ECON 557 – Advanced Data Analysis

Michael T. Sandfort

Department of Economics
Masters in Applied Economics Program
Georgetown University

January 13, 2023

Preliminaries

Who We Are

- Professor: Michael Sandfort (ms4232)
- TA: Pranjal Pranjal (pp712)
- You:
  - Have taken Econometrics (ECON553) and Data Analysis (ECON554), and want still more. (!)
  - Have a basic familiarity with calculus, matrix algebra, probability and statistical inference.

Why We Are Here

- Extend your current skill set for data description, prediction and inference.
- Contemporary tools are classified with all sorts of exciting names like "data science," "machine learning," and "statistical learning," usually in the context of work with "big data."
- But most of these tools have a long pedigree in econometrics, statistics and computer science.
- Many are also related to one another in ways that may be surprising.
- We will emphasize the connections between some contemporary tools (e.g., LASSO) and tools that are already in your toolbox (e.g., regression).
- At the end of the course, you will be expected both to know how to apply the estimators we discuss as well as to understand their advantages and disadvantages.

Expectations and Assessment

- Read the syllabus!
- Weekly homeworks (60%).
- Midterm (20%) and final exam (20%).
- Attendance and participation.

Homeworks

- Homeworks are an important part of the course.
- Each will give you an opportunity to develop a better understanding of a topic raised in class and/or experience applying an estimator in practice.
- I expect the solution each of you turns in to be the product of your own reflection on the course material.
- Homeworks are typically due at the start of class on the due date.
- Submit via Canvas. Late submissions are not accepted. Network connections are sometimes flaky at the least advantageous moment – please do not wait until 4:59:59 to submit your assignment.
- I have a fairly formal process for homework submission – please adhere to it.

Submitting Your Work

- You must submit THREE files:
  - (A) **Discussion (*.pdf)**: Most important part. Should contain your answer to each question posed in the assignment. I should not need to read either of the other two files to find your answers. **Please note that raw regression output (even if highlighted or marked up) does not constitute a discussion of your results – I will not review raw regression output without the context of a discussion.** Please name this file `LastName_FirstName_HWnn_A.pdf`.
  - (B) **Code (*.R)**: The **R** code implementing any estimations or calculations you performed in support of your discussion. Please name this file `LastName_FirstName_HWnn_B.R`.
  - (C) **Session transcript (*.txt)**: The transcript resulting from running the above **R** code. This file should contain, e.g., the coefficient estimates supporting your discussion. Please name this file `LastName_FirstName_HWnn_C.txt`.

What If I Have A Question about Course Material?

- Great! Having questions is a feature, not a bug in the learning process.
- Ask in class – class participation has externalities.
- Email your TA or instructor.
- If it's something that can't be hashed out efficiently in an email, we can set up a virtual room to talk through it.
- I will try to address questions about administrative issues of broad interest (e.g., "What's going to be on the exam?", "Did the instructor make a typo in the solution key for PS4?") at the start of class or during breaks.
- Other questions not about the course material (e.g., individual assessments, grades) should be handled individually by email directly with the instructor.

Lecture 1: Basic Data Analysis with R/RStudio

The R Language: Statistics and Programming

- A language for "computing with data," designed to "turn ideas into software, quickly and faithfully."
- R is an interpreted language (like Stata or SAS), not a compiled language (like Fortran, C or Java). But its functions can be linked to compiled code in other languages (e.g., C++ via `Rcpp`) to speed execution.
- R defaults to interactive mode, but can be made to read scripts using the `source()` command. If you use R this way, you will need a separate "text editor" to write scripts.
- At the ">" prompt, type in an expression and you get a result – and a fresh prompt.

RStudio is an integrated development environment (IDE) for R. It contains:

- An R console (the "interactive mode" described above);
- A script editor;
- A viewer window (for plots and Rmarkdown documents);
- An environment browser.

Data Analysis: Typical Workflow

A typical workflow for data analysis includes the following steps:

- Identify physical media/logical properties of data.
- Make data available to analytical tools ("ingest", "import").
- Examine, transform, and analyze data.
- Discuss and publish results.

Gross Data File Structure

When a data set comes across your desk, you may have to investigate (however briefly) its gross structure.

With some file types (e.g., *.csv, *.xls[x], *.dta) this process is so ingrained you may not even think about it.

The most common issues relate to:

- File format
- File encoding
- End-of-record and end-of-file markers (esp. embedded CR)
- Field delimiters (esp. embedded commas)
- Field metadata (esp. dates, number-like strings, and fixed-precision numerics)

The available tools for assessing these issues are, unfortunately, somewhat OS-dependent. We won't have much to say about them in this course.

Data File Ingest/Import

This is the point in the workflow where your data file (the serialized stream of 0's and 1's, either on disk or over the wire) becomes a structured object, typically held in memory (RAM). Examples:

- Excel will "read" its native *.xls[x] or "import" a *.csv file into a workbook/worksheet for interactive analysis.
- Stata will `use` its native *.dta or `insheet` a *.csv file into a data set for interactive analysis.
- R will `load()` its native *.rda or `read_csv()` a *.csv file into a data frame for interactive analysis.

The end of each of these procedures is a data structure in memory. In R, most of the `read` family of operations yield a data.frame (or a tibble).

The `data.frame` is itself built on two even more fundamental data structures in R: the list and the vector.

Examine, Transform and Analyze

Once you have a `data.frame` or a `tibble` in R, you can use several tools to look it over, including:

- `str()` will show the row count and each variable's name and type.
- `head()` will show the first few rows of your data (six rows, by default).

After looking it over, you will often find your data...

- ...is disaggregated, when you want it grouped by some feature;
- ...is unsorted, when you want it sorted by some alpha/numeric;
- ...is in "wide" format, when you want it "long" (or vice versa);
- ...contains irrelevant dates/regions, when you want only a subset;
- ...is missing key calculated fields.

For transforming data, a full suite of tools is available in the dplyr package, which I highly recommend. We'll work through a few examples shortly.

Most of this course will be spent on the "analyze" step – our example will demonstrate one mode of analysis you should be very familiar with (OLS) momentarily.

Discuss, Publish and Update

One aim of this course is to give you the tools to discuss and explain your quantitative analysis in a thoughtful way, including, for example, the results from related models as context.

The format for "publishing" your results will depend to a large degree on your workplace. For this course, I recommend RMarkdown within RStudio (free) as a fairly lightweight tool for integrating discussion with results. This makes it easy to find the code that generated your results, in the event you want to go back and review your results ("How did I do this?") at the end of the semester.

Professionally, you should also be prepared to update your work as new data arrives or as peers, editors or reviewers suggest alternative modeling approaches. A source code management (SCM) system is a very useful way to manage revisions and track different versions of your work and the underlying data. It is fairly straightforward to integrate SCM (e.g., git) with RStudio, but this topic is outside the scope of our course.

Example: Gross Data File Structure

If we have time, a couple of examples exploring issues that can arise with gross file structure.

Unix `bash` tools: `file`, `head`, `tail`, `grep`, `iconv`, `od`, `wc`.

For Unicode basic multilingual plane, examine the file `hello.txt`.

For delimiters and encoding, look at the USGS populated places data set `POP_PLACES_20210825.txt`. Is it all ASCII?

Example: Data File Ingest/Import

The "data files" in this example are from the US Census Bureau – longitudinal weights from the Survey of Income and Program Participation (SIPP) at `https://www.census.gov/programs-surveys/sipp/data/datasets/2021-data/2021.html`.

They are useful for this example because the same information is provided in several different file formats, all commonly in use. Other than that, we won't be making use of these files.

Example: Data File Ingest/Import

There is support in "base R" for loading/saving native R files and for import of delimited and fixed width files. The "CSV" file provided is actually delimited with pipes (|) rather than with commas:

```
> myData <- read.csv("lgtwgt2021yr4.csv")
> head(myData,2) # Oops!
             ssuid.pnum.spanel.finyr4
1 00011455225318|101|2018|32552.637126
2  00011481677018|101|2018|9546.924851
> myData <- read.delim("lgtwgt2021yr4.csv",header=TRUE,sep="|")
> head(myData,2) # Much better!
        ssuid pnum spanel    finyr4
1 11455225318  101   2018 32552.637
2 11481677018  101   2018  9546.925
```

Save the file as older "RData" (*.rda) file:

```
> save(myData,file="lgtwgt2021yr4.rda")
> rm(myData) # Delete the data file from memory after saving
> head(myData,2) # Now this fails (myData doesn't exist)
Error in head(myData, 2):  object 'myData' not found
> load("lgtwgt2021yr4.rda") # (Re)load the data
```

Also support for newer "RDS" format (default for some packages now). See `saveRDS()`.

Example: Data File Ingest/Import

The readr package contains some updated versions of some of these base functions and delivers a tibble rather than a data.frame.

```
> myData <- read_delim("lgtwgt2021yr4.csv") # NOT read.delim()!
Error in read_delim("lgtwgt2021yr4.csv"):  could not find function "read_delim"
> library(readr) # This package defines the function read_delim()
> myData <- read_delim("lgtwgt2021yr4.csv")
Rows:  15070 Columns:  4
-- Column specification --------------------------------------------------------
Delimiter:  "|"
chr (1):  ssuid
dbl (3):  pnum, spanel, finyr4
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.
> myData
# A tibble: 15,070 x 4
   ssuid           pnum spanel finyr4
   <chr>          <dbl>  <dbl>  <dbl>
 1 00011455225318   101   2018 32553.
 2 00011481677018   101   2018  9547.
 3 00011481677018   102   2018 12955.
 4 00013345043118   101   2018 32997.
 5 00013355998818   101   2018 15533.
 6 00028503462618   101   2018 14267.
 7 00028503462618   102   2018 15070.
 8 00028503464618   101   2018 24074.
 9 00028504944018   101   2018 21041.
10 00028507348618   101   2018 11697.
# ... with 15,060 more rows
```

Note the difference between the underscore and dot in the function name! (More on packages in a bit.)

Example: Data File Ingest/Import

Import of Stata, SAS, and SPSS files available through the `haven` package. As above, these yield a tibble:

```
> library(haven)

> myData_Stata <- read_dta("lgtwgt2021yr4.dta")
> head(myData_Stata,2)
# A tibble: 2 x 4
  spanel ssuid           pnum finyr4
   <dbl> <chr>          <dbl>  <dbl>
1   2018 00011455225318   101 32553.
2   2018 00011481677018   101  9547.

> myData_SAS <- read_sas("lgtwgt2021yr4.sas7bdat")
> head(myData_SAS,2)
# A tibble: 2 x 4
  ssuid           pnum spanel finyr4
  <chr>          <dbl>  <dbl>  <dbl>
1 00011455225318   101   2018 32553.
2 00011481677018   101   2018  9547.
```

All the same data, but packaged in different file formats.

Example: Data File Ingest/Import

Import of Excel (*.xls[x]) files available through the `readxl` package.

This information on cyclically adjusted price-earnings (CAPE) ratios comes from Robert Shiller at Yale (`http://www.econ.yale.edu/~shiller/data.htm`).

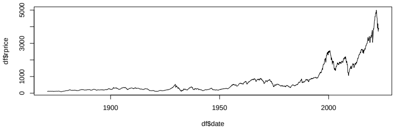
As you can see, the first sheet isn't very informative. The next slide shows that it takes a little work to make this information usable. We'll cover a few of these basic data manipulation techniques next.

```
> library(readxl)
> read_excel("ie_data.xls")
# A tibble: 0 x 1
# ... with 1 variable:
#   The data and CAPE Ratio on this spreadsheet were developed by Robert J. Shiller using various public sources.  Neither Robe
```

Example: Data File Ingest/Import

The more realistic version:

```
> df <- read_excel("ie_data.xls")
> # df <- read_excel("ie_data.xls",sheet="Data")
> # df <- read_excel("ie_data.xls",sheet="Data",skip=8)
> # df <- read_excel("ie_data.xls",sheet="Data",skip=8,col_names=F)
> suppressMessages(
+   df <- read_excel("ie_data.xls",sheet="Data",skip=8,col_names=F,col_types="text")
+ )
> df$date_text <- paste0(substr(df$"...1",1,4),"-",as.character(round(100*as.numeric(substr(df$"...1",5,8),0))))
> df$date_text[1]
[1] "1871-1"
> df$date <- as.Date(paste0(df$date_text,"-01"),"%Y-%m-%d")
> df$date[1]
[1] "1871-01-01"
> df$rprice <- as.numeric(df$"...8")
> plot(df$date,df$rprice,type="l")
```



Simple Stuff

- Arithmetic

```
> 1 + 2
[1] 3
```

- Operator precedence

```
> 2 + 3 * 4
[1] 14
```

- Exponents

```
> 2^3
[1] 8
```

- Basic algebra and trig functions

```
> exp(1)
[1] 2.718282
> sqrt(10)
[1] 3.162278
```

- A few constants

```
> pi
[1] 3.141593
```

Atomic Data Types

- Logical (Boolean: `TRUE` or `FALSE`; can be abbreviated `T` or `F`)
- Integer
- Numeric (double-precision)
- Complex (we won't use this)
- Character string (string)

Type conversion via `as.numeric()` and `as.character()`.

There is a largest representable integer

```
> .Machine$integer.max
[1] 2147483647
```

and a numeric "machine precision"

```
> .Machine$double.eps
[1] 2.220446e-16
```

Atomic Data Types

- Logical

```
> a <- T # T (or TRUE) and F (or FALSE)
> typeof(a)
[1] "logical"
```

- Integer

```
> b <- 1L # The "L" is needed to mark integers
> typeof(b)
[1] "integer"
```

- Double

```
> c <- 1
> typeof(c)
[1] "double"
```

- Character string

```
> d <- "Econometrics"
> typeof(d)
[1] "character"
```

- Type conversion

```
> as.character(b)
[1] "1"
> z <- as.numeric(as.character(b))
> typeof(z)
[1] "double"
```

String Functions

Working with character strings can sometimes be challenging:

- Encoding information is often missing.
- Truncation of text fields is common.
- Monetary units and other punctuation are often included (though unwanted) in otherwise numeric values.
- Different upper/lower case conventions for the same field may be adopted in different data sets.
- Minor variations in spelling can mess up a text match.

You can save yourself a lot of grief by always:

- Store unique identifiers as character strings.
- Store postal codes (e.g., US Zipcodes) as character strings.

String Functions

- Concatenate strings

```
> paste("ab","de",sep="ZZZ")
[1] "abZZZde"
> paste0("ab","de") # paste0 uses null separator
[1] "abde"
> paste(c("1","2","3"),sep="-") # ?
[1] "1" "2" "3"
> paste(c("1","2","3"),sep="-",collapse=" ") # paste is vectorized
[1] "1 2 3"
```

- Character substitution

```
> sub(",","","$15,000,000,000")
[1] "$15000,000,000"
> gsub(",","","$15,000,000,000")
[1] "$15000000000"
> gsub("$","","$15,000,000,000")
[1] "$15,000,000,000"
> gsub("[$,]","","$15,000,000,000")
[1] "15000000000"
```

- Pattern matching

```
> grep("a",c("abcdefedcba","aba","mm","a"))
[1] 1 2 4
```

Vectors

- A vector is an array of atomic data types.
- In a given vector, all elements must be of the same atomic data type – if not they will be silently "promoted."
- Vectors are a simple data structure – you can type in a "vector of vectors" but it will be automatically flattened to a non-nested array. This can be useful to "grow" an existing vector.
- Vectors in R are NOT the same as a matrix with one dimension equal to one (common "gotcha").
- Most often created with `c()`, `rep()`, or `seq()`.
- Most math functions automatically map over vectors (no loops required)

Vectors

- Create a numeric vector

```
> v <- c(1,2,3)
> v
[1] 1 2 3
```

- Try this with mixed atomic types (promotion)

```
> s <- c(1,2,"3")
> s
[1] "1" "2" "3"
```

- Attempt to create a nested vector; “grow” it

```
> v <- c(c(1,2),c(3,4)); v
[1] 1 2 3 4
> v <- c(v,5); v
[1] 1 2 3 4 5
```

- `rep()` is very flexible

```
> rep(c(1,2),times=5)
 [1] 1 2 1 2 1 2 1 2 1 2
> rep(c(1,2),each=5)
 [1] 1 1 1 1 1 2 2 2 2 2
```

- Math functions mapped over vectors automatically

```
> x <- seq(0,2*pi,by=pi/2)
> sin(x)
[1]  0.000000e+00  1.000000e+00  1.224647e-16 -1.000000e+00 -2.449294e-16
```

Getting and Setting Vector Elements

- The `[]` operator selects elements from a vector.
- The argument to `[]` can be an index or a vector of indices.
- Negative index (or vector of indices) gets all elements except those indexed.
- Can also use a logical vector to index.
- The `[]<-` operator assigns (overwrites) elements.

Getting and Setting Vector Elements

- Start with a vector and simple index

```
> x <- c(1,3,5,7)
> x[2]
[1] 3
```

- Now get all but the second element

```
> x[-2]
[1] 1 5 7
```

- Vectorized comparison and logical indexing

```
> x > 3
[1] FALSE FALSE  TRUE  TRUE
> x[x>3]
[1] 5 7
```

- Assign 4 as the third element

```
> x[3] <- 4; x
[1] 1 3 4 7
```

- Multiple assignment and recycling

```
> x[c(1,3)] <- 2; x
[1] 2 3 2 7
> x[] <- c(98,99); x
[1] 98 99 98 99
```

Lists

- Lists are the most flexible container class in R (key/value pairs).
- They can hold arbitrary data structures and mixed atomic types without promotion.
- As with vectors, lists have a `length()`.
- Lists can be created with or without explicit keys. Implicit keys are the integer indices (like vectors).

Lists

- A list with two items and implicit keys

```
> l <- list(1,"1");l
[[1]]
[1] 1

[[2]]
[1] "1"
```

- A list with two items and explicit keys

```
> a <- list(foo=1,bar="1"); a
$foo
[1] 1

$bar
[1] "1"
```

- A nested list

```
> ll <- list(a,a); ll
[[1]]
[[1]]$foo
[1] 1

[[1]]$bar
[1] "1"


[[2]]
[[2]]$foo
[1] 1

[[2]]$bar
[1] "1"
```

Getting and Setting List Items

- Like vectors, list elements can be accessed and set using indexing.
- The syntax for getting a subset of elements (which is returned as a list) is `[]`.
- The argument to `[]` can be a numeric vector (i.e., numeric indices) or character vector (i.e., the key strings).
- The syntax for getting a single element is `[[]]`.
- The argument to `[[]]` can be a (single) numeric index or a (single) key string.
- Lists also support access using the `$` operator, as shown in the example.

Getting and Setting List Items

- Create a list and select the first two elements

```
> point <- list(x=0,y=1,z=2); point[1:2]
$x
[1] 0

$y
[1] 1
```

- Another way to select the first two elements

```
> # point[c("x","y")]
```

- Select the second element by index

```
> point[[2]]
[1] 1
> # point[2] # What does this return?
```

- Select the second element by key

```
> point[["y"]]
[1] 1
```

- Select the second element by name

```
> point$y
[1] 1
```

Data Frames

- Data frames are a data structure for holding what we usually think of as a "data set."
- They are probably the data structure you will use most in R.
- A data frame is nothing more than a list of vectors, all of the same length.
- Because a list can hold mixed atomic data types, we can have a data frame where:
  - Vector (variable) #1 can hold customer IDs (integers)
  - Vector (variable) #2 can hold customer names (character strings)
- Data frames have a variety of useful utilities, among which are `str()` which provides information on the structure, and `nrow()` which gives the observation count.

Data Frames

- cars is a built-in data frame; look at the structure

```
> data(cars)
> str(cars)
'data.frame':	50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

- How many observations?

```
> nrow(cars)
[1] 50
```

- What do the first five lines look like?

```
> head(cars,5)
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
```

- Sample mean of the variable “speed.”

```
> mean(cars$speed)
[1] 15.4
```

Matrices

- As we just discussed, a matrix is a rectangular array of numbers $\mathbf{A} = \{a_{ij}\}$.
- All of the standard matrix operations discussed earlier are supported in R.
- Addition is `+`.
- Matrix multiplication is `%*%`, not `*` (which will do element-by-element multiplication).
- Matrix transposes can be computed with `t()`.
- Matrix inverses can be computed with `solve()`, which also solves linear systems $\mathbf{X}\mathbf{b} = \mathbf{y}$.
- One BIG gotcha with matrices: extracting a single row or column by default returns an R vector, which is not a matrix object.

Getting Help

- R has a built-in help system with useful information and examples.
- `help(mean)` provides information on the `mean()` command.
- A shortcut is typing `?mean` instead.
- `help.search("histogram")` will search the help database for topics that include the word "histogram."
- A shortcut is typing `??histogram` instead.
- `example(plot)` provides some examples for the `plot()` function.

Installing Packages

- Additional functionality can be added to R using packages.
- Probably the easiest way within RStudio is to just choose "Tools" $\rightarrow$ "Install Packages". Type the name of the package you want and be sure that "Install dependencies" is checked.
- From the R console, packages can be installed using `install.packages("MyPackagename")`. Be sure to use quotes.

```
> install.packages("lattice")
```

- When you install packages, you may need to choose (via a picklist) a CRAN mirror from which to download the source and/or binary files. If you are using RStudio (highly recommended), it uses its own mirrors.
- Once you have installed the package, you must then type `library(MyPackagename)` before attempting to use the functions provided by the package. Be sure not to use quotes this time.

```
> library(lattice)
```

Managing Filesystem I/O

- From within the RStudio editor, you can highlight a section of code and click "Run" in the upper right-hand corner. The code will execute in the console window.
- From the command line (or console window itself), use `source("MyFilename.R")` to read in the file `MyFilename.R` and execute its contents.
- Use `sink("MyFilename.log")` to redirect output to a log file and use `sink()` to restore output to the screen.
- Use `print()` or `cat()` to print information to the screen from within loops and functions. Useful for debugging!

Example: Examine, Transform and Analyze – The Pearson-Lee Data

The Pearson-Lee Data

- To start with, let's bring the Pearson-Lee data on father and son heights into R to work with.
- If I have the data set `pearson.rda` already in R format in my working directory, then I can just say

```
> load("pearson.rda")
```

- If I have the text data set `pearson.dat` in my working directory, which is tab-delimited but doesn't have variable names (column headers) then I can say

```
> plData = read.table(
+         file="pearson.dat",
+         col.names=c("f_hgt","s_hgt")
+         )
```

- In my case, I'm actually keeping the data set in another directory, so I say

```
> plData = read.table(
+         file=paste0(myDataPath,"pearson.dat"),
+         col.names=c("f_hgt","s_hgt")
+         )
```

  where `myDataPath` is a string variable I've defined elsewhere in the program.

Descriptive Analysis – Univariate Data

- Example: In the Pearson-Lee data, data on son heights alone.
- For each son i, observe $y_i =$height in inches.
- Sample mean shows where the data set is located:

$$\widehat{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

```
> mean(plData$s_hgt)
[1] 68.68407
```

- What are the units of the sample mean?

Descriptive Analysis – Univariate Data

- Sample variance shows how spread out the data set is:

$$s^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \overline{y})^2$$

```
> var(plData$s_hgt)
[1] 7.922545
```

- Could also use the standard deviation (not the standard error!)

$$s = \sqrt{s^2}$$

```
> sd(plData$s_hgt)
[1] 2.814702
```

- What are the units of the standard deviation?
- Other commonly reported items: median ($q_{.5}$), interquartile range ($[q_{.25}, q_{.75}]$), max, min.

Descriptive Analysis – Univariate Data

- You can get many of the above statistics compactly with the `summary()` command:

```
> summary(plData$s_hgt)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  58.51   66.93   68.62   68.68   70.47   78.36 
```

Descriptive Analysis – Univariate Data

- In R, the procedure for least squares regression is `lm()` (linear model).
- The argument to `lm()` is a formula which mostly does what you would expect, although there are a few exceptions.
- Here's a very simple regression of son height on a constant to compute the sample mean:

```
> res.mean <- lm(s_hgt~1,data=plData)
> summary(res.mean)

Call:
lm(formula = s_hgt ~ 1, data = plData)

Residuals:
     Min       1Q   Median       3Q      Max
-10.1770  -1.7528  -0.0682   1.7819   9.6807

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 68.68407    0.08573   801.2   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.815 on 1077 degrees of freedom
```

Descriptive Analysis – Univariate Data

Questions:

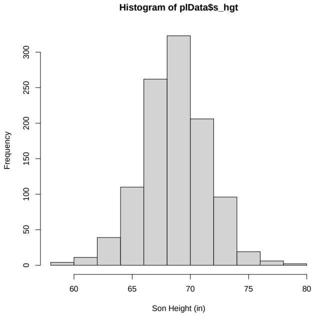
- What is the "residual standard error" measuring? What are its units?
- How is it different from the "standard error" reported with the coefficient estimate? What are its units?
- Why isn't the "residual mean" reported?

Descriptive Analysis – Univariate Data

How to show all of the values as a picture:

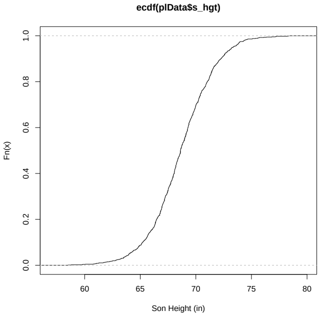
Histogram

```
> hist(plData$s_hgt,xlab="Son Height (in)")
```



Empirical Dist. Func.

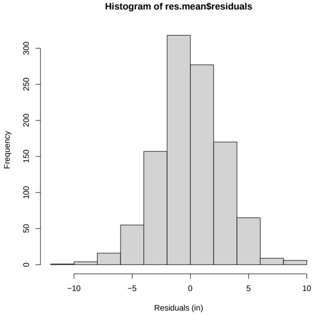
```
> plot(ecdf(plData$s_hgt),xlab="Son Height (in)")
```



How to show all of the residuals as a picture:

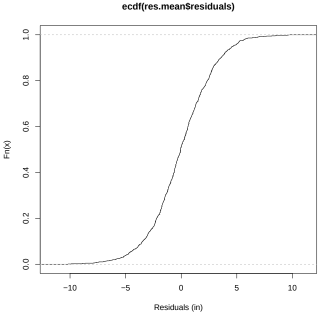
Histogram

```
> hist(res.mean$residuals,xlab="Residuals (in)")
```



Empirical Dist. Func.

```
> plot(ecdf(res.mean$residuals),xlab="Residuals (in)")
```



The Empirical Distribution

- The empirical cumulative distribution function (ECDF or just empirical distribution) is so important that we'll define it here for use later on.
- The empirical distribution of a univariate sample $\{y_i\}_{i=1\ldots N}$ is

$$\widehat{F}_N(A) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\{y_i \in A\}$$

- Taking the sets A to be the intervals $(-\infty, s]$, we have

$$\widehat{F}_N(s) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\{y_i \leq s\}$$

- This latter is a step function which jumps by $\frac{1}{N}$ at every sample point y_i.
- There is a similar definition for bivariate and other multivariate samples $\{\mathbf{z}_i\}$.

Descriptive Analysis – Bivariate Data

- Example: In the Pearson-Lee data, father height (x_i) and son height (y_i).
- Represent a single observation by ordered pair $\mathbf{z}_i = (x_i, y_i)$.
- Make the R code a little cleaner:

```
> attach(plData)
```

- How to show where the data set is "located":
  - Sample Means: $\overline{x}$,$\overline{y}$

```
> mean(f_hgt) # Using "attach", I don't have to reference plData!
[1] 67.6871
> mean(s_hgt)
[1] 68.68407
```

Descriptive Analysis – Bivariate Data

- How to show how spread out the data is:
  - Sample Variance of x $= s_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2$
  - Standard Deviation of x $= s_x = \sqrt{s_x^2}$

```
> sd(f_hgt)
[1] 2.744868
```

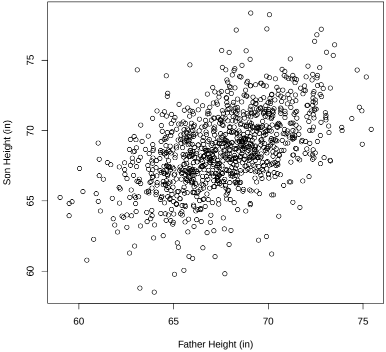
  - Sample Variance of y $= s_y^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \overline{y})^2$
  - Standard Deviation of y $= s_y = \sqrt{s_y^2}$

```
> sd(s_hgt)
[1] 2.814702
```

Descriptive Analysis – Bivariate Data

A scatterplot shows all of the values as a picture:

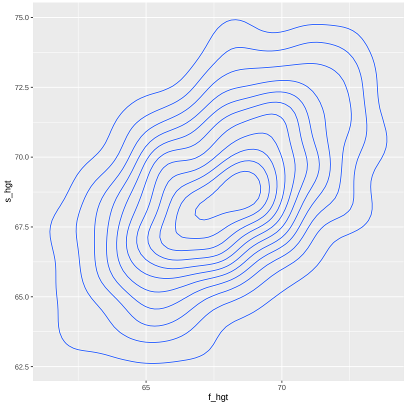
```
> plot(s_hgt~f_hgt,data=plData,xlab="Father Height (in)",ylab="Son Height (in)")
```



Descriptive Analysis – Bivariate Data

So does a density contour:

```
> suppressMessages(library(ggplot2))
> print(ggplot(plData,aes(x=f_hgt,y=s_hgt))+geom_density2d())
```



Descriptive Analysis – Bivariate Data

- Because an observation in bivariate data consists of two pieces of information (x_i and y_i), we might also be tempted to think about the relationship between the two.
- The association between two variables in a bivariate data set is captured by the sample correlation coefficient, r.
- $$r = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{x_i - \overline{x}}{s_x} \cdot \frac{y_i - \overline{y}}{s_y}\right)$$
- ```
  > cor(f_hgt,s_hgt)
  [1] 0.5013383
  > detach(plData) # Reverses "attach()"
  ```
- The sample correlation coefficient necessarily satisfies $-1 \leq r \leq 1$. What are its units?
  ```

Descriptive Analysis – Bivariate Data

The regression lines for bivariate data are computed from the five summary statistics we've just discussed:

1. The sample mean of x: $\overline{x}$
2. The sample mean of y: $\overline{y}$
3. The standard deviation of x: s_x
4. The standard deviation of y: s_y
5. The sample correlation coefficient between x and y: r

At the moment, this is simply a descriptive exercise. We do not need to invoke the typical OLS assumptions because we are not making any inferences.

Descriptive Analysis – Bivariate Data

Using the five statistics above:

- The regression line of y on x is

$$y = \widehat{\alpha} + \widehat{\beta} x$$

  where $\widehat{\alpha} = \overline{y} - \widehat{\beta}\overline{x}$ and $\widehat{\beta} = r\frac{s_y}{s_x}$.
- The standard deviation line (sample major axis) is

$$y = \alpha^* + \beta^* x$$

  where $\alpha^* = \overline{y} - \beta^*\overline{x}$ and $\beta^* = \text{sgn}(r)\frac{s_y}{s_x}$.

The standard deviation line is not particularly important historically, but it is useful as a means of cementing in your head the difference between the regression of y on x and the "reverse regression" of x on y!

Descriptive Analysis – Bivariate Data

For the Pearson-Lee data, we have

```
> attach(plData)
> print( b.hat <- cor(f_hgt,s_hgt)*sd(s_hgt)/sd(f_hgt) )
[1] 0.514093
```

and

```
> print( a.hat <- mean(s_hgt) - b.hat * mean(f_hgt) )
[1] 33.8866
```

What are the units of $\widehat{\alpha}$ and $\widehat{\beta}$?

Descriptive Analysis – Bivariate Data

Of course we can use `lm()` to regress son height y on father height x:

```
> res.ols <- lm(s_hgt~f_hgt,data=plData)
> summary(res.ols)

Call:
lm(formula = s_hgt ~ f_hgt, data = plData)

Residuals:
    Min      1Q  Median      3Q     Max
-8.8772 -1.5144 -0.0079  1.6285  8.9685

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 33.88660    1.83235   18.49   <2e-16 ***
f_hgt        0.51409    0.02705   19.01   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

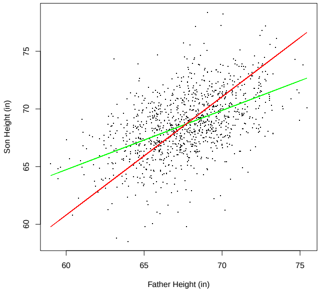
Residual standard error: 2.437 on 1076 degrees of freedom
Multiple R-squared:  0.2513,Adjusted R-squared:  0.2506
F-statistic: 361.2 on 1 and 1076 DF,  p-value: < 2.2e-16
```

But notice that the canned routines also deliver a bunch of extra junk that's irrelevant to our descriptive analysis.

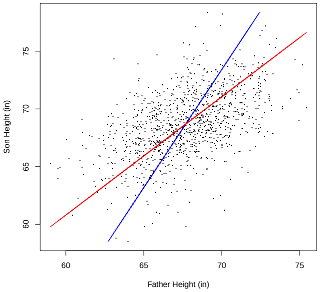
Bivariate Regression in the Pearson-Lee Data

- The red line is the standard deviation line.

Regression of y on x



Regression of x on y



Descriptive Analysis – Summary

- All three of the following are descriptive analyses of the relationship between father height and son height in the Pearson-Lee data:
  - The regression of son height (y) on father height (x): $y = \widehat{\alpha} + \widehat{\beta}x$;
  - The regression of father height (x) on son height (y): $x = \widehat{\widehat{\alpha}} + \widehat{\widehat{\beta}}y$;
  - The standard deviation line: $y = \alpha^* + \beta^* x$.
- Without additional criteria, none of these is distinguished from the others as particularly good.
- Neither do we have a formal criterion for whether the data summary is "too long" (too many covariates) or "too short."

Example: Data Transformation – Histogram Overlay

We saw before that a histogram can be a useful way to understand univariate data.

Suppose we want to see the difference between two different univariate distributions. One way to do this is by overlaying two histograms.

In order to do this easily (e.g., in `ggplot2`), we need the data laid out in "long form" with the groups (i.e., "fathers" and "sons") specified by a single grouping variable. But for our regression, we had the data in "wide form" with a separate variable for each group ("father height" and "son height").

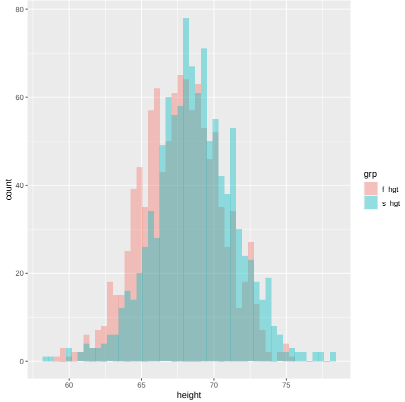
Is there a way to switch easily between long form and wide form? Indeed there is! `pivot_wider` and `pivot_longer`.

Example: Data Transformation – Histogram Overlay

```
> head(plData)
     f_hgt    s_hgt
1 65.04851 59.77827
2 63.25094 63.21404
3 64.95532 63.34242
4 65.75250 62.79238
5 61.13723 64.28113
6 63.02254 64.24221
> suppressMessages(library(tidyverse))
> plotData <- pivot_longer(plData,cols=c(f_hgt,s_hgt),names_to="grp",values_to="height")
> head(plotData)
# A tibble: 6 x 2
  grp   height
  <chr>  <dbl>
1 f_hgt   65.0
2 s_hgt   59.8
3 f_hgt   63.3
4 s_hgt   63.2
5 f_hgt   65.0
6 s_hgt   63.3
> myPlot <- ggplot(plotData,aes(x=height,fill=grp)) +
+   geom_histogram(color=NA,alpha=.4,position="identity",bins=50)
```

Example: Data Transformation – Histogram Overlay

```
> myPlot
```



Example: Data Transformation – dplyr

One very useful toolset is the dplyr package for data transformation. If you have any familiarity with spreadsheet tools or with SQL, you will probably find these tools to be fairly intuitive.

One element introduced in dplyr actually has a long pedigree on the Unix command line – the pipe. Pipes are a programming tool allowing the output of one command to "flow" to the input of another command. In this way, simple tools (like head, tail, grep, as we saw earlier), can be used to build up an expressive language of data manipulation.

In R, a pipe is indicated by $|>$. The yield from the expression before the pipe is passed as the first argument to the function following the pipe.

Most of the material on the next five slides comes straight from Hadley Wickham's excellent text "R for Data Science." The full (and free!) text for this book can be found at https://r4ds.had.co.nz/index.html. The tools (including dplyr and ggplot2) are mostly contained in the tidyverse package.

Example: Data Transformation – The Data

Bring in some Bureau of Transportation Statistics data on flights departing New York City in 2013 (documented in `?flights`)

```
> library(nycflights13)
> library(tidyverse)
> flights
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Example: Data Transformation – `filter()`

The `filter()` function is used to select observations from a data frame/tibble:

```
> temp0 <- filter(flights, month == 1, day == 1)
> temp0
# A tibble: 842 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Note that I could have used a pipe here:

```
> temp1 <- flights |> filter(month==1, day == 1)
> identical(temp0,temp1)
[1] TRUE
```

Arguments are ANDed together. To get OR, XOR, NOT or other logicals, use the discussion from Basics.

Example: Data Transformation – arrange()

In addition to extracting rows, we can change the order of rows. This can be particularly helpful following aggregation/summarize operations to see changes by year, which US state had the most individuals in poverty, etc.

```
> arrange(flights,year, month, day)
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
> flights |> arrange(desc(dep_delay))
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     9      641            900      1301     1242           1530
 2  2013     6    15     1432           1935      1137     1607           2120
 3  2013     1    10     1121           1635      1126     1239           1810
 4  2013     9    20     1139           1845      1014     1457           2210
 5  2013     7    22      845           1600      1005     1044           1815
 6  2013     4    10     1100           1900       960     1342           2211
 7  2013     3    17     2321            810       911      135           1020
 8  2013     6    27      959           1900       899     1236           2226
 9  2013     7    22     2257            759       898      121           1026
10  2013    12     5      756           1700       896     1058           2020
# ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Example: Data Transformation – `select()`

We can select particular fields as well. There is a very flexible syntax for selection based on regular expressions (patterns), as shown below.

```
> flights_sml <- select(flights,
+   year:day,
+   ends_with("delay"),
+   distance,
+   air_time
+ )
> flights_sml
# A tibble: 336,776 x 7
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1         2        11     1400      227
 2  2013     1     1         4        20     1416      227
 3  2013     1     1         2        33     1089      160
 4  2013     1     1        -1       -18     1576      183
 5  2013     1     1        -6       -25      762      116
 6  2013     1     1        -4        12      719      150
 7  2013     1     1        -5        19     1065      158
 8  2013     1     1        -3       -14      229       53
 9  2013     1     1        -3        -8      944      140
10  2013     1     1        -2         8      733      138
# ... with 336,766 more rows
```

Example: Data Transformation – `mutate()`

If a calculated field is not present in our data set, we can add it.

```
> fltInfo <- flights_sml |> mutate(
+   gain = dep_delay - arr_delay,
+   speed = distance / air_time * 60
+ )
```

Example: Data Transformation – group() and summarize()

Grouping and summarizing work as you would expect. There are a wide variety of built-in summary statistics (mean, median, count) as well as hooks allowing you to define your own.

```
> by_day <- group_by(flights, year, month, day)
> summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))
`summarise()` has grouped output by 'year', 'month'.  You can override using the
`.groups` argument.
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

Example: Data Transformation – Putting It All Together

Putting the steps together shows the real power of pipes:

```
> delays <- flights |>
+   group_by(dest) |>
+   summarize(
+     count = n(),
+     dist = mean(distance, na.rm = TRUE),
+     delay = mean(arr_delay, na.rm = TRUE)
+   ) |>
+   filter(count > 20, dest != "HNL")
> delays
# A tibble: 96 x 4
   dest  count  dist delay
   <chr> <int> <dbl> <dbl>
 1 ABQ     254 1826   4.38
 2 ACK     265  199   4.85
 3 ALB     439  143  14.4
 4 ATL   17215  757. 11.3
 5 AUS    2439 1514.  6.02
 6 AVL     275  584.  8.00
 7 BDL     443  116   7.05
 8 BGR     375  378   8.03
 9 BHM     297  866. 16.9
10 BNA    6333  758. 11.8
# ... with 86 more rows
```