Big Data and Economics

Regression analysis in R

Kyle Coombs

Bates College | DCS/ECON 368

Contents

Software requirements . . . 1
Regression basics . . . 3
Nonstandard errors . . . 6
Dummy variables and interaction terms . . . 10
Marginal effects . . . 13
Presentation . . . 17
Other models . . . 23
Instrumental variables . . . 23
Further resources . . . 27

This lecture covers 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. 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**
- Already used: **tidyverse**, **hrbrthemes**, **listviewer**, **tidycensus**, **tigris**

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

```r
## 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,tidycensus,tigris)
## 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 to mention up front is that we are using the Opportunity Atlas today for our regressions. Some of these regressions will be a bit simple, but the point is to show you how to run them in R with data that are meaningful. We'll also use the `fips_codes` data from the **tidycensus** package to merge in county names and state abbreviations.

The Opportunity Atlas file lives on GitHub, so we'll download it from there (*Note*: I am using the githack URL. You could also sync your fork, pull to your cloned repo, and navigate to `lectures/10-regression/` and the file is there too.). I've also amended the file to make it smaller so it can be easily pushed to GitHub. *Note*: This is bad practice generally, do not store data files on GitHub unless you have a really good reason to do so.

```
# UNCOMMENT THIS DOWNLOAD THE FIRST TIME YOU RUN THIS
# Download the data from GitHub into your working directory -- uncomment the line
# Create opp atlas object
opp_atlas <- read_csv("https://raw.githack.com/big-data-and-economics/big-data-class-materials/main/lec
```

```
## Rows: 3219 Columns: 10
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): czname, state_abb, state_name, county_name
## dbl (6): state, county, cz, kfr_pooled_pooled_p25, poor_share1990, ann_avg_j...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
#Quickly renaming fips data to make it easier to merge
fips <- fips_codes %>%
  rename(state_abb=state,
    state=state_code,
    county_name=county,
    county=county_code) %>%
  mutate(across(c(state,county),as.numeric))
```

```
# Look at file to refresh your memory
opp_atlas
```

```
## # A tibble: 3,219 x 10
##    state county    cz czname     kfr_pooled_pooled_p25 poor_share1990
##    <dbl>  <dbl> <dbl> <chr>                      <dbl>          <dbl>
##  1     1      1 11101 Montgomery                 0.362          0.140
##  2     1      3 11001 Mobile                     0.389          0.140
##  3     1      5 10301 Eufaula                    0.349          0.256
##  4     1      7 10801 Tuscaloosa                 0.363          0.213
##  5     1      9 10700 Birmingham                 0.392          0.148
##  6     1     11  9800 Auburn                     0.346          0.376
##  7     1     13 11101 Montgomery                 0.357          0.316
##  8     1     15  9600 LaGrange                   0.362          0.148
##  9     1     17  9600 LaGrange                   0.341          0.185
## 10     1     19  6600 Rome                       0.365          0.175
## # i 3,209 more rows
## # i 4 more variables: ann_avg_job_growth_2004_2013 <dbl>, state_abb <chr>,
## #   state_name <chr>, county_name <chr>
```

```
# We want to join these now, note that I do not need to specify the variable names because they are the

# Rename kfr for the kids with 25th percentile parents,
# This is the mean income percentile of the 25th percentile parents
```

```r
opp_atlas <- opp_atlas %>%
  left_join(fips) %>%
  rename(kfr_p25=kfr_pooled_pooled_p25)
```

```
## Joining with `by = join_by(state, county, state_abb, state_name, county_name)`
```

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.

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

You'll note that the `lm()` call includes a reference to the data source. 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 `opp_atlas` is the only data frame in our global environment at the time.

Let's run a simple bivariate regression of

```r
ols1 = lm(kfr_p25 ~ poor_share1990, data = opp_atlas)
ols1
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990, data = opp_atlas)
##
## Coefficients:
##    (Intercept)  poor_share1990
##        0.54146         0.08594
```

You might immediately notice that the coefficient seems wrong. A higher share that are poor in 1990 is associated with greater income mobility? We'll get there in a second.

First, let's talk about the object we have. 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 (used in the APIs 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 = kfr_p25 ~ poor_share1990, data = opp_atlas)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.928 -0.172 -0.135 -0.091 98.459
```

```
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.541460   0.100150   5.407  6.9e-08 ***
## poor_share1990 0.085944   0.006457  13.310  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.602 on 3217 degrees of freedom
## Multiple R-squared:  0.0522, Adjusted R-squared:  0.0519
## F-statistic: 177.2 on 1 and 3217 DF,  p-value: < 2.2e-16
```

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)    0.54146011 0.100149807  5.406502 6.895922e-08
## poor_share1990 0.08594437 0.006457027 13.310207 2.181347e-39
```

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)      0.541     0.100       5.41 6.90e- 8   0.345     0.738
## 2 poor_share1990   0.0859    0.00646    13.3  2.18e-39   0.0733    0.0986
```

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.

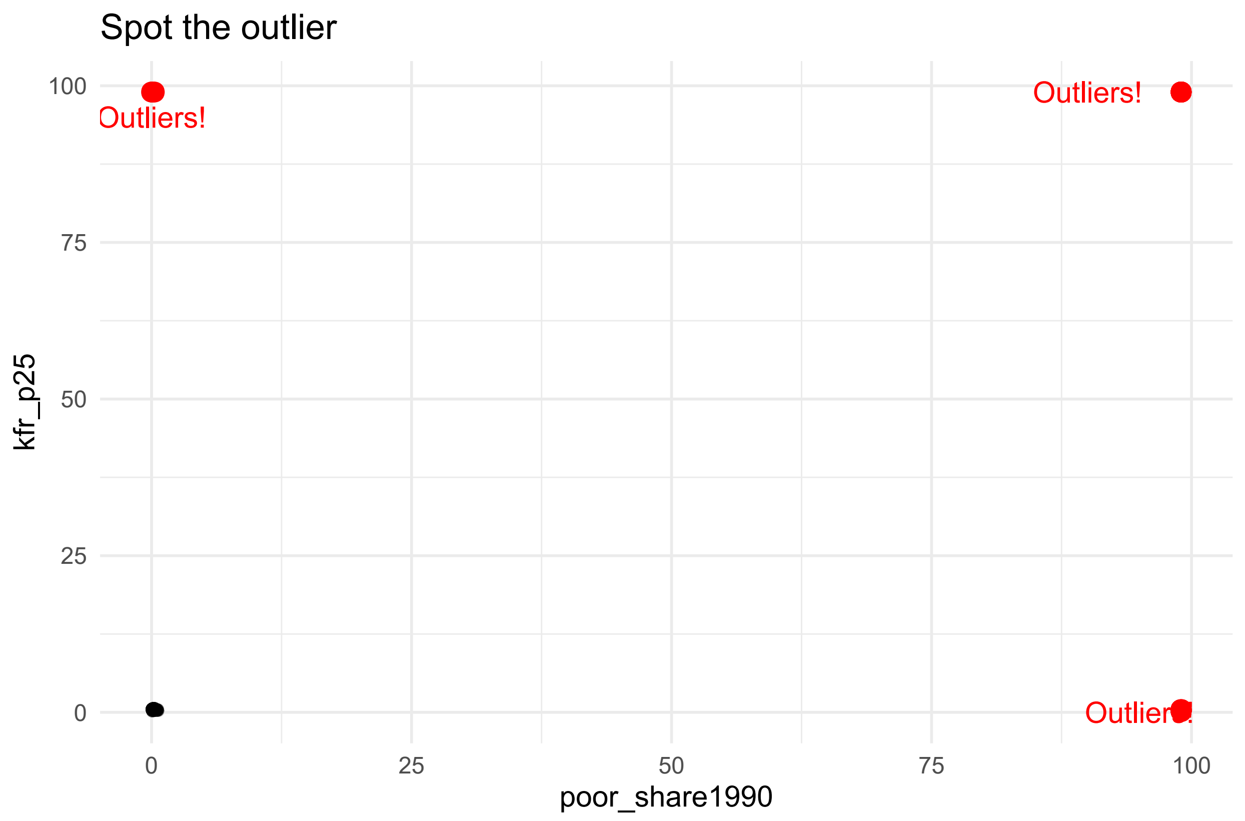
```
glance(ols1)
```

```
## # 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    0.0522        0.0519  5.60      177. 2.18e-39     1 -10114. 20233. 20251.
## # 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 0.052. Different species and homeworlds aside, we may have an extreme outlier in our midst…



Remember: Always plot your data...

It looks like NAs were replaced with 99 by someone… ahem… someone who wants to remind you to plot your data and check for quirks like this. Maybe we should exclude outliers 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. But it would take just a little elbow grease (with help from ChatGPT or CoPilot) to switch to **data.table** if you prefer.

```
opp_atlas_filter =
  opp_atlas %>%
  filter(kfr_p25!=99 & poor_share1990!=99)
  # filter(!(grepl("Jabba", name))) ## Regular expressions also work

ols2 = lm(kfr_p25 ~ poor_share1990, data = opp_atlas_filter)
summary(ols2)
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990, data = opp_atlas_filter)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.29396 -0.04179 -0.01081  0.03110  0.26356
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)       0.46353    0.00247  187.67   <2e-16 ***
## poor_share1990 -0.20497    0.01355  -15.13   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06027 on 3134 degrees of freedom
## Multiple R-squared:  0.06804,    Adjusted R-squared:  0.06774
## F-statistic: 228.8 on 1 and 3134 DF,  p-value: < 2.2e-16
```

2) Subset directly in the `lm()` call Running a regression directly on a subsetted data frame is equally easy.

```
ols2a = lm(kfr_p25 ~ poor_share1990, data = opp_atlas %>% filter(kfr_p25!=99 & poor_share1990!=99))
summary(ols2a)
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990, data = opp_atlas %>% filter(kfr_p25 !=
##     99 & poor_share1990 != 99))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.29396 -0.04179 -0.01081  0.03110  0.26356
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     0.46353    0.00247  187.67   <2e-16 ***
## poor_share1990 -0.20497    0.01355  -15.13   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06027 on 3134 degrees of freedom
## Multiple R-squared:  0.06804,    Adjusted R-squared:  0.06774
## F-statistic: 228.8 on 1 and 3134 DF,  p-value: < 2.2e-16
```

The overall model fit is much improved by the exclusion of this outlier, with R2 increasing to 0.068. 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.

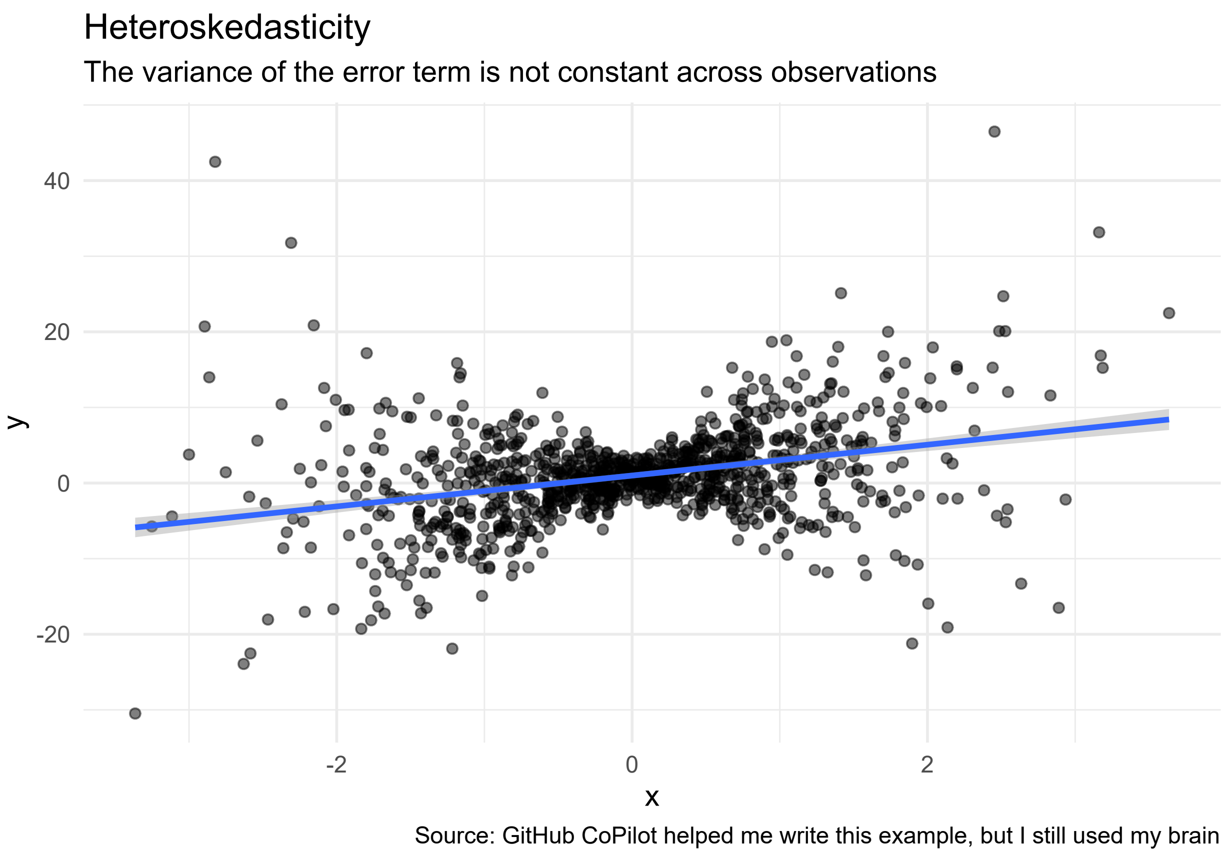
Robust standard errors

One of the primary reasons that you might want to use robust standard errors is to account for heteroskedasticity. What is heteroskedasticity, well it is when the variance of the error term is not constant across observations. This is a problem because it violates one of the key assumptions of OLS regression, namely that the error term is homoskedastic (i.e. constant variance).[1] I present an example below with fake data (cause it is easier than forcing it out of real data.)

[1] See Causal Inference: The Mixtape for more details.

```r
# Create an example of heteroskedasticity

# This creates a simple regression, but with variance that increases with x
hetero_df <- tibble(
  x = rnorm(1000),
  y = 1 + 2 * x + rnorm(1000, sd = 1 + 5 * abs(x))
)

# Plot the data
hetero_df %>%
  ggplot(aes(x = x, y = y)) +
  geom_point(alpha = 0.5) +
  geom_smooth(method = "lm", se = TRUE) +
  labs(
    title = "Heteroskedasticity",
    subtitle = "The variance of the error term is not constant across observations",
    caption = "Source: GitHub CoPilot helped me write this example, but I still used my brain"
  )
```

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



There are many ways to deal with heteroskedasticity, but one of the most common is to use 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.

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

```
ols1_robust = lm_robust(kfr_p25 ~ poor_share1990, data = opp_atlas_filter)
# tidy(ols1_robust, conf.int = TRUE) ## Could tidy too
ols1_robust
```

```
##                  Estimate  Std. Error   t value     Pr(>|t|)   CI Lower
## (Intercept)     0.4635271 0.002243062 206.64928 0.000000e+00  0.4591291
## poor_share1990 -0.2049684 0.012138310 -16.88607 2.699319e-61 -0.2287682
##                  CI Upper   DF
## (Intercept)     0.4679251 3134
## poor_share1990 -0.1811685 3134
```

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.[2] For example, you can specify Stata robust standard errors (`reg y x, robust`) 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. tl;dr: A few years ago several people realized Stata was reporting different SEs than they expected.)

```
lm_robust(kfr_p25 ~ poor_share1990, data = opp_atlas_filter, se_type = "stata")
```

```
##                  Estimate  Std. Error   t value    Pr(>|t|)   CI Lower
## (Intercept)     0.4635271 0.002241778 206.76765 0.00000e+00  0.4591316
## poor_share1990 -0.2049684 0.012126374 -16.90269 2.08495e-61 -0.2287448
##                  CI Upper   DF
## (Intercept)     0.4679226 3134
## poor_share1990 -0.1811919 3134
```

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. See this 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
```

```
##    (Intercept) poor_share1990
##     0.07192355     0.02334502
```

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:

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

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

```
##
## t test of coefficients:
```

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

```
##
##                  Estimate Std. Error t value  Pr(>|t|)
## (Intercept)    0.541460   0.071924  7.5283 6.638e-14 ***
## poor_share1990 0.085944   0.023345  3.6815 0.0002357 ***
## ---
## 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)
```

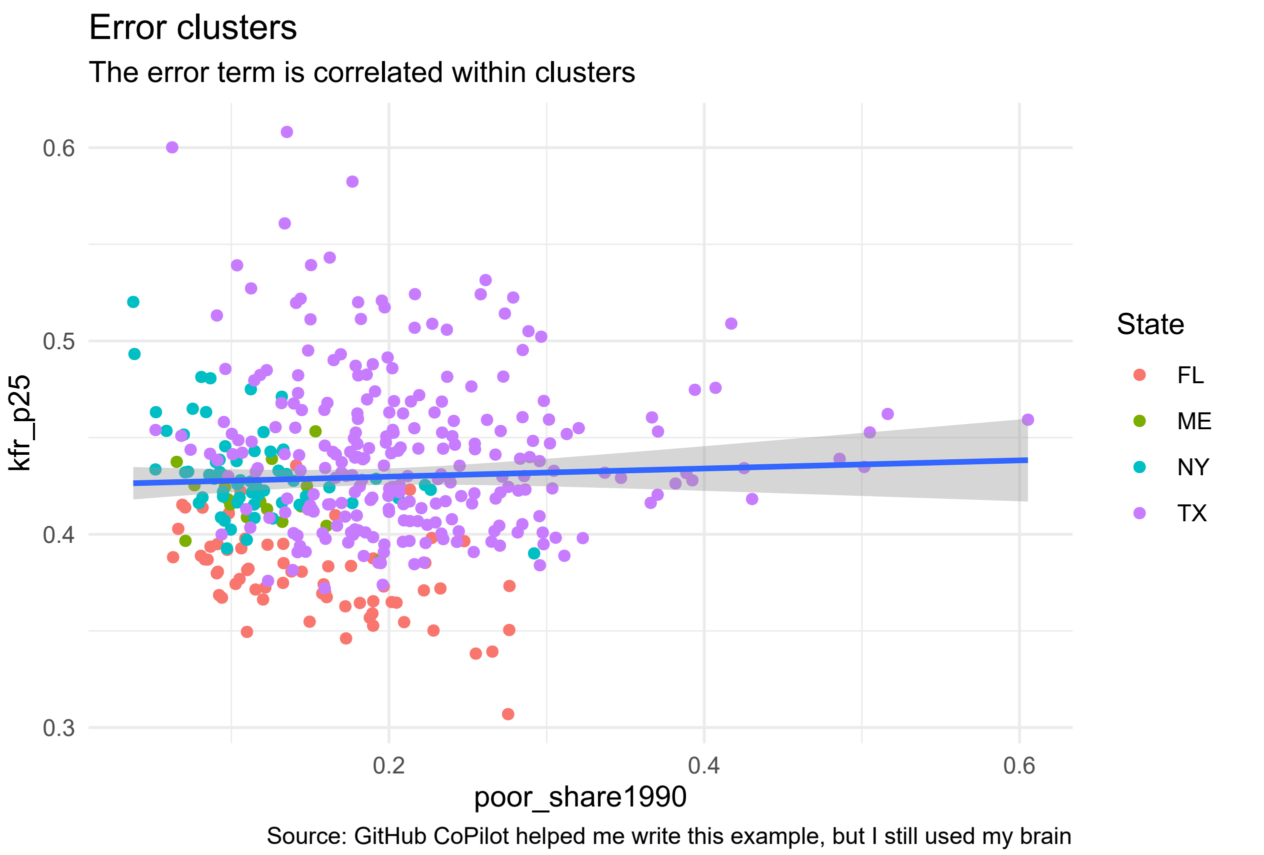
```
## # A tibble: 2 x 7
##   term           estimate std.error statistic  p.value conf.low conf.high
##   <chr>             <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
## 1 (Intercept)      0.541     0.0719      7.53 6.64e-14   0.400      0.682
## 2 poor_share1990   0.0859    0.0233      3.68 2.36e- 4   0.0402     0.132
```

Clustered standard errors

Another way that standard errors can violate homoskedasticity is for the error to be "clustered" by groups in the data. A classic example is students in the same classroom will all have the same teacher and so their learning outcomes will be correlated. For example, all of you have the same teacher (me), so you will all learn the same coding habits. This is a problem because it violates the assumption that the error term is independent across observations. Here's an example with the Opportunity Atlas data.

```r
# Let's look at just a few states, so we can easily see clusters
filter(opp_atlas_filter, state_abb %in% c('FL','NY','TX','ME')) %>%
  ggplot(aes(x=poor_share1990,y=kfr_p25)) +
  geom_point(aes(col=state_abb)) +
  geom_smooth(method='lm') +
  labs(
    col = 'State',
    title = "Error clusters",
    subtitle = "The error term is correlated within clusters",
    caption = "Source: GitHub CoPilot helped me write this example, but I still used my brain"
  )
```

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



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(kfr_p25 ~ poor_share1990, data = opp_atlas_filter, clusters = state)
```

```
##                  Estimate Std. Error   t value     Pr(>|t|)   CI Lower
## (Intercept)     0.4635271 0.01092559 42.425831 6.474980e-29  0.4412395
## poor_share1990 -0.2049684 0.04822963 -4.249844 4.104134e-04 -0.3057273
##                  CI Upper       DF
## (Intercept)     0.4858147 30.83756
## poor_share1990 -0.1042095 19.53519
```

Dummy variables and interaction terms

For the next section, we'll need to create a dummy variable. We'll create two: one for whether a county is in the South and one for all major regions of the US.

```r
# Get the regions and do minor cleaning
options(tigris_use_cache=TRUE)
states <- tigris::states(year=2015) %>%
  as.data.frame() %>%
  select(state=STATEFP, region=REGION) %>%
  mutate(state=as.numeric(state),
    region=as.numeric(region))

opp_atlas_filter <- mutate(opp_atlas_filter,
  in_south = ifelse(state_abb %in% c("AL", "AR",
    "FL", "GA", "KY", "LA", "MS", "NC", "OK",
     "SC", "TN", "TX", "VA", "WV"), "South", "North")) %>%
  left_join(states)
```

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.[3]

Here's an example where we explicitly tell R that "in_south" 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.

```
summary(lm(kfr_p25 ~ poor_share1990 + as.factor(in_south), data = opp_atlas_filter))
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990 + as.factor(in_south),
##     data = opp_atlas_filter)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.306973 -0.036301 -0.008869  0.029395  0.234286
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                0.466033   0.002221 209.793  < 2e-16 ***
## poor_share1990            -0.063778   0.013226  -4.822 1.49e-06 ***
## as.factor(in_south)South -0.057829   0.002114 -27.351  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05416 on 3133 degrees of freedom
## Multiple R-squared:  0.2477, Adjusted R-squared:  0.2472
## F-statistic: 515.7 on 2 and 3133 DF,  p-value: < 2.2e-16
```

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^{TM}. 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 "in_south" instead; R knows it must be ordered
## for it to be included as a regression variable
summary(lm(kfr_p25 ~ poor_share1990 + in_south, data = opp_atlas_filter))
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990 + in_south, data = opp_atlas_filter)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.306973 -0.036301 -0.008869  0.029395  0.234286
##
```

[3] Factors are variables that have distinct qualitative levels, e.g. "male", "female", "non-binary", etc.

```
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)     0.466033   0.002221 209.793  < 2e-16 ***
## poor_share1990 -0.063778   0.013226  -4.822 1.49e-06 ***
## in_southSouth  -0.057829   0.002114 -27.351  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05416 on 3133 degrees of freedom
## Multiple R-squared:  0.2477, Adjusted R-squared:  0.2472
## F-statistic: 515.7 on 2 and 3133 DF,  p-value: < 2.2e-16
```

What happens if I use region? It thinks region is a numeric variable and so it does not create dummy variables for it.

```
summary(lm(kfr_p25 ~ poor_share1990 + region, data = opp_atlas_filter))
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990 + region, data = opp_atlas_filter)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.27441 -0.04197 -0.01136  0.03161  0.25427
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)     0.492303   0.003943 124.859   <2e-16 ***
## poor_share1990 -0.174849   0.013758 -12.709   <2e-16 ***
## region         -0.012640   0.001362  -9.284   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05947 on 3133 degrees of freedom
## Multiple R-squared:  0.09299,    Adjusted R-squared:  0.09241
## F-statistic: 160.6 on 2 and 3133 DF,  p-value: < 2.2e-16
```

We can fix this by telling R that region is a factor.

```
summary(lm(kfr_p25 ~ poor_share1990 + as.factor(region), data = opp_atlas_filter))
```

```
##
## Call:
## lm(formula = kfr_p25 ~ poor_share1990 + as.factor(region), data = opp_atlas_filter)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.29983 -0.03550 -0.00631  0.02925  0.22547
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         0.445325   0.003856 115.500  < 2e-16 ***
## poor_share1990     -0.076094   0.013024  -5.843 5.66e-09 ***
## as.factor(region)2  0.031849   0.003994   7.973 2.15e-15 ***
## as.factor(region)3 -0.034673   0.004066  -8.528  < 2e-16 ***
## as.factor(region)4  0.014484   0.004449   3.255  0.00114 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05323 on 3131 degrees of freedom
## Multiple R-squared:  0.2738, Adjusted R-squared:  0.2729
## F-statistic: 295.2 on 4 and 3131 DF,  p-value: < 2.2e-16
```

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.[4] 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 location's income mobility for children in the 25th percentile and its 1990 poverty rate differs by region. That is, we want to run a regression of the form,

$$KFR_{P25} = \beta_0 + \beta_1 D_{South} + \beta_2 + \beta_3 D_{South} \times 1990PovertyShare$$

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

```
ols_ie = lm(kfr_p25 ~ in_south * poor_share1990, data = opp_atlas_filter)
summary(ols_ie)
```

```
##
## Call:
## lm(formula = kfr_p25 ~ in_south * poor_share1990, data = opp_atlas_filter)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.307278 -0.036168 -0.008956  0.029301  0.234545
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)                   0.466705   0.003116 149.762  < 2e-16 ***
## in_southSouth                -0.059169   0.004842 -12.221  < 2e-16 ***
## poor_share1990               -0.068708   0.020780  -3.306 0.000955 ***
## in_southSouth:poor_share1990  0.008289   0.026944   0.308 0.758382
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05417 on 3132 degrees of freedom
## Multiple R-squared:  0.2477, Adjusted R-squared:  0.247
## F-statistic: 343.7 on 3 and 3132 DF,  p-value: < 2.2e-16
```

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.

[4] 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, Grant McDermott has strong feelings that interaction effects are most widely misunderstood and misapplied concept in econometrics. However, that's a topic for another day.

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.[5] 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 income mobility varied by 1990 poverty rate and region. 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 in_southSouth    -0.0578   0.00212    -27.3  2.17e-164  -0.0620   -0.0537
## 2 poor_share1990  -0.0650   0.0138      -4.70 2.61e-  6  -0.0922   -0.0379
```

If we want to compare marginal effects at specific values — e.g. how the AME of 1990 poverty rate on income mobility differs across north and south — then that's easily done too.

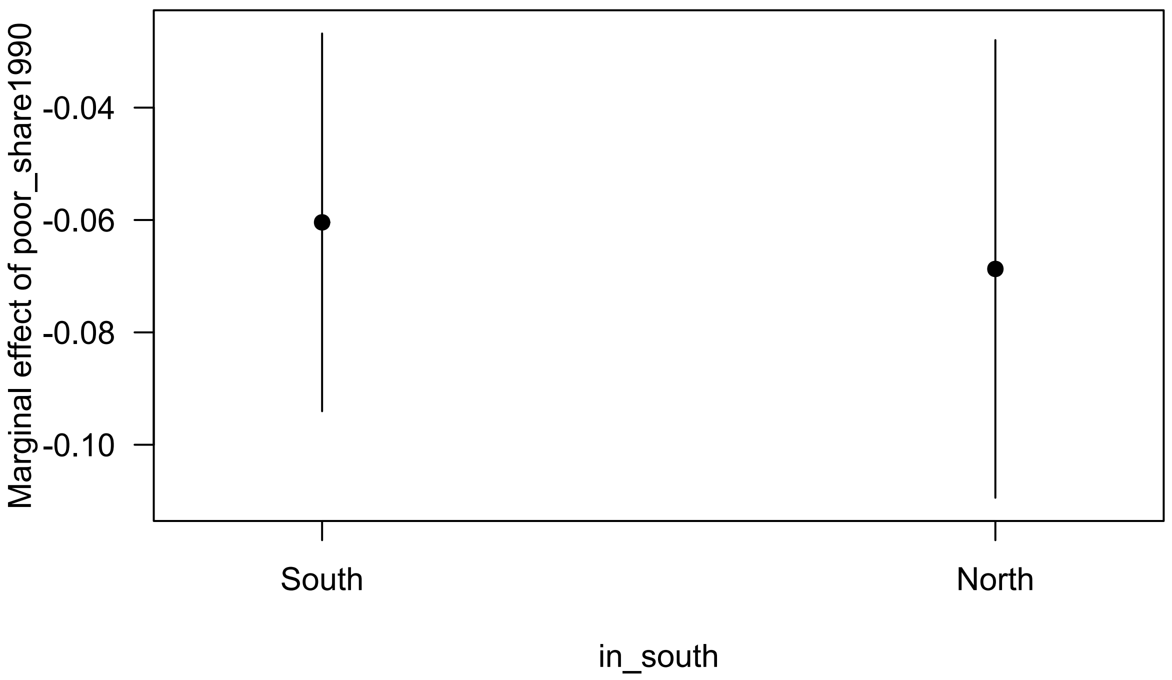
```
ols_ie %>%
  margins(
    variables = "poor_share1990", ## The main variable we're interested in
    at = list(in_south = c("North","South")) ## How the main variable is modulated by at specific value
    ) %>%
  tidy(conf.int = TRUE) ## Tidy it (optional)
```

```
## # 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 poor_share~ in_south    North     -0.0687    0.0208     -3.31 9.45e-4  -0.109
## 2 poor_share~ in_south    South     -0.0604    0.0172     -3.52 4.27e-4  -0.0940
## # 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.

```
cplot(ols_ie, x = "in_south", dx = "poor_share1990", what = "effect",
      data = opp_atlas_filter)
```

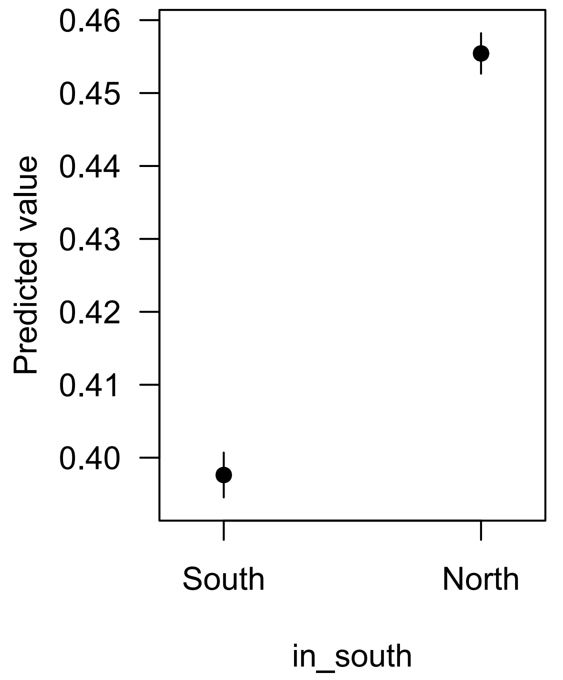
[5] I do, however, want to flag that it does not yet support **fixest** (or **lfe**) models. But there are workarounds in the meantime.

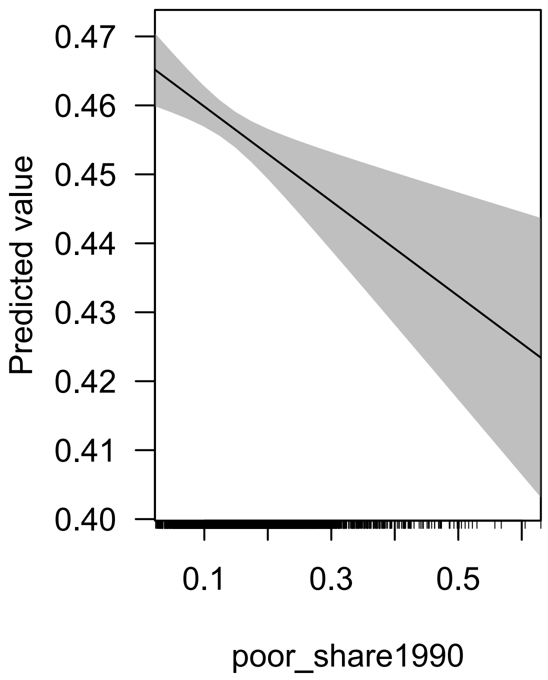


In this case,it doesn't make much sense to read a lot into the larger standard errors on the female group; that's being driven by a very small sub-sample size.

Finally, you can also use `cplot()` to plot the predicted values of your outcome variable (here: "kfr_p25"), 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 = "in_south", what = "prediction", data = opp_atlas_filter)
cplot(ols_ie, x = "poor_share1990", what = "prediction", data = opp_atlas_filter)
```





```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. Grant 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(kfr_p25 ~ in_south * poor_share1990, data = opp_atlas_filter) ## Original model
ols_ie_marg2 = lm(kfr_p25 ~ in_south / poor_share1990, data = opp_atlas_filter)
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)             0.467   0.00312    150.   0           0.461     0.473
## 2 in_southSouth          -0.0592  0.00484    -12.2  1.38e-33  -0.0687   -0.0497
## 3 in_southNorth:poor_s~  -0.0687  0.0208      -3.31 9.55e- 4  -0.109    -0.0280
```

Table 1: Relationship between Average Income Percentile in 2015 of Children born in 25th percentile and 1990 poverty rate and region

	OLS	Interaction
Poverty Rate (1990)	0.086***	−0.069***
	(0.006)	(0.021)
South x Poverty Rate (1990)		0.008
		(0.027)
South		−0.059***
		(0.005)
Constant	0.541***	0.467***
	(0.100)	(0.003)
Num.Obs.	3219	3136
R2	0.052	0.248
R2 Adj.	0.052	0.247
AIC	20 233.0	−9381.3
BIC	20 251.2	−9351.1
Log.Lik.	−10 113.508	4695.656
RMSE	5.60	0.05

+ $p < 0.1$, * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

```
## 4 in_southSouth:poor_s~  -0.0604   0.0172      -3.52 4.34e- 4  -0.0940   -0.0268
```

Note that the marginal effects on the two south × poverty rate interactions (i.e. -0.069 and -0.06) 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 the aforementioned blog post if you'd like to learn more.

Presentation

Tables

Regression tables There are loads of different options here. A great tool for 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.

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

## Note: msummary() is an alias for modelsummary() add variable names
msummary(list("OLS"=ols1, "Interaction"=ols_ie),
         stars=TRUE, ## Output type
         coef_map = c("poor_share1990"="Poverty Rate (1990)",
          "in_southSouth:poor_share1990" = "South x Poverty Rate (1990)",
          "in_southSouth" = "South",
          "(Intercept)"="Constant"),
          title="Relationship between Average Income Percentile in 2015 of Children born in 25th percen
```

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

	North (N=1744)		South (N=1392)			
	Mean	Std. Dev.	Mean	Std. Dev.	Diff. in Means	Std. Error
kfr_p25	0.5	0.1	0.4	0.0	-0.1	0.0
poor_share1990	0.1	0.1	0.2	0.1	0.1	0.0
ann_avg_job_growth_2004_2013	0.0	0.0	0.0	0.0	0.0	0.0

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(~ in_south,
                    data = opp_atlas_filter %>%
                    select(kfr_p25:ann_avg_job_growth_2004_2013,in_south))
```

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(opp_atlas_filter %>%
  select(kfr_p25:ann_avg_job_growth_2004_2013, in_south),
   group = 'in_south',
   out = otype)
```

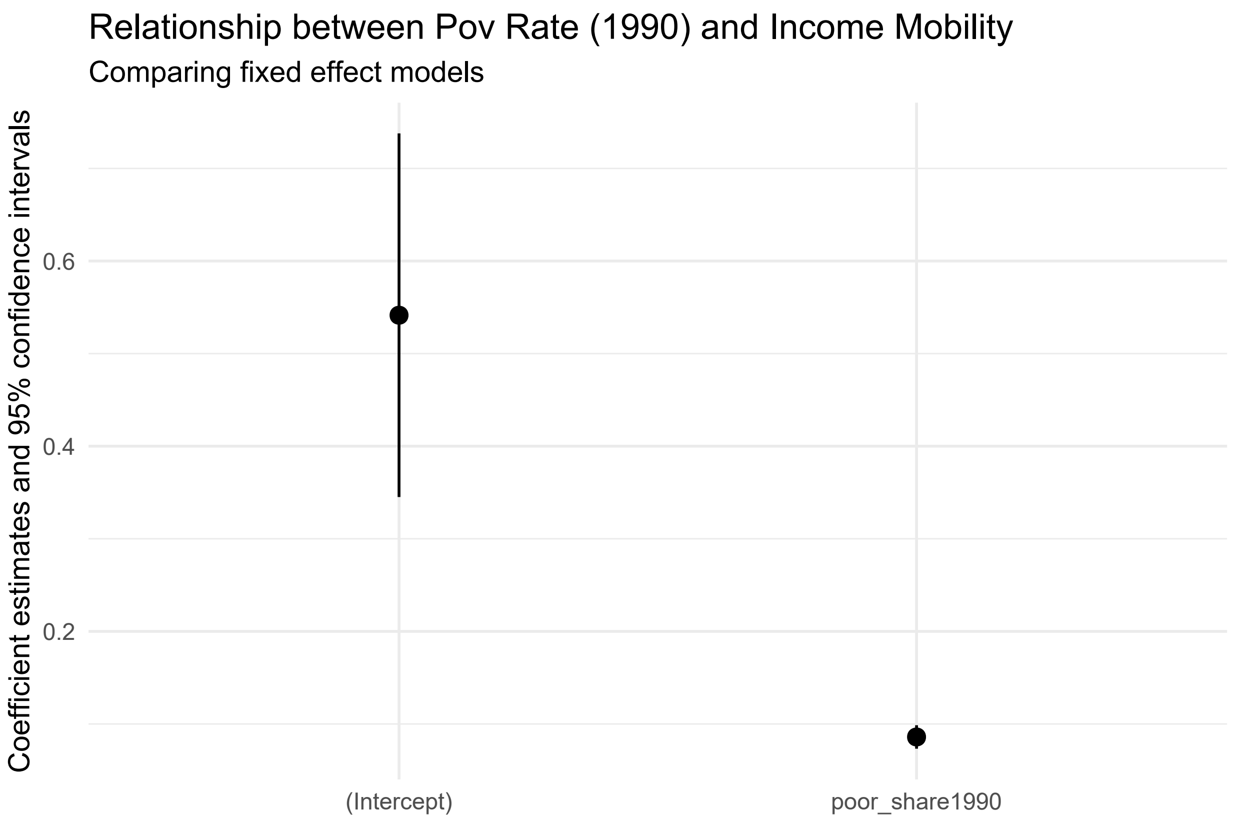
```
##                       Variable     N    Mean    SD     N    Mean    SD
## 1                     in_south North               South
## 2                      kfr_p25  1744    0.46 0.062  1392     0.4 0.044
## 3               poor_share1990  1744    0.14 0.062  1392     0.2 0.085
## 4 ann_avg_job_growth_2004_2013  1739 -0.0028 0.014  1392 -0.0022 0.016
```

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

Figures

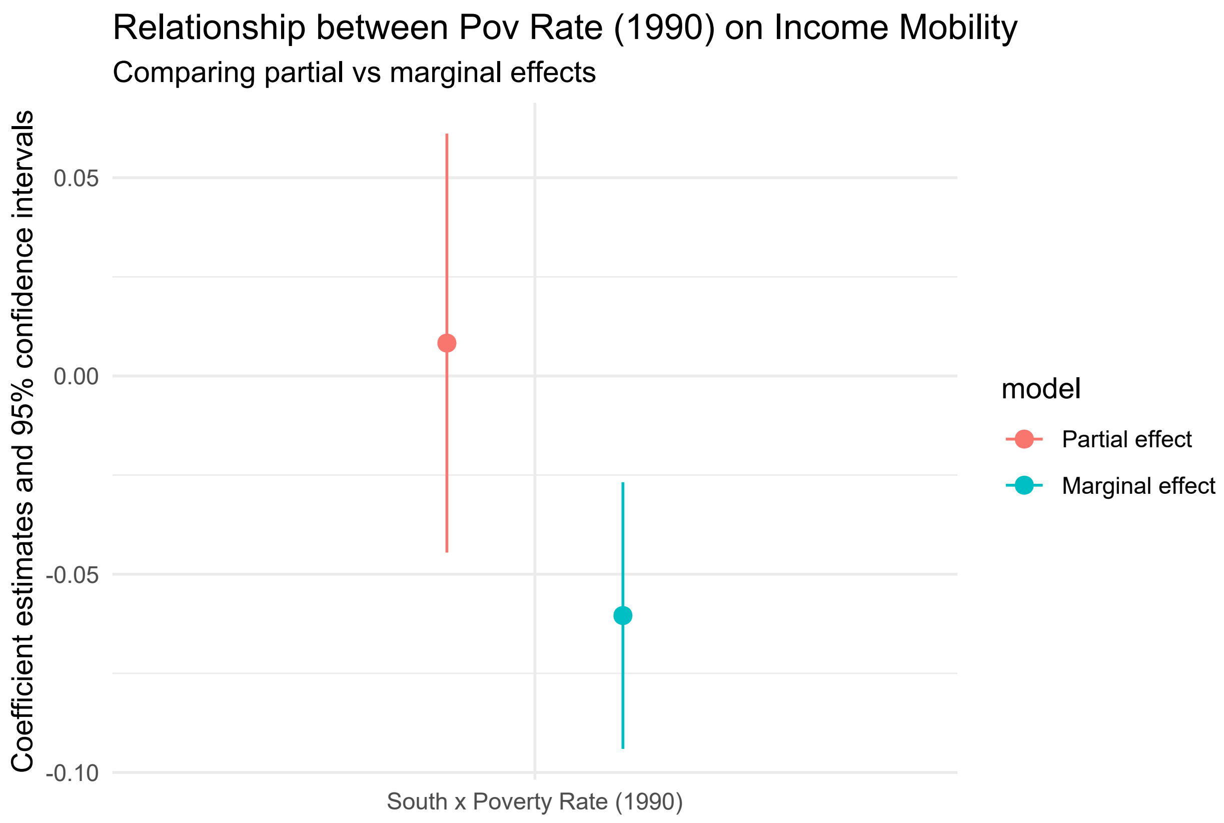
Coefficient plots We've already worked through an example of how to extract and compare model coefficients here. I use this "manual" approach to visualizing coefficient estimates all the time. However, our focus on **modelsummary** in the preceding section provides a nice segue to another one of the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

```
# 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 = "Relationship between Pov Rate (1990) and Income Mobility",
    subtitle = "Comparing fixed effect models"
    )
```



Or, here's another example where we compare the (partial) In South × Poverty Share coefficient from our earlier interaction model, with the (full) marginal effect that we obtained later on.

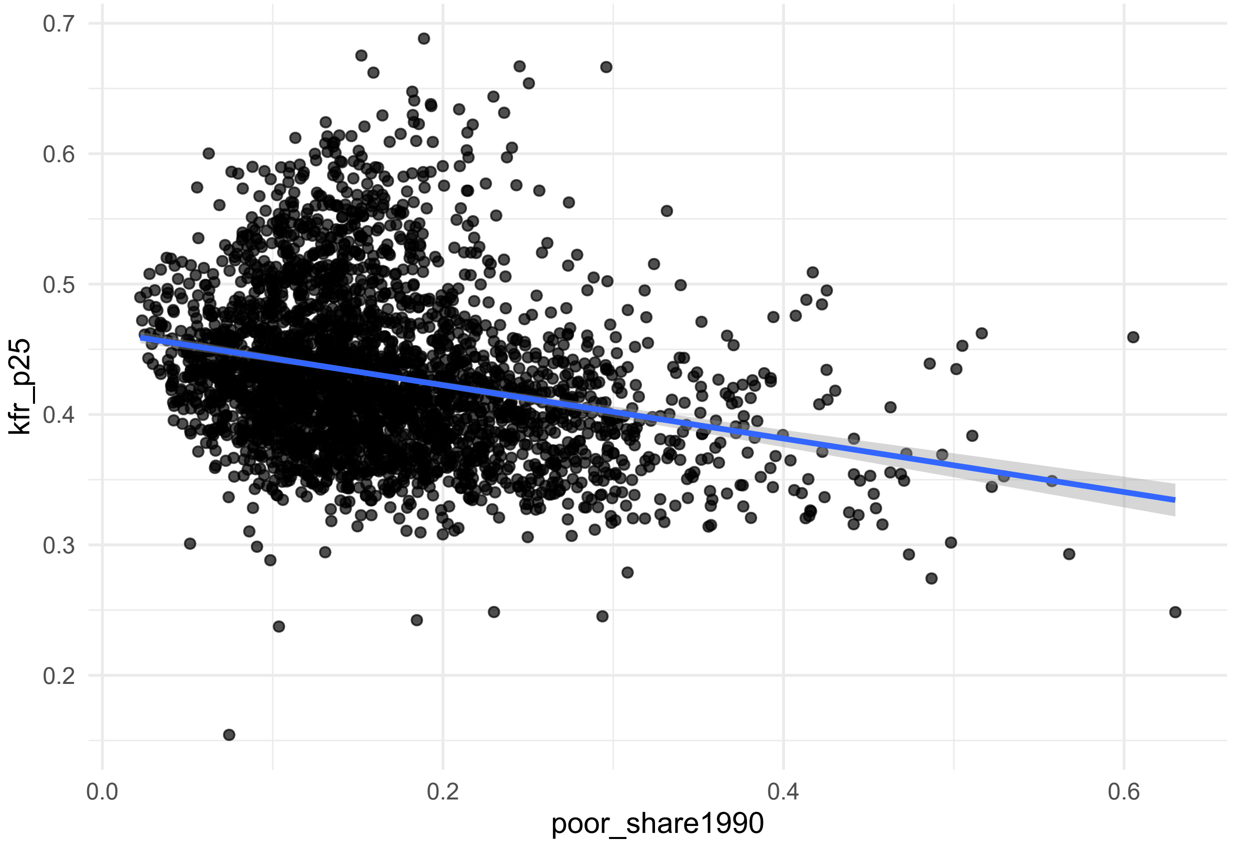
```
ie_mods = list('Partial effect' = ols_ie, 'Marginal effect' = ols_ie_marg2)

modelplot(ie_mods, coef_map = c("in_southSouth:poor_share1990" = "South x Poverty Rate (1990)")) +
  coord_flip() +
  labs(
    title = "Relationship between Pov Rate (1990) on Income Mobility",
    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 `opp_atlas_filter` data frame that excludes those bad NA outliers:

```r
ggplot(opp_atlas_filter, aes(x = poor_share1990, y = kfr_p25)) +
    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 from earlier.[6] This time, however, we'll estimate our model on a training dataset that only consists of the first 1000 counties ranked by poverty. Here's how you would do it.

```
## Estimate a model on a training sample of the data (shortest 30 characters)
ols1_train = lm(kfr_p25 ~ poor_share1990, data = opp_atlas_filter %>% filter(rank(poor_share1990) <= 10

## Use our model to predict the average income percentile of children born at 25th
## percentile using the full dataset
## Note that I'm including a 95% prediction interval. See ?predict.lm for other
## intervals and options.
predict(ols1_train, newdata = opp_atlas_filter, interval = "prediction") %>%
  head(5) ## Just print the first few rows
```

```
##         fit       lwr       upr
## 1 0.4381707 0.3327544 0.5435870
## 2 0.4381430 0.3327216 0.5435643
## 3 0.4314182 0.3234046 0.5394317
## 4 0.4339106 0.3271717 0.5406494
## 5 0.4376971 0.3321882 0.5432060
```

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

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

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.[7]

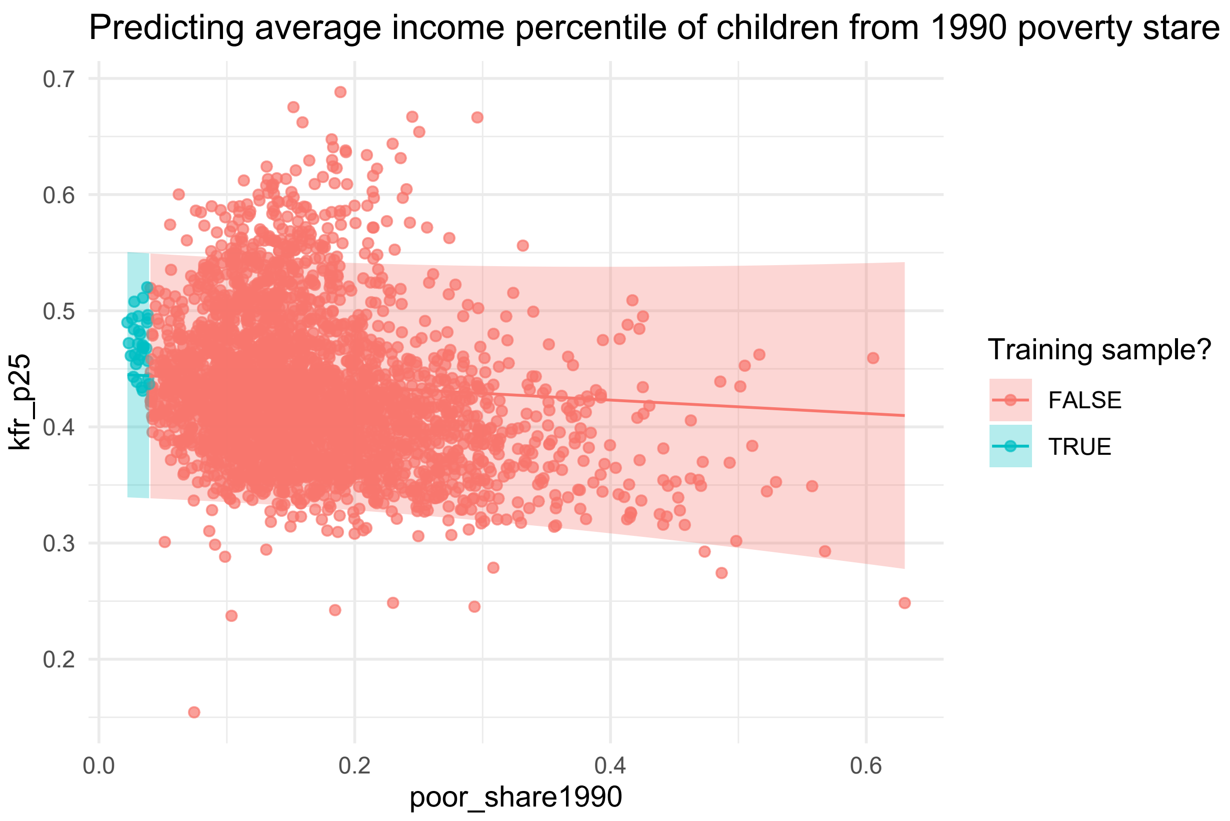
```
## Alternative to predict(): Use augment() to add .fitted and .resid, as well as
## .conf.low and .conf.high prediction interval variables to the data.
opp_atlas_filter = augment(ols1_train, newdata = opp_atlas_filter, interval = "prediction")

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

```
## # A tibble: 6 x 16
##   .fitted .lower .upper  .resid state county    cz czname kfr_p25 poor_share1990
##     <dbl>  <dbl>  <dbl>   <dbl> <dbl>  <dbl> <dbl> <chr>    <dbl>          <dbl>
## 1   0.438  0.333  0.544 -0.0765     1      1 11101 Montg~   0.362          0.140
## 2   0.438  0.333  0.544 -0.0493     1      3 11001 Mobile   0.389          0.140
## 3   0.431  0.323  0.539 -0.0820     1      5 10301 Eufau~   0.349          0.256
## 4   0.434  0.327  0.541 -0.0705     1      7 10801 Tusca~   0.363          0.213
## 5   0.438  0.332  0.543 -0.0458     1      9 10700 Birmi~   0.392          0.148
## 6   0.424  0.311  0.538 -0.0788     1     11  9800 Auburn   0.346          0.376
## # i 6 more variables: ann_avg_job_growth_2004_2013 <dbl>, state_abb <chr>,
## #   state_name <chr>, county_name <chr>, in_south <chr>, region <dbl>
```

We can now see how well our model — again, only estimated on the 1000 counties with least poverty in 1990 — performs against all of the data.

```
opp_atlas_filter %>%
  ggplot(aes(x = poor_share1990, y = kfr_p25, col = rank(poor_share1990)<=30, fill = rank(poor_share199
  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 average income percentile of children from 1990 poverty stare",
    caption = "Line of best fit, with shaded regions denoting 95% prediction interval."
    )
```

[7]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.

Other models

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 two different options today using the `ivreg::ivreg()` and `estimatr::iv_robust()` (we'll do another with `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 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).[8] 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
## ---
```

[8] 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.

```
## 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.

```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.[9] 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
```

[9] 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.

```
##
## 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
```

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 using the cars dataset that comes bundled with R.

```
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`

## 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)
- Bayesian regression: **rstanarm** (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.

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.