Moving from IDK to IDE

Getting Started with R for Laboratory Medicine
Sunday Aug 04, 2019
AACC 2019, Anaheim CA

Daniel T. Holmes, MD (dtholmes@mail.ubc.ca)
Department of Pathology and Laboratory Medicine, University of British Columbia
Stephen R Master, MD PhD (masters@email.chop.edu)
Department of Pathology and Laboratory Medicine, Children's Hospital of Philadelphia

Contents

0.1 Preparation for the Course 2

1 Lesson 1: Basics and Data Types **2**
1.1 What is R? 2
1.2 Algebra 3
1.3 Comments 4
1.4 Variables 4
1.5 Data Types 5
1.6 Vectors 6

2 Lesson 2: Matrices, Dataframes and Lists **9**
2.1 Matrices 9
2.2 Data Frames 10
2.3 Lists 16

0.1 Preparation for the Course

This Short Course requires you to bring your own laptop. Plugs will be provided on the tables so that you do not have to rely on your battery life. You must install the necessary software before you arrive. It can take up to 30 mins and you do not want to be stuck doing this in the session because you get behind.

You need to install both the R programming Language and the R-Studio interface to R. While one can use R without R-Studio, we will all use it to make things uniform. The R programming language is available for download from many different places. Here are three places in Canada, http://cran.stat.sfu.ca/; http://cran.utstat.utoronto.ca/; https://mirror.its.dal.ca/cran/. Choose the download that is appropriate for your laptop (whether Linux, Mac or Windows). Rstudio can be downloaded and installed free for personal use from: https://www.rstudio.com/products/RStudio/. You want the Open Source Edition.

Throughout this handout, we'll be showing you R code in a shaded box like this

```
R code goes here
```

and output will come be in an unshaded area after two "#" signs

```
## [1] "Here is some R output"
```

1 Lesson 1: Basics and Data Types

1.1 What is R?

The purpose of this course is to show you that many of the data management and data analysis task you *wish* you could do in Excel are, in fact, very easy if you move to the statistical programming language R.

R is what is known as a "scripting language". This means that R does not build portable, stand–alone programs like commercial software we usually purchase. In contrast, R requires the R interpreter to be pre-installed on the computer before we can use any of its utilities. The same is true of other popular scripting languages like Python, PHP, Ruby and JavaScript. In principle this means that if you are using R on a company laptop and you don't have administrative privileges, you are going to need IT to help you install it.

R has been built for Windows, Linux and Mac, which means that R code you write can be run by other people on other systems *without any alterations to the code*, provided that they have installed the R interpreter. There are occasional exceptions to this rule but any code alterations are–most often–trivial.

The programs you write in R are saved as text or "ASCII" files that you save like any old document. You can use any text editor to write your program. This might include programs like Notepad or Notepad++ in the Windows environment

or Gedit, eMacs, vi, Nano or Sublime Text in the Linux/Mac environments. In this course we will use a text editor in an environment specifically built for R called RStudio. This program has the benefit that it can both allow you to edit your text and run it without leaving the RStudio program. RStudio has many other convenient features that you will discover should you choose to continue this jou–R–ney. We could spend a lot of time on the background, but we do not have a great deal of time together so we should just jump right in.

1.2 Algebra

R can act as a calculator. It follows these basic rules or algebra.

Operation	Expression	R Code
Addition	$x + y$	`x + y`
Subtraction	$x - y$	`x - y`
Multiplication	$x \times y$	`x * y`
Division	$x \div y$	`x / y`
Exponents	x^y	`x^y` or `x**y`
Logarithm	$\log(x)$	`log(x)`
Exponential	e^x	`exp(x)`
Trig Functions	$\sin(x)$	`sin(x)`

There are some other useful built-in functions in R. These are:

- `abs(x)`
- `sqrt(x)`
- `floor(x)`
- `round(x, digits = n)`
- `signif(x, digits = n)`

1.2.1 Exercise

1. Experimentation is a great way to find out what a function does. Determine the square root of 64 with the `sqrt()` function and using the fact that the square root of 64 is really $64^{1/2}$
2. Try the following:
   - `ceiling(1.2)`
   - `ceiling(1.49)`
   - `ceiling(1)` What does ceiling do?
3. Try the following:
   - `round(1.4503,1)`
   - `round(exp(1),4)`
   - `round(pi,4)`

- Now try typing `round(12314,-1)` and `round(12314,-2)`. What's happening here?

4. What does `trunc(5.99)` give you? What does `trunc(-3.43)` give you? How is `trunc()` different than `floor()`?

1.3 Comments

Note that anything we type on an R line that comes after the `#` sign is ignored by R. This is very useful for including comments in your code and helps to remind you (and others) what you were thinking. Because anything after the `#` is ignored, we can either put a comment on its own line:

```
# here is a comment
```

or after some code that is going to be executed

```
2 + 2  # we had better get the answer "4"
```

```
## [1] 4
```

1.4 Variables

Variables allow you to store your data so that it can be easily retrieved at a later time. For example, suppose you calculated the standard deviation of a data set and you had to use this result over and over again (and you did not want to type it our each time!). The best way to reuse this value is to store it in a variable.

```
my.sd <- 0.352
my.sd
```

```
## [1] 0.352
```

```
1.96 * my.sd
```

```
## [1] 0.68992
```

Notice that assigning the variable is performed with an "arrow" <- . The arrow can actually go the other way too but we don't do that all too much.

```
a <- 5 -> b
a
```

```
## [1] 5
```

```
b
```

```
## [1] 5
```

Getting rid of a variable is sometimes convenient, and the way to do this is with the `rm()` function.

```
rm(a)
a
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

To list all active variables type `ls()`. To remove all active variables type `rm(list = ls)`. This can also be achieved by clicking the broom icon in the **Environment** tab of the top right pane of Rstudio.

1.5 Data Types

We will encounter a variety of different data types during this course, and each has specific applications.

```
var.1 <- TRUE # a logical variable
class(var.1)
```

```
## [1] "logical"
```

```
var.2 <- 32.5 # a numeric variable
class(var.2)
```

```
## [1] "numeric"
```

```
var.3 <- "Michael" # a character variable
class(var.3)
```

```
## [1] "character"
```

Also, we can make integer variables and factor variables. R guesses which you want, and if it guesses wrong you may need to force it to assume the correct data type. You will see this later on. Don't assume that R has correctly read your mind on the data type you wanted.

```
var.4 <- 5
class(var.4)
```

```
## [1] "numeric"
```

```
var.4 <- as.integer(var.4)  # coerces the value to the class of integer
class(var.4)
```

```
## [1] "integer"
```

```
var.5 <- "4"
class(var.5)
```

```
## [1] "character"
```

```
7 * var.5                   # what happened?
```

```
## Error in 7 * var.5: non-numeric argument to binary operator
```

```r
var.5 <- as.numeric(var.5)  # coerces the character into a numeric
class(var.5)
```

```
## [1] "numeric"
```

```r
7 * var.5                   # ahhh no error now.
```

```
## [1] 28
```

1.6 Vectors

Vectors are a way to store multiple data points in a single variable. They are like a column from an Excel file and we will use them a lot. To define a vector you have to use the combine `c()` function to group the data.

```r
x <- c(5,3,6,4,7,2,6) # defines the variable x
class(x)              # what class will this be?
```

```
## [1] "numeric"
```

Importantly, every member of the vector must be of the same type and if they are not, R will choose a data type that will be compatible with all the elements.

```r
y <- c(5,3,6,4,7,2,"Hi There")
class(y)              # What happened?
```

```
## [1] "character"
```

```r
y
```

```
## [1] "5"        "3"        "6"        "4"        "7"        "2"
## [7] "Hi There"
```

Let's explore some algebra:

```r
x + 2   # What does it do?
```

```
## [1] 7 5 8 6 9 4 8
```

```r
x + x       # How is this different from x+2
```

```
## [1] 10  6 12  8 14  4 12
```

```r
x / 2       # What does this do?
```

```
## [1] 2.5 1.5 3.0 2.0 3.5 1.0 3.0
```

```r
x * x       # What does this do?
```

```
## [1] 25  9 36 16 49  4 36
```

```r
x / x       # What does this tell you about dividing vectors?
```

```
## [1] 1 1 1 1 1 1 1
```

```
sum(x)  # What does this calculate?
```

```
## [1] 33
```

```
mean(x) # And this?
```

```
## [1] 4.714286
```

```
sd(x)       # And this?
```

```
## [1] 1.799471
```

```
length(x)   # And this? This is really useful.
```

```
## [1] 7
```

```
c(x, x)
```

```
##  [1] 5 3 6 4 7 2 6 5 3 6 4 7 2 6
```

```
rep(x,3)
```

```
##  [1] 5 3 6 4 7 2 6 5 3 6 4 7 2 6 5 3 6 4 7 2 6
```

What if we wanted to find out an individual value from x?

```
x[2]  # Note the SQUARE brackets.
```

```
## [1] 3
```

```
# OK, makes sense
```

What if we wanted to know which value of x was 6, if any?

```
which(x  ==  6)
```

```
## [1] 3 7
```

CAREFUL "==" is used to compare two values to see if they are equal; don't confuse this with and "=" or "<-", which are for assignment.

```
x[which(x  ==  6)]    # should give us back a number of 6's of course.
```

```
## [1] 6 6
```

```
x  ==  6    #this gives us a logical vector answering the question "Is the value 6?"
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
y <- x  ==  6
y
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
x[y]    #this should just give us back the 6's
```

```
## [1] 6 6
```

```
x[x  ==  6]
```

```
## [1] 6 6
```

1.6.1 Exercise

1. Define a variable, `days`, which contains all the days of the week.
2. Now imagine that you move to a planet where there are an 8th and 9th day of the week, called "Chillday" and "Sleepday". Can you use `c()` to add these days to your days variable so that you do not have to retype everything?

1.6.2 Exercise

1. Type `1:10` and see what happens.
2. Now type `x <- 1:10`. What did this do? Find out by asking R what `x` is.
3. Now type `x <- 5:10`. What did this do?
4. Type ?seq in the console to find out what the `seq()` function does. Can you replicate the results of `x <- 5:10` with `seq()`? Why is `seq()` more flexible?

1.6.3 Exercise

1. Write an expression in R that will always return the last value in a vector named `z`.
2. Test your expression's success. We'll start by generating a sequence of letters of the alphabet that terminates randomly.

```
z <- letters[1:round(runif(1,0,26),0)]  # don't worry about why this works right now.
z
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r"
```

```
z[length(z)]                            # check what the last value is
```

```
## [1] "r"
```

Now apply your expression to identify the last letter in the sequence you generated.

2 Lesson 2: Matrices, Dataframes and Lists

2.1 Matrices

A matrix is a 2D analogue of the vector. Matrices may not seem all that important but there are certain R statistical functions requiring their use and so you are certainly going to encounter them.

```
days <- 1:28  # Generates a sequence of 28 integers
days          # confirm that you have done what you thought
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28
```

```
# Note: another way to do this is with days <- seq(from = 1, to = 28, by = 1)
```

Now, let's convert this to something that looks more like the calendar layout of February.

```
matrix(days, nrow = 4, ncol = 7)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    5    9   13   17   21   25
## [2,]    2    6   10   14   18   22   26
## [3,]    3    7   11   15   19   23   27
## [4,]    4    8   12   16   20   24   28
```

```
# Huh? What happened?
#
# Let's try again
matrix(days, nrow = 4, ncol = 7, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4    5    6    7
## [2,]    8    9   10   11   12   13   14
## [3,]   15   16   17   18   19   20   21
## [4,]   22   23   24   25   26   27   28
```

```
# Ahhh
```

2.1.1 Exercise

- February 2016 started on a Monday but was a leap-year. Make a matrix for February 2016 and fill days that are not in February with NA. We'll talk later about different places that NA pops up, but for now just know that it's R's way of denoting data that is "**N**ot **A**vailable". Hint for this exercise: start by using the `c()` command to prepend the appropriate

number of NA's to the beginning and the end of variable days. Assign the name feb.2016 to your matrix.

To get a specific value of your matrix, you simply refer to the row and column.

```
feb.2016[3, 4] #Gives entry from row 3 column 4.  Should return 17 as the result.
```

```
## [1] 17
```

```
# To get all the values in the column, you simply omit the row number.
feb.2016[ ,6] # Gives us what we need: all the Saturdays.
```

```
## [1]  5 12 19 26 NA
```

```
# ...and the same trick works for columns
feb.2016[3, ] # Gives us week 3.
```

```
## [1] 14 15 16 17 18 19 20
```

2.1.2 Exercise

The raw speed data from your latest bike ride can be exported from your Garmin and brought into R. The ride happens to be exactly 38 mins duration and data is sampled every second. This means that there will be 2280 measurements in total. Execute this code to import the data from a file.

```
speed.data <- read.csv("Data_Files/speed.csv")
speed.data <- speed.data[,1]
```

- Now, convert this data into a matrix of 38 columns where each column is the speed data of one minute.

2.2 Data Frames

Data frames are the closest thing you are going to get to Excel-like storage of your data. When you read your data into R from a file (if you have saved it from Excel in a standard format), it will become a data frame.

Let's convert our February 2016 matrix to a data frame, and then we will look at how to import Excel-like data to a data frame. We will spend some time working with data frames because they are going to be our bread and butter.

First, we can convert a matrix to a data frame:

```
feb.2016 <- as.data.frame(feb.2016)
feb.2016
```

```
##   V1 V2 V3 V4 V5 V6 V7
## 1 NA  1  2  3  4  5  6
## 2  7  8  9 10 11 12 13
```

```
## 3 14 15 16 17 18 19 20
## 4 21 22 23 24 25 26 27
## 5 28 29 NA NA NA NA NA
```

```
# this has uninformative column names, but we can name the columns as we can in Excel
names(feb.2016) <- c("Sun","Mon","Tue","Wed","Thu","Fri","Sat")
feb.2016
```

```
##   Sun Mon Tue Wed Thu Fri Sat
## 1  NA   1   2   3   4   5   6
## 2   7   8   9  10  11  12  13
## 3  14  15  16  17  18  19  20
## 4  21  22  23  24  25  26  27
## 5  28  29  NA  NA  NA  NA  NA
```

```
# much better
```

Second, we can build a data frame from vectors as follows.

```
odd.nums <- c(1,3,5,7)
even.nums <- c(2,4,6,8)
numbers <- data.frame(odds = odd.nums, evens = even.nums)
numbers
```

```
##   odds evens
## 1    1     2
## 2    3     4
## 3    5     6
## 4    7     8
```

```
# alternatively, numbers <- as.data.frame(rbind(odds,evens))
```

So, the columns of a data frame could be the results of different data fields in your study, name, health number, sex, age, date of last visit, blood pressure, medications, creatinine, hemoglobin etc. Whatever you could store in an Excel sheet could be in a dataframe. We often want to pull out specific columns from a data frame–usually to perform statistical tests.

Pulling out a column is easy. We can do it by the column name or we can do it by the column number.

```
# by column name
feb.2016$Tue # gives all the Tuesdays in Feb 2016
```

```
## [1]  2  9 16 23 NA
```

```
# by column number
feb.2016[,3] # gives all rows of the third column, which is the same
```

```
## [1]  2  9 16 23 NA
```

```
# You can pull out data from an individual cell.
feb.2016[2,3]
```

```
## [1] 9
```

```
# Or you can do the same with the $ approach because feb.2016$Tue is a vector
feb.2016$Tue[2]
```

```
## [1] 9
```

If you need to pull more than one column out, you can do so by numbers or the names:

```
# by number
feb.2016[,3:4]
```

```
##   Tue Wed
## 1   2   3
## 2   9  10
## 3  16  17
## 4  23  24
## 5  NA  NA
```

```
#by name - this is particularly convenient when dealing with large dataframes
feb.2016[,c("Tue","Wed")]
```

```
##   Tue Wed
## 1   2   3
## 2   9  10
## 3  16  17
## 4  23  24
## 5  NA  NA
```

Let's go back to that bike speed data we have above. We will start by turning it into a data frame.

```
speed.frame <- as.data.frame(speed.mat)
names(speed.frame ) <- paste0("min_",1:38)
#head(speed.frame)
```

2.2.1 Exercise

- Using the speed.frame dataframe, find the average speed of the 20th minute of your ride.

Note that if you try to find the average by row instead of a column, you will run into a problem that illustrates something about data frames, namely that if you extract a row, you can't just simply do algebra on it because there is generally no guarantee that different columns of a data frame will all be numbers.

```
min.start.avg <- mean(speed.frame[1,]) #does not work
min.start.avg <- mean(speed.mat[1,]) #does work
```

Here are some things you can do with data in a data frame. You can ask R to tell you things about your data points. For example, you could ask R for some descriptive statistics of the last minute of your ride:

```
mean(speed.frame$min_38)
```

```
## [1] 25.15944
```

```
median(speed.frame$min_38)
```

```
## [1] 23.103
```

```
sd(speed.frame$min_38)
```

```
## [1] 4.482852
```

```
summary(speed.frame$min_38) # a statistical summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   21.50   22.52   23.10   25.16   26.81   38.38
```

```
quantile(speed.frame$min_38 ,probs = c(0.25,0.50, 0.75)) # specific quantiles
```

```
##     25%     50%     75%
## 22.5216 23.1030 26.8137
```

But this does not work when you try to take the grand mean:

```
mean(speed.frame)
```

```
## [1] NA
```

But this approach does work for matrices

```
mean(speed.mat)
```

```
## [1] 30.9961
```

Remember, we'll say more later about "NA", but for now notice that R gives us a nasty warning message. The reason that the `mean` function does not operate on data frames in this case because the data in a data frame is often not numeric.

You can also ask R to tell you which values have certain properties.

```
# did you dip below 25 kph in the last minute of your ride?
speed.frame$min_38 < 25
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
## [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [56]  TRUE  TRUE  TRUE  TRUE  TRUE
```

But the code authors acknowledge that it is a pain to have to write out those $ signs all the time, so try this:

```
attach(speed.frame)
min_1 # now min_1 is a local variable
```

```
##  [1] 14.1984 17.2188 18.8964 20.0448 21.3732 22.4172 22.9752 23.2992
##  [9] 23.3424 23.1228 22.6908 22.2876 21.9456 21.6648 21.3768 21.1140
## [17] 20.9412 20.7648 20.6244 20.5056 20.1960 19.9080 19.3608 18.9612
## [25] 18.8460 18.8424 18.9324 19.1232 19.2600 19.5696 20.3868 20.0412
## [33] 19.2276 18.0828 16.9488 16.2684 15.5304 15.0048 14.7204 14.3928
## [41] 14.3172 14.5800 15.1416 15.1884 12.4380 12.3840 12.8736 13.3128
## [49] 13.6764 14.0724 14.4720 14.8104 15.1596 15.4800 15.6384 15.8400
## [57] 16.0596 16.2900 16.4952 16.6860
```

```
min_2 # ...and so is min_2!
```

```
##  [1] 17.0064 17.4456 17.8488 18.1260 18.3996 18.6264 18.9576 19.4436
##  [9] 18.9756 19.3752 19.8648 20.4372 21.0564 21.6972 22.6152 23.4756
## [17] 24.1200 24.5772 24.8796 25.1820 25.5852 25.9020 26.1864 26.4996
## [25] 26.8560 27.2232 27.6732 28.2204 28.7856 29.2536 29.6244 29.9376
## [33] 30.2544 30.5064 30.6576 30.8412 31.0320 31.2156 31.3128 31.3488
## [41] 31.3164 31.3200 31.3452 31.4280 31.4712 31.3632 31.3020 31.2048
## [49] 31.0608 30.9024 30.7404 30.5892 30.3804 30.0780 29.5488 28.7388
## [57] 28.0620 28.0980 28.4004 28.5876
```

Note that if you alter the attached variables, they *do not* alter the original data frame. To remove these variables we type:

```
detach(speed.frame)
```

There are lots of reasons not to use `attach`, but it is good for quick and dirty things.

2.2.2 Exercise

- We are going to read in a little more data from your bike ride. This time the data will contain time, cadence, heart rate, distance, speed and power.

```
ride.data <- read.csv("Data_Files/ride_file.csv")
head(ride.data)
```

```
##   secs cad hr      km     kph watts
## 1    1   1 80 0.00347 14.1984   360
## 2    2  75 81 0.00788 17.2188   360
```

```
## 3    3  77 83 0.01290 18.8964   255
## 4    4  91 84 0.01817 20.0448   245
## 5    5  99 86 0.02400 21.3732   334
## 6    6 101 87 0.03020 22.4172   249
```

Isolate the data from your ride for which your speed was over 65 kph.

What you have just done is called subsetting your data frame, and it is a very frequent task that we are going to be doing more of in the next hour. Because it is such a common task (e.g. pulling out all the subjects less than 40 years, pulling out all the male subjects, excluding an outlier), there is a specific command for it:

```
subset(ride.data, kph > 65)
```

```
##      secs cad  hr      km     kph watts
## 1387 1387  96 129 10.7909 65.7180    88
## 1388 1388  96 129 10.8098 66.7512    88
## 1389 1389  96 129 10.8284 67.1940    95
## 1390 1390  96 129 10.8471 66.3840    96
## 1450 1450 102 134 11.7990 66.8952   131
## 1451 1451 102 135 11.8178 67.6728   131
## 1452 1452 103 134 11.8365 67.0788   131
## 1453 1453 103 134 11.8553 65.0520   123
```

You can build other logical constraints to isolate data of interest in vectors, matrices and data frames by using `&` (AND), `|` (OR), and `!` (NOT). For example:

```
x <- c(3,4,2,7,5,8,9,5,-2,34,15,7)
# values of x greater than 2 and less than 7. Use the & for AND.
x[x > 2 & x < 7]
```

```
## [1] 3 4 5 5
```

```
# values of x not less than 15
x[!(x < 15)]
```

```
## [1] 34 15
```

```
# which is the same as
x[x >=  15]
```

```
## [1] 34 15
```

```
# values of x less than 3 or greater than 5. Use the | for OR.
x[x < 3 | x > 5]
```

```
## [1]  2  7  8  9 -2 34 15  7
```

```
# all values except the first three: use a minus sign
x[-(1:3)]
```

```
## [1]  7  5  8  9  5 -2 34 15  7
```

and with the data frame we might ask for rows your speed was under 20 km/h but your power was over 350 watts.

```
subset(ride.data, kph < 15 & watts > 350)
```

```
##      secs cad  hr       km     kph watts
## 1       1   1  80  0.00347 14.1984   360
## 1282 1282  98 162 10.08370  6.8832   386
## 1291 1291  98 163 10.10570  9.4284   359
## 1292 1292  98 163 10.10830  9.6624   356
## 1293 1293  98 163 10.11100  9.8424   354
## 1294 1294  98 164 10.11380 10.0656   362
## 1295 1295  98 164 10.11670 10.3860   360
## 1296 1296 100 164 10.11960 10.7208   360
## 1297 1297 100 164 10.12260 11.0880   353
## 1298 1298 102 165 10.12580 11.4624   353
## 1302 1302 108 166 10.13940 13.5828   353
```

(Note that – just like algebra – we can use parenthesis to make absolutely clear what order we are using to evaluate the expressions)

2.3 Lists

Lists are another way of storing data in a conveniently accessible way. Usually they are used for data types that are different in structure (or store different types of data) but are related because they are part of the same analysis. For example, R frequently provides output of statistical analysis in the form of a list. Suppose you had a vector, a data frame and a matrix:

```
a <- c("Doe","a","deer")
b <- data.frame(lyrics1 = c("a","female","deer"), lyrics2 = c("Ray","a","drop"))
d <- matrix(seq(from = 0, to = 100, length.out = 25), nrow = 5, ncol = 5)
```

But they were all related to some specific problem and you wanted to bundle them together in another variable. This is where you would use a list.

```
my.list <- list(a,b,d)
my.list
```

```
## [[1]]
## [1] "Doe"  "a"    "deer"
##
## [[2]]
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
```

```
## 3    deer    drop
##
## [[3]]
##           [,1]     [,2]     [,3]     [,4]      [,5]
## [1,]  0.000000 20.83333 41.66667 62.50000  83.33333
## [2,]  4.166667 25.00000 45.83333 66.66667  87.50000
## [3,]  8.333333 29.16667 50.00000 70.83333  91.66667
## [4,] 12.500000 33.33333 54.16667 75.00000  95.83333
## [5,] 16.666667 37.50000 58.33333 79.16667 100.00000
```

Components of the list are addressed using a double bracket notation. For example:

```
my.list[[2]] # gives the data frame.
```

```
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
## 3    deer    drop
```

If you happen to give the components of the list names, you can address with the $ notation:

```
my.list <- list(one = a,two = b,three = d)
my.list
```

```
## $one
## [1] "Doe"  "a"    "deer"
##
## $two
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
## 3    deer    drop
##
## $three
##           [,1]     [,2]     [,3]     [,4]      [,5]
## [1,]  0.000000 20.83333 41.66667 62.50000  83.33333
## [2,]  4.166667 25.00000 45.83333 66.66667  87.50000
## [3,]  8.333333 29.16667 50.00000 70.83333  91.66667
## [4,] 12.500000 33.33333 54.16667 75.00000  95.83333
## [5,] 16.666667 37.50000 58.33333 79.16667 100.00000
```

```
my.list$two # gives the same dataframe data
```

```
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
## 3    deer    drop
```

R very often uses lists for statistical reports. For example, the `lm()` function is used to generate a regression report by R. You can assign the output of `lm()` to a variable and this variable contains a very useful list.

```
x <- 1:10
y <- 2 * x + rnorm(10,0,1)  # generates some fake data
reg <- lm(y ~ x)
str(reg)                    # shows what variables that reg stores.

# ...but they generate a lot of output, so try it on your own

reg$residuals
reg$fitted.values
```

```
# OK, fine--we'll show one...
reg$coefficients
```

```
## (Intercept)           x
##  -0.7340475   2.0885229
```

As a final trivia point, data frames are really just lists of columns, each of which is an equal-length vector.

Data Cleansing I've Tried Scrubbing Even Soaking

Getting Started with R for Laboratory Medicine
Sunday Aug 04, 2019
AACC 2019, Anaheim CA

Dennis Orton

7/23/2019

Contents

1 Lesson 3: Cleansing and beautifying data - garbage to gold 1
1.1 Setting up and formatting your data . . . 2
1.2 A sign of the times:`lubridate()` . . . 3
1.3 Pulling the Strings . . . 5
1.4 Coping with Non-numeric Data . . . 8

2 Lesson 4: Meet the 'tidyverse' - Gather, Join, Filter, and Clean 12
2.1 Packages . . . 12
2.2 `gather()` and `spread()` . . . 13
2.3 `arrange()`, `filter()`, and `select()` . . . 17
2.4 Acknowledgements . . . 18

1 Lesson 3: Cleansing and beautifying data - garbage to gold

Lets face it, a real-life data set is never going to be as clean as the examples we will give you in a controlled environment such as a workshop with a time limit, so we're not likely going to be able to cover every situation you may find yourself in. What we can do is try to give you some examples of the types of data formatting issues we have faced in our experiences and show you some examples of how to deal with troublesome data in general.

The main data formats that we encounter are factor, numeric, character, and date. The purpose of this section is to review these formats, how to convert between them, and how to "clean" imperfections out of the data to generate usable datasets. we will start with general data structure and work out from there.

1.1 Setting up and formatting your data

The most common form of data structure in the Clinical Lab is likely going to be a `dataframe`. Whether you are importing `.csv` files to plot patient comparison data, or want to summarize your QC running mean and SD, this is likely going to be your format of choice. For this section, we will start by synthesizing a "practice" data set.

Note that if you ever post questions online, you will need to know how to generate a "representative" data set, so knowing how to generate a dataframe can be helpfull as well!

```
# Start by constructing a dataframe
date <- c("June 28, 2019", "June 29, 2019",
          "June 30, 2019", "July 1, 2019",
          "July 2, 2019")
time <- c("18:45", "16:36", "7:30", "14:22", "12:36")
patient.ID <- c("A", "B", "C", "D", "E") # Here is a vector of patient IDs
result <- c(5, 6, 4, 7, 5) # Here is a vector of results
df <- data.frame(patient.ID, result, date, time)
# This generates a dataframe with two columns (variables) and five rows (observations)
df
```

```
##   patient.ID result          date  time
## 1          A      5 June 28, 2019 18:45
## 2          B      6 June 29, 2019 16:36
## 3          C      4 June 30, 2019  7:30
## 4          D      7  July 1, 2019 14:22
## 5          E      5  July 2, 2019 12:36
```

Never assume that R has interpreted what you plan to do with this dataframe correctly. It's good practice to always check the format of your data before moving on!

```
str(df)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ patient.ID: Factor w/ 5 levels "A","B","C","D",..: 1 2 3 4 5
##  $ result    : num  5 6 4 7 5
##  $ date      : Factor w/ 5 levels "July 1, 2019",..: 3 4 5 1 2
##  $ time      : Factor w/ 5 levels "12:36","14:22",..: 4 3 5 2 1
```

Notice that the function `data.frame()` has interpreted the date and patient.ID variables as Factors! This is an issue if we are going to try to summarize these results by date or by patient ID. To change these results to character variables, we can use `as.character()` or just tell R that we want to import them as characters to begin with.

```
df <- data.frame(patient.ID, result, date, time, stringsAsFactors = FALSE)
str(df)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ patient.ID: chr  "A" "B" "C" "D" ...
##  $ result    : num  5 6 4 7 5
##  $ date      : chr  "June 28, 2019" "June 29, 2019" "June 30, 2019" "July 1, 2019" ...
##  $ time      : chr  "18:45" "16:36" "7:30" "14:22" ...
```

Okay, now we have something to work with. From here we can change the formats for each variable as necessary.

```
df$patient.ID <- as.character(df$patient.ID) # Change to a character variable
df$result <- as.numeric(df$result) # This was already numeric, but I wanted to show you
```

Dates are a little more complicated . . .

1.2 A sign of the times:`lubridate()`

The `lubridate()` package allows us to deal with dates and times and do algebra on them as we would with other vectors. This represents a major advantage over the handling of dates using base R packages.

Lubridate makes logical assumptions about what you probably mean based on typical date formats.

The functions that lubridate uses are `mdy()`, `ymd()` and `dmy()`. They have very predictable behavior.

```
library(lubridate)

mdy("Aug-20,1755")
```

```
## [1] "1755-08-20"
```

```
mdy("Aug/20/1755")
```

```
## [1] "1755-08-20"
```

```
mdy("08-20-1755")
```

```
## [1] "1755-08-20"
```

```
mdy("08201755")
```

```
## [1] "1755-08-20"
```

```
mdy("August 20, 1755")
```

```
## [1] "1755-08-20"
```

```
mdy("August 20 1755")
```

```
## [1] "1755-08-20"
```

```
#all work perfectly without any explicit statements about format.
```

The real gold with lubridate is how it can handle times too!

```
#calculating the number of days since the Declaration of Independence
then <- mdy_hms("July 04,  1776 14:32:45")
then
```

```
## [1] "1776-07-04 14:32:45 UTC"
```

```
now <- ymd_hms(Sys.time())
now
```

```
## [1] "2019-07-24 16:51:02 UTC"
```

```
delta <- difftime(now, then, units = "days")
delta #wow
```

```
## Time difference of 88773.1 days
```

```
#calculating the number of days Canada has been a country
then <- mdy_hms("July 01,  1867 11:17:21")
then
```

```
## [1] "1867-07-01 11:17:21 UTC"
```

```
now <- ymd_hms(Sys.time())
now
```

```
## [1] "2019-07-24 16:51:02 UTC"
```

```
delta <- difftime(now, then, units = "days")
delta
```

```
## Time difference of 55540.23 days
```

Now lets apply the `dmy()` and `dmy_hm()` formats to our `df` dataframe.

```
df$date <- mdy(df$date) # note that I had to change "dmy()" to "mdy()".
```

It's fairly common for dates and times to not be listed in the same column. We can deal with that in a dataframe by using `paste()`. This function is fairly intuitive, then we can apply the `dmy_hm()` function to format the new column.

```
# paste the date and time using a space as a separator into a new
# column called "date.time" in the "df" dataframe.
date.time <- paste(date, time, sep = " ")
df <- data.frame(patient.ID, result, date.time, stringsAsFactors = FALSE)
```

```
df$date.time <- mdy_hm(date.time)
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ patient.ID: chr  "A" "B" "C" "D" ...
##  $ result    : num  5 6 4 7 5
##  $ date.time : POSIXct, format: "2019-06-28 18:45:00" "2019-06-29 16:36:00" ...
```

1.2.1 Exercise

1. Read the file Potassium.csv into a variable called `K.data`. This file contains real K^+ data extracted from SunQuest from Dan's lab for one month from the two ER bays. It contains order time, collection time, receive time and result time. Use the `head()` function to get an idea of how the dates and times are formatted. All the K^+ results presented were run on an ABL800 whole blood analyzer directly from an unspun PST tube.
   - Convert the order, receive, and result times into dates
   - Calculate the order-to-result times and store it a column called TAT ("turn around time"). Use the function `difftime()` to calculate the time difference.
     - What is the median and IQR of the TAT?
     - What is the 90th and 99th percentiles of the TAT?
     - What is the maximum value of TAT?
   - Calculate the receive-to-result times and store it a column called lab.TAT (the analysis time).
     - What is the median and IQR of the lab.TAT?
     - What is the 90th and 99th percentiles of the lab.TAT?
     - What is the maximum value of lab.TAT? What day did it occur?
     - What strange finding have you discovered?

1.3 Pulling the Strings

1.3.1 Manipulating data with grep(), gsub() and regular expressions

In programming, the word 'string' refers to anything treated as text. Strings may be one word or several words, and can include other characters as well. For example, the word "computer" is a string, as is the sentence "I have 4 computers." There are an array of tools in R specifically for working with strings. In general lab data, a string may be a sample identifier, a patient name or hospital admission number, a gender, or a comment on a sample result. We can use strings to build a structure from which to extract information.

Here we have a list of Sample Identifiers where the prefix "C" indicates a chemistry sample while "H" indicates a hematology sample and "Z" indicates a

QC material :

```
sample.ids <- c("C001", "C002", "H001", "H002", "ZC001", "ZC002", "ZH001", "ZH002")
```

identify which samples correspond to QC Samples using grep().

```
library(stringr)
grepl("Z", sample.ids)
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
# returns a TRUE or FALSE to the question "does the string contain the pattern ZC?"
grep("Z", sample.ids)
```

```
## [1] 5 6 7 8
```

```
# returns the location of strings that contain the patern ZC
```

We often want to filter out only the QC or patient data from our datasets, so lets look at how to rename them using `gsub()`

```
gsub("Z", "QC", sample.ids) #what did this do?
```

```
## [1] "C001"   "C002"   "H001"   "H002"   "QCC001" "QCC002" "QCH001" "QCH002"
```

```
gsub("ZC001", "QC", sample.ids) #okay, now we're getting somewhere
```

```
## [1] "C001"  "C002"  "H001"  "H002"  "QC"    "ZC002" "ZH001" "ZH002"
```

This is great, but replacing each string one by one is a little silly. What if we can look for the pattern "Z" followed by any number of other characters and replace them all with "QC"? You can. It's called a "Regular Expression", and just to show you the magic:

```
gsub("Z.*", "QC", sample.ids)
```

```
## [1] "C001" "C002" "H001" "H002" "QC"   "QC"   "QC"   "QC"
```

Included in this course package is a "Cheat Sheet" which shows you all of the regular expressions and what their uses are. In this case the "." represents "Any Character" while the "*" symbol represents "matches at least 0 times". So in essence, the above script is saying to match any string that contains the pattern "Z" and has any number of characters following it.

Other useful notations include "^" meaning "begins with,"|" meaning "or", and "&" meaning "and". So few other ways to do the same thing we already did:

```
gsub("^Z.*", "QC", sample.ids) # begins with "Z" and has text after
```

```
## [1] "C001" "C002" "H001" "H002" "QC"   "QC"   "QC"   "QC"
```

```
# useful to restrict the pattern to look only at strings that start with Z
gsub("ZC.*|ZH.*", "QC", sample.ids) #change ZC or ZH containing string to "QC"
```

```
## [1] "C001" "C002" "H001" "H002" "QC"   "QC"   "QC"   "QC"
```

if you want to specify Chemistry or Hematology QC, you will need to do multiple replacements and define the varaible each time.

```
sample.ids <- gsub("ZC.*", "Chemistry QC", sample.ids)
sample.ids <- gsub("ZH.*", "Hematology QC", sample.ids)
sample.ids
```

```
## [1] "C001"          "C002"          "H001"          "H002"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

Now to identify patient samples as well. Samples starting with "C" are chemistry specimens and those starting with "H" are hematology, then each letter is followed by some series of numbers from 0 to 9. The regular expression for this would be to use "" which means "one of".

```
# works if there is always a zero after the "C" in the patient result
gsub("^C0.*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "H001"          "H002"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
#works for any number after the "C"
gsub("^C[0-9].*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "H001"          "H002"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# now we'll include an "or" so we can convert chem and heme samples to "Patient"
gsub("^C[0-9].*|^H[0-9].*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "Patient"       "Patient"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# side note that this works using letters as well as numbers
gsub("^[CH]0.*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "Patient"       "Patient"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# this works because your string
# 1) "^" starts with,
# 2) "one of C or H",
# 3) followed by a zero and
# 4) "." any character, 5) which appears "*" at least 0 times

gsub("^[A-Z]0.*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "Patient"       "Patient"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# this will work with any letters from A to Z
```

1.4 Coping with Non-numeric Data

Now that we know how to fix our data, we will want to process the numeric data. To do so, lets use another dataset. This is a made-up QC dataset containing data for liver panel tests Albumin, ALT, Ammonia, Total Bilirubin, and Direct Bilirubin. There are 8 columns including identifiers for Test, Specimen ID, Test Site, QC Mnemonic, QC Lot number, Analyzer Name, Result, and Date/time of result. There are 7 days of QC data in here for three sites from six analyzers, so we're looking at lots of QC data. R can handle much much more than that, but this is a good start.

First, use `read.csv()` to import the data file "QCData_Jun2019.csv" to a dataframe called `qc.data`. Start by surveying the data using `head()` and `str()`

```
qc.data <- read.csv(file = "Data_Files/QCData_Jun2019.csv", sep = ",")
str(qc.data)
```

```
## 'data.frame':    570 obs. of  8 variables:
##  $ Test       : Factor w/ 5 levels "ALB","ALT","AMM",..: 5 5 5 5 5 5 5 5 5 5 ...
##  $ Spec_Num   : Factor w/ 277 levels "0106:C00046Q",..: 1 2 3 6 8 9 10 11 12 15 ...
##  $ Site       : Factor w/ 3 levels "A","B","C": 1 1 1 3 3 3 2 2 2 2 ...
##  $ QC_Mnemonic: Factor w/ 7 levels "AMML1","BIL3",..: 7 5 6 2 6 5 7 5 6 7 ...
##  $ QC_Lot     : Factor w/ 7 levels "31851","31852",..: 5 1 2 5 2 1 6 1 2 7 ...
##  $ Analyzer   : Factor w/ 6 levels "Byfuglien","Conner",..: 3 3 3 2 2 2 6 5 5 5 ...
##  $ Result     : Factor w/ 308 levels "105.4","105.7",..: 143 306 253 182 294 80 164 107 3
##  $ Result_Date: Factor w/ 232 levels "2018-06-01 10:41",..: 7 8 9 10 12 13 14 15 16 17 ..
```

```
# notice all of the columns are being imported as Factors we didn't set
# stringsAsFactors to FALSE

head(qc.data)
```

```
##   Test     Spec_Num Site QC_Mnemonic  QC_Lot   Analyzer Result
## 1 BILT 0106:C00046Q    A      CMBIL3 BC18093 Hellebuyck  363.4
## 2 BILT 0106:C00064Q    A      CBLIQ1   31851 Hellebuyck    ERR
## 3 BILT 0106:C00066Q    A      CBLIQ2   31852 Hellebuyck   78.7
## 4 BILT 0106:C00109Q    C        BIL3 BC18093     Conner  387.7
## 5 BILT 0106:C00116Q    C      CBLIQ2   31852     Conner   90.8
## 6 BILT 0106:C00120Q    C      CBLIQ1   31851     Conner   16.8
##       Result_Date
## 1 2018-06-01 2:36
## 2 2018-06-01 3:02
## 3 2018-06-01 3:05
## 4 2018-06-01 7:06
```

```
## 5 2018-06-01 7:14
## 6 2018-06-01 7:15
```

```
# gives you an idea of the type of data you're dealing with
```

Okay, now we just have to format the columns as we wish.

```
as.numeric(qc.data$Result)
```

Wait–what just happened here? This makes no sense at all. These are all integers between 1 and 308. Do you see what has happened?

R's default handling of converting of a factor variable to a numeric variable is to convert it to the number of the factor (1 through 308 in this case). We personally find this irritating, but it is evidently working as intended.

So the correct thing to do is:

```
as.numeric(as.character(qc.data$Result))
```

```
## Warning: NAs introduced by coercion
```

The default R behaviour of treating strings as factors can be over–ridden when the csv file is read by specifying `stringsAsFactors = FALSE`, same as the `data.frame()` function we used before. When one does this, the `as.character` part of the expression above is unnecessary.

```
qc.data <- read.csv(file = "Data_Files/QCData_Jun2019.csv", stringsAsFactors = FALSE)
# re-read the data using stringAsFactors = FALSE
qc.data$Result <- as.numeric(qc.data$Result)
```

```
## Warning: NAs introduced by coercion
```

```
str(qc.data) # results are numeric
```

```
## 'data.frame':    570 obs. of  8 variables:
##  $ Test       : chr  "BILT" "BILT" "BILT" "BILT" ...
##  $ Spec_Num   : chr  "0106:C00046Q" "0106:C00064Q" "0106:C00066Q" "0106:C00109Q" ...
##  $ Site       : chr  "A" "A" "A" "C" ...
##  $ QC_Mnemonic: chr  "CMBIL3" "CBLIQ1" "CBLIQ2" "BIL3" ...
##  $ QC_Lot     : chr  "BC18093" "31851" "31852" "BC18093" ...
##  $ Analyzer   : chr  "Hellebuyck" "Hellebuyck" "Hellebuyck" "Conner" ...
##  $ Result     : num  363.4 NA 78.7 387.7 90.8 ...
##  $ Result_Date: chr  "2018-06-01 2:36" "2018-06-01 3:02" "2018-06-01 3:05" "2018-06-01 7:
```

```
head(qc.data) # data looks good
```

```
##   Test     Spec_Num Site QC_Mnemonic  QC_Lot   Analyzer Result
## 1 BILT 0106:C00046Q    A      CMBIL3 BC18093 Hellebuyck  363.4
## 2 BILT 0106:C00064Q    A      CBLIQ1   31851 Hellebuyck     NA
## 3 BILT 0106:C00066Q    A      CBLIQ2   31852 Hellebuyck   78.7
```

```
## 4 BILT 0106:C00109Q    C        BIL3 BC18093     Conner  387.7
## 5 BILT 0106:C00116Q    C      CBLIQ2   31852     Conner   90.8
## 6 BILT 0106:C00120Q    C      CBLIQ1   31851     Conner   16.8
##       Result_Date
## 1 2018-06-01 2:36
## 2 2018-06-01 3:02
## 3 2018-06-01 3:05
## 4 2018-06-01 7:06
## 5 2018-06-01 7:14
## 6 2018-06-01 7:15
```

Something else somewhat surprising has happened. What do you notice? See the NA? This is what all non-numerics become, even if one is < 40 and another is > 200 ("NA" stands for "not available"). In this case, some text is present within the QC data. So, be careful. If we want to preserve something of what was actually in the data file, we will need another approach. If we are happy to have all non-numerics display NAs, then what we have here is fine.

But if we now want to look at the statistics of the Results column, the NA results cannot contribute. It would be good to figure out what is going on. In this case, all we want to do is remove the lines of data that are non-numeric because they contain nonsense information, but keep in mind that this may not be the case if you are looking at $>$ or $<$ values in your patient data. You can use `gsub()` to remove or replace these symbols with whatever you desire.

Identify rows of data that contain NA using `is.na()`

```
is.na(qc.data$Result)
# this is a logical variable that asks "is this value NA?" for each row of data.
# Neat, but not useful here.
```

This uses the function `is.na()` to generate a dataframe containing lines that have NA in the Result column. It's a good idea to use this to check what data you're removing before you actually do it.

```
# there are four values in the dataset that are now NA, corresponding to
# specific sample ID's
qc.data[is.na(qc.data$Result),]
```

```
##     Test     Spec_Num Site QC_Mnemonic QC_Lot   Analyzer Result
## 2   BILT 0106:C00064Q    A      CBLIQ1  31851 Hellebuyck     NA
## 9   BILT 0106:C00183Q    B      CBLIQ2  31852  Scheifele     NA
## 503  ALB 0306:C00242Q    A      CBLIQ1  31851  Byfuglien     NA
## 534  ALB 0506:C00118Q    C      CBLIQ1  31851     Conner     NA
##         Result_Date
## 2   2018-06-01 3:02
## 9   2018-06-01 8:13
## 503 2018-06-03 8:51
## 534 2018-06-05 7:11
```

So what we want is to use an ! (represents "is not") ahead of the function `is.na()` to result the rows of data that "are not NA"

```
num.qcdata <- qc.data[!is.na(qc.data$Result),]
head(num.qcdata)
```

```
##   Test     Spec_Num Site QC_Mnemonic   QC_Lot   Analyzer Result
## 1 BILT 0106:C00046Q    A      CMBIL3  BC18093 Hellebuyck  363.4
## 3 BILT 0106:C00066Q    A      CBLIQ2    31852 Hellebuyck   78.7
## 4 BILT 0106:C00109Q    C        BIL3  BC18093     Conner  387.7
## 5 BILT 0106:C00116Q    C      CBLIQ2    31852     Conner   90.8
## 6 BILT 0106:C00120Q    C      CBLIQ1    31851     Conner   16.8
## 7 BILT 0106:C00127Q    B      CMBIL3 BC18093A    Wheeler  376.3
##       Result_Date
## 1 2018-06-01 2:36
## 3 2018-06-01 3:05
## 4 2018-06-01 7:06
## 5 2018-06-01 7:14
## 6 2018-06-01 7:15
## 7 2018-06-01 7:18
```

```
str(num.qcdata) # perfect, no more NAs!
```

```
## 'data.frame':    566 obs. of  8 variables:
##  $ Test       : chr  "BILT" "BILT" "BILT" "BILT" ...
##  $ Spec_Num   : chr  "0106:C00046Q" "0106:C00066Q" "0106:C00109Q" "0106:C00116Q" ...
##  $ Site       : chr  "A" "A" "C" "C" ...
##  $ QC_Mnemonic: chr  "CMBIL3" "CBLIQ2" "BIL3" "CBLIQ2" ...
##  $ QC_Lot     : chr  "BC18093" "31852" "BC18093" "31852" ...
##  $ Analyzer   : chr  "Hellebuyck" "Hellebuyck" "Conner" "Conner" ...
##  $ Result     : num  363.4 78.7 387.7 90.8 16.8 ...
##  $ Result_Date: chr  "2018-06-01 2:36" "2018-06-01 3:05" "2018-06-01 7:06" "2018-06-01 7:
```

1.4.1 Exercise

Let's apply what we just learned using `gsub()` our QC data set. * Replace the short names in the "Test" column "ALB", "ALT", "AMM", "BILD", and "BILT" with their full names. * Format the Result_Date column as a date using `lubridate()` as we did before.

2 Lesson 4: Meet the 'tidyverse' - Gather, Join, Filter, and Clean

There is a paradigm of R programming referred to as the "tidyverse" that is particularly useful for certain types of data summary and data visualization. It allows for rapid-high level commands accomplishing a great deal in few lines of highly readable code. The challenge of using tidyverse packages is that they are under very rapid development and this can mean that updates in the packages can cause your code to stop functioning. Additionally, many statistical packages use traditional "base R" and they may not cooperate with tidyverse output. However, in the context of our work in health care, we are usually doing fairly simple one-off reports for research or answering a specific question so using the tidyverse makes sense.

2.1 Packages

Before we can start using these functions we will have to install and load them. This is because the 'tidyverse' does not exist in 'base' R (that is, the pre-loaded set of function which are installed when you install the R program). Rather, the tidyverse is a set of packages. Packages are themselves sets of functions which have been written by whomever authoured the package. These user-created functions have been 'packaged' and made available for anyone to download and use.

Packages are what makes R great. Lots can be done in base R but often to acheive what you want, you end up having to write your own functions. This is fine if you like programming and find coding your own functions enjoyable. Sometimes, though, we just want to 'get it done' - and whatever 'it' is that you are trying to do, chances are that someone out there has done it before and written a package for you to make it easier.

A final word about packages - they are only as good as the person who wrote them. The tidyverse is a group of packages which have been written by Hadley Wickham (who created RStudio) and his team - which means that we can rely on them being functional, updated and usually bug-free. The same cannot always be said of smaller packages, so if you are ever poking around for a package to fill a niche need, read the documentation and do some google searches to see how others have been able to work with the package before spending too much time trying to code with it.

2.1.1 Exercise

- Hopefully you installed the tidyverse when you first installed R as per the pre-course instructions. Just in case, the following code will install the

tidyverse package if it is not already there.

```
if("tidyverse" %in% rownames(installed.packages()) == FALSE) {install.packages("tidyverse")}
```

This command asks R to download the most current version of the tidyverse from CRAN, which is an online package repository. Note that you need to be connected to the internet for this to work.

Since quite a few packages are being downloaded, this may take a few minutes. Perhaps a good time for a coffee break.

Once tidyverse is installed, we need to load it by calling library():

```
library(tidyverse)
```

You only need to install a package once, but you will need to load any packages you need each time you restart R.

2.2 gather() and spread()

Now that we are in the tidyverse, the first thing we want to introduce is what is meant by "tidy data".

When humans prepare spreadsheet data, they typically have each row represent all the observations on single subject. This is usually pretty easy to look at but it happens to have a number of disadvantages from a statistical programming standpoint.

For example, in traditional "untidy" data, you might have each subject on a row and all of their lab tests represented as columns: Sodium, Potassium, Chloride, Bicarbonate, Creatinine, pH, Troponin etc. But in the tidy data paradigm, all of the blood tests should be factors under a single column labelled "Test" and then each row represents a single observation, not a single patient.

It is frequently necessary to jump back and forth between the traditional view, which we call "data wide" and the tidy view which we call "data long". This is accomplished with the functions `gather()` and `spread()`.

Let's starting exploring these funtions by reading in some data on "anthropometric" (for want of a better term) measurements on opossum.

```
possum <- read_csv("Data_Files/possum.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   X1 = col_character(),
##   case = col_double(),
##   site = col_double(),
```

```
##   Pop = col_character(),
##   sex = col_character(),
##   age = col_double(),
##   hdlngth = col_double(),
##   skullw = col_double(),
##   totlngth = col_double(),
##   taill = col_double(),
##   footlgth = col_double(),
##   earconch = col_double(),
##   eye = col_double(),
##   chest = col_double(),
##   belly = col_double()
## )
```

```
# we used read_csv here (a tidyverse function) instead of read.csv (base R)
# the anthropomorphic measurements of the possums are wide
head(possum)
```

```
## # A tibble: 6 x 15
##   X1     case  site Pop   sex     age hdlngth skullw totlngth taill
##   <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>
## 1 C3        1     1 Vic   m         8    94.1   60.4     89    36
## 2 C5        2     1 Vic   f         6    92.5   57.6     91.5  36.5
## 3 C10       3     1 Vic   f         6    94     60       95.5  39
## 4 C15       4     1 Vic   f         6    93.2   57.1     92    38
## 5 C23       5     1 Vic   f         2    91.5   56.3     85.5  36
## 6 C24       6     1 Vic   f         1    93.1   54.8     90.5  35.5
## # ... with 5 more variables: footlgth <dbl>, earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

We can convert them to data long format with the `gather()` function. In the this function we must determine three things:

1. What columns are to be gathered together as factors of a similar type?– This parameter referred to as the "key" and in our output the column will be named "Possum_Metric".
2. What do you want to call the column that contains the associated values?– This parameter is referred to as the "value" and in our output the column will be named "Result".
3. Which columns (by name or number) are to be gathered?–In this case it is columns 7–15. You can also denote which columns you *don't* want gathered – so we could also write this as "-c(1:6)".

```
possum.long <- gather(possum, key = Possum_Metric, value = Result, 7:15)
head(possum.long,10)
```

```
## # A tibble: 10 x 8
##    X1     case  site Pop   sex     age Possum_Metric Result
```

```
##    <chr> <dbl> <dbl> <chr> <chr> <dbl> <chr>          <dbl>
##  1 C3        1     1 Vic   m         8 hdlngth         94.1
##  2 C5        2     1 Vic   f         6 hdlngth         92.5
##  3 C10       3     1 Vic   f         6 hdlngth         94
##  4 C15       4     1 Vic   f         6 hdlngth         93.2
##  5 C23       5     1 Vic   f         2 hdlngth         91.5
##  6 C24       6     1 Vic   f         1 hdlngth         93.1
##  7 C26       7     1 Vic   m         2 hdlngth         95.3
##  8 C27       8     1 Vic   f         6 hdlngth         94.8
##  9 C28       9     1 Vic   f         9 hdlngth         93.4
## 10 C31      10     1 Vic   f         6 hdlngth         91.8
```

As it turns out, this permits very slick calculations which allow you to rapidly generate summary statistics based on variables of your choosing.

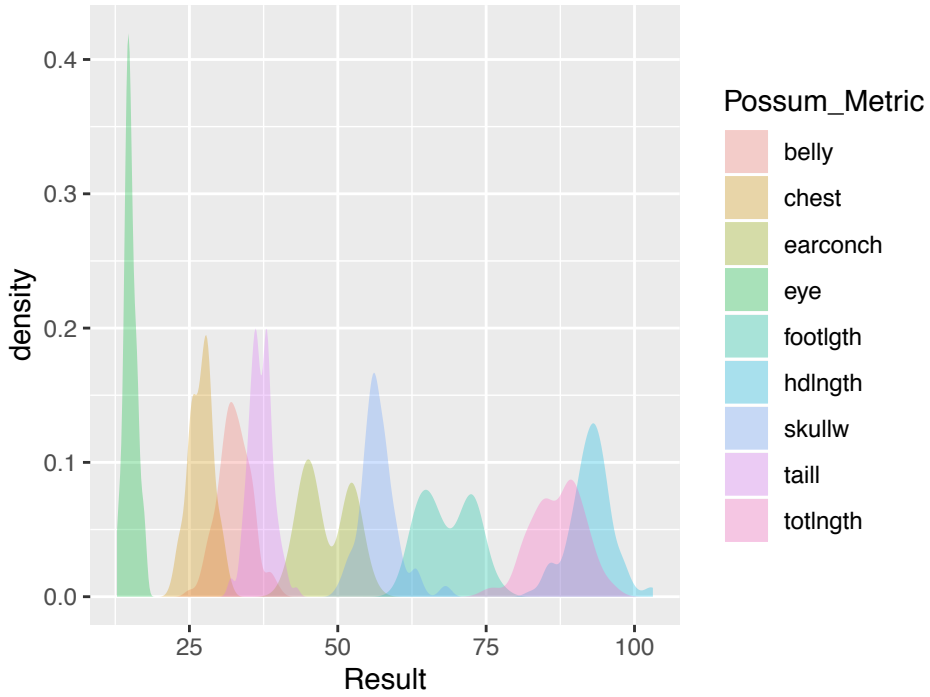
```
#long data permits rapid calculations
group_by(.data = possum.long, key = Possum_Metric) %>%
  summarise(Mean = mean(Result, na.rm = TRUE),
            SD = sd(Result, na.rm = TRUE),
            "%CV" = (100*sd(Result, na.rm = TRUE)/mean(Result, na.rm = TRUE)))
```

```
## # A tibble: 9 x 4
##   key       Mean    SD `%CV`
##   <chr>    <dbl> <dbl> <dbl>
## 1 belly     32.6  2.76  8.48
## 2 chest     27    2.05  7.58
## 3 earconch  48.1  4.11  8.54
## 4 eye       15.0  1.05  6.98
## 5 footlgth  68.5  4.40  6.42
## 6 hdlngth   92.6  3.57  3.86
## 7 skullw    56.9  3.11  5.47
## 8 taill     37.0  1.96  5.29
## 9 totlngth  87.1  4.31  4.95
```

and plots:

```
#long data permits rapid visualizations (which we will cover later)
library(ggplot2)
p <- ggplot(possum.long, aes(x = Result, fill = Possum_Metric)) +
  geom_density(alpha = 0.3, color = NA)
p
```

```
## Warning: Removed 1 rows containing non-finite values (stat_density).
```



If we want to convert back to 'untidy' data, this is accomplished by `spread()`. For this function again we need to determine our "key" and "value" columns:

1. The "key" column will be "spread" across the top of the table (this column becomes the column names).
2. The "value" column contains the values we want to populate these new columns.

```
possum.wide <- spread(possum.long, key = Possum_Metric, value = Result)
head(possum.wide, 10)
```

```
## # A tibble: 10 x 15
##    X1     case  site Pop   sex     age belly chest earconch   eye footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl> <dbl> <dbl>    <dbl> <dbl>    <dbl>
##  1 A1       29     1 Vic   f         3  32    24       51.8  14       74.9
##  2 A2       30     1 Vic   f         2  33    24.5     50.8  14.5     70.6
##  3 A3       31     1 Vic   m         3  31    27       52.5  14.5     68
##  4 A4       32     1 Vic   f         4  34    28       52    14.9     74.8
##  5 AD1      33     1 Vic   m         3  30    24       51.8  14.8     70.8
##  6 BB13     35     2 Vic   m         4  35.5  28       55.5  16.4     71.2
##  7 BB15     36     2 Vic   m         7  36    25.5     52    14.9     74.3
##  8 BB17     37     2 Vic   f         2  31.5  28       52    13.6     71.2
##  9 BB25     38     2 Vic   m         7  30    27       49.5  15.9     68.4
## 10 BB31     39     2 Vic   f         1  25    25       53.4  13       68.7
## # ... with 4 more variables: hdlngth <dbl>, skullw <dbl>, taill <dbl>,
```

```
## #   totlngth <dbl>
```

2.2.1 Exercise

Using the `num.qc` dataset you generated earlier, summarise the QC data running means and sds according to test, qc mnemonic, and site to a new dataframe called `qc.means`.

2.3 `arrange()`, `filter()`, and `select()`

There are some more very useful and simple functions in the tidyverse which we will need before we get too much further.

The first is `arrange()` and it does pretty much what you think it would. Let's try it out on the QC data:

```
arrange(num.qcdata, Site)
```

If we want to arrange from largest to smallest, we would ask for the ages in descending `desc()` order:

```
arrange(num.qcdata, desc(Site))
```

What happens if we `arrange()` something that is not a character?

```
arrange(num.qcdata, Result) # numeric variable
arrange(num.qcdata, Result_Date) # date variable
```

Next up, `filter()` and `select()`. Filtering means choosing *rows* while selecting means choosing *columns*. You can filter rows based on what the rows contain, but you can only select columns by name or position.

```
filter(num.qcdata, Site == "B") # all results from site B
filter(num.qcdata, Result < 10) # all results less than 10

# if you want to filter by multiple criteria, simply join the
# filter criteria using an & symbol
filter(num.qcdata, Site == "A" & Test == "AMM" & QC_Mnemonic == "CBALC1")

# can also filter by date range
filter(num.qcdata, Result_Date >= "2018-06-01" & Result_Date < "2018-06-04")
```

```
select(num.qcdata, Test, Site, Result) # returns the named columns
select(num.qcdata, 1:7) # returns columns 1 to 7
```

2.3.1 Exercise

- From the num.qcdata data, create a table which only has the specimen number, site and result for Albumin CBLIQ1 QC run on June 3, 2018. Arrange this table by site.

2.4 Acknowledgements

- Dan Holmes, Stephen Master, Will Slade & Janet Simons's Intro to R Workshop

Stat of the Union

Getting Started with R for Laboratory Medicine
Sunday Aug 04, 2019
AACC 2019, Anaheim CA

Stephen Master, Dan Holmes, Will Slade, Janet Simons

7/23/2019

Contents

1 Lesson 5: Basic statistics and regression with R **1**
1.1 Correlation 2
1.2 Comparing Mean and Central Tendency 4
1.3 Regression 8
1.4 Weighted Least Squares 12
1.5 Outlier Effects in OLS 14
1.6 Passing Bablok Regression 16
1.7 Deming Regression 17
1.8 Non-Linear Regression 19
1.9 Putting it all together: Linearity Testing 23

1 Lesson 5: Basic statistics and regression with R

OK, we have some data, it's formatted the way we want, and we know how to do some basic calculations with it. Let's get fancier. In order to dig in and really understand data from our clinical laboratory, we often need to perform some sort of statistical test. Since R began life as a statistical language, it is ideally suited for this task. Most, if not all, common statistical procedures–t-tests, nonparametric comparisons, ANOVA, you name it–have built-in functions within R. Better yet, because R is so widely used within the statistical community, pretty much any statistical procedure that you can think of or will read about is available as an R package. This makes R a seriously powerful tool for analyzing your lab data–much, much more powerful than Excel. Let's look at how to do some simple statistical calculations in R.

1.1 Correlation

Let's start with one of the most common situations. We often want to calculate the correlation between two variables (perhaps we want to know whether there's a nice, linear relationship between lab results from two different platforms). The simplest way to calculate this in R is using the function `cor()`. For example:

```
x <- c(1,2,3,4,5)
y <- c(1.1,1.9,7,6,8)
cor(x,y)
```

```
## [1] 0.9108501
```

But be warned: `cor()`, like many R functions, needs you to declare what you want done with missing values. To see what we mean:

```
x <- c(1,2,3,4,NA)
y <- c(1.1,1.9,7,6,8)
cor(x,y)
```

```
## [1] NA
```

This gives us NA as a result with no error. Useless. `help(cor)` tells us what is going on.

```
cor(x,y,use = "complete.obs") #not terribly intuitive
```

```
## [1] 0.8713087
```

The `mean()` function also does not like NAs, but the syntax of handling them is more widely used throughout the language (specifically, `mean(x,na.rm = TRUE)`).

Onward... let's load in some data comparing tube types:

```
tube.data <- read.csv("Data_Files/tube_data.csv")
head(tube.data)
```

```
##   Subject LiHep EDTA SST
## 1       1  11.4 13.2 9.7
## 2       2   4.7  5.3 4.6
## 3       3   7.8  8.8 6.9
## 4       4  10.4 11.7 9.4
## 5       5   9.5 11.1 8.6
## 6       6  10.2 12.3 8.6
```

Now, what do you suppose this produces?

```
cor(tube.data$LiHep, tube.data$EDTA)
```

```
## [1] 0.9976726
```

Conveniently, we can also get `cor()` to do multiple correlations at once.

```
cor(tube.data[,2:4])
```

```
##           LiHep      EDTA       SST
## LiHep 1.0000000 0.9976726 0.9651111
## EDTA  0.9976726 1.0000000 0.9633474
## SST   0.9651111 0.9633474 1.0000000
```

As nice as this is, we are not a huge fans of `cor()` because it lacks p-values and confidence intervals for the correlation. For this reason, we often use `cor.test()` (which automatically copes with missing values, by the way).

```
x <- c(1,2,3,4,NA)
y <- c(1.1,1.9,7,6,8)
cor.test(x,y)
```

```
##
##  Pearson's product-moment correlation
##
## data:  x and y
## t = 2.511, df = 2, p-value = 0.1287
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.5521545  0.9972745
## sample estimates:
##       cor
## 0.8713087
```

The drag is that `cor.test()` does not calculate the whole correlation matrix for you... but it does give you just about everything else you would want to know about the correlation.

This brings us to another side point. When you are doing a statistical test like this, you can store the whole analysis in a variable that you can call on later to, say, put in a plot or use in another calculation.

```
my.cor <- cor.test(tube.data$EDTA,tube.data$SST)
my.cor # displays
```

```
##
##  Pearson's product-moment correlation
##
## data:  tube.data$EDTA and tube.data$SST
## t = 15.236, df = 18, p-value = 9.925e-12
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.9078399 0.9856741
## sample estimates:
```

```
##       cor
## 0.9633474
```

```
str(my.cor) # tells you all the variables stored in my.cor
```

```
## List of 9
##  $ statistic  : Named num 15.2
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named int 18
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 9.93e-12
##  $ estimate   : Named num 0.963
##   ..- attr(*, "names")= chr "cor"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "correlation"
##  $ alternative: chr "two.sided"
##  $ method     : chr "Pearson's product-moment correlation"
##  $ data.name  : chr "tube.data$EDTA and tube.data$SST"
##  $ conf.int   : num [1:2] 0.908 0.986
##   ..- attr(*, "conf.level")= num 0.95
##  - attr(*, "class")= chr "htest"
```

```
my.cor$estimate # is your correlation
```

```
##       cor
## 0.9633474
```

```
paste(round((my.cor$estimate)^2,3), "is my R-squared, the coefficient of determination")
```

```
## [1] "0.928 is my R-squared, the coefficient of determination"
```

1.2 Comparing Mean and Central Tendency

1.2.1 The t-test

As you know, the t-test is used to compare the means of two groups to see if there is a statistically significant difference. Usually, our "null hypothesis" is that there is no difference between the groups, and this is how we most often use this test in clinical chemistry.

When we are comparing groups that are not *a priori* connected (e.g. testosterone levels in males from Philadelphia to testosterone levels in males from British Columbia), we would use the t-test in its unpaired form. The syntax is: `t.test(x,y)`–where `x` and `y` are vectors of the data points for each group.

However, we are very often comparing data that is paired–e.g., collections from the same individuals on different tube types, or lipid levels pre- and post-statin therapy, or Vitamin D levels in summer vs. winter for the same people.

If we are doing a paired t-test, the syntax is only slightly different: `t.test(x,y, paired = TRUE)`.

1.2.2 Exercise

- Use t-tests to compare the mean results from the three different tube types.
- Are there difference in results by the different tube types?
- Why is doing multiple t-tests bad practice?
- What is the "right" way to do this analysis?

If you look at `help(t.test)`, you can find the syntax for changing the confidence level to something other than 0.95, using one-sided vs. two sided, etc.

For completeness: `t.test()` has an alternate syntax for data that is arranged so that tube types are stored as factors all in one column and results are stored in a second column. To show you, we will store the tube type data differently.

(Oh, by the way: this uses the "tidyr" library, so make sure you run `install.packages("tidyr")` before moving on...)

```
library(tidyr)
tube.data.2 <- gather(tube.data, "Tube", "PTH", -Subject)
tube.data.2 # do this on you own to see what happens

# also, since this approach requires only two factors
#   in the column for comparison:
data.for.t.test <- subset(tube.data.2, Tube=="EDTA" | Tube=="SST")
data.for.t.test # try this also
```

```
t.test(PTH ~ Tube, data=data.for.t.test, paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  PTH by Tube
## t = 2.7367, df = 19, p-value = 0.01311
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.2222684 1.6677316
## sample estimates:
## mean of the differences
##                   0.945
```

1.2.3 The Problem of Multiple Comparisons

You can circumvent the whole problem of multiple comparisons by using the function `pairwise.t.test()`, which makes corrections in the p-value to adjust

for the fact that you are doing the t-test multiple times. This prevents the so-called "Type I Error", which is a "false positive" (rejection of the null hypothesis when it is actually true, or–loosely–erroneously declaring a difference when there is none).

In any case, if we apply:

```
pairwise.t.test(tube.data.2$PTH,tube.data.2$Tube ,p.adj = "bonf",paired = TRUE)
```

We automatically get all the p-values for the pairwise comparisons with $p < 0.05$ considered significant **after the multiple comparisons effect has been accounted for**. What does our output mean?

```
##
##  Pairwise comparisons using paired t tests
##
## data:  tube.data.2$PTH and tube.data.2$Tube
##
##       EDTA   LiHep
## LiHep 0.0019 -
## SST   0.0393 1.0000
##
## P value adjustment method: bonferroni
```

Here we have chosen the very conservative Bonferroni method to adjust the p-value. There are multiple other less conservative approaches to the p-value adjustment. Type `help(p.adjust)` for details.

1.2.4 The Wilcoxon Signed Rank and Rank Sum Tests

The t-test assumes that the results from the two groups to be compared are normally distributed about the respective means. While this is something we can test for (with the Shapiro Wilk Test or the Kolmogorov Smirnov test), you can also convince yourself with a histogram that it is not likely true. When you want to compare two groups in a "non-parametric" fashion (lingo for "no assumptions about distribution"), you can use the Wilcoxon Signed Rank Test (which is the non-parametric analog of the paired t-test). If the data is not paired, the Wilcoxon Rank Sum Test is performed when we use the same syntax. This is the analog of the unpaired t.test and is also called the "Mann Whitney" test.

```
wilcox.test(tube.data$EDTA,tube.data$SST, paired = TRUE)
```

```
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  tube.data$EDTA and tube.data$SST
## V = 139.5, p-value = 0.01979
```

```
## alternative hypothesis: true location shift is not equal to 0
```

```
#alternatively
wilcox.test(PTH ~ Tube,data = data.for.t.test, paired = TRUE)
```

```
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  PTH by Tube
## V = 139.5, p-value = 0.01979
## alternative hypothesis: true location shift is not equal to 0
```

1.2.5 Basic ANOVA

Obviously, ANOVA is a course unto itself - but it is straightforward to do a basic (unpaired) ANOVA which is the multivariable analog of the (unpaired) t-test. We can make some mock data for this. Suppose we are comparing average vitamin D concentrations in unsupplemented individuals from Norway, Germany and Italy – 100 otherwise matched subjects from each country.

```
#fake data
set.seed(100)
norway.D <- rnorm(100,70,20)
deutsch.D <- rnorm(100,75,20)
italy.D <- rnorm(100,80,20)

#put in a data frame
nationality <- c(rep("N",100), rep("D",100), rep("I",100))
vitD <- c(norway.D, deutsch.D, italy.D)
my.data <- data.frame(nationality,vitD)
str(my.data) #note what R did to the nationality!
```

```
## 'data.frame':    300 obs. of  2 variables:
##  $ nationality: Factor w/ 3 levels "D","I","N": 3 3 3 3 3 3 3 3 3 3 ...
##  $ vitD       : num  60 72.6 68.4 87.7 72.3 ...
```

```
fit <- aov(vitD ~ nationality)
summary(fit)
```

```
##              Df Sum Sq Mean Sq F value   Pr(>F)
## nationality   2   5200  2599.9   7.076 0.000996 ***
## Residuals   297 109133   367.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

1.2.6 Repeated Measures ANOVA

Performing an ANOVA analysis that is suitable to compare the PTH results from the different tube types is less trivial. This would be a within-subjects or "repeated measures" ANOVA. The code is as follows but the subject is beyond the scope of time that we have.

```
tube.data.2$Subject <- factor(tube.data.2$Subject) #aov needs the subjects to be factors
fit <- aov(PTH ~ Tube + Error(Subject/Tube), data=tube.data.2)
summary(fit)
```

```
##
## Error: Subject
##           Df Sum Sq Mean Sq F value Pr(>F)
## Residuals 19  722.3   38.02
##
## Error: Subject:Tube
##           Df Sum Sq Mean Sq F value  Pr(>F)
## Tube       2  10.45   5.225    7.79 0.00146 **
## Residuals 38  25.48   0.671
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

1.3 Regression

It's time to *progress* to the next topic: *regression* (ha! See what we did there?). Regression turns up all the time in clinical chemistry: calibration curves, assay comparisons during validation, harmonization, looking at relationships between analytes... very useful. In this section, we will cover ordinary least squares (OLS), Deming, and Passing Bablok. These latter two forms of regression have found a neurotic devotion in the Clinical Chemistry literature, and so you have to use them when you publish. They are not available in Excel. The nice thing about Passing Bablok is that it is very resistant to the effect of an outlier, though it is certainly not the only form of robust regression.

1.3.1 Ordinary Least Squares

Let's start with OLS and let's use the tube-type dataset we used in the last section. Let's plot the PTH results of EDTA and SST against one another, since we know that there are statistical differences in the mean and median. The function we will use is called `lm()`

```
tube.data <- read.csv(file = "Data_Files/tube_data.csv")
OLS.reg <- lm(EDTA ~ SST,data = tube.data)
summary(OLS.reg)
```

```
##
## Call:
## lm(formula = EDTA ~ SST, data = tube.data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.64067 -0.90371 -0.04158  0.68455  2.44498
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.22948    0.62417   -1.97   0.0644 .
## SST          1.33713    0.08776   15.24 9.93e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.176 on 18 degrees of freedom
## Multiple R-squared:  0.928,  Adjusted R-squared:  0.924
## F-statistic: 232.1 on 1 and 18 DF,  p-value: 9.925e-12
```

```
#alternative syntax
lm(tube.data$EDTA ~ tube.data$SST)
```

```
##
## Call:
## lm(formula = tube.data$EDTA ~ tube.data$SST)
##
## Coefficients:
##   (Intercept)  tube.data$SST
##        -1.229          1.337
```

Now let's look at all of the things that R calculates:

```
str(OLS.reg)
```

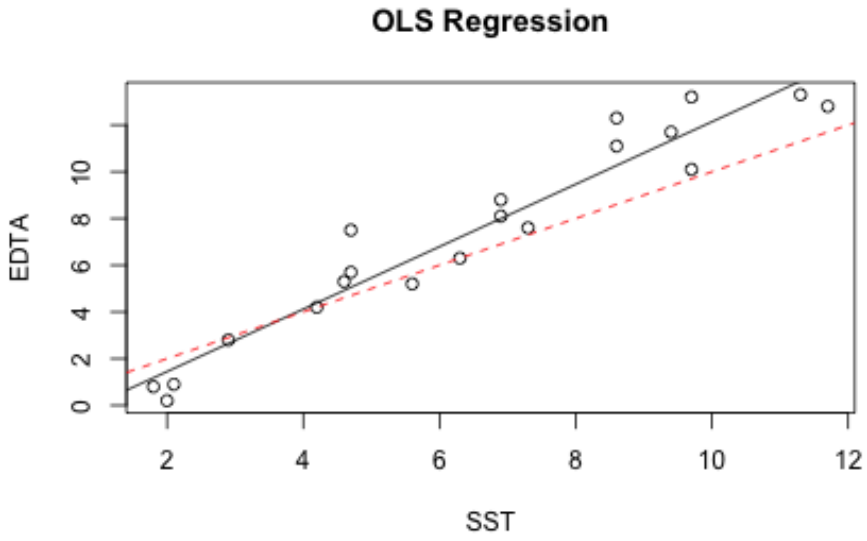
```
## List of 12
##  $ coefficients : Named num [1:2] -1.23 1.34
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "SST"
##  $ residuals    : Named num [1:20] 1.459 0.379 0.803 0.36 0.83 ...
##   ..- attr(*, "names")= chr [1:20] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:20] -33.071 17.919 0.536 0.11 0.574 ...
##   ..- attr(*, "names")= chr [1:20] "(Intercept)" "SST" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:20] 11.74 4.92 8 11.34 10.27 ...
##   ..- attr(*, "names")= chr [1:20] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:20, 1:2] -4.472 0.224 0.224 0.224 0.224 ...
##   .. ..- attr(*, "dimnames")=List of 2
```

```
##   .. .. ..$ : chr [1:20] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "SST"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.22 1.18
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 18
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = EDTA ~ SST, data = tube.data)
##  $ terms        :Classes 'terms', 'formula'  language EDTA ~ SST
##   .. ..- attr(*, "variables")= language list(EDTA, SST)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "EDTA" "SST"
##   .. .. .. ..$ : chr "SST"
##   .. ..- attr(*, "term.labels")= chr "SST"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(EDTA, SST)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "EDTA" "SST"
##  $ model        :'data.frame':   20 obs. of  2 variables:
##   ..$ EDTA: num [1:20] 13.2 5.3 8.8 11.7 11.1 12.3 5.2 8.1 0.8 6.3 ...
##   ..$ SST : num [1:20] 9.7 4.6 6.9 9.4 8.6 8.6 5.6 6.9 1.8 6.3 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language EDTA ~ SST
##   .. .. ..- attr(*, "variables")= language list(EDTA, SST)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "EDTA" "SST"
##   .. .. .. .. ..$ : chr "SST"
##   .. .. ..- attr(*, "term.labels")= chr "SST"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(EDTA, SST)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "EDTA" "SST"
##  - attr(*, "class")= chr "lm"
```

This is probably more information than you wanted for right now, but it's a fairly complete list of the things that you might like to know at some point. To

take a look at your data and add the regression line, you can type:

```
plot(EDTA ~ SST, data = tube.data, main = "OLS Regression")
#to add the regression line
abline(OLS.reg$coefficients)
# abline(OLS.reg$coefficients[1], OLS.reg$coefficients[2]) does same thing
# abline(OLS.reg) also does the same but is less intuitive
# lines(tube.data$SST,OLS.reg$fitted.values) does the same thing
#
# to add the line of identity...
abline(0,1, col = "red", lty = 2)
```



If you enter `plot(OLS.reg)`, you can cycle through some diagnostic plots of the regression.

BONUS MATERIAL: R just keeps getting easier. If you like, the `broom` package will give you a nice data frame-like variable with all the things you might want from `lm()`. Here's how it looks:

```
library(broom)
OLS.reg <- lm(EDTA ~ SST,data = tube.data)
tidy(OLS.reg)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    -1.23    0.624      -1.97 6.44e- 2
```

```
## 2 SST             1.34    0.0878     15.2  9.93e-12
```

Even better, the `broom` package can also give you the residual errors (the difference between your regression line and the original data at every point) using a function called `augment()`.

```
augment(OLS.reg)
```

```
## # A tibble: 20 x 9
##     EDTA   SST .fitted .se.fit .resid   .hat .sigma  .cooksd .std.resid
##    <dbl> <dbl>   <dbl>   <dbl>  <dbl>  <dbl>  <dbl>    <dbl>      <dbl>
##  1  13.2   9.7   11.7    0.388  1.46  0.109    1.15 0.105        1.31
##  2   5.3   4.6    4.92   0.309  0.379 0.0691   1.21 0.00413      0.334
##  3   8.8   6.9    8.00   0.266  0.803 0.0511   1.19 0.0132       0.701
##  4  11.7   9.4   11.3    0.369  0.360 0.0985   1.21 0.00569      0.323
##  5  11.1   8.6   10.3    0.324  0.830 0.0757   1.19 0.0221       0.734
##  6  12.3   8.6   10.3    0.324  2.03  0.0757   1.10 0.132        1.80
##  7   5.2   5.6    6.26   0.273 -1.06  0.0540   1.18 0.0244      -0.925
##  8   8.1   6.9    8.00   0.266  0.103 0.0511   1.21 0.000219     0.0902
##  9   0.8   1.8    1.18   0.485 -0.377 0.170    1.21 0.0127      -0.352
## 10   6.3   6.3    7.19   0.263 -0.894 0.0501   1.19 0.0161      -0.780
## 11   5.7   4.7    5.06   0.305  0.645 0.0671   1.20 0.0116       0.568
## 12   2.8   2.9    2.65   0.408  0.152 0.120    1.21 0.00129      0.138
## 13  10.1   9.7   11.7    0.388 -1.64  0.109    1.13 0.133       -1.48
## 14   0.9   2.1    1.58   0.464 -0.678 0.155    1.20 0.0362      -0.628
## 15   7.5   4.7    5.06   0.305  2.44  0.0671   1.04 0.166        2.15
## 16  13.3  11.3   13.9    0.500 -0.580 0.181    1.20 0.0328      -0.545
## 17   0.2   2      1.44   0.471 -1.24  0.160    1.16 0.127       -1.15
## 18   4.2   4.2    4.39   0.329 -0.186 0.0782   1.21 0.00116     -0.165
## 19   7.6   7.3    8.53   0.273 -0.932 0.0540   1.19 0.0189      -0.814
## 20  12.8  11.7   14.4    0.531 -1.61  0.203    1.13 0.302       -1.54
```

Pretty easy, huh?

1.4 Weighted Least Squares

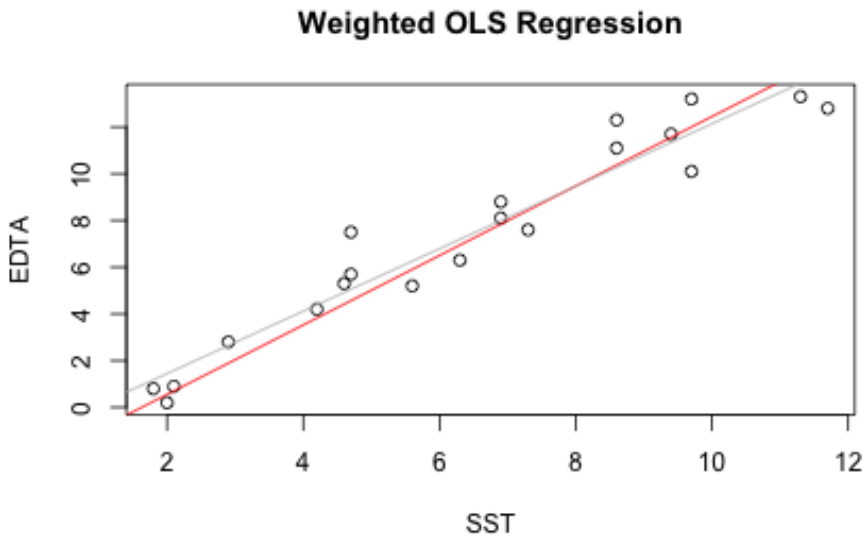
Now, sometimes you want to perform weighted regression. This is what we do on the mass spectrometer when we want to improve the recoveries of the low-level calibrators and the expense of the high level calibrators. In other words, when accuracy at the low end is clinically required, we weight the regression. How we weight is fairly arbitrary, but the bigger the weight, the more fitting effect a point has.

```
w.OLS.reg <-  lm(EDTA ~ SST, data = tube.data, weights = 1/EDTA)
summary(w.OLS.reg)
```

```
##
```

```
## Call:
## lm(formula = EDTA ~ SST, data = tube.data, weights = 1/EDTA)
##
## Weighted Residuals:
##     Min      1Q  Median      3Q     Max
## -0.7859 -0.2977  0.1973  0.4062  1.0712
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2.42248    0.27376  -8.849 5.66e-08 ***
## SST          1.48698    0.07372  20.170 8.32e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5016 on 18 degrees of freedom
## Multiple R-squared:  0.9576, Adjusted R-squared:  0.9553
## F-statistic: 406.8 on 1 and 18 DF,  p-value: 8.319e-14
```

```r
plot(EDTA ~ SST, data = tube.data, main = "Weighted OLS Regression")
#to add the regression line
abline(w.OLS.reg, col = "red")
#put the unweighted line in for comparison
abline(OLS.reg, col = "gray")
```

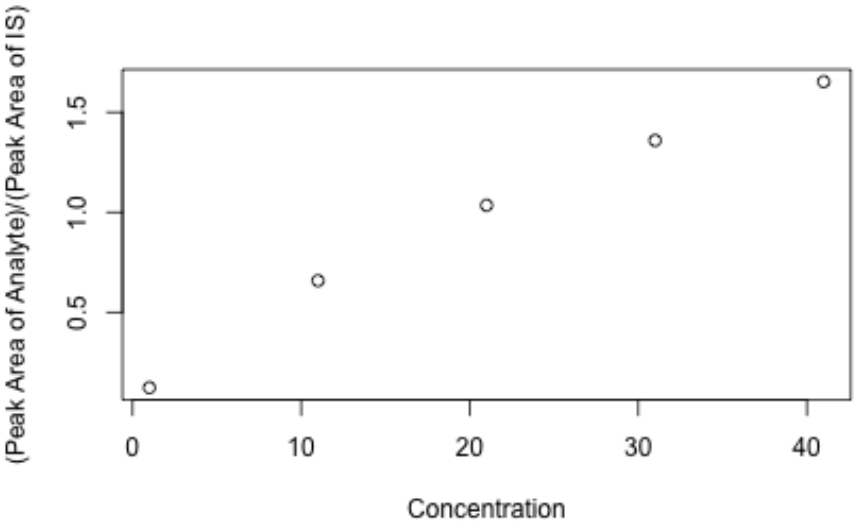


```
#What do you notice?
```

1.4.1 Exercise

- Here is some fake cal curve data that shows the characteristic flattening we see when we are trying to extend our linear range too far:

```
conc <- seq(from = 1, to = 50, by = 10)
response <- (conc/20)^(0.7)
plot(conc, response, xlab = "Concentration", ylab = "(Peak Area of Analyte)/(Peak Area of IS
```



- Find and plot the OLS regression line. Add it to the plot using abline() in green.
- Find and plot the $1/x^2$ weighted OLS regression line. Add it to the plot using `abline()` in purple.
- What is the effect of the weighting?

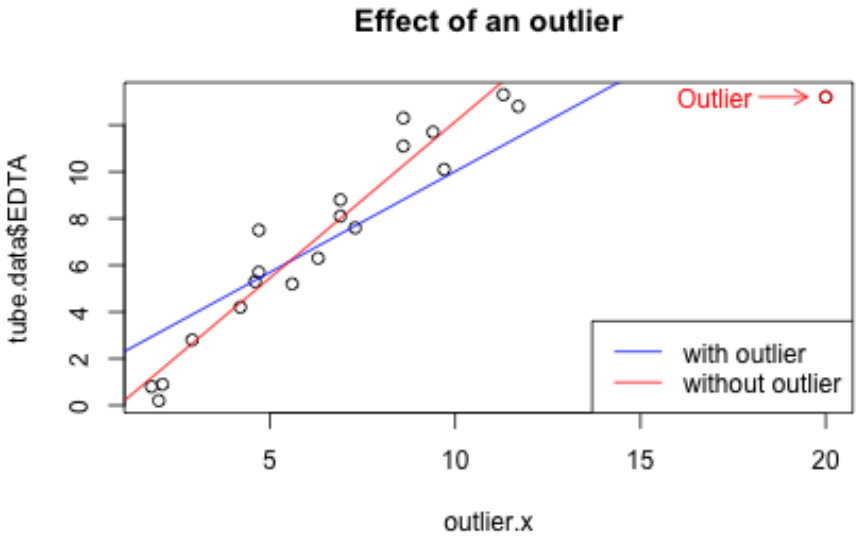
1.5 Outlier Effects in OLS

Outlier effects are significant with OLS regression. To illustrate, we can do the following (you don't need to understand the code, you just need to see the effect an outlier has):

```r
outlier.x <- tube.data$SST
outlier.x[1] <- 20 #introduce a single outlier point
summary(lm(tube.data$EDTA ~ outlier.x))
```

```
##
## Call:
## lm(formula = tube.data$EDTA ~ outlier.x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.4511 -1.0334  0.1041  1.6111  3.4931
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.3805     0.9574   1.442    0.167
## outlier.x     0.8635     0.1180   7.320 8.47e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.198 on 18 degrees of freedom
## Multiple R-squared:  0.7486, Adjusted R-squared:  0.7346
## F-statistic: 53.59 on 1 and 18 DF,  p-value: 8.471e-07
```

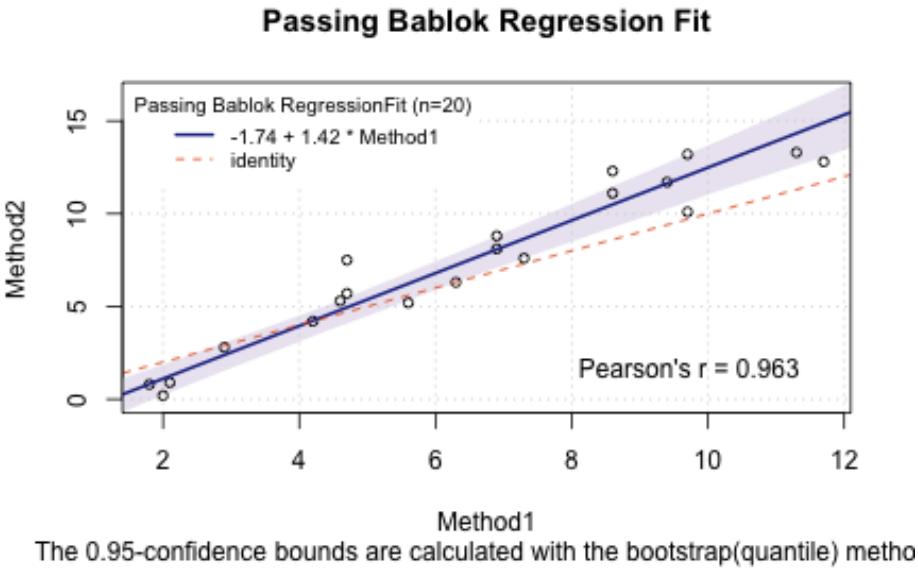
```r
plot(outlier.x, tube.data$EDTA, main = "Effect of an outlier")
abline(lm(tube.data$EDTA ~ outlier.x), col = "blue") #regression with outlier
abline(OLS.reg, col = "red") #regression without outlier
legend("bottomright",c("with outlier","without outlier"), lty = c(1,1), col = c("blue","red"
points(20, 13.2, col = "red")
text(17, 13.2,"Outlier", col = "red")
arrows(x0 = 18.2, y0 = 13.2, x1 = 19.5, y1 = 13.2, length = 0.1, col = "red")
```



1.6 Passing Bablok Regression

Now that you have seen the effect of an outlier, you can see that it would be great to have a regression method that more resistant to the effect of outliers. Passing Bablok regression is such a method. However, this function is not built-in to R. The good thing is that there are packages that include it and we can install them. Statisticians at Roche have contributed a package called "mcr" that contains PB regression (remember to run `install.packages("mcr")` before this next step).

```
library("mcr")
PB.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "PaBa")
plot(PB.reg) # plots a nice generic plot automatically
```



The 0.95-confidence bounds are calculated with the bootstrap(quantile) metho

1.6.1 Exercise

- Use the `mcreg()` function to show that PB regression is more resistant to an outlier than OLS regression. For your x data use outlier.x and for your y data use tube.data$EDTA. Plot the regression line. Use `abline()` to add the regression line from the PB.reg model shown immediately above.

Note that PB regression cannot be weighted by virtue of how the method works. There is no minimization of residuals, so there is nothing to weight. PB regression is very computationally intensive for larger data sets. For this reason, the code authors of the "mcr"" package have developed a method called `PaBaLarge` for large data sets. It's not exact, but it's very close. It would be called like this:

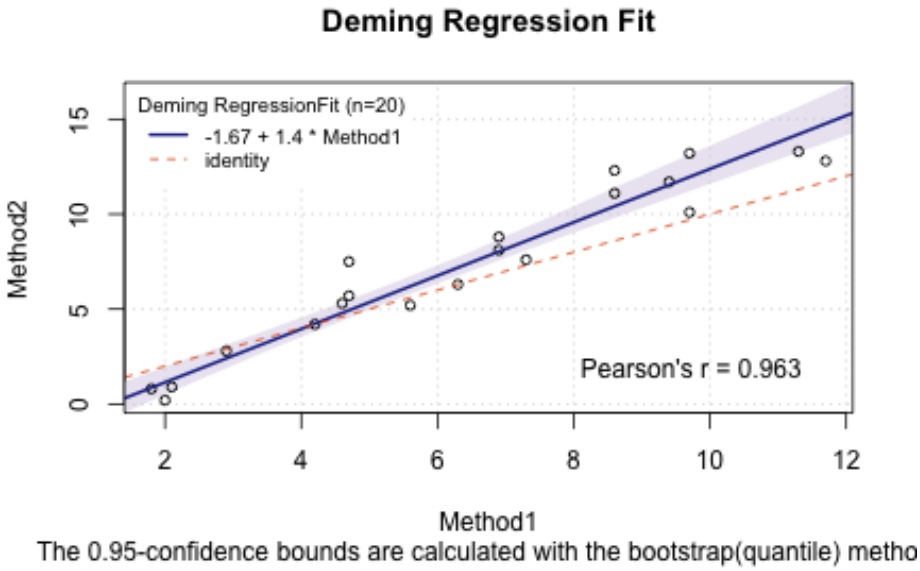
```
PB.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "PaBaLarge")
```

1.7 Deming Regression

OLS regression assumes that there is no error in the x-axis data. This is only true if the x-axis is mass spectrometry and the y-axis is an immunoassay (ba-dum-*ching*). OK–that was facetious. Deming regression also assumes that the ratios of the variances (i.e. CVs) is known for the two methods. This can only be meaningfully known if both x and y results are run in duplicate. Generally we don't do this because of the expense. For the most part, if the precision

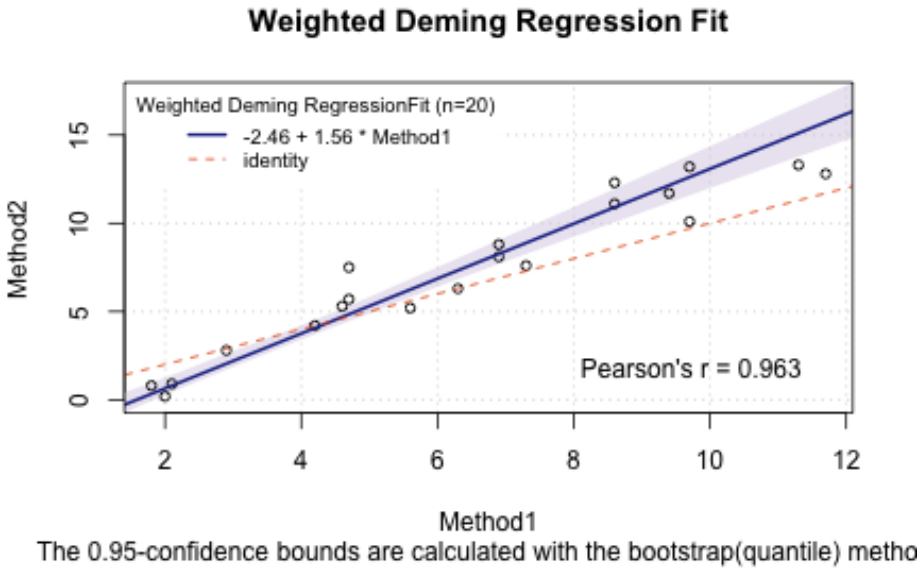
behavior of the two methods is approximately the same, then this value, called δ, is taken to be its default value of 1. The "mcr" package has both a Deming and a weighted Deming regression.

```
Deming.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "Deming")
plot(Deming.reg)
```



The 0.95-confidence bounds are calculated with the bootstrap(quantile) metho

```
WDeming.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "WDeming")
plot(WDeming.reg)
```



BONUS: If you do not like the syntax of the "mcr" package approach (because it uses a weird class for its results), you can also use the "MethComp" package (http://cran.r-project.org/web/packages/MethComp/MethComp.pdf) or "Deming" (http://cran.r-project.org/web/packages/deming/deming.pdf) package. Both have Deming and PB regression with output more similar to what you have seen before.

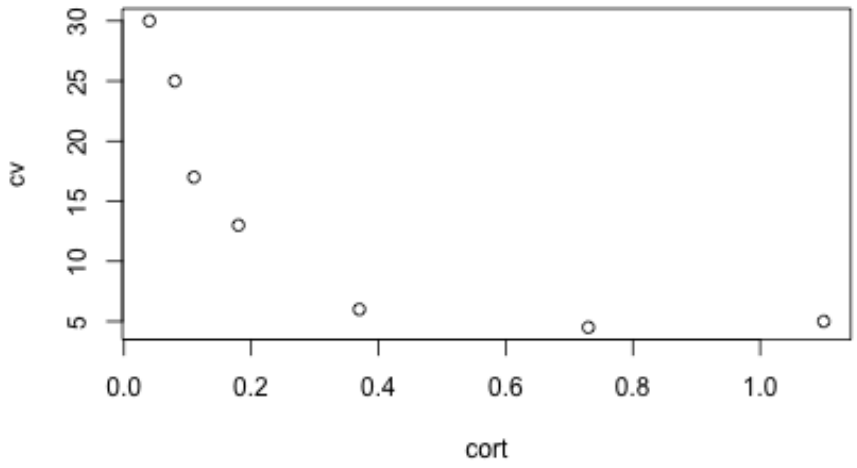
1.8 Non-Linear Regression

OK, so much for linear regression. Now, fitting lines to data often works fairly well. When it does, linear regression is great. Unfortunately, though, the world isn't always linear. It is more than occasionally necessary to fit a series of points that lie on a curve. For this next technique, we have to have some idea what shape of function we want to fit to our data... but if we do, based on some reasoned physical principle, we can determine the "best fit" using any function you want: polynomial, exponential, sinusoidal, etc.

Just to illustrate the principle, let's take a common task as an example. When we estimate the LOQ (limit of quantification) for an assay, we typically have to determine the level at which we hit some critical %CV (say, 20%CV). Suppose that you have the following data for the precision of a cortisol assay at different concentrations (ug/dL). Enter the following initial data into R.

```
cort <- c(0.04,0.08,0.11,0.18,0.37,0.73,1.1)
cv <- c(30, 25, 17,13,6,4.5,5)
```

```
cv.data <- data.frame(cort,cv)
plot(cort,cv)
```



. . . giving this graph.

Now we can invoke the `nls()` function, which performs non-linear least squares. The syntax is `nls(formula, data, start)`. The `formula` is the functional form you want to fit to with unknowns included as variables, the `data` is the data you are fitting (in this case `cort` and `cv`), and `start` is a list of your best initial guesses for your unknowns.

In our case, after looking at the graph, we decide that we are going to fit these data with a hyperbolic function. We are going to make no assumptions about the parameters of our function. We will say that $cv = A/(cort - B) + C$, where A, B, and C are unknowns. We will start with guess that A is 1, and we'll let B and C start at 0 (this might seem like magic, but we'll explain in a minute where the guess for A comes from). We could do better, but it illustrates the process.

```
cort.reg<-nls(cv~A/(cort-B)+C, data=cv.data,start=list(A=1,B=0,C=0),trace=TRUE)
```

```
## 336.6289 :  1 0 0
## 36.06774 :   2.31633959 -0.04666037  1.48408860
## 16.5338 :   2.92634608 -0.05934555  0.92116764
## 16.21782 :   2.87710038 -0.05631155  1.02340803
## 16.21589 :   2.89710314 -0.05699503  0.99149744
## 16.21582 :   2.89304599 -0.05686381  0.99829462
```

```
## 16.21582 :   2.89384518 -0.05688962  0.99697183
## 16.21582 :   2.89368861 -0.05688456  0.99723164
## 16.21582 :   2.89371946 -0.05688556  0.99718040
```

```
summary(cort.reg)
```

```
##
## Formula: cv ~ A/(cort - B) + C
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## A  2.89372    0.93632   3.091   0.0366 *
## B -0.05689    0.02927  -1.943   0.1239
## C  0.99718    2.00787   0.497   0.6455
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.013 on 4 degrees of freedom
##
## Number of iterations to convergence: 8
## Achieved convergence tolerance: 3.366e-06
```

The parameter trace=TRUE shows the work that `nls()` is doing as it tries to find a solution from the start values.

SIDE NOTE:

If the start values really stink, `nls()` will choke and tell you so. You will need to find better guesses for the start values. This is a broad topic that is beyond the scope of our discussion, but we promised that we would talk about how we guessed at A=1. We could start by asking what a simpler function (say, $cv = A/cort$) would look like. If we fit that simpler function, we can get the following:

```
c.reg <- nls(cv~A/cort, data=cv.data, start=list(A=0), trace=TRUE)
```

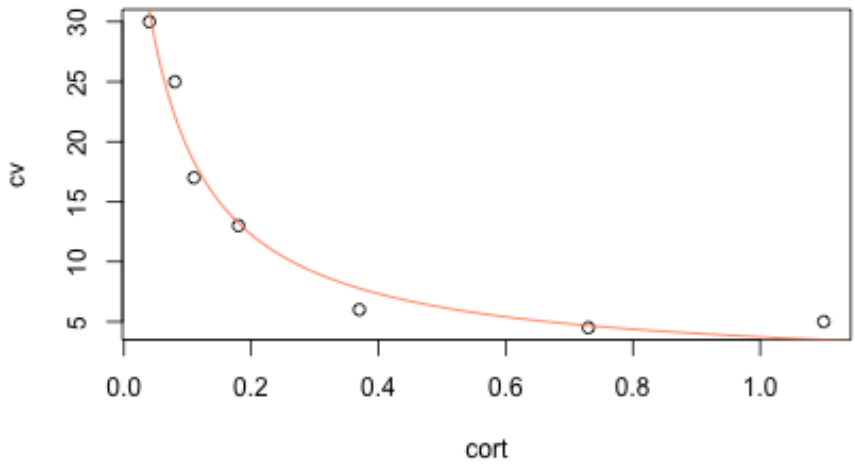
```
## 2064.25 :  0
## 149.5393 :  1.454733
```

Since A ends up pretty close to 1, we can plug that into our original formula with $B = 0$ and $C = 0$. If we had wanted to get even closer, we could have plugged in 1.45. Sometimes you can ignore all this and just plug in 1 for everything and have it work... but not always. When you start getting weird errors, come back and reread this section.

OK, we were successful

So now I can show my fit:

```
cort.fit <- seq(from=0, to=1.2, by=0.005) # make a bunch of tightly spaced points
cv.fit <- predict(cort.reg , list(cort=cort.fit)) # let our regression predict
plot(cort,cv)
lines(cort.fit,cv.fit,col="coral") # ...and plot it
```



Now it's time to estimate our LOQ. Let's look at our fitted line for a point near 20%:

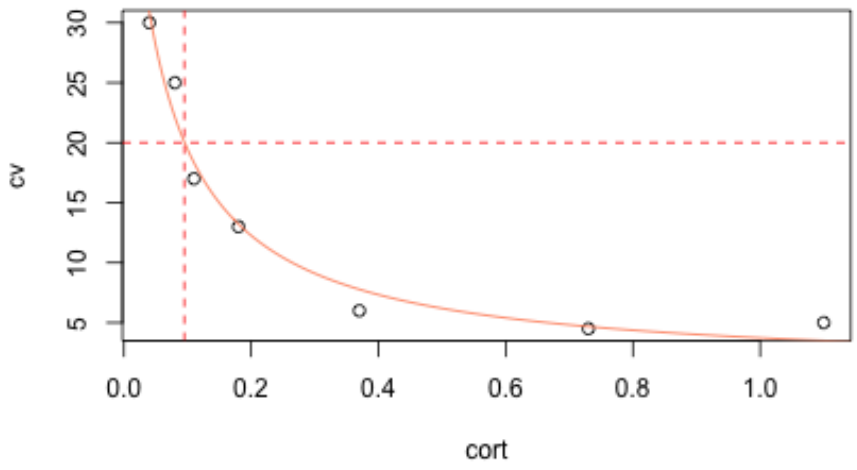
```
which((abs(cv.fit-20))<0.1) #tells us if there are any fitted values of the cv that are wit
```

```
## [1] 20
```

```
cort.fit[20] #result of the which function ? gives 0.095 back
```

```
## [1] 0.095
```

```
plot(cort,cv)
lines(cort.fit,cv.fit,col="coral")
abline (h = 20, col = "red",lty = 2)
abline(v = 0.095, col = "red", lty = 2) #to confirm
```



And there you have it–a nonlinear curve fit to estimate the measured level of an analyte that we're comfortable reporting numerically!

Note that we didn't have to use a hyperbolic fit with *nlm*(); we could have used an exponential function, a *sin*() function... pretty much anything we thought looked like it could provide a reasonable fit.

1.9 Putting it all together: Linearity Testing

Let's try one more application that will let us use multiple types of regression. The CLSI guidelines suggest evaluation of linearity using the method of Emancipator and Kroll. In this approach, the data is fit with a linear regression (OLS, not Deming or PB; why?), then is it fit with quadratic regression, and then cubic regression. The quadratic and cubic fits are compared to each other based on which one has the lowest summed squared residuals, and the winner is then compared to the linear fit using a difference plot.

We have provided calibration curve data in the file "Cal_Curve_Data.csv". After importing the data, we will first determine the linear regression fit with $1/x^2$ weighting. This is accomplished by including the option `weights = 1/conc^2`.

```
###Solution###
cal.data <- read.csv(file = "Data_Files/cal_curve_data.csv",  header = TRUE, sep = ",")
#linear
lin.mod <- lm(signal~conc, data = cal.data, weights = 1/conc^2)
```

OK, now we want to generate quadratic and cubic regressions for comparison.

We can use the *nlm*() function that we learned about in the last section. We will perform quadratic regression by fitting the curve to a function of the form: `signal ~ A + B*conc + C*conc^2`. It likely does not matter if we weight the regression, but for consistency we will weight with $1/x^2$. Similarly, we'll perform cubic regression by fitting the curve to a function of the form: `signal ~ A + B*conc + C*conc^2 + D*conc^3`.

```
#quadratic
quad.mod <- nls(signal~A+B*conc+C*conc^2,data = cal.data, start = list(A = 0,B = 1,C = 0), w
#cubic
cube.mod <- nls(signal~A+B*conc+C*conc^2+D*conc^3,data = cal.data, start = list(A = 0,B = 1,
```

OK, so which of the two polynomial fits is best?

```
summary(quad.mod)
```

```
##
## Formula: signal ~ A + B * conc + C * conc^2
##
## Parameters:
##     Estimate Std. Error t value Pr(>|t|)
## A -2.558e-01  2.041e-01  -1.253  0.27836
## B  1.342e+00  1.124e-02 119.411 2.95e-08 ***
## C -1.297e-04  2.041e-05  -6.355  0.00314 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.01529 on 4 degrees of freedom
##
## Number of iterations to convergence: 1
## Achieved convergence tolerance: 2.491e-07
```

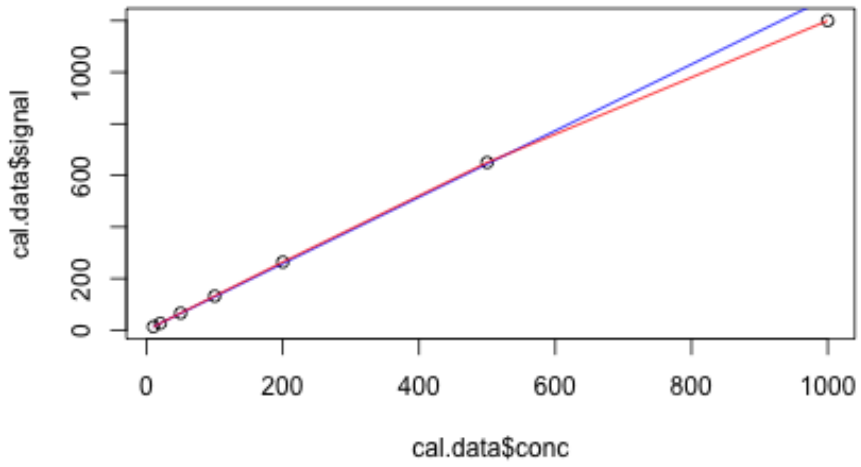
```
summary(cube.mod)
```

```
##
## Formula: signal ~ A + B * conc + C * conc^2 + D * conc^3
##
## Parameters:
##     Estimate Std. Error  t value Pr(>|t|)
## A  2.569e-02  9.087e-03    2.828 0.066319 .
## B  1.317e+00  6.086e-04 2164.512 2.17e-10 ***
## C  4.697e-05  3.324e-06   14.129 0.000768 ***
## D -1.642e-07  3.010e-09  -54.552 1.36e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0005603 on 3 degrees of freedom
##
```

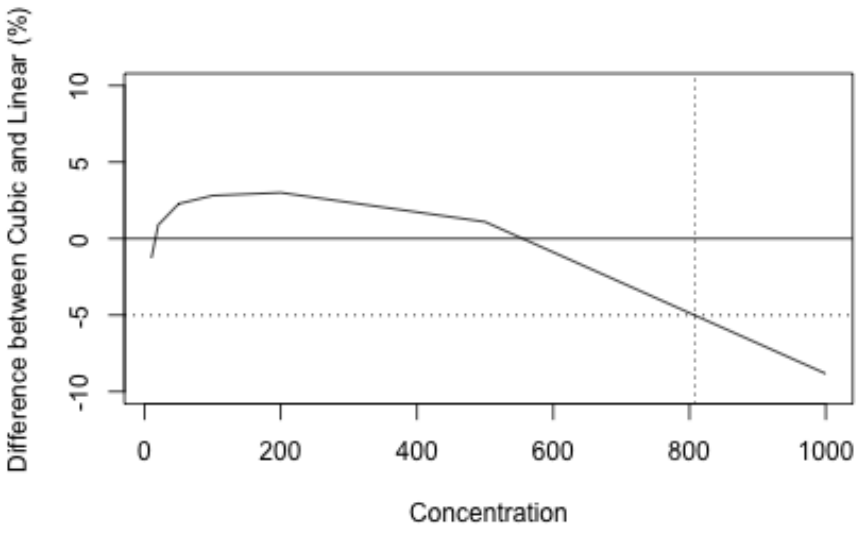
```
## Number of iterations to convergence: 1
## Achieved convergence tolerance: 4.03e-06
```

For the sake of argument, let's suppose we can tolerate up to 5% non-linearity. We can generate a difference plot of the fitted values of the best polynomial fit against the linear fit. This will then allow us to estimate how far our linear range extends...

```
# first do a traditional plot of the fits...
plot(cal.data$conc,cal.data$signal)
lines(cal.data$conc,predict(lin.mod), col = "blue")
lines(cal.data$conc,predict(cube.mod), col = "red")
```




```
# ...now prepare a difference plot
y <- (predict(cube.mod)-predict(lin.mod))/cal.data$conc*100
x <- cal.data$conc
plot(x,y,type = "l", ylim = c(-10,10), ylab = "Difference between Cubic and Linear (%)",xlab
abline(h = 0)
abline(h = -5, lty = 3)
abline(v = 808,lty = 3)
```




```
#Linear to about 800
```

Take a deep breath–this section was a challenge (pretty useful, though!). Don't be discouraged if it seemed to fly by the first time; you have the notes and all the code, and if necessary you can give it another try when you're fresh and rested.

. . . and congratulations! You've taken your first steps toward impressing your colleagues as an R stats genius!

Dealing with my Vizness

Getting Started with R for Laboratory Medicine
Sunday Aug 04, 2019
AACC 2019, Anaheim CA

Shannon Haymond, PhD (s-haymond@northwestern.edu)
Northwestern University Feinberg School of Medicine

7/23/2019

Contents

1 Lesson 6: Data Visualization with `ggplot2` 1
1.1 Useful resources for `ggplot2` . . . 2
1.2 What is ggplot? . . . 2
1.3 Plotting with Continuous Variables . . . 3
1.4 Plotting with Discrete Variables . . . 11
1.5 Layer it on!! . . . 16
1.6 Creating and combining multiple small plots . . . 21
1.7 Saving plots . . . 22
1.8 Summary . . . 23
1.9 Acknowledgements . . . 23

2 Lesson 7: Reports and Reproducible Workflows using `R` 23
2.1 `R Markdown` and `R Notebooks` . . . 23

1 Lesson 6: Data Visualization with `ggplot2`

So far, we've seen how to do some 'quick and dirty' plots with plotting functions like `hist()` and `plot()` which are built into base R. There is another graphics package for R called `lattice`. The tidyverse has its own paradigm for creating graphics called `ggplot`. The advantage to using `ggplot` over base R functions is that the gpplot paradigm comes with many built in defaults to make your plots look nice without having to code too much customization. As we go through some examples, note how ggplot has automatically chosen colour schemes, scales, and axis labels for us, without us specifying any of this. Of course, we can override these, but having some usable defaults built in makes it very fast to produce nice plots.

1.1 Useful resources for `ggplot2`

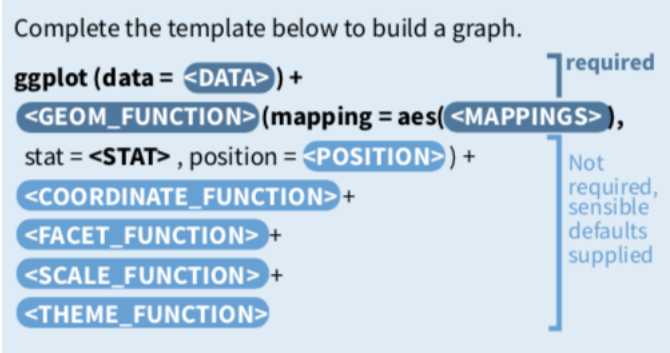
For inspiration and for help creating graphics with `ggplot2`, bookmark these pages:

- THE R GRAPH GALLERY http://www.r-graph-gallery.com/portfolio/ggplot2-package/
- COOKBOOK for R >> Graphs http://www.cookbook-r.com/Graphs/
- CHEAT SHEET for `ggplot2` https://www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf

1.2 What is ggplot?

The "gg" in "ggplot" stands for "grammar of graphics", and the basic idea is this: when you plot data, you are creating a visual representation of numeric or categorical information within a coordinate system. The most basic example is a scatterplot; the position of a point on the x axis reflects one variable, and the position on the y axis reflects another variable. That works well for simple examples, but often we have a large number of parameters that we'd like to display. Ideally, we want a clear, flexible framework that maps arbitrary variables to arbitrary visual elements or aesthetics, such as x position, y position, size, color, shape, transparency, etc. This would let us rapidly explore different ways of looking at our data to see what is the most helpful. `ggplot` provides this sort of framework, with a clean mapping of variables to output.

In other words, `ggplot2` maps data to aesthetics and it does so in layers. There are several types of layers we'll learn about, including geometric objects, statistical transformations, and position adjustments. We can see how this works by examining the syntax. We initialize a plot with `ggplot()` and then add the layers with instructions for mapping. We can also add other functions to further customize our graphic.



Let's illustrate this using data from the 2003-2004 NHANES Survey that measured iron status markers in children aged 3-5 years old.

Read in the data file, NHANES_FeMarkers_3to5y.csv, clean it up, and take a look at the data.

```
nhanes_fe <- read_csv(file = "Data_Files/NHANES_FeMarkers_3to5y.csv",
                      col_types = cols(Subject = col_factor(),
                                       Gender = col_factor(),
                                       Age_months = col_integer(),
                                       Race_ethn = col_factor())) %>%
             mutate(Gender = recode(Gender,
                                    `1` = "Male", `2` = "Female"),
                    Race_ethn = recode(Race_ethn,
                                       `1` = "Mexican American",
                                       `2` = "Other Hispanic",
                                       `3` = "Non-Hispanic White",
                                       `4` = "Non-Hispanic Black",
                                       `5` = "Other Race - Including Multi-Racial"))

glimpse(nhanes_fe) #demographics and lab values
```

```
## Observations: 295
## Variables: 10
## $ Subject    <fct> 21046, 21121, 21131, 21208, 21222, 21235, 21251, 21...
## $ Gender     <fct> Male, Female, Female, Male, Male, Male, Male, Femal...
## $ Age_months <int> 52, 56, 49, 53, 46, 52, 52, 47, 37, 49, 50, 59, 37,...
## $ Race_ethn  <fct> Mexican American, Mexican American, Other Hispanic,...
## $ Ft_ngdL    <dbl> 59, 26, 15, 53, 48, 27, 22, 24, 14, 24, 41, 33, 7, ...
## $ Hgb_gdL    <dbl> 13.0, 13.3, 12.3, 14.5, 11.4, 12.5, 13.1, 12.7, 13....
## $ MCV_fL     <dbl> 78.6, 84.9, 85.3, 84.7, 75.0, 81.1, 86.6, 86.6, 84....
## $ Fe_ugdL    <dbl> 17, 81, 55, 92, 57, 48, 103, 59, 36, 15, 54, 44, 76...
## $ TIBC_ugdL  <dbl> 346, 310, 279, 331, 278, 401, 420, 353, 356, 328, 3...
## $ TfSat_pct  <dbl> 4.9, 26.1, 19.7, 27.8, 20.5, 12.0, 24.5, 16.7, 10.1...
```

1.3 Plotting with Continuous Variables

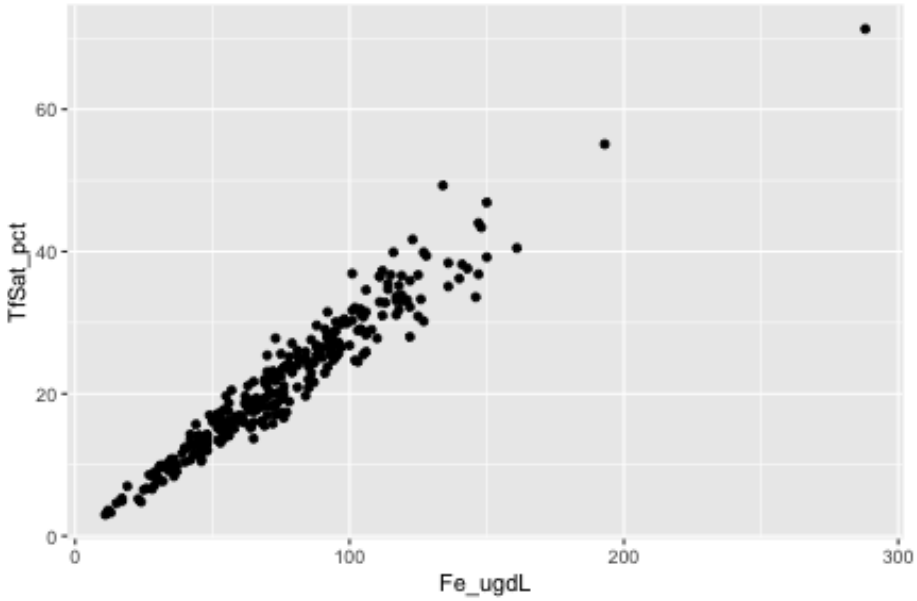
Scatterplots are one of the most commonly used graphics in laboratory medicine. Let's create a scatterplot to look at the relationship between iron and transferrin saturation. `ggplot()` wants us to provide some data, a mapping of the data onto parameters, and a geometry with which to render that data.

Here's that template again.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Let's fill it in and run the code.

```
ggplot(nhanes_fe) +
  geom_point(aes(x = Fe_ugdL, y = TfSat_pct))
```



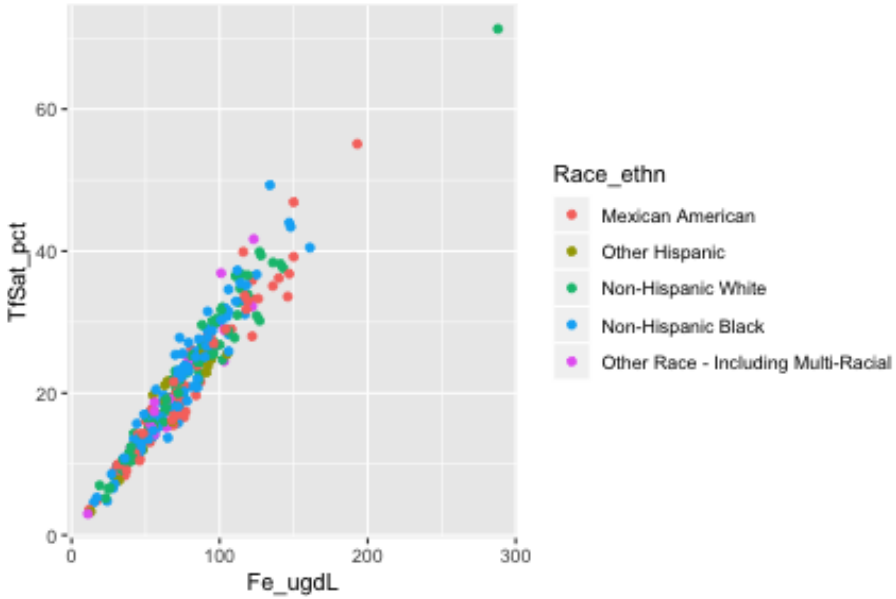
Notice that our `ggplot()` command had three parts: DATA (the nhanes_fe object), a set of aesthetic MAPPINGS (x and y in this case), and a GEOM_FUNCTION (`geom_point()`) for rendering the geometry. This doesn't look like our ordinary function calls, but you can think of the `+` as saying "OK, add this geometry rendering layer to the plot that I just made".

The "aesthetics" that are specified within the `aes()` call are where the real fun starts. The aesthetics for x and y can be specified in a global `aes()` call in the main `ggplot()` call, or, locally, within the function call for each layer. We can further customize for color, size, shape, etc. inside the geometry call.

Notes on global vs local settings (assuming `inherit.aes = TRUE`):
* Mappings and data that appear in `ggplot()` will apply globally to every layer.
* Mappings and data that appear in the layer function calls will add to or override the global mappings for that layer **only**.

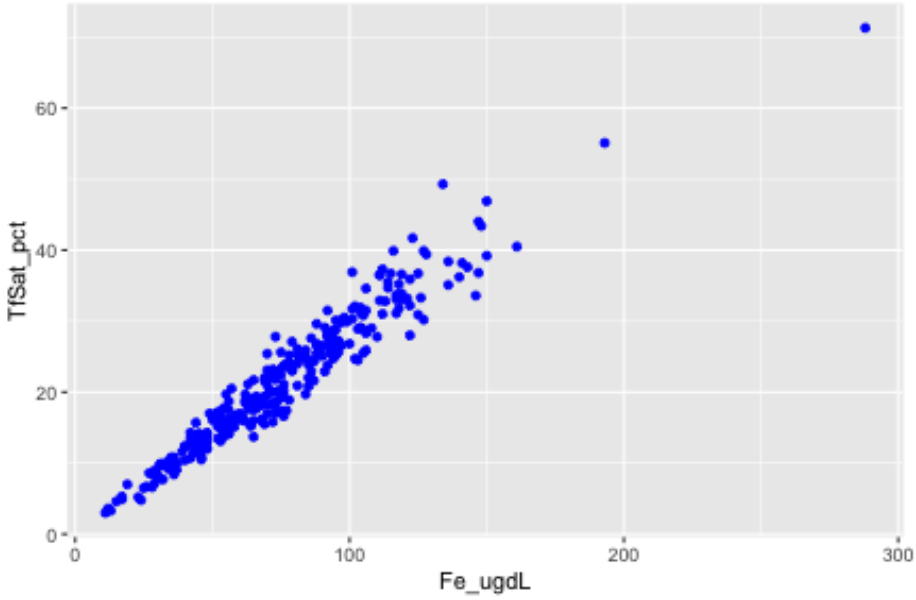
For example, let's look at transferrin saturation vs iron again, and map the race/ethnicity of the subject to color. We'll do this by setting the data mappings globally and then the color aesthetic locally within the `geom_point()` call.

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(aes(color = Race_ethn))
```

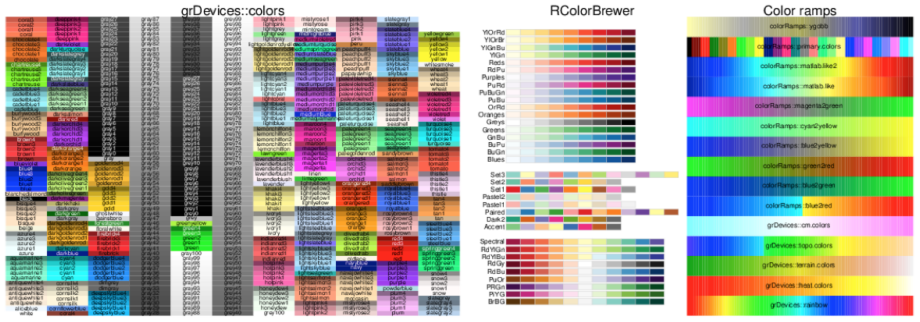


Mapping the color (as above) is different than setting the color (as below). Notice what happens when we specify color *outside* of the `aes()` call:

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(color = "blue")
```

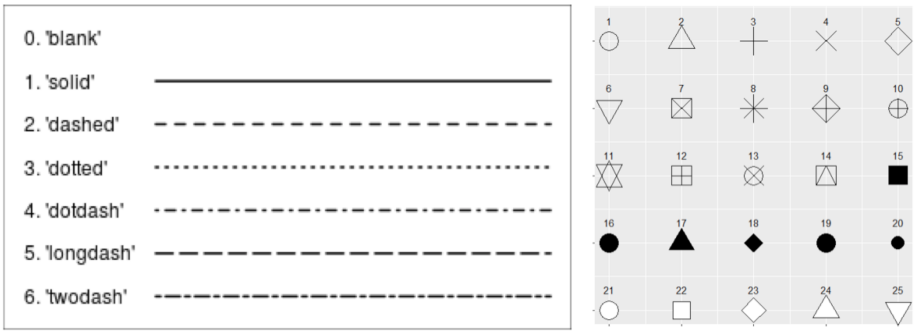


There are more than 600 colors available within R. Colors can be specified by name, RBG, or by hexadecimal code. There are around 50 different color palettes and ramps available, though we will not discuss those in this session. You can also create your own colors and palettes.



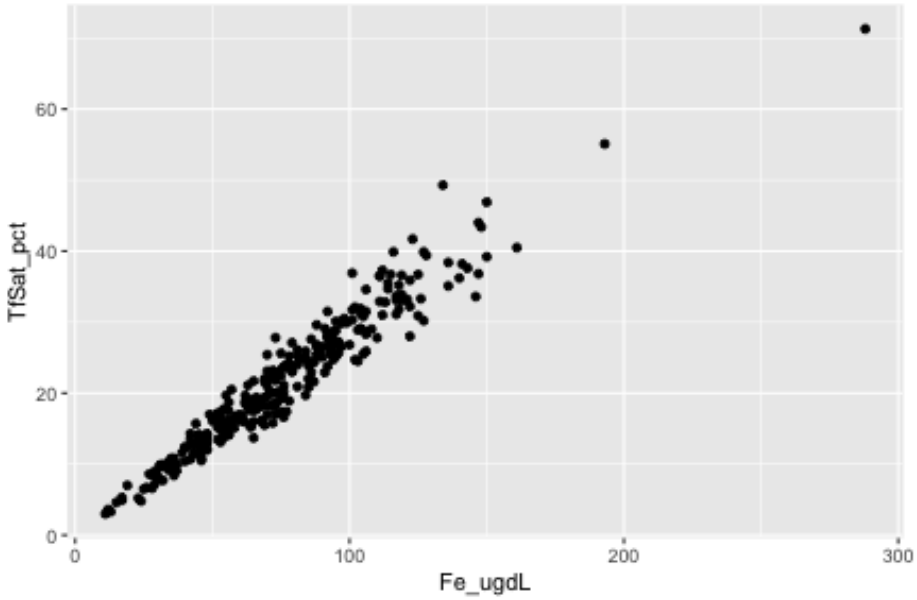
You can access a full sheet of the colors here: http://bc.bojanorama.pl/wp-content/uploads/2013/04/rcolorsheet.pdf

There are 6 types of lines and 25 choices for symbol shape:



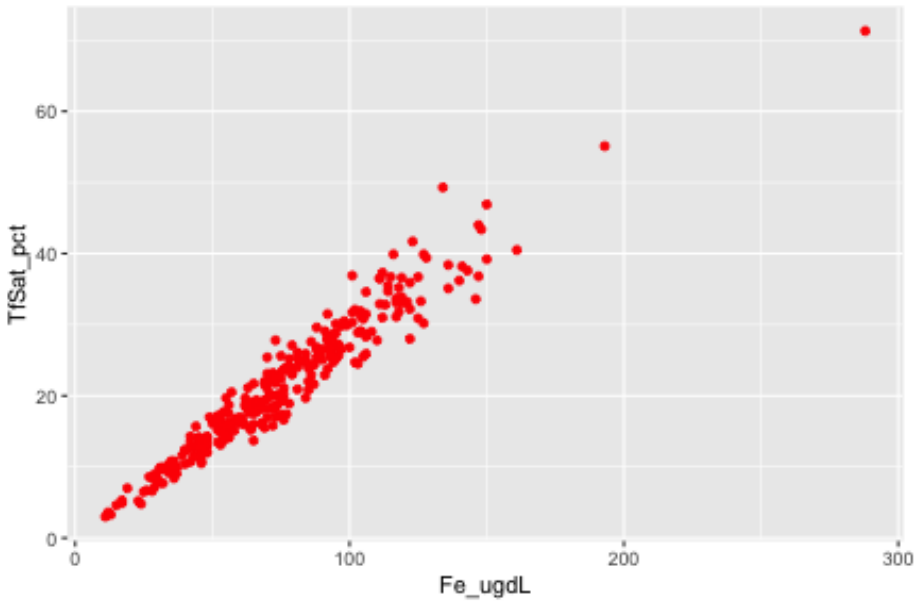
Notice that shapes 1, 16, 19, 20, and 21 are different types of circles. This brings up the difference between the `fill` and `color` arguments. Remember, we used `color` to change the color of the circles above. What happens if we do something similar with `fill`?

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(fill = "blue")
```



That seems unexpected!
Let's figure out how `color` and `fill` work together for different symbols.
Starting with the default shape – for most geoms this is shape 19, the solid circle. Shape 19 only respects color, but whichever color is specified is used for both the fill and the stroke (border).

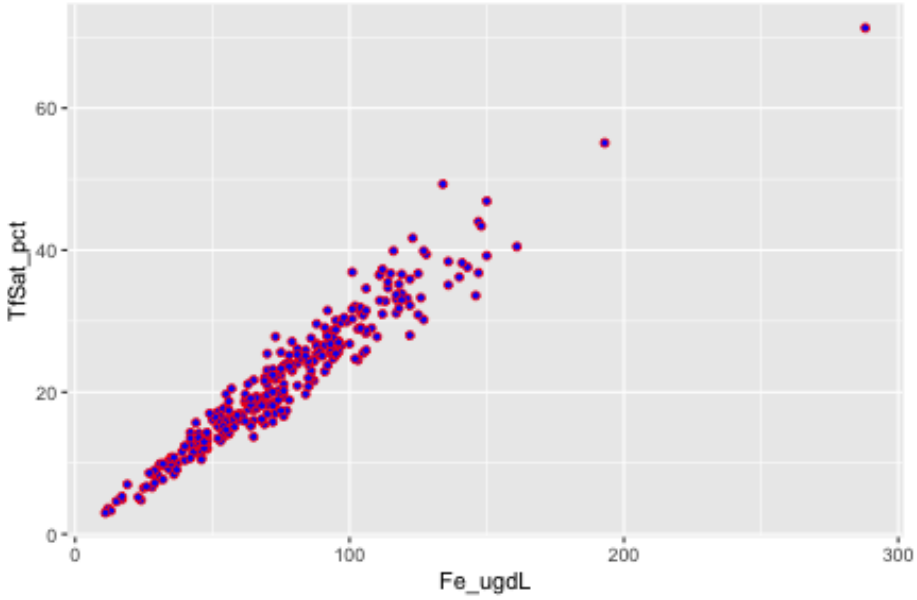
```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(fill = "blue", color = "red")
```



Fill has no effect for this shape. Strangely, shape 16 only has a fill (and no border), but this is controlled by color – fill is ignored.

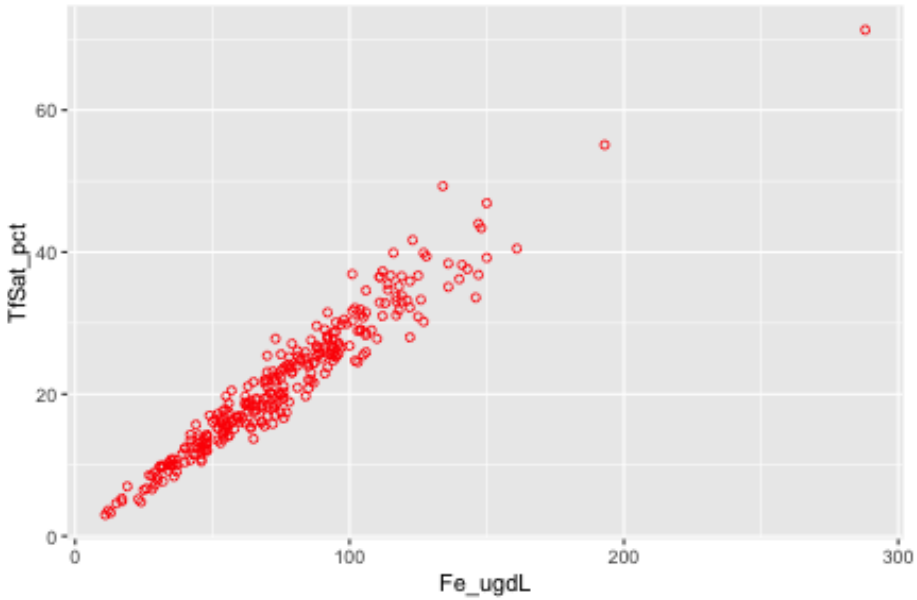
Let's try it with shape 21, the solid circle with a border.

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(shape = 21, fill = "blue", color = "red")
```



And, finally, let's try it for shape 1, the open circle (border only, no fill).

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(shape = 1, fill = "blue", color = "red")
```

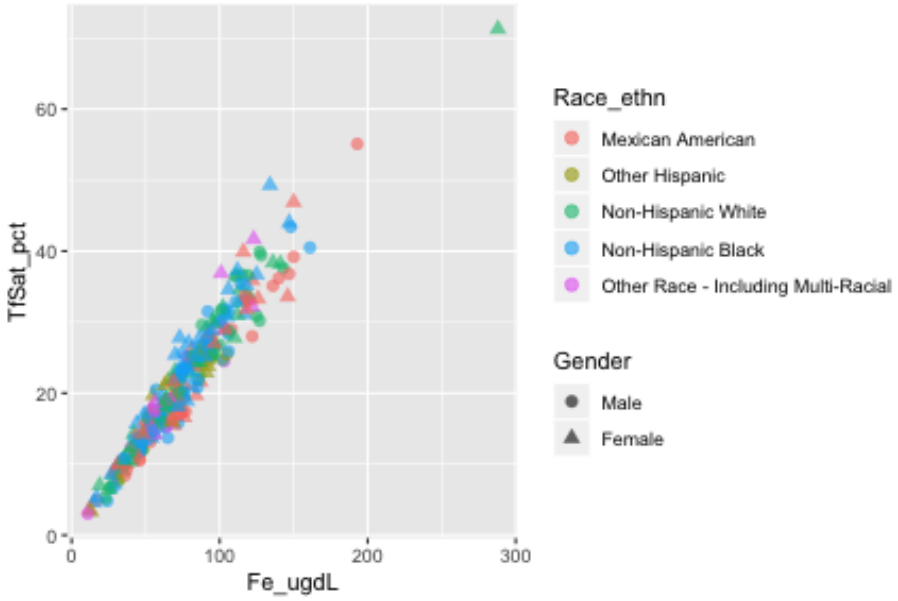


We'll come back to this again in the section on bar plots. This is included so

you may remember there is something about `fill` and `color` when you are troubleshooting unexpected behavior in graphs.

We can also combine aesthetic mappings. Let's map color to race/ethnicity and change the shape based on gender:

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point(aes(color = Race_ethn, shape=Gender), alpha = 0.6, size = 2.5)
```



1.3.1 YOUR TURN EXERCISE

Work with a neighbor.
Using the code examples and information above,

(1) modify the scatterplot by changing the color, size, alpha, and/or shape aesthetics of your graph

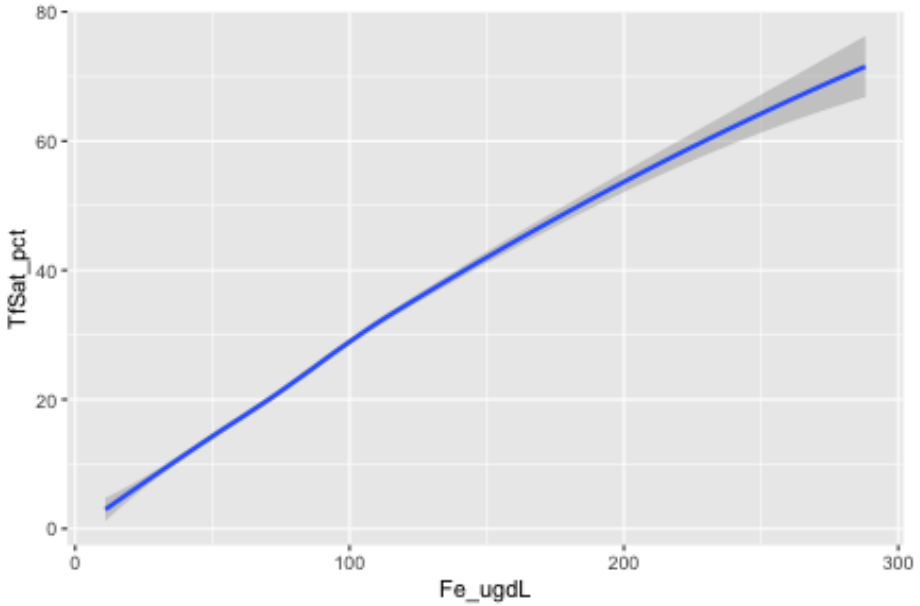
(2) use the `ggplot2` CHEAT SHEET or an internet search to figure out how to add a title to your plot

(3) use the `ggplot2` CHEAT SHEET or an internet search to figure out how to change the theme of your plot (i.e., get rid of the grey grid background)

There are dozens of geometries at your disposal - you can see them on the CHEAT SHEET. Some other useful graphs for continuous variables are `geom_line()` and `geom_smooth()`.

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_smooth()
```

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



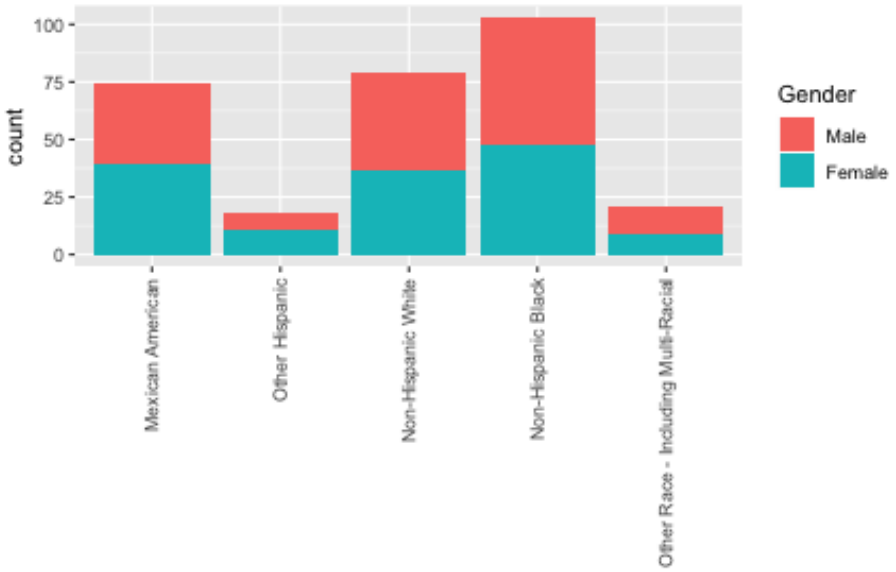
1.4 Plotting with Discrete Variables

If we want to compare counts or continuous values across discrete variables, we need a different set of plots. These also exist in `ggplot2` as specific geoms. There are two types of bar charts: `geom_bar()` and `geom_col()`. `geom_bar()` makes the height of the bar proportional to the number of cases in each group. If you want the heights of the bars to represent values in the summarized data, use `geom_col()` instead. `geom_bar()` uses `stat_count()` by default: it counts the number of cases at each x position. `geom_col()` uses `stat_identity()`: it leaves the data as is. This means we do not need to provide a y variable for `geom_bar()`, but we do for `geom_col()`.

Since our data is not summarized into counts, we'll use `geom_bar()`. If we have the bar fill by a categorical variable, we see a stacked bar plot showing the relative numbers from each group. The categories are plotted based on the order of the levels (here Mexican American = 1).

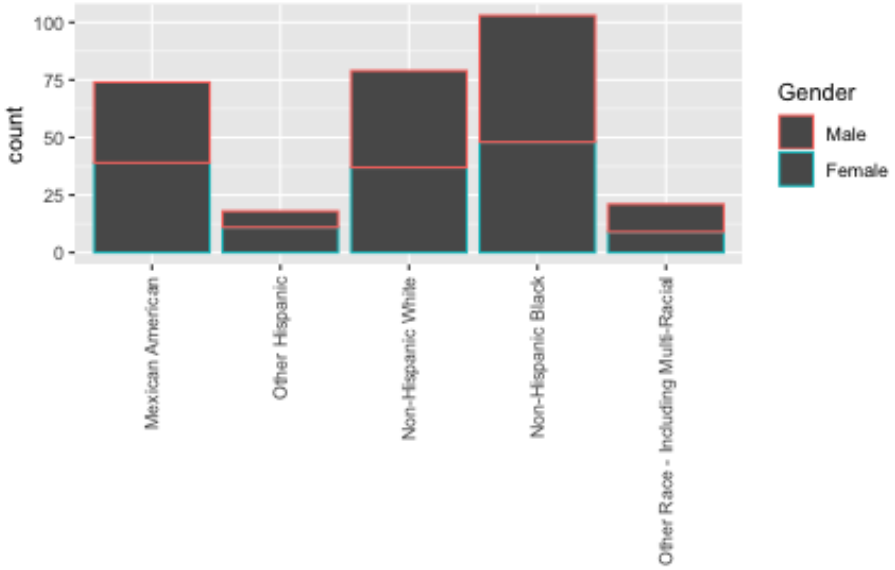
```
ggplot(nhanes_fe, aes(x = Race_ethn)) +
  geom_bar(aes(fill = Gender)) +
```

```
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust = 0.5)) +
  labs(x = "") #hides x axis label
```



Note we used `fill` here to color the bars rather than `color` - what happens if you use `color` instead?

```
ggplot(nhanes_fe, aes(x = Race_ethn)) +
  geom_bar(aes(color = Gender)) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust = 0.5)) +
  labs(x = "")
```

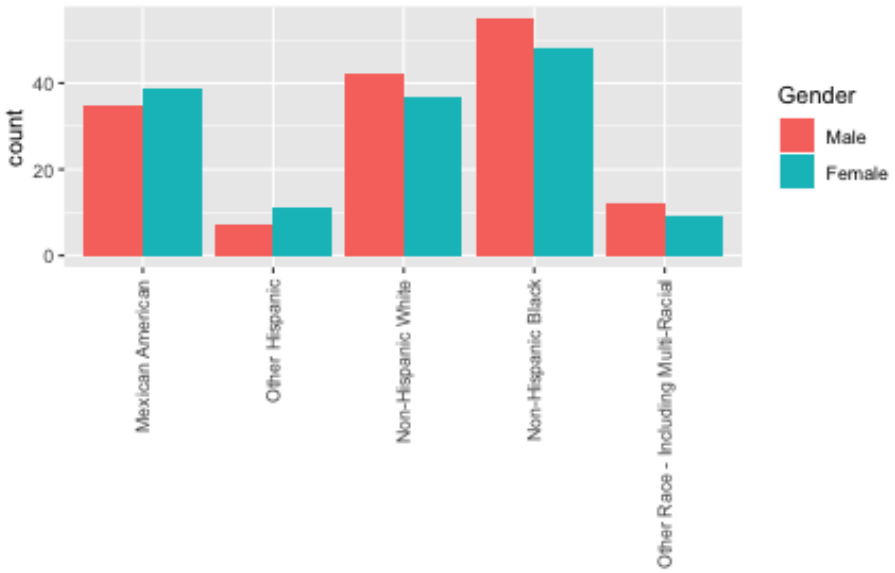


Ah, a similar effect as we saw above for the circle shapes.

If we don't want a stacked bar plot (the default), we can specify the `position` argument to change the arrangement. The CHEAT SHEET shows the effects of the different position adjustments. The dodge and fill are commonly used for bar plots.

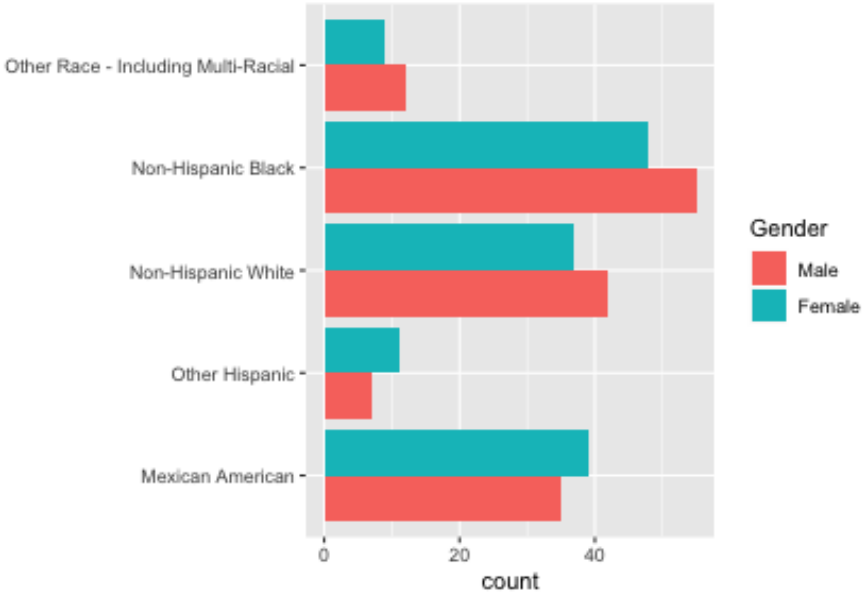
Let's create a bar plot where the bars from each category are placed next to each other.

```
ggplot(nhanes_fe, aes(x = Race_ethn)) +
  geom_bar(aes(fill = Gender), position = "dodge") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust = 0.5)) +
  labs(x = "")
```



Some data visualization experts would suggest that the orientation of this bar plot is not ideal, given the long names of the race/ethnicity labels. In this case, we can do a quick transformation of the axes using `coord_flip()`.

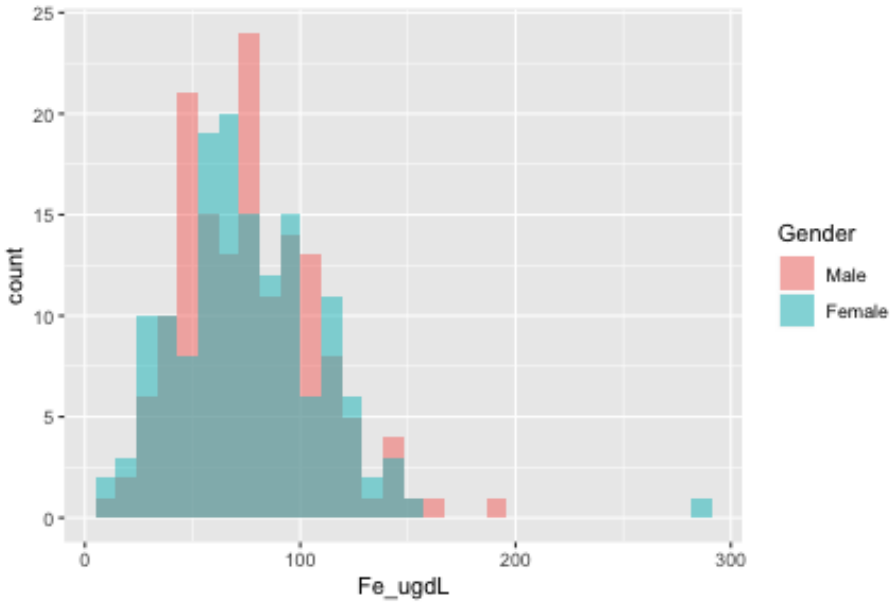
```
ggplot(nhanes_fe, aes(x = Race_ethn)) +
  geom_bar(aes(fill = Gender), position = "dodge") +
  labs(x = "") +
  coord_flip()
```



And of course, good old histograms get their own geom, `geom_histogram()`. We can fill by a categorical variable and use transparency and position to visualize the overlap in the distributions:

```
ggplot(nhanes_fe, aes(x = Fe_ugdL)) +
  geom_histogram(aes(fill = Gender), alpha=0.5, position="identity")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



1.4.1 YOUR TURN EXERCISE

Work with a neighbor.
Using the code examples and information above,

(1) instead of plotting the histograms for iron by gender, create overlapped histograms for each race/ethnicity

(2) use the `ggplot2` CHEAT SHEET or an internet search to figure out how to change the histograms for each group to the density function for each group, colored by group

(3) use the `ggplot2` CHEAT SHEET or an internet search to figure out how to change the position of the legend of your plot (i.e., move it to the top or bottom)

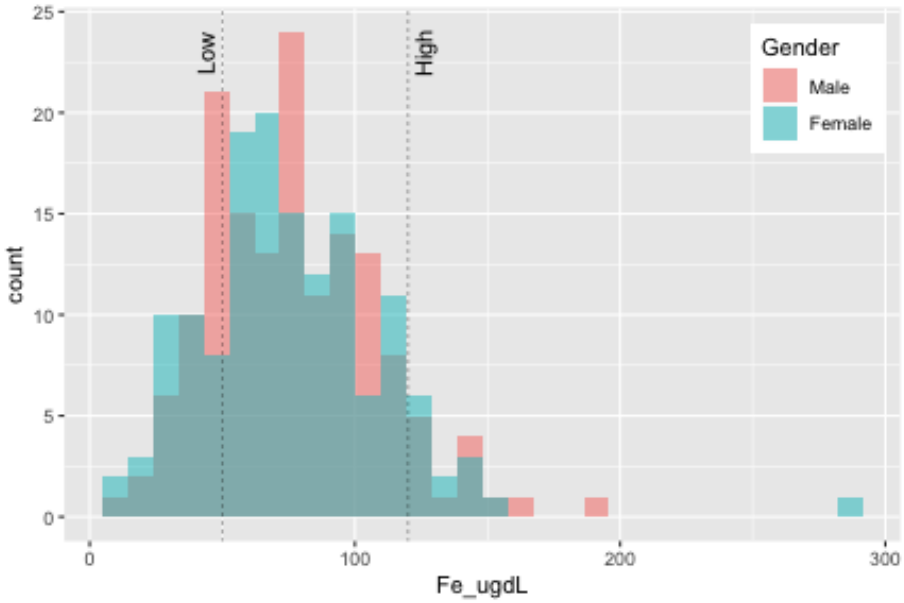
1.5 Layer it on!!

`ggplot()` objects can be layered and layered upon, including multiple geoms, labels, custom scales, statistical results, and more. Here are a couple of examples:

Adding lines and labels
Let's add lines to to our histogram plot from above, showing the reference range limits, and label them Low and High:

```
ggplot(nhanes_fe, aes(x = Fe_ugdL)) +
  geom_histogram(aes(fill = Gender), alpha=0.5, position="identity") +
  geom_vline(aes(xintercept = 50), linetype = 3, size = 0.3) +
  annotate("text", x = 44, y = 23, label = "Low", angle = 90) + #notice syntax
  geom_vline(aes(xintercept = 120), linetype = 3, size = 0.3) +
  annotate("text", x = 126, y = 23, label = "High", angle = 90) +
  theme(legend.position = c(0.9, 0.85))
```

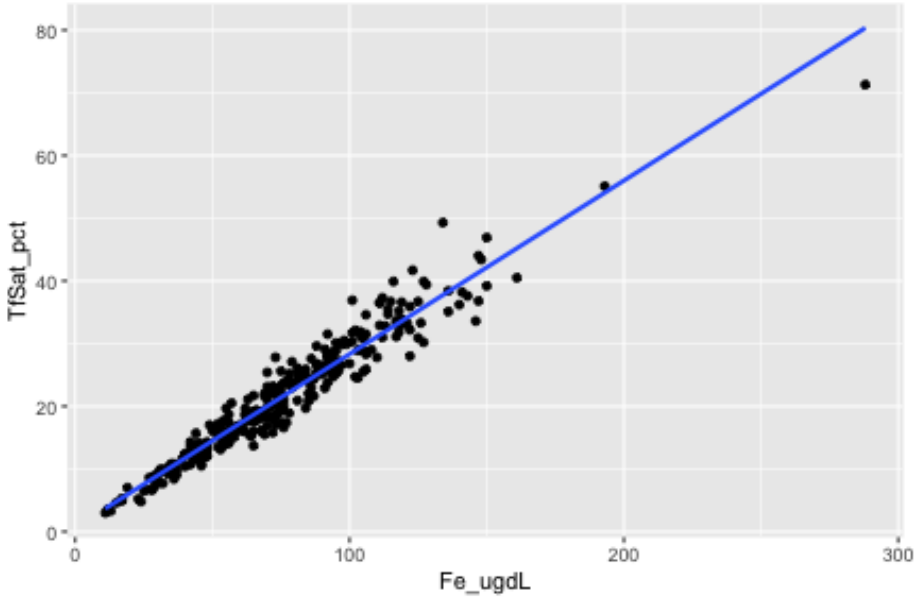
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Other options for adding labels to plots include `geom_label()` and `geom_text()`.

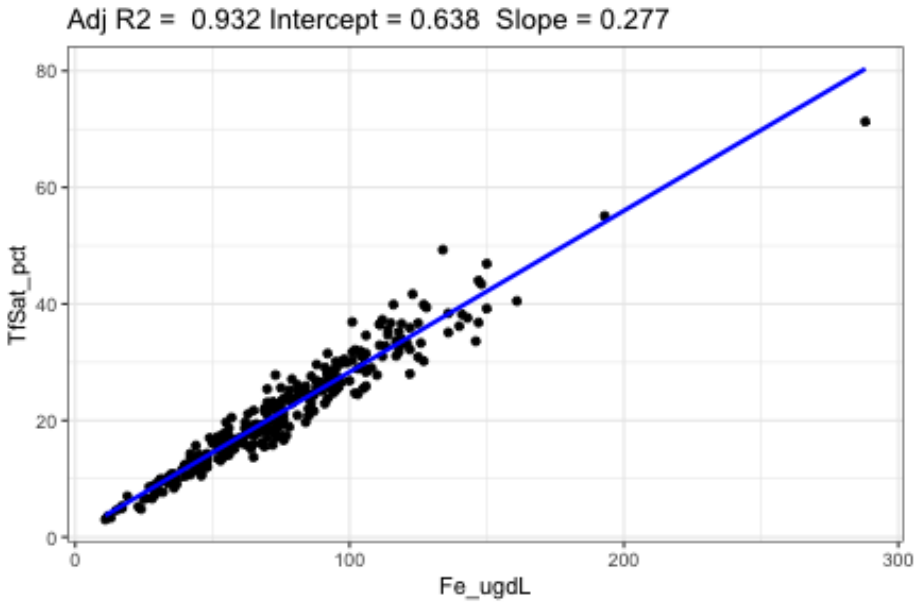
We can add a best fit regression line to a scatter plot:

```
ggplot(nhanes_fe, aes(x = Fe_ugdL, y = TfSat_pct)) +
  geom_point() +
  geom_smooth(method = "lm", se=FALSE)
```



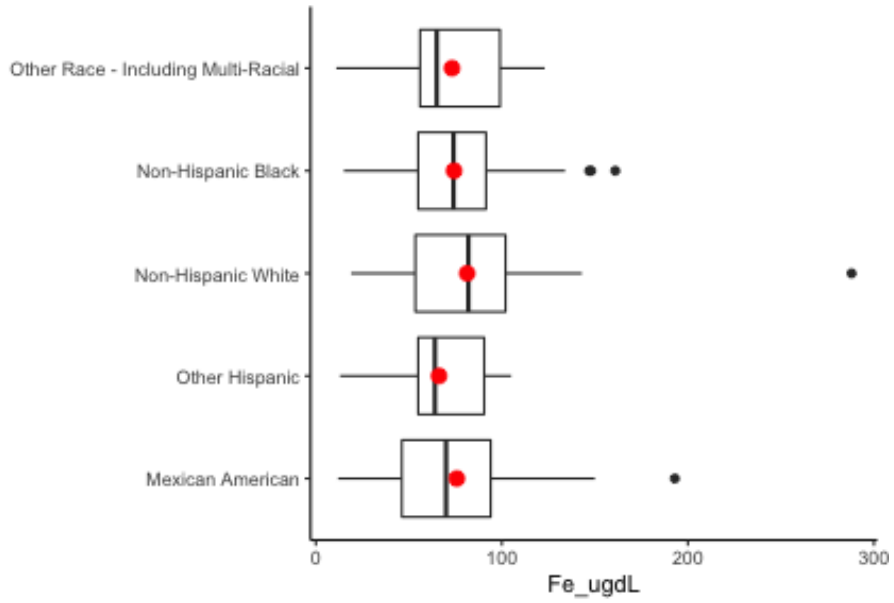
To add the best fit line equation with R2 value is a bit more complicated, but here's one way to do it (modified from:

```
ggplotRegression <- function(fit){

require(ggplot2)

ggplot(fit$model, aes_string(x = names(fit$model)[2], y = names(fit$model)[1])) +
  geom_point() +
  geom_smooth(method = "lm", color = "blue", se = FALSE) +
  labs(title = paste("Adj R2 = ",signif(summary(fit)$adj.r.squared, 3),
                     "Intercept =",signif(fit$coef[[1]],3 ),
                     " Slope =",signif(fit$coef[[2]], 3))) +
  theme_bw()
}

ggplotRegression(lm(TfSat_pct ~ Fe_ugdL, data = nhanes_fe))
```



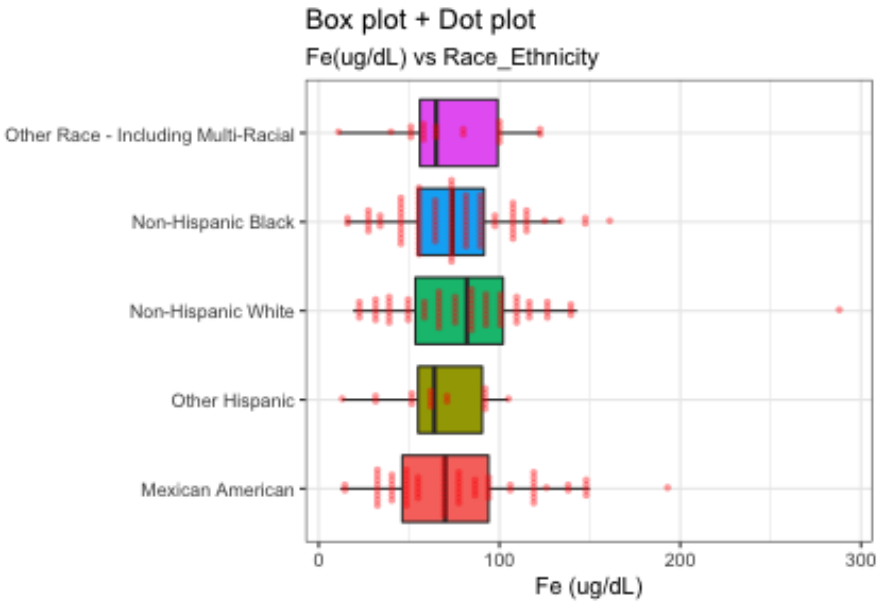
Layering plots and stats output
We can also layer grouped boxplots with plots of statistical summary (e.g., mean) values. This can be done using the `stat` argument within the `geom_point()` call (shown below) or by using a `stat_summary()` (shown in next lesson).

```
ggplot(nhanes_fe, aes(x = Race_ethn, y = Fe_ugdL)) +
  geom_boxplot() +
  geom_point(stat = "summary", fun.y = "mean", color = "red", size = 3) +
  labs(x = "") +
  theme_classic() +
  coord_flip()
```



Layering multiple plots

```
ggplot(nhanes_fe, aes(Race_ethn, Fe_ugdL))+
  geom_boxplot(aes(fill = Race_ethn),
               outlier.shape = NA)  +
  geom_dotplot(binaxis = 'y',
               stackdir = 'center',
               dotsize = 0.5,
               fill = "red",
               col = NA,
               alpha = 0.4,
               binwidth = 8,
               stackratio = 0.7) +
  theme_bw() +
  theme(legend.position = "none") +
  labs(title="Box plot + Dot plot",
       subtitle="Fe(ug/dL) vs Race_Ethnicity",
       x = "",
       y = "Fe (ug/dL)") +
  coord_flip()
```



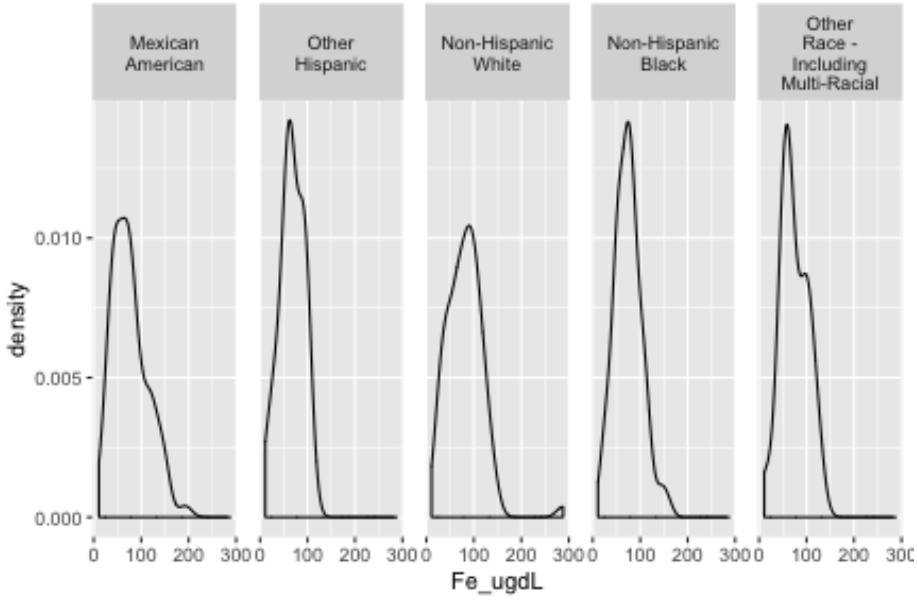
1.6 Creating and combining multiple small plots

`ggplot2` has several faceting functions to divide plots into subplots based on categorical variable values. A different package, `gridExtra`, is needed to arrange multiple independent plots into a single figure.

Instead of overlapping density plots by race/ethniciy, let's create individual plots for each group using faceting. `facet_grid()` arranges the subplots into columns or rows, depending on how the faceting is specified. The data and plot type may dictate which orientation is best.

As columns:

```
ggplot(nhanes_fe, aes(x = Fe_ugdL)) +
  geom_density() +
  facet_grid(~Race_ethn, labeller = label_wrap_gen(10)) + #wraps group labels
  theme(panel.spacing.x = unit(3.6, "mm")) #removes overlap on x axis
```



1.6.1 YOUR TURN EXERCISE

Work with a neighbor.
Using the code examples and information above,

(1) use the `ggplot2` CHEAT SHEET or an internet search to figure out how to change the code to facet the density plots for iron by race/ethnicity into rows.

As rows:

1.7 Saving plots

There are several ways to save plots in R:
(1) Preview or knit a document from Rmd. (2) Export from Plots pane in RStudio. (3) Use `ggsave()` to save the last plot rendered.

```
ggsave("my_plot.png", width = 5, height = 3.5) #you can specify the size
```

A more advanced and comprehensive approach is to include code for knitr options in the r setup chunk to save all figures in a document to a specified folder when the knitted document is created, for example:

knitr::opts_chunk$set(fig.path = “images/”)

The file names default to the name of the code chunk.

1.7.1 YOUR TURN EXERCISE

Work with a neighbor.
Using the code examples and information above,

(1) Determine your working directory.
(2) Create a new folder named 'My_Plots'.

(3) Save your last plot to this new folder and find it there.

1.8 Summary

- The `ggplot2` library is very powerful for creating and customizing high-quality visualizations.

- Graphics are created as layers with mappings of variables to visual elements or aesthetics.

1.9 Acknowledgements

- National Health and Nutrition Examination Survey: Datasets and Codebooks

- Dan Holmes, Stephen Master, Will Slade & Janet Simons's Intro to R Workshop

- Hadley Wickham & Garrett Grolemund's R for Data Science book

- Amelia McNamara's Introduction to R & RStudio, deck 02: Visualization

- Jake Thompson's Tidy Data Science Workshop: Data Visualization

2 Lesson 7: Reports and Reproducible Workflows using `R`

2.1 `R Markdown` and `R Notebooks`

The file type you've been working in during this course is `R Markdown` (extension .Rmd). This type of file is very useful, as you've seen, for writing code mixed with text and for viewing output interactively and independently. `R Markdown` is also great for rendering formatted, report-style documents to PDF, HTML, Word, etc. In fact, the PDF materials from this course were created from an `R`

`Markdown` document. There is another type of file, `R Notebook`, which is very similar. These types of files promote reproducible workflows since the results are together with the data, code, and rationale (if commented or described) used to produce them.

2.1.1 YOUR TURN EXERCISE

(1) Let's open a new `R Markdown` file.

(2) Once we've taken a look, we'll open the file named 'YourTATReport.Rmd', found in the course folder.

- Add your name as the author and execute the code within the file.
- Knit the file to either Word or HTML.

Note: 'YourTATReport.Rmd' is a file that we will use to learn about some of the top features of `R Markdown`. It was designed so you could also use it as a template to create a monthly TAT summary report for your own lab.

Monthly Lab TAT Report

Your Name Here

Report date: July 23 2019

Data

This report describes laboratory turn-around-times (TAT) for tests ordered between November 30 2018 and December 31 2018. TAT values less than 5 minutes or greater than the 99th percentile were excluded from calculations.

Calculations

Turn-around-times in minutes, are calculated as follows:
* TAT = Result time (min) - Order time (min)
* Lab TAT = Result time (min) - Receive time (min)

Summary Tables for each Location by Test (All Priorities)

Table 1: Emergency

Test	Avg TAT	Avg Lab TAT	Max TAT	Max Lab TAT	N
ALT(ALT)	109	41	1448	168	1365
Creatinine(CR)	137	22	1448	131	2011
Hemoglobin(HB)	101	14	1423	161	2361
INR(INR)	73	24	1154	155	1591
Potassium(K)	137	22	1448	151	2043
Troponin T(CTROPT)	89	42	1083	166	1707
TSH(TSH)	161	64	1455	165	465

Table 2: Inpatient

Test	Avg TAT	Avg Lab TAT	Max TAT	Max Lab TAT	N
ALT(ALT)	447	43	1410	167	940
Creatinine(CR)	537	39	1516	170	6428
Hemoglobin(HB)	562	14	1348	169	4490
INR(INR)	381	24	1352	100	1644
Potassium(K)	533	39	1516	170	6673
Troponin T(CTROPT)	281	40	1284	168	713
TSH(TSH)	444	66	1304	167	189

Table 3: Outpatient

Test	Avg TAT	Avg Lab TAT	Max TAT	Max Lab TAT	N
ALT(ALT)	218	68	1566	170	1873

Test	Avg TAT	Avg Lab TAT	Max TAT	Max Lab TAT	N
Creatinine(CR)	227	62	1566	169	3047
Hemoglobin(HB)	221	33	1556	163	2804
INR(INR)	300	40	1564	150	962
Potassium(K)	287	56	1566	169	2315
Troponin T(CTROPT)	617	49	1528	146	311
TSH(TSH)	207	99	1546	170	242

Table 4: Outside

Test	Avg TAT	Avg Lab TAT	Max TAT	Max Lab TAT	N
ALT(ALT)	73	74	159	159	138
Creatinine(CR)	70	70	158	158	160
Hemoglobin(HB)	51	52	158	159	157
INR(INR)	57	60	139	148	19
Potassium(K)	65	65	153	155	55
Troponin T(CTROPT)	53	54	53	54	1
TSH(TSH)	100	99	150	141	10

Plots of Lab TAT for Stats by Test and Location