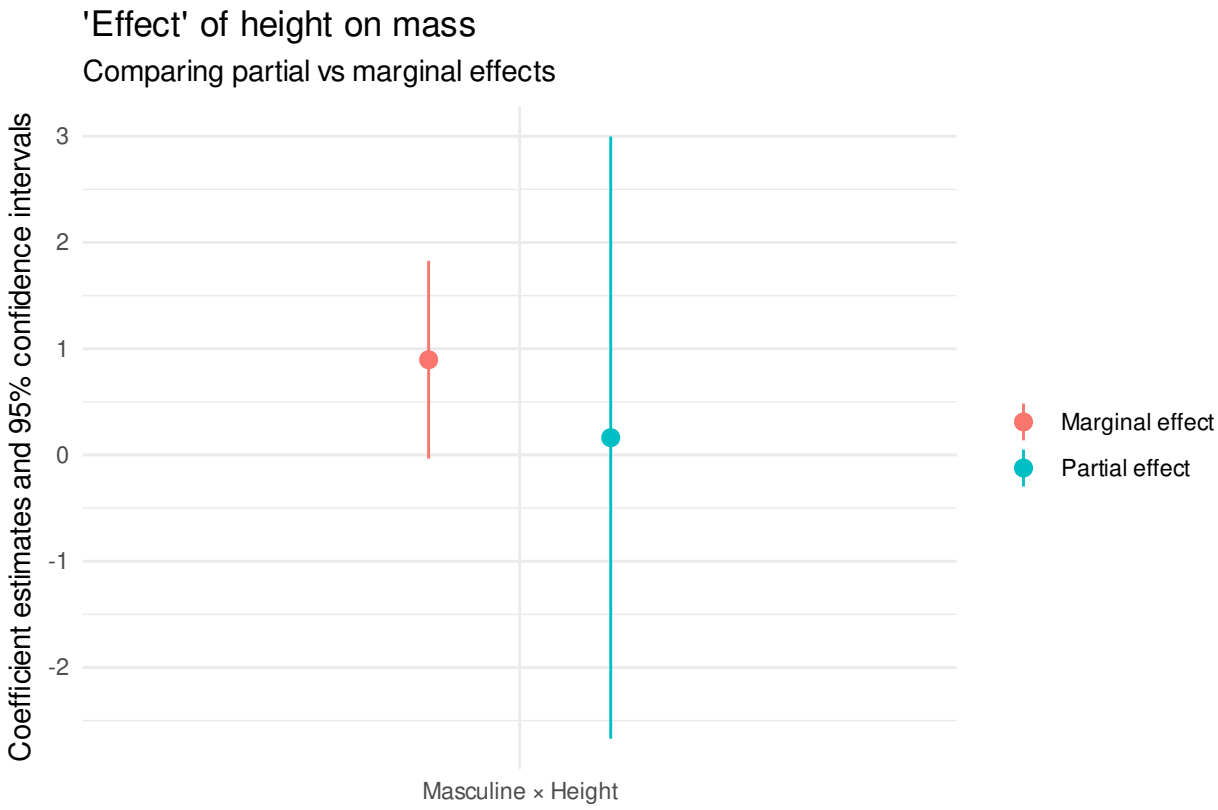
```
# library(modelsummary) ## Already loaded
mods = list('FE, no clustering' = summary(ols_fe, se = 'standard'),  # Don't cluster SEs
            'HDFE, twoway clustering' = ols_hdfe)

modelplot(mods) +
  ## You can further modify with normal ggplot2 commands ...
  coord_flip() +
  labs(
    title = "'Effect' of height on mass",
    subtitle = "Comparing fixed effect models"
    )
```



Or, here's another example where we compare the (partial) Masculine × Height coefficient from our earlier interaction

model, with the (full) marginal effect that we obtained later on.

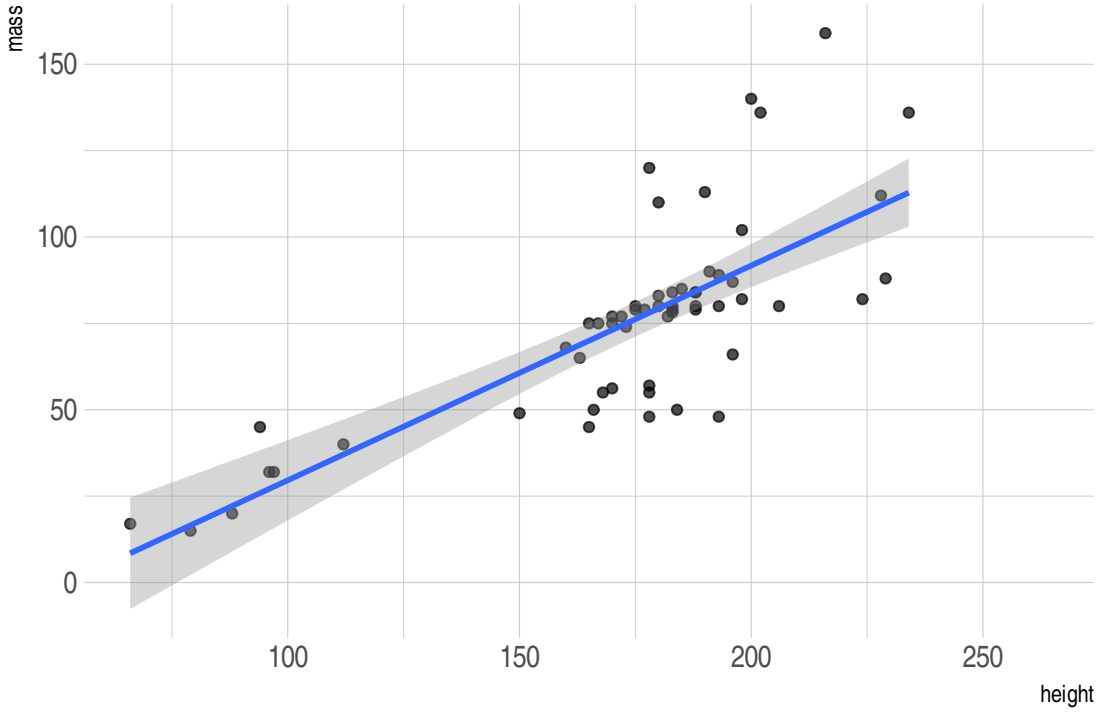
```
ie_mods = list('Partial effect' = ols_ie, 'Marginal effect' = ols_ie_marg2)

modelplot(ie_mods, coef_map = c("gendermasculine:height" = "Masculine × Height")) +
  coord_flip() +
  labs(
    title = "'Effect' of height on mass",
    subtitle = "Comparing partial vs marginal effects"
    )
```



Prediction and model validation The easiest way to visually inspect model performance (i.e. validation and prediction) is with **ggplot2**. In particular, you should already be familiar with `geom_smooth()` from our earlier lectures, which allows you to feed a model type directly in the plot call. For instance, using our `starwars2` data frame that excludes that slimy outlier, Jabba the Hutt:

```
ggplot(starwars2, aes(x = height, y = mass)) +
    geom_point(alpha = 0.7) +
    geom_smooth(method = "lm") ## See ?geom_smooth for other methods/options
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Now, I should say that `geom_smooth()` isn't particularly helpful when you've already constructed a (potentially complicated) model outside of the plot call. Similarly, it's not useful when you want to use a model for making predictions on a *new* dataset (e.g. evaluating out-of-sample fit).

The good news is that the generic `predict()` function in base R has you covered. For example, let's say that we want to re-estimate our simple bivariate regression of mass on height from earlier.[15] This time, however, we'll estimate our model on a training dataset that only consists of the first 30 characters ranked by height. Here's how you would do it.

```
## Estimate a model on a training sample of the data (shortest 30 characters)
ols1_train = lm(mass ~ height, data = starwars %>% filter(rank(height) <= 30))

## Use our model to predict the mass for all starwars characters (excl. Jabba).
## Note that I'm including a 95% prediction interval. See ?predict.lm for other
## intervals and options.
predict(ols1_train, newdata = starwars2, interval = "prediction") %>%
  head(5) ## Just print the first few rows
```

```
##        fit       lwr       upr
## 1 68.00019 46.307267  89.69311
## 2 65.55178 43.966301  87.13725
## 3 30.78434  8.791601  52.77708
## 4 82.69065 60.001764 105.37954
## 5 57.22718 35.874679  78.57968
```

Hopefully, you can already see how the above data frame could easily be combined with the original data in a **ggplot2** call. (I encourage you to try it yourself before continuing.) At the same time, it is perhaps a minor annoyance to have to combine the original and predicted datasets before plotting. If this describes your thinking, then there's even more good news because the **broom** package does more than tidy statistical models. It also ships the `augment()` function, which provides a convenient way to append model predictions to your dataset. Note that `augment()` accepts exactly the same arguments as `predict()`, although the appended variable names are slightly different.[16]

[15] I'm sticking to a bivariate regression model for these examples because we're going to be evaluating a 2D plot below.

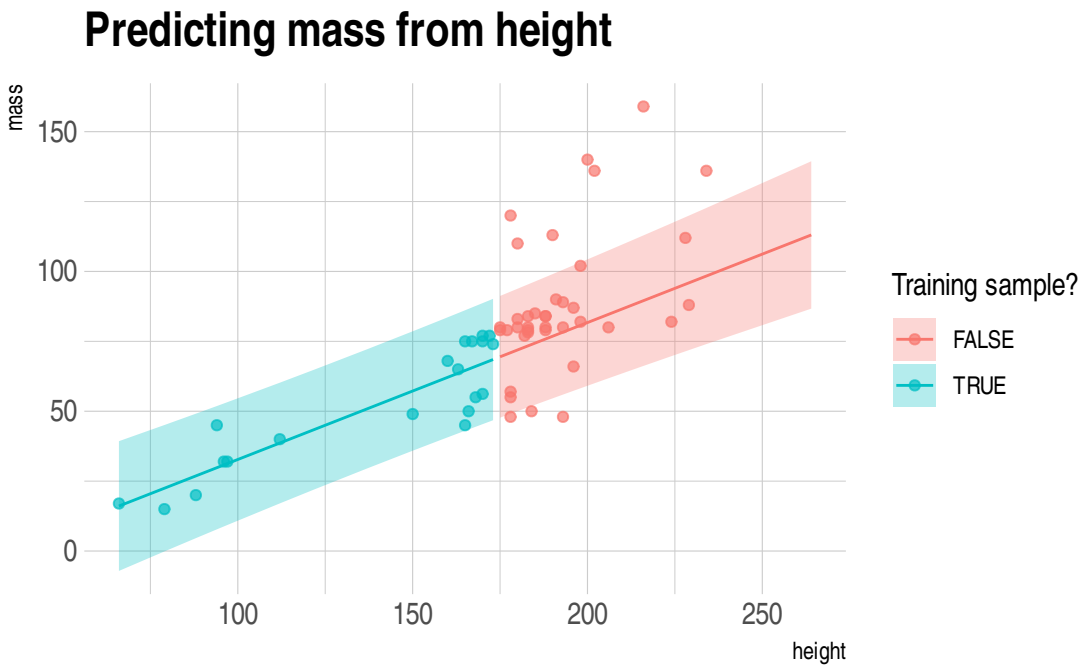
[16] Specifically, we' re adding ".fitted", ".resid", ".lower", and ".upper" columns to our data frame. The convention adopted by `augment()` is to always prefix added variables with a "." to avoid overwriting existing variables.

```
## Alternative to predict(): Use augment() to add .fitted and .resid, as well as
## .conf.low and .conf.high prediction interval variables to the data.
starwars2 = augment(ols1_train, newdata = starwars2, interval = "prediction")

## Show the new variables (all have a "." prefix)
starwars2 %>% select(contains("."), everything()) %>% head()
```

```
## # A tibble: 6 x 18
##   .fitted .lower .upper .resid name  height  mass hair_color skin_color
##     <dbl>  <dbl>  <dbl>  <dbl> <chr>  <int> <dbl> <chr>      <chr>
## 1    68.0  46.3    89.7   9.00 Luke~    172    77 blond      fair
## 2    65.6  44.0    87.1   9.45 C-3PO    167    75 <NA>       gold
## 3    30.8   8.79   52.8   1.22 R2-D2     96    32 <NA>       white, bl~
## 4    82.7  60.0   105.   53.3  Dart~    202   136 none       white
## 5    57.2  35.9    78.6  -8.23 Leia~    150    49 brown      light
## 6    70.9  49.1    92.8  49.1  Owen~    178   120 brown, gr~ light
## # ... with 9 more variables: eye_color <chr>, birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

We can now see how well our model — again, only estimated on the shortest 30 characters — performs against all of the data.

```
starwars2 %>%
  ggplot(aes(x = height, y = mass, col = rank(height)<=30, fill = rank(height)<=30)) +
  geom_point(alpha = 0.7) +
  geom_line(aes(y = .fitted)) +
  geom_ribbon(aes(ymin = .lower, ymax = .upper), alpha = 0.3, col = NA) +
  scale_color_discrete(name = "Training sample?", aesthetics = c("colour", "fill")) +
  labs(
    title = "Predicting mass from height",
    caption = "Line of best fit, with shaded regions denoting 95% prediction interval."
    )
```



Line of best fit, with shaded regions denoting 95% prediction interval.

Further resources

- Ed Rubin has outstanding teaching notes for econometrics with R on his website. This includes both undergrad- and graduate-level courses. Seriously, check them out.
- Several introductory texts are freely available, including *Introduction to Econometrics with R* (Christoph Hanck *et al.*), *Using R for Introductory Econometrics* (Florian Heiss), and *Modern Dive* (Chester Ismay and Albert Kim).
- Tyler Ransom has a nice cheat sheet for common regression tasks and specifications.
- Itamar Caspi has written a neat unofficial appendix to this lecture, *recipes for Dummies*. The title might be a little inscrutable if you haven't heard of the `recipes` package before, but basically it handles "tidy" data preprocessing, which is an especially important topic for machine learning methods. We'll get to that later in course, but check out Itamar's post for a good introduction.
- I promised to provide some links to time series analysis. The good news is that R's support for time series is very, very good. The Time Series Analysis task view on CRAN offers an excellent overview of available packages and their functionality.
- Lastly, for more on visualizing regression output, I highly encourage you to look over Chapter 6 of Kieran Healy's *Data Visualization: A Practical Guide*. Not only will learn how to produce beautiful and effective model visualizations, but you'll also pick up a variety of technical tips.