Quick and Tidy Guide to Econometrics

Ryan Safner

Fall 2019

Contents

Overview

This is a quick guide to using R for basic econometrics and data analysis tasks (i.e. manipulating data, running regressions, and making plots and tables) in an *opinionated* way, using the `tidyverse` packages and grammar. Use this as document a shortcut or cheatsheet to refer to the packages, commands, and syntax necessary to perform these tasks.

It is wise to always **load tidyverse**[1] at the beginning of each R session or markdown document, as most of the guide makes use of packages and commands that are *assumed* to already be loaded with tidyverse.

```
library(tidyverse)
```

Throughout this guide, I use an `example_data.csv` file for data. To see how I made this data with R, see the Appendix.

[1]With `library(tidyverse)`.

Summary Cheatsheet

General Tips

- Make an R Project to organize files
- Always start with `library(tidyverse)`
- Never `View()` or `install.packages()` in a `.Rmd` file

Data Import

`df<-read_*("file.*")`

Data Wrangling

Command	Does	Example
`select()`	Keep desired columns (variables)	`gapminder %>% select(pop)`
`filter()`	Keep desired rows (observations)	`gapminder %>% filter(country=="France")`
`arrange()`	Reorder rows (e.g. in numerical order)	`gapminder %>% arrange(pop)`
`mutate()`	Create new variables	`gapminder %>% mutate(GDP = gdpPerCap * pop)`
`summarize()`	Collapse data into summary statistics	`gapminder %>% summarize(avg_GDP = mean(gdpPerCap))`
`group_by()`	Perform any of the above functions by groups/categories	`gapminder %>% group_by(country) %>% summarize(avg_LE = mean(LifeExp))`

Regressions

`reg<-lm(y~x+z, data = df)`

- Extensions:
  - Polynomial: `I(var_name^2)`
  - Log: `log(var_name)`
  - Dummy/Fixed Effects (if not already a `factor` or dummy): `factor(category_var)`
  - Interaction: `var_1:var_2`
- Viewing:
  - `summary(reg)` for full output
  - `broom::tidy(reg)` to view coefficients in tidy tibble
  - `broom::glance(reg)` to view regression statistics in tidy tibble
  - `broom::augment(reg)` to add regression-based observations (i.e. $\hat{u}_i$, $\hat{Y}_i$)
- Making Regression Tables:
  - `huxtable::huxreg(reg)`

Plotting

```
ggplot(data = df)+ # layer defining dataframe for data source
  # aesthetics layer to map variables to aesthetics
```

```
aes(x = X,
    y = Y,
    color = shape)+ # will color by shape
# geometries layer(s)
geom_point()+
geom_smooth(method = "lm") # add a regression line
```

R Basics

- R is "object-oriented":
  - *Assign* values in objects: `my_object<-my_values`
  - *Overwrite* objects: `my_object<-my_new_values`
  - *Run functions* on objects: `function_name(my_object)`
- Data types:
  - `numeric` ("`double`" or "`integer`") data are numbers
    * can use for math and statistics
  - `character`: strings of text
    * values must always be `"in quotes"`
  - `factor`: indicates membership in one of several possible categories or groups
- Object types:
  - `vector`: collection of objects
    * create with `c()` function
  - `data.frame` or `tibble`: each row is a vector of same data type[2]
    * rows are observations
    * columns are variables
- Packages:
  - *Install* any package with 'install.packages("package_name")
    * Only necessary once, if package doesn't already exist
  - *Load* a package for each session needed with `library("package_name")`
  - *Description of packages we use*:

Package	Use
`tidyverse`	For tibbles (`tibble`), `%>%` operator (`magrittr`), data import (`readr`), data wrangling (`dplyr`), plotting (`ggplot2`)
`broom`	For tidy regression outputs
`huxtable`	For making regression tables
`car`	For regression tests (heteroskedasticity, outliers, F-test)
`lmtest`	
`estimatr`	For regression with heteroskedasticity-robust standard errors

[2] Henceforth, with `tidyverse`, I refer to all `data.frames` as `tibbles`

Data Wrangling

Import

- Import data with `read_*()` where the `*` represents the file extension (e.g. `csv`, `tsv`, `xls`, `xlsx`, `dta`)[3] and inside the parentheses you place the location of the file on your computer or web URL in quotes
- Be sure to assign your data to a tibble!

```
#my_df<-read_csv("https://metricsf19.classes.ryansafner.com/data/example_data.csv")

my_df<-read_csv("example_data.csv")
```

Looking at Data

- type the name of the tibble to print its contents
- `str()` gives the structure of a tibble
- `head()` prints the first few rows of a tibble
- `View()` will open the tibble in a separate window for inspection[4]
- `glimpse()` gives the structure in a horizontal way

```
my_df
```

```
## # A tibble: 100 x 5
##        X     Z       U Shape        Y
##    <dbl> <dbl>   <dbl> <chr>    <dbl>
##  1  9.74  10.8  0.576  Square    9.60
##  2 11.8   12.0  0.991  Circle    2.58
##  3 10.3   10.7 -0.781  Circle    4.91
##  4  9.03  14.4  0.412  Triangle 24.7
##  5  9.37  18.0 -0.675  Square   34.5
##  6 10.3   14.7  1.23   Square   21.4
##  7  9.57  17.4 -0.0248 Square   32.5
##  8 10.7   16.7 -0.750  Circle   24.6
##  9  9.71  15.5 -0.541  Square   25.0
## 10 10.0   15.0 -0.0772 Square   22.3
## # ... with 90 more rows
```

```
str(my_df)
```

```
## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 100 obs. of  5 variables:
##  $ X    : num  9.74 11.79 10.33 9.03 9.37 ...
##  $ Z    : num  10.8 12 10.7 14.4 18 ...
##  $ U    : num  0.576 0.991 -0.781 0.412 -0.675 ...
##  $ Shape: chr  "Square" "Circle" "Circle" "Triangle" ...
##  $ Y    : num  9.6 2.58 4.91 24.67 34.45 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   X = col_double(),
##   ..   Z = col_double(),
##   ..   U = col_double(),
##   ..   Shape = col_character(),
```

[3] Loading `tidyverse` automatically loads the `readr` package, allowing you to load `csv` and `tsv` files without loading any package. Other file types require loading other packages, such as `readxl` (for Excel files) or `haven` or `foreign` (which are pretty good at reading any other type).

[4] *Do not run this command in a markdown document or it will not knit!*

```
##    ..    Y = col_double()
##    .. )
```

```r
head(my_df)
```

```
## # A tibble: 6 x 5
##        X     Z      U Shape        Y
##    <dbl> <dbl>  <dbl> <chr>    <dbl>
## 1  9.74  10.8  0.576 Square    9.60
## 2 11.8   12.0  0.991 Circle    2.58
## 3 10.3   10.7 -0.781 Circle    4.91
## 4  9.03  14.4  0.412 Triangle 24.7
## 5  9.37  18.0 -0.675 Square   34.5
## 6 10.3   14.7  1.23  Square   21.4
```

```r
glimpse(my_df)
```

```
## Observations: 100
## Variables: 5
## $ X     <dbl> 9.737951, 11.785850, 10.334175, 9.028176, 9.371203, 10.272492 ...
## $ Z     <dbl> 10.76144, 12.02853, 10.72002, 14.41761, 18.03719, 14.68002, 1 ...
## $ U     <dbl> 0.57631430, 0.99107639, -0.78134698, 0.41238420, -0.67469748, ...
## $ Shape <chr> "Square", "Circle", "Circle", "Triangle", "Square", "Square", ...
## $ Y     <dbl> 9.598392, 2.579783, 4.905085, 24.673541, 34.452718, 21.389187 ...
```

General Data Manipulation

The following table provides the major verbs for manipulating data. Further sections below provide examples for i. *subsetting* data ii. *transforming* data iii. *summarizing* data

Verb (from `dplyr`)	Action
`select()`	Keep desired columns (variables)
`filter()`	Keep desired rows (observations)
`arrange()`	Reorder rows (e.g. in numerical order)
`mutate()`	Create new variables
`summarize()`	Collapse data into summary statistics
`group_by()`	Perform any of the above functions by groups/categories

Most verbs allow you to manipulate data according to some **condition(s)**. Popular operators for performing conditional operations are listed in the table below:

Command	Effect
$<$; $>$	Less than; greater than
$\leqslant$; $\geqslant$	Less than or equal to; greater than or equal to
$==$; $\neq$	Is equal to; is not equal to
`%in%`	Is in the set of [a vector of options]
`is.na()`	Is missing (NA)

Finally, for each command, you can alternatively:

```r
# Just view the output
my_df %>% verb()
```

```
# Assign to a different object
my_df_2<-my_df %>% verb()
my_df_2 # then view the output

# Assign to the original object (and overwrite it)
my_df<-my_df %>% verb()
my_df # then view the output
```

Subset Data

- To *subset* data and take only a portion of the data set by some condition(s) for various purposes, use
  - `select()` to subset by columns (variables)
  - `filter()` to subset by rows (observations)

```
# look only at data for "circles" AND where X>10
my_df %>%
  filter(Shape=="Circle",
         X>10)
```

```
## # A tibble: 17 x 5
##        X     Z       U Shape       Y
##    <dbl> <dbl>   <dbl> <chr>   <dbl>
##  1  11.8  12.0  0.991  Circle   2.58
##  2  10.3  10.7 -0.781  Circle   4.91
##  3  10.7  16.7 -0.750  Circle  24.6
##  4  11.5  11.4  0.0275 Circle   1.12
##  5  11.9  12.8  1.02   Circle   4.65
##  6  10.2  15.9  0.0326 Circle  24.7
##  7  11.6  14.3 -1.05   Circle  10.6
##  8  10.4  16.9  0.929  Circle  28.4
##  9  10.6  10.3 -2.05   Circle   0.707
## 10  10.4  13.2 -1.59   Circle  12.3
## 11  11.5  19.2 -0.372  Circle  30.9
## 12  10.9  19.5 -1.54   Circle  33.7
## 13  10.2  15.0 -1.38   Circle  20.3
## 14  11.1  16.7  0.803  Circle  24.4
## 15  10.2  19.6  0.967  Circle  39.0
## 16  11.4  12.4 -0.745  Circle   4.36
## 17  13.0  10.9 -0.232  Circle -12.4
```

```
# look only at X and Y
my_df %>%
  select(X,Y)
```

```
## # A tibble: 100 x 2
##        X     Y
##    <dbl> <dbl>
##  1  9.74  9.60
##  2 11.8   2.58
##  3 10.3   4.91
##  4  9.03 24.7
##  5  9.37 34.5
##  6 10.3  21.4
##  7  9.57 32.5
```

```
## 8 10.7  24.6
## 9  9.71 25.0
## 10 10.0  22.3
## # ... with 90 more rows
```

Transform Data

- Data transformation uses the `mutate()` command to either
  - create a new variable `mutate(new_name = conditions on existing variables)`
  - change a variable (and overwrite it) `mutate(existing_variable = conditions on existing_variable)`

```
# take the log of Y,
# and the log of X,
# and make new variable V, which is 0.5 times X times Z,
# and change the class of Shape from to a character variable to a factor variable

my_df<-my_df %>%
  mutate(log_Y=log(Y),
         log_X=log(X),
         V = 0.5*(X*Z),
         Shape = as.factor(Shape))
my_df
```

```
## # A tibble: 100 x 8
##        X     Z       U Shape        Y log_Y log_X     V
##    <dbl> <dbl>   <dbl> <fct>    <dbl> <dbl> <dbl> <dbl>
##  1  9.74  10.8  0.576  Square    9.60 2.26   2.28  52.4
##  2 11.8   12.0  0.991  Circle    2.58 0.948  2.47  70.9
##  3 10.3   10.7 -0.781  Circle    4.91 1.59   2.34  55.4
##  4  9.03  14.4  0.412  Triangle 24.7  3.21   2.20  65.1
##  5  9.37  18.0 -0.675  Square   34.5  3.54   2.24  84.5
##  6 10.3   14.7  1.23   Square   21.4  3.06   2.33  75.4
##  7  9.57  17.4 -0.0248 Square   32.5  3.48   2.26  83.4
##  8 10.7   16.7 -0.750  Circle   24.6  3.20   2.37  89.7
##  9  9.71  15.5 -0.541  Square   25.0  3.22   2.27  75.4
## 10 10.0   15.0 -0.0772 Square   22.3  3.11   2.30  75.2
## # ... with 90 more rows
```

Summarize Data

- Create summary statistics for datasets with `summarize()`. This will create a tibble of summary statistics.

Below is a table of popular statistics-based commands for summarizing data. Except for the first two, place a variable inside each command, and optionally set it equal to a name for the statistic to be outputted.

Command	Does
`n()`	Number of observations (nothing goes in parentheses!)
`n_distinct()`	Number of unique observations (nothing goes in parentheses!)
`sum()`	Sum all observations of a variable
`mean()`	Average of all observations of a variable
`median()`	50th percentile of all observations of a variable

Command	Does
`sd()`	Standard deviation of all observations of a variable
`min()`	Minimum value of a variable
`max()`	Maximum value of a variable
`quantile(x, 0.25)`	Specified percentile (example 25th percentile) of a variable
`first()`	First value of a variable
`last()`	Last value of a variable
`nth(x, 2)`	Specified position of a variable (example 2nd)

```
# find number of obs, and mean and sd of X and Y
my_df %>%
  summarize(n(),
            Mean_X = mean(X),
            Std_dev_X = sd(X),
            Mean_Y = mean(Y),
            Std_dev_Y = sd(Y))
```

```
## # A tibble: 1 x 5
##   `n()` Mean_X Std_dev_X Mean_Y Std_dev_Y
##   <int>  <dbl>     <dbl>  <dbl>     <dbl>
## 1   100   10.2      1.05   21.5      12.4
```

Grouped-Summaries

- You can run summary statistics by group by first using `group_by(categorical_variable)` and then `summarize()`:

```
# get mean of X and Y for each Shape
my_df %>%
  group_by(Shape) %>%
  summarize(mean_X = mean(X),
            mean_Y = mean(Y))
```

```
## # A tibble: 3 x 3
##   Shape    mean_X mean_Y
##   <fct>     <dbl>  <dbl>
## 1 Circle    10.4    21.1
## 2 Square     9.96   21.3
## 3 Triangle  10.3    22.4
```

Categorical Data

- For categorical data (`factors`), you can quickly produce a frequency table of each category with `count(factor_variable_name)`
- `distinct()` shows the distinct values of a specified variable (often useful for finding the different categories)

```
# count by shape
my_df %>%
  count(Shape)
```

```
## # A tibble: 3 x 2
##   Shape        n
##   <fct>    <int>
## 1 Circle      27
```

```
## 2 Square      47
## 3 Triangle    26
```

```
# get the distinct shapes
my_df %>%
  distinct(Shape)
```

```
## # A tibble: 3 x 1
##   Shape
##   <fct>
## 1 Square
## 2 Circle
## 3 Triangle
```

Correlation

- You can quickly produce a correlation table (2+ variables) so long as they are `numeric` (i.e. not `character` or `factor`):

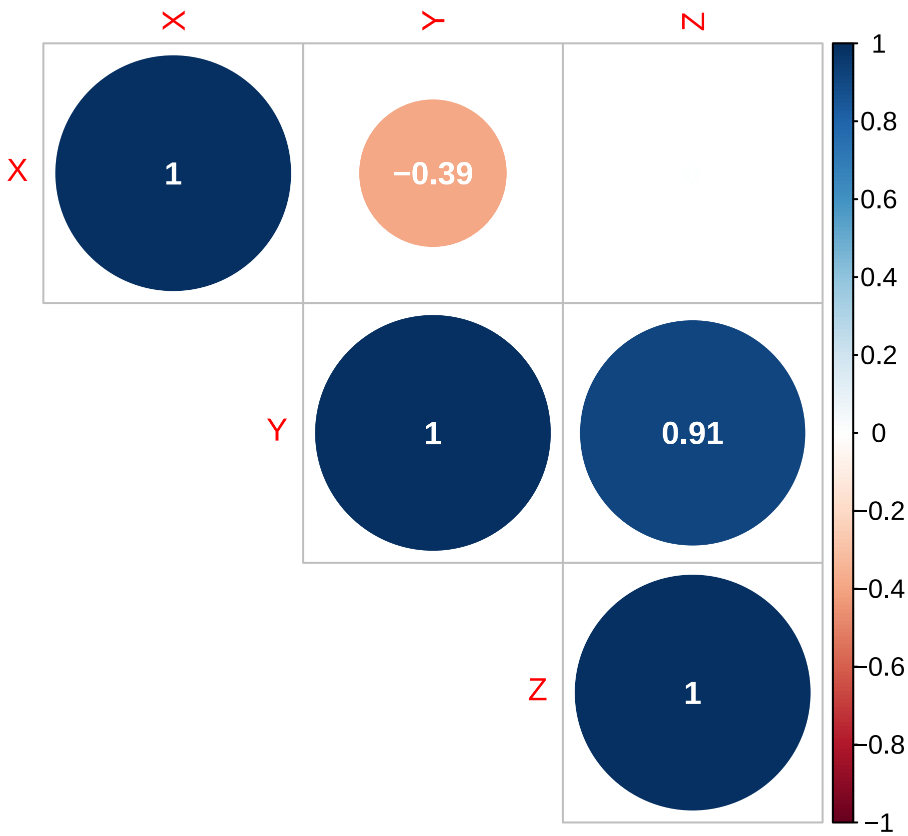
```
my_df %>%
  select(X,Y,Z) %>%
  cor()
```

```
##              X          Y           Z
## X  1.000000000 -0.3872977 0.007279217
## Y -0.387297726  1.0000000 0.912385345
## Z  0.007279217  0.9123853 1.000000000
```

- A nice **correlogram** can be made with the `corrplot` package:

```
library(corrplot)

my_df %>%
  select(X,Y,Z) %>%
  cor() %>%
  corrplot(.,
           method = "circle", # choose circle, square, ellipse, number, pie, shade, color
           type = "upper", # show only upper triangle of matrix
           addCoef.col="white") # add white numbers
```


X
Y
Z
X
Y
Z
1
−0.39
1
0.91
1
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1

Regression

The primary task in econometrics is running a regression on data. Regression uses the linear model `lm()` command where "Y" is regressed on all X variables, connected with +s, where the data is sourced from your tibble.

```
lm(Y~X+Z, data = my_df)
```

This will output the coefficients only. To get full information on coefficients, standard errors, hypothesis testing, and regression fit, pipe into the `summary()` command, or save the regression as an object and then run `summary()` on it. I show both methods below:

```
my_df %>%
  lm(data = ., # pipes my_data into the data = argument
     Y~X+Z) %>%
  summary()
```

```
##
## Call:
## lm(formula = Y ~ X + Z, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.2913 -0.8377 -0.0387  0.7535  3.7508
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 13.51488    1.51898   8.897 3.22e-14 ***
## X           -4.65217    0.13328 -34.904  < 2e-16 ***
## Z            3.65230    0.04504  81.089  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.397 on 97 degrees of freedom
## Multiple R-squared:  0.9876, Adjusted R-squared:  0.9874
## F-statistic:  3877 on 2 and 97 DF,  p-value: < 2.2e-16
```

```
my_reg_1<-lm(Y~X+Z, data = my_df)
summary(my_reg_1)
```

```
##
## Call:
## lm(formula = Y ~ X + Z, data = my_df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.2913 -0.8377 -0.0387  0.7535  3.7508
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 13.51488    1.51898   8.897 3.22e-14 ***
## X           -4.65217    0.13328 -34.904  < 2e-16 ***
## Z            3.65230    0.04504  81.089  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.397 on 97 degrees of freedom
## Multiple R-squared:  0.9876, Adjusted R-squared:  0.9874
```

```
## F-statistic:  3877 on 2 and 97 DF,  p-value: < 2.2e-16
```

Interpretation of Output

- The top row `Residuals` describes the distribution of the residuals.
- The `Coefficients` table describes the OLS parameters $\hat{\beta}_j$'s, where each row is a right-hand side variable, starting with `(Intercept)` ($\hat{\beta}_0$), then ($\hat{\beta}_1$), etc.
  - `Estimate` column describes the OLS parameters ($\hat{\beta}_0, \hat{\beta}_1, \cdots$)
  - `Std. Error` column describes the standard error of each parameter
  - `t value` column describes the test statistic for a hypothesis test where H_0 : that particular $\hat{\beta}_j = 0$
  - `Pr(>|t|)` column is the *p*-value on that hypothesis test for that parameter (roughly, we're looking for it to be less than 0.05, if it is, there will be `*` stars to the right of it.
- The bottomw three rows describes the goodness of fit of the regression
  - `Residual standard error` is the Standard Error of the Regression σ_u
  - `Multipled R-squared` is R^2, `Adjusted R-squared` is $\bar{R}^2$
  - `F-statistic` is the F-statistic on an *All F-test* (all betas are equal to 0), and associated `p-value` on that test

Tidying Output with Broom

- The `broom` package allows us to output regressions into tidy tibbles that we can easily print, work with, and extract individual parameters or statistics from for further analysis

Command	Does
`tidy()`	Takes the saved `reg_object` and makes a tibble of the coefficients table only
`augment()`	Create dataset with calculated values (e.g. `.fitted`, `.resid`)
`glance()`	Get statistics of regression fit (e.g. `r.squared`, `sigma`)

```r
# load broom
library(broom)

# tidy to get coefficients in a tidy tibble
my_reg_1_tidy<-tidy(my_reg_1)
my_reg_1_tidy
```

```
## # A tibble: 3 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    13.5      1.52        8.90 3.22e-14
## 2 X              -4.65     0.133     -34.9  1.02e-56
## 3 Z               3.65     0.0450     81.1  6.18e-91
```

```r
# glance (original lm object) to view statistics
glance(my_reg_1)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <int>  <dbl> <dbl> <dbl>
## 1     0.988         0.987  1.40     3877. 2.87e-93     3  -174.  356.  366.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

```
# "r.squared" and "adj.r.squared" are self-explanatory
# "sigma" is the Standard Error of the Regression (SER)
# "statistic" is the F-statistic on the All-F test
# "p.value" is the p-value from that All-F test

my_reg_1_aug<-augment(my_reg_1)
my_reg_1_aug
```

```
## # A tibble: 100 x 10
##        Y     X     Z .fitted .se.fit  .resid   .hat .sigma    .cooksd .std.resid
##    <dbl> <dbl> <dbl>   <dbl>   <dbl>   <dbl>  <dbl>  <dbl>      <dbl>      <dbl>
##  1  9.60  9.74  10.8    7.52   0.248  2.08   0.0315   1.39 0.0249          1.51
##  2  2.58 11.8   12.0    2.62   0.293 -0.0372 0.0441   1.40 0.0000114      -0.0272
##  3  4.91 10.3   10.7    4.59   0.244  0.314  0.0306   1.40 0.000547        0.228
##  4 24.7   9.03  14.4   24.2    0.209  0.502  0.0223   1.40 0.00100         0.363
##  5 34.5   9.37  18.0   35.8    0.219 -1.34   0.0246   1.40 0.00797        -0.974
##  6 21.4  10.3   14.7   19.3    0.142  2.05   0.0103   1.39 0.00755         1.47
##  7 32.5   9.57  17.4   32.7    0.192 -0.203  0.0188   1.40 0.000138       -0.147
##  8 24.6  10.7   16.7   24.6    0.174  0.0518 0.0155   1.40 0.00000734      0.0374
##  9 25.0   9.71  15.5   25.1    0.154 -0.0983 0.0121   1.40 0.0000204      -0.0708
## 10 22.3  10.0   15.0   21.7    0.141  0.652  0.0102   1.40 0.000758        0.469
## # ... with 90 more rows
```

```
# ".fitted" are predicted (Y-hat) values from the model
# ".resid" are the residuals (u-hat) for each X-value
```

- `augment` is particularly useful for plotting fitted or residual values, as in a residual plot, where you can set `aes(y = .resid)`.

Extensions

Categorical Data: Dummy Variables

For categorical data, you can run a regression with a dummy variable if that variable takes on the values of 0 or 1.

If the variable has multiple possible categories, you can use (or make) a dummy variable for *each* of the n categories and include all $n - 1$ dummy variables in the regression (to avoid the dummy variable trap!).

If your variable exists as a `factor` variable (e.g. the value of each observation for that variable is the name of the category), you can simply add that variable in the regression and R will automatically create a dummy for each category and include $n - 1$ dummies in a regression:

```
# run a regression with Shape, a factor variable
## which has "Circle," "Triangle," and "Square" for categories

shape_reg<-lm(Y~Shape, data = my_df)
summary(shape_reg)
```

```
##
## Call:
## lm(formula = Y ~ Shape, data = my_df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -33.446  -8.783  -0.010  10.234  21.661
##
## Coefficients:
```

```
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     21.069      2.416   8.720 7.74e-14 ***
## ShapeSquare      0.206      3.032   0.068    0.946
## ShapeTriangle    1.319      3.450   0.382    0.703
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.55 on 97 degrees of freedom
## Multiple R-squared:  0.00182,    Adjusted R-squared:  -0.01876
## F-statistic: 0.08841 on 2 and 97 DF,  p-value: 0.9155
```

```r
# Note R made two dummies:
# # ShapeTriangle for Triangle (0 or 1)
# # ShapeSquare for Square (0 or 1)
# # and left out Circle as the reference category
```

Interpretation:

- $\hat{\beta}_0$ is the average value of Y for the reference category
  - e.g. `Circles` have an average Y of 19.63
- Each $\hat{\beta}$ is the *difference* between that category and the reference category
  - e.g. `Triangle` has an average Y that is 4.66 larger than `Circle`
  - e.g. `Square` has an average Y that is 0.15 larger than `Circle`

Interaction Terms

To interact two variables in a regression and create an interaction term, simply add them to the regression with `:` or `*` between them.

```r
interact_reg<-lm(Y~X+Z+X:Z, data = my_df)
summary(interact_reg)
```

```
##
## Call:
## lm(formula = Y ~ X + Z + X:Z, data = my_df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.01411 -0.75425 -0.06623  0.81317  2.76908
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 47.28197    5.33968   8.855 4.28e-14 ***
## X           -7.99364    0.52518 -15.221  < 2e-16 ***
## Z            1.28283    0.36586   3.506 0.000693 ***
## X:Z          0.23436    0.03599   6.511 3.41e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.169 on 96 degrees of freedom
## Multiple R-squared:  0.9914, Adjusted R-squared:  0.9912
## F-statistic:  3701 on 3 and 96 DF,  p-value: < 2.2e-16
```

Polynomial Models

To run a polynomial regression, simply add a higher order variable, which you can first `mutate()`, or use the `I()` command to create a quadratic (or higher order) term in your regression:

```
reg_quad<-lm(Y~X+I(X^2), data = my_df)
summary(reg_quad)
```

```
##
## Call:
## lm(formula = Y ~ X + I(X^2), data = my_df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -20.487  -8.755  -1.634  11.213  19.902
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -120.513     75.032  -1.606   0.1115
## X             32.862     14.779   2.224   0.0285 *
## I(X^2)        -1.839      0.724  -2.540   0.0127 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.22 on 97 degrees of freedom
## Multiple R-squared:  0.203,  Adjusted R-squared:  0.1866
## F-statistic: 12.35 on 2 and 97 DF,  p-value: 1.663e-05
```

Logarithmic Models

To run a regression with a logged variable, you can first `mutate()` the logged variable, or use the `log()` command to create a logged variable in your regression:

```
reg_log_log<-lm(log(Y)~log(X), data = my_df)
summary(reg_log_log)
```

```
##
## Call:
## lm(formula = log(Y) ~ log(X), data = my_df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9873 -0.2975  0.0962  0.6634  1.1164
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  10.7762     1.9425   5.548 2.51e-07 ***
## log(X)       -3.4454     0.8396  -4.103 8.48e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.85 on 97 degrees of freedom
##   (1 observation deleted due to missingness)
## Multiple R-squared:  0.1479, Adjusted R-squared:  0.1391
## F-statistic: 16.84 on 1 and 97 DF,  p-value: 8.482e-05
```

Regression Output Tables

There are several methods to make regression output tables, but I have found best use with the `huxtable` package's `huxreg()` command. There are three parts to the command (each separated by commas):

1. Add your regression `lm` objects, separated by commas.

- Optionally define a `"column name" = your_lm_object` for each.

2. Optionally rename and omit your X-variables as desired inside `coefs = c()`

- To change a variable's name down each row, set `"desired name" = var_name`.
- Any X-variable not listed inside `coefs = c()` will be omitted from the table!

3. Optionally rename and omit statistics as desired inside `stats = c()`

- To change a variable's name down each row, set `"desired name" = stat_name`.
- Any statistics not listed inside `stats = c()` will be omitted from the table!

```
# default example
library(huxtable)
my_reg_1 %>% huxreg()
```

	(1)
(Intercept)	13.515 ***
	(1.519)
X	-4.652 ***
	(0.133)
Z	3.652 ***
	(0.045)
N	100
R2	0.988
logLik	-173.792
AIC	355.584

*** $p < 0.001$; ** $p < 0.01$; * $p < 0.05$.

```
# heavily customized
library(huxtable)
huxreg(my_reg_1,
       shape_reg,
       interact_reg,
       reg_quad,
       reg_log_log,
       coefs = c("Constant" = "(Intercept)",
                 "X" = "X",
                 "Z" = "Z",
                 "Square" = "ShapeSquare",
                 "Triangle" = "ShapeTriangle",
                 "X:Z", "X:Z",
                 "$X^2$" = "I(X^2)",
                 "ln(X)" = "log(X)"),
       statistics = c("N" = "nobs",
                      "$R^2$" = "r.squared",
                      "SER" = "sigma"),
       note = NULL, # suppress footnote for stars, to insert Fixed Effects Row below
       number_format = 3) %>% # round to three decimals
```

```r
# Adding a lot of custom parts to table below

add_rows(c("Fixed Effects", "None", "Shape", "None", "None", "None"), # add fixed effects row
       after = 17) %>% # insert after 17th row

# allow math to render as R^2 and X^2
set_escape_contents(c(14,20), c(1,1), FALSE) %>% # R^2 is rows 14 and 20, column 1

# add centered "Y" in second row
insert_row(c("",rep("Y",4)),"ln(Y)", after = 1) %>%
merge_cells(c(2,2,2), 2:5) %>%

# create borders
set_all_borders(0) %>% # remove all borders to manually set my own
set_top_border(1, 1:6, 2) %>% # add border size 2 to first row, columns 1:6
set_top_border(2, 1:6, 1) %>% # add border size 1 to second row, columns 1:6
set_top_border(3, 2:5, 1) %>% # add border size 1 to second row, columns 2:5
set_top_border(19, 2:6, 1) %>% # add border size 1 to second row, columns 2:5
set_bottom_border(22, 1:6, 2) %>% # add border size 2 to 22nd row, columns 1:6

# caption the table
set_caption( "Regression Results")
```

Table 7: Regression Results

	(1)	(2)	(3)	(4)	(5)
	Y				ln(Y)
Constant	13.515 ***	21.069 ***	47.282 ***	-120.514	10.776 ***
	(1.519)	(2.416)	(5.340)	(75.032)	(1.942)
X	-4.652 ***		-7.994 ***	32.862 *	
	(0.133)		(0.525)	(14.779)	
Z	3.652 ***		1.283 ***		
	(0.045)		(0.366)		
Square		0.206			
		(3.032)			
Triangle		1.319			
		(3.450)			
X:Z			0.234 ***		
			(0.036)		
X^2				-1.839 *	
				(0.724)	
ln(X)					-3.445 ***
					(0.840)
Fixed Effects	None	Shape	None	None	None
N	100	100	100	100	99
R^2	0.988	0.002	0.991	0.203	0.148
SER	1.397	12.554	1.169	11.218	0.850

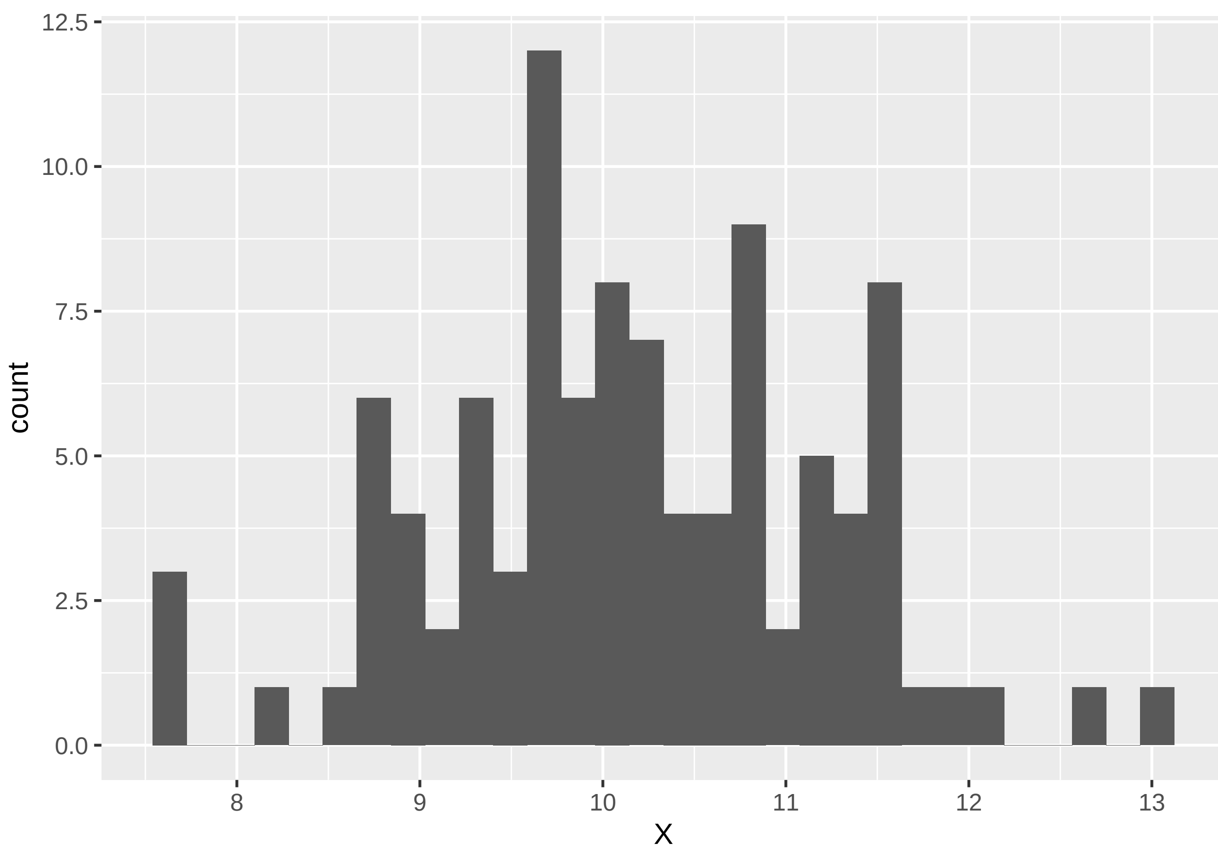
Plots

Plots of any kind can be made with `ggplot2`, which uses a “grammar of graphics” to build plots layer by layer. Some possible layers are described in the table below, *required* layers are boldened:

Command	Layer	Description
`data =`	**Data**	Defines what tibble to use for the data
`+aes()`	**Aesthetics**	Defines what variables from data will be mapped to markings
`+geom_*()`	**Geometry**	Defines what markings to make, e.g. `point`, `histogram`, `line`, `smooth` (for regression lines)
`+coord_*()`	Coordinates	Scales for axes
`+scale_*()`	Scales	Define the range of values
`+facet_*()`	Facets	Group into subplots

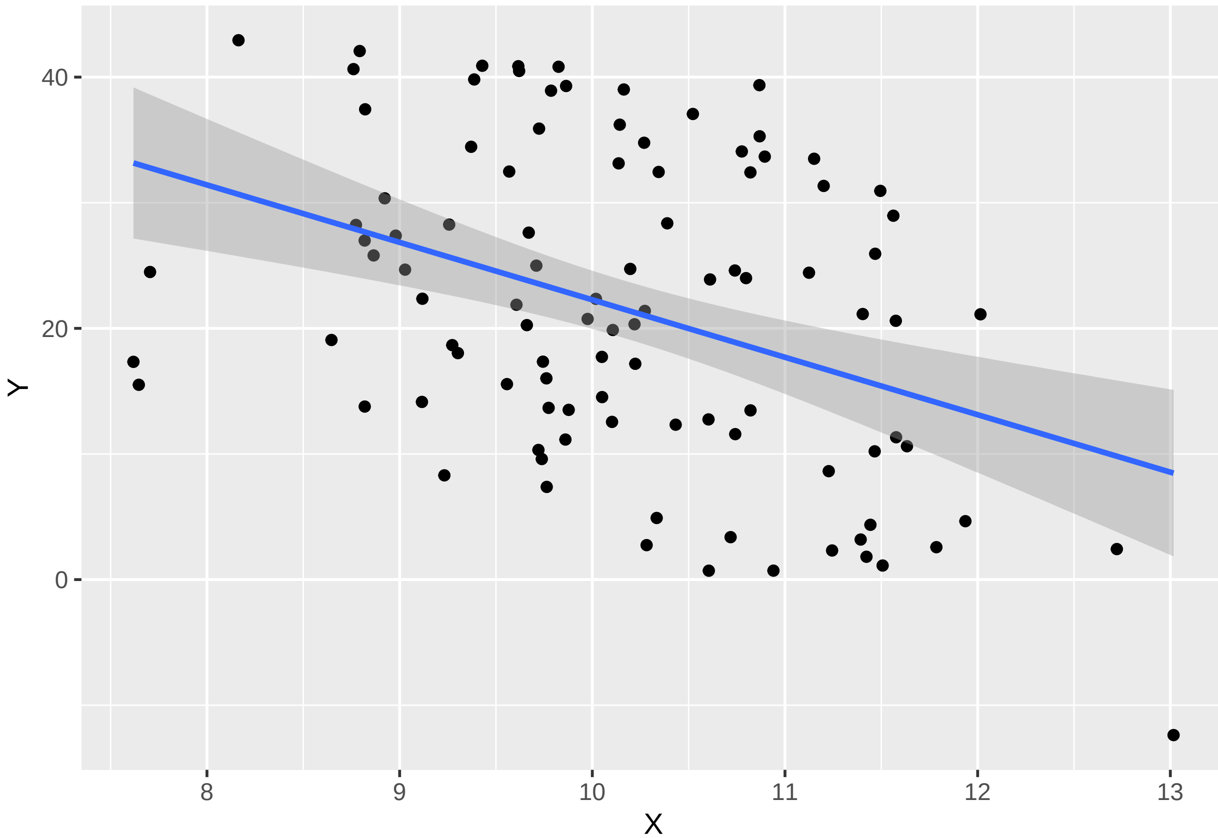
Histogram

```
ggplot(data = my_df)+
  aes(x = X)+
  geom_histogram()
```



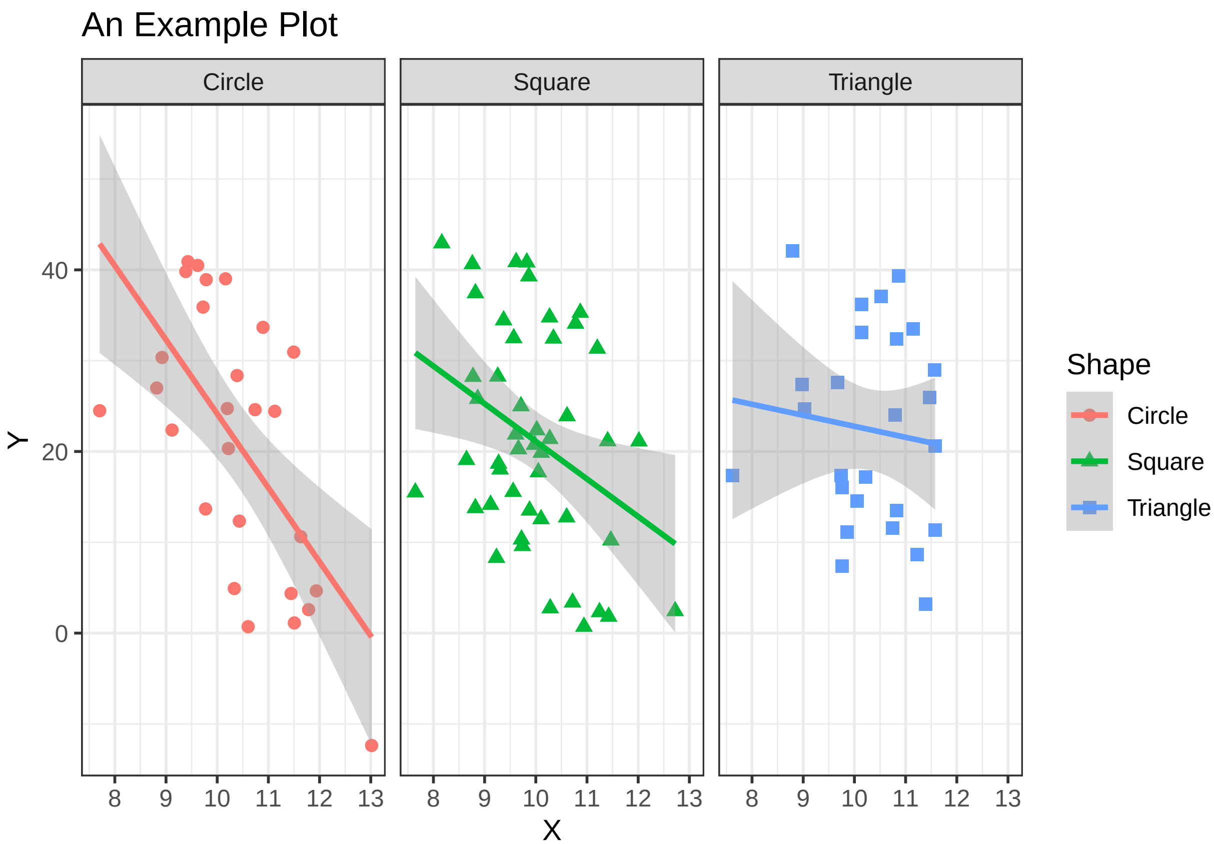
Scatterplot (with Regression Line)

```
ggplot(data = my_df)+
  aes(x = X,
      y = Y)+
  geom_point()+
  geom_smooth(method = "lm")
```



Customized Example

```
ggplot(data = my_df)+
  aes(x = X,
      y = Y)+
  geom_point(aes(shape = Shape,
                 color = Shape),
             size = 2)+
  geom_smooth(method = "lm",
              aes(color = Shape))+
  labs(x = "X",
       y = "Y",
       title = "An Example Plot")+
  facet_wrap(~Shape)+
  theme_bw()
```


An Example Plot
Circle
Square
Triangle
Y
X
40
20
0
8
9
10
11
12
13
Shape
Circle
Square
Triangle