Big Data and Economics

Regression analysis in R

Kyle Coombs

Bates College | DCS/ECON 368

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, we won't cover 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**, **rstanarm**
- Already used: **tidyverse**, **hrbrthemes**, **listviewer**

A convenient way to install (if necessary) and load everything is by running the below code chunk.

```
## 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, modelsummary,rstanarm)
## Make sure we have at least version 0.6.0 of ivreg
if (numeric_version(packageVersion("ivreg")) < numeric_version("0.6.0")) install.packages("ivreg")

## My preferred ggplot2 plotting theme (optional)
theme_set(theme_minimal())
```

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 Sk~    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 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
##  5 Leia Or~    150    49 brown      light      brown           19   fema~ femin~
##  6 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
##  7 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
##  8 R5-D4        97    32 <NA>       white, red red             NA   none  mascu~
##  9 Biggs D~    183    84 black      light      brown           24   male  mascu~
## 10 Obi-Wan~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
## # i 77 more rows
## # i 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.

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

```
ols1 = lm(mass ~ height, data = starwars)
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.

```r
# 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.

```r
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:

```r
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.

```r
# 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.

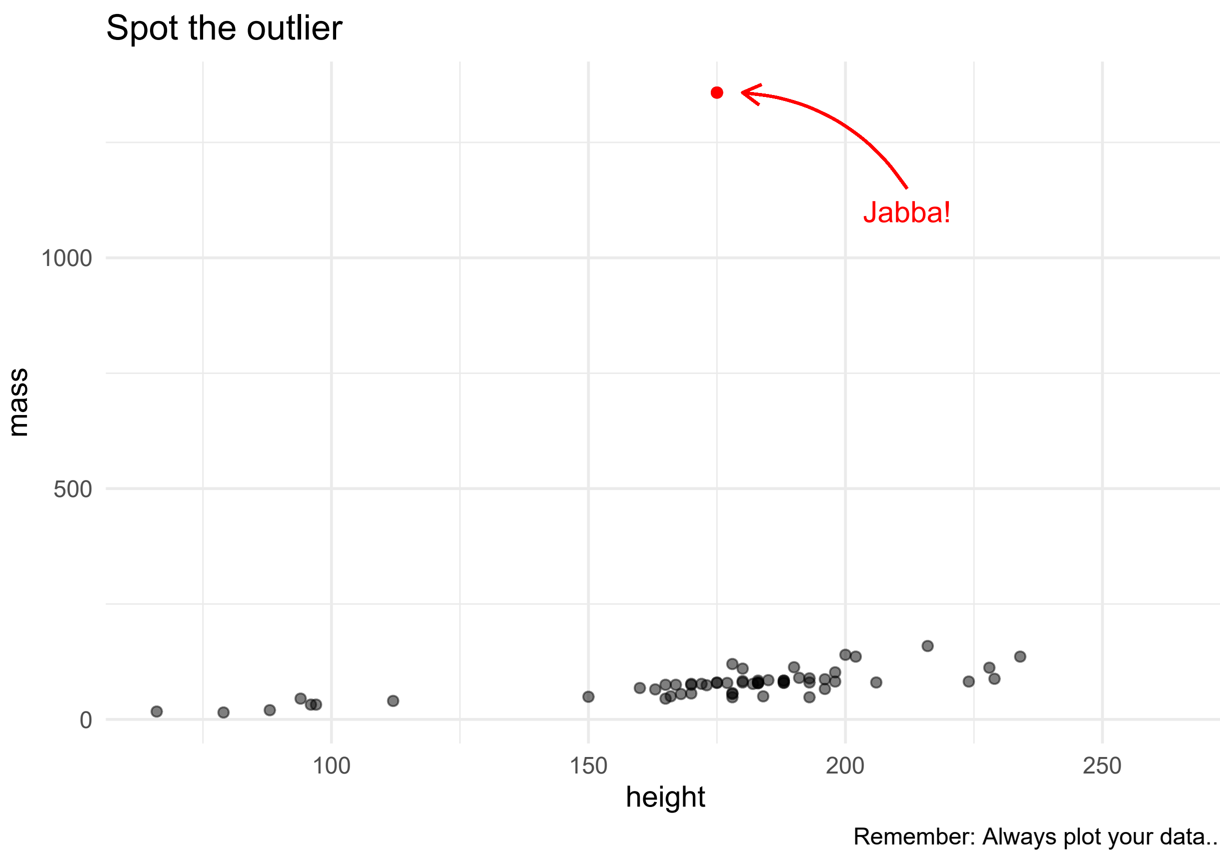
```
glance(ols1)
```

```
## # 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.
## # i 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…



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 %>%
```

```r
  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.

```r
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, here I'll demonstrate using 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 On 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 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
```

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

```
## (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, bootstrapped, etc. — but here's how it would work for the present case:

```r
# 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.

```r
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:

```r
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.

```r
humans = 
  starwars %>% 
  filter(species=="Human") %>%
  select(where(Negate(is.list))) ## Drop list columns (optional)
humans
```

```
## # A tibble: 35 x 11
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke Sk~    172    77 blond      fair       blue            19   male  mascu~
##  2 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
##  3 Leia Or~    150    49 brown      light      brown           19   fema~ femin~
##  4 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
##  5 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
##  6 Biggs D~    183    84 black      light      brown           24   male  mascu~
##  7 Obi-Wan~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
##  8 Anakin ~    188    84 blond      fair       blue            41.9 male  mascu~
##  9 Wilhuff~    180    NA auburn, g~ fair       blue            64   male  mascu~
## 10 Han Solo    180    80 brown      fair       brown           29   male  mascu~
## # i 25 more rows
## # i 2 more variables: homeworld <chr>, species <chr>
```

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 where we explicitly tell R that "gender" is a factor. 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.

```r
summary(lm(mass ~ height + as.factor(gender), data = humans))
```

```
##
## Call:
## lm(formula = mass ~ height + as.factor(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 *
## as.factor(gender)masculine  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
```

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 Works™. 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.

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

```r
## 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, if 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,

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

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

```r
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
```

[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.

```
## -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
```

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).[4] 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
    )
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.

[4] 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.

```r
## 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.[5] But different strokes for different folks.

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()`.

```r
# 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
```

[5] 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.

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)
summary(glm_logit)
```

```
##
## Call:
## glm(formula = am ~ cyl + hp + wt, family = binomial, data = mtcars)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 19.70288    8.11637   2.428   0.0152 *
## cyl          0.48760    1.07162   0.455   0.6491
## hp           0.03259    0.01886   1.728   0.0840 .
## wt          -9.14947    4.15332  -2.203   0.0276 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 43.2297  on 31  degrees of freedom
## Residual deviance:  9.8415  on 28  degrees of freedom
## AIC: 17.841
##
## Number of Fisher Scoring iterations: 8
```

Alternatively, you may recall me saying earlier that **fixest** supports nonlinear models. So you could (in this case, without fixed-effects) also estimate:

```
feglm(am ~ cyl + hp + wt, data = mtcars, family = binomial)
```

```
## GLM estimation, family = binomial, Dep. Var.: am
## Observations: 32
## Standard-errors: IID
##              Estimate Std. Error   t value Pr(>|t|)
## (Intercept) 19.702883   8.540119  2.307097 0.021049 *
## cyl          0.487598   1.127568  0.432433 0.665427
## hp           0.032592   0.019846  1.642249 0.100538
## wt          -9.149471   4.370163 -2.093623 0.036294 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## Log-Likelihood: -4.92075   Adj. Pseudo R2: 0.633551
##           BIC: 23.7          Squared Cor.: 0.803395
```

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`
```

```r
## Get marginal effects for the above logit model
# logitmfx(am ~ cyl + hp + wt, atmean = TRUE, data = mtcars) ## Can also estimate directly
logitmfx(glm_logit, atmean = TRUE, data = mtcars)
```

```
## Call:
## logitmfx(formula = glm_logit, data = mtcars, atmean = TRUE)
##
## Marginal Effects:
##          dF/dx  Std. Err.       z  P>|z|
## cyl  0.0537504  0.1132652  0.4746 0.6351
## hp   0.0035927  0.0029037  1.2373 0.2160
## wt  -1.0085932  0.6676628 -1.5106 0.1309
```

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.[6] 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.

```r
# 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.

```r
# summary(ols_ie_marg) ## Same effect
tidy(ols_ie_marg, conf.int = TRUE)
```

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

```
## # 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
```

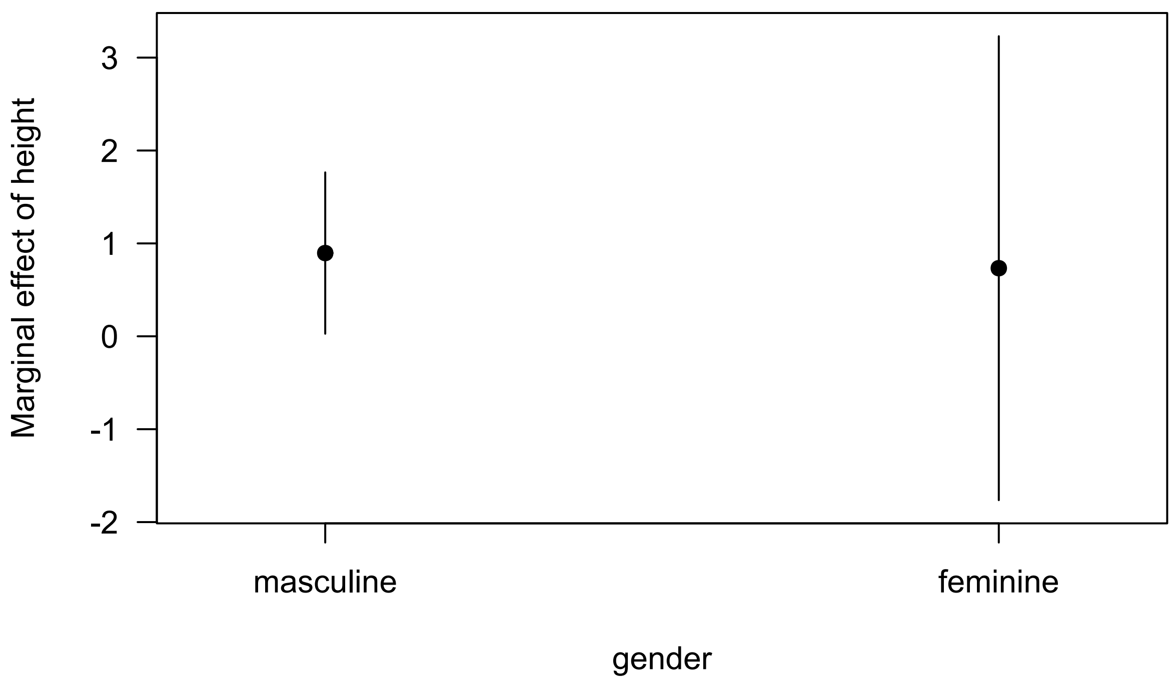
If we want to compare marginal effects at specific values — e.g. how the AME of height on mass differs across genders — then that's easily done too.

```r
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
    ) %>%
  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   0.0278
## 2 height gender      feminine     0.733     1.27      0.576  0.565   -1.76
## # i 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.

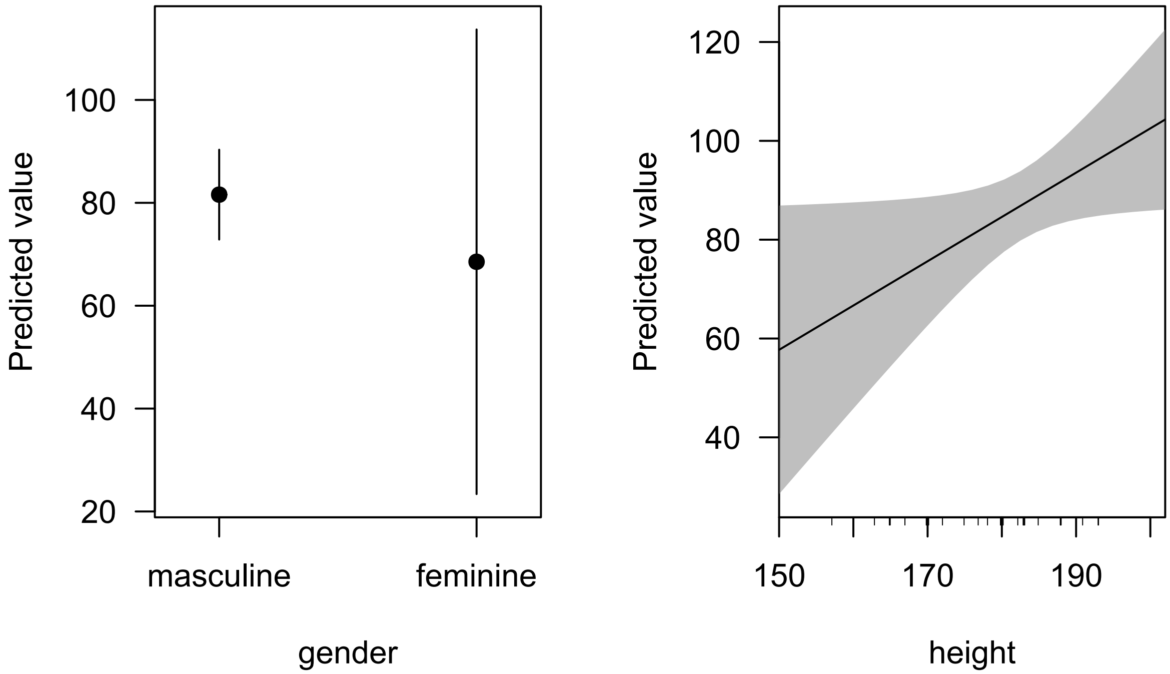
```r
cplot(ols_ie, x = "gender", dx = "height", what = "effect",
      data = humans)
```



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:

```r
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)
```




```r
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.

```r
# 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)             -61.0     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
```

	(1)	(2)
(Intercept)	−13.810	−61.000
	(111.155)	(204.057)
height	0.639	0.733
	(0.626)	(1.274)
gendermasculine		−15.722
		(219.544)
gendermasculine × height		0.163
		(1.349)
Num.Obs.	59	22
R2	0.018	0.444
R2 Adj.	0.001	0.352
AIC	777.0	188.9
BIC	783.2	194.4
Log.Lik.	−385.503	−89.456
F	1.040	
RMSE	166.50	14.11

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. We've already seen the excellent `etable()` function from **fixest** above.[7] However, my own personal favourite tool or creating and exporting regression tables is the **modelsummary** package (link). 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))
```

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.

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

[7] Note that `etable()` is limited to fixest models only.

		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	Pct.	N	Pct.		
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')

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

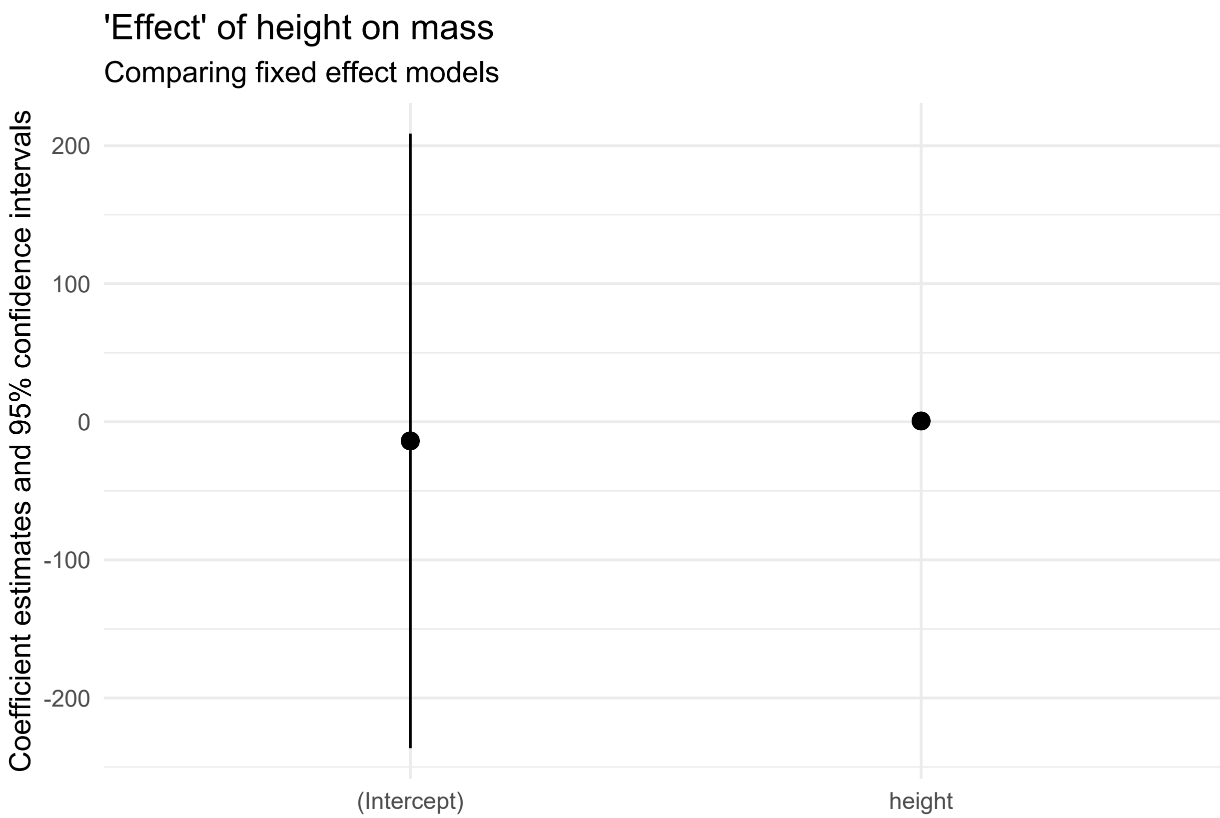
```
##          Variable        N Mean SD          N Mean  SD
## 1          gender feminine          masculine
## 2          height        8  160  7         23  182 8.2
## 3            mass        3   56 16         19   87  17
## 4      birth_year        5   46 19         20   55  26
## 5       eye_color        9                 26
## 6        ... blue        3  33%             9  35%
## 7   ... blue-gray        0   0%             1   4%
## 8       ... brown        5  56%            12  46%
## 9        ... dark        0   0%             1   4%
## 10      ... hazel        1  11%             1   4%
## 11     ... yellow        0   0%             2   8%
```

Lastly, Stata users in particular might like the `qsu()` and `descr()` functions from the lightning-fast **collapse** package (link).

Figures

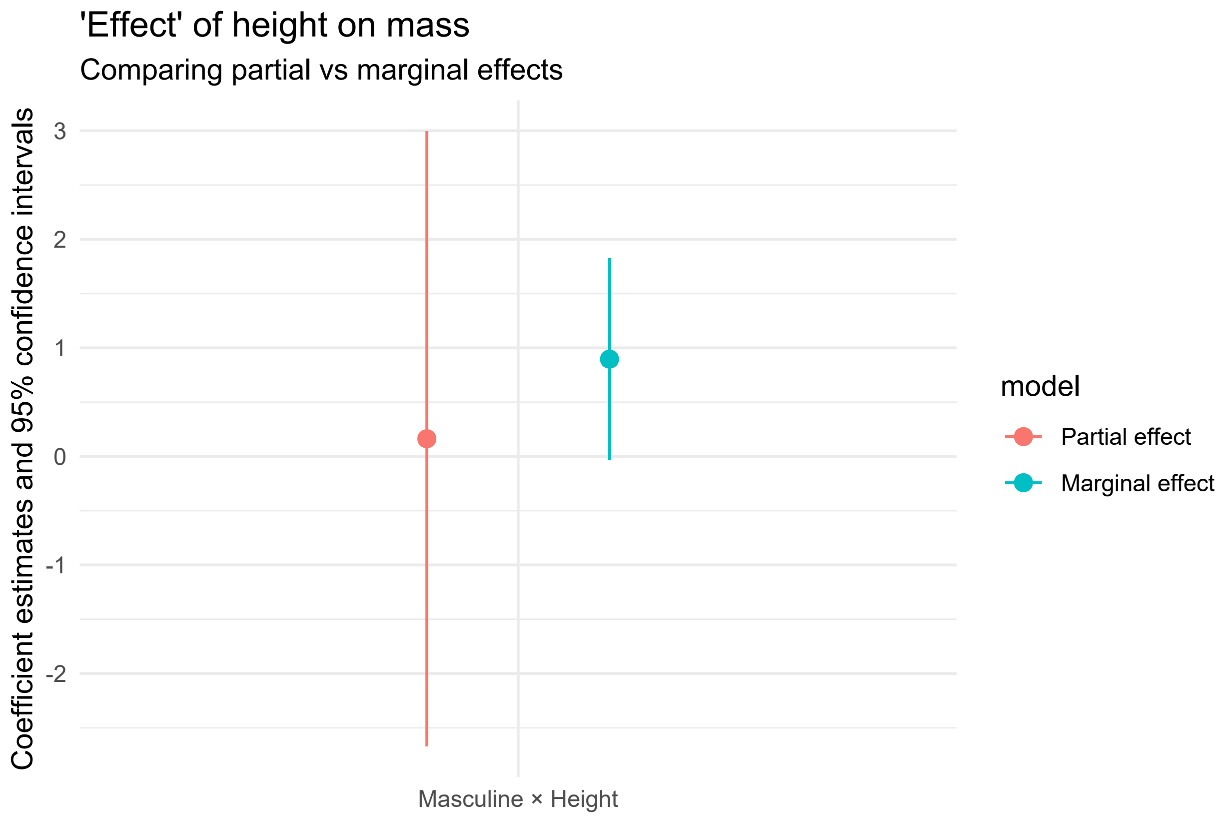
Coefficient plots We've already worked through an example of how to extract and compare model coefficients here. I use this "manual" approach to visualizing coefficient estimates all the time. However, our focus on **modelsummary** in the preceding section provides a nice segue to another one of the 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.

```r
# library(modelsummary) ## Already loaded
mods = list('No clustering' = summary(ols1, se = 'standard'))

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.

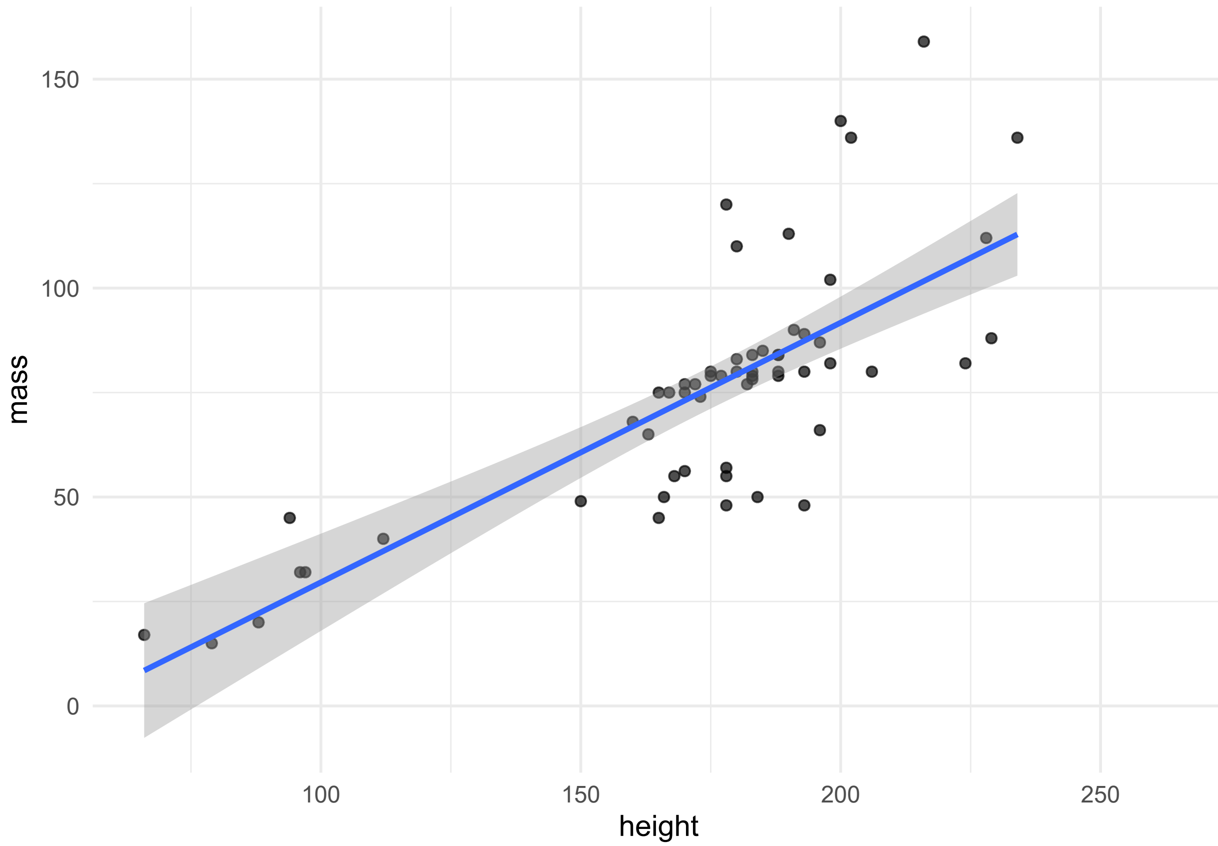
```r
ie_mods = list('Partial effect' = ols_ie, 'Marginal effect' = ols_ie_marg2)

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.[8] 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

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

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.[9]

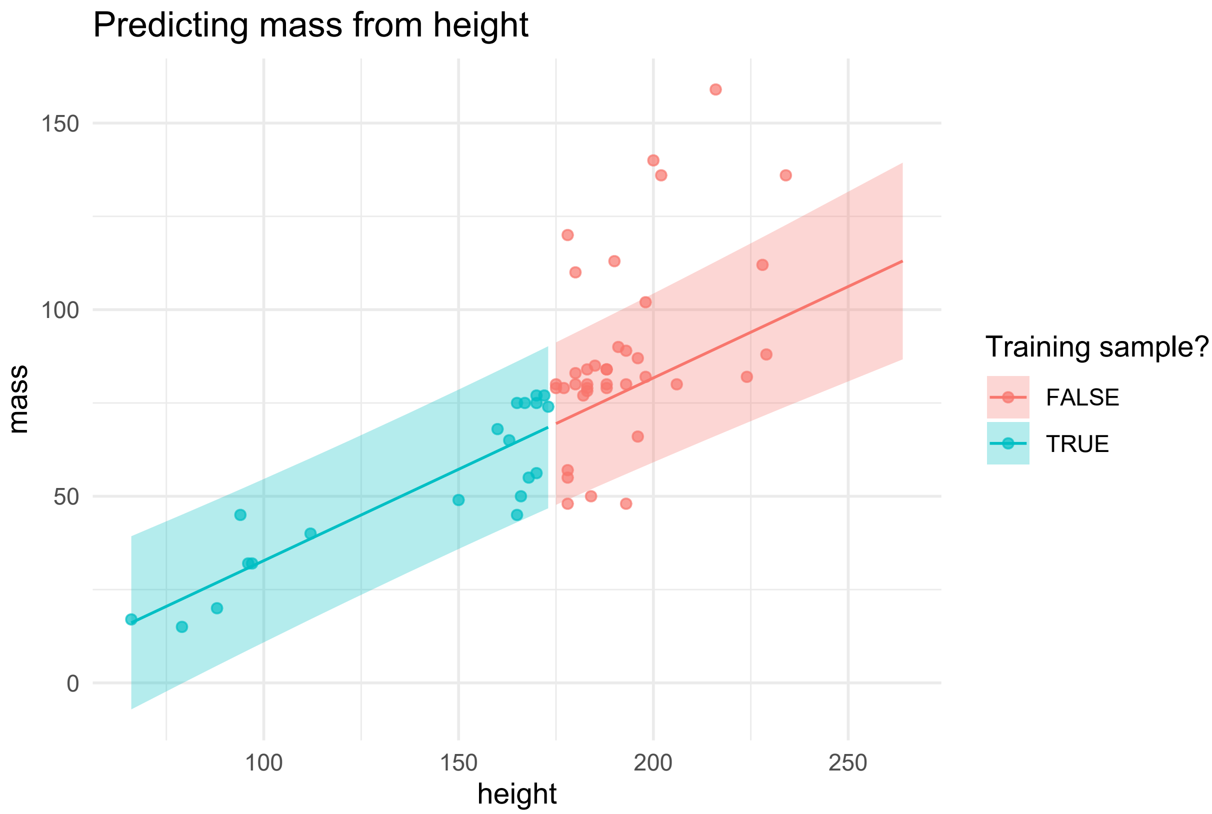
```
## Alternative to predict(): Use augment() to add .fitted and .resid, as well as
## .conf.low and .conf.high prediction interval variables to the data.
starwars2 = augment(ols1_train, newdata = starwars2, interval = "prediction")

## 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 Skywalker    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  Darth Vader       202   136 none       white
## 5    57.2  35.9    78.6  -8.23 Leia Organa       150    49 brown      light
## 6    70.9  49.1    92.8  49.1  Owen Lars         178   120 brown, gr~ light
## # i 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."
    )
```

[9] 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.



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.