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

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)
```

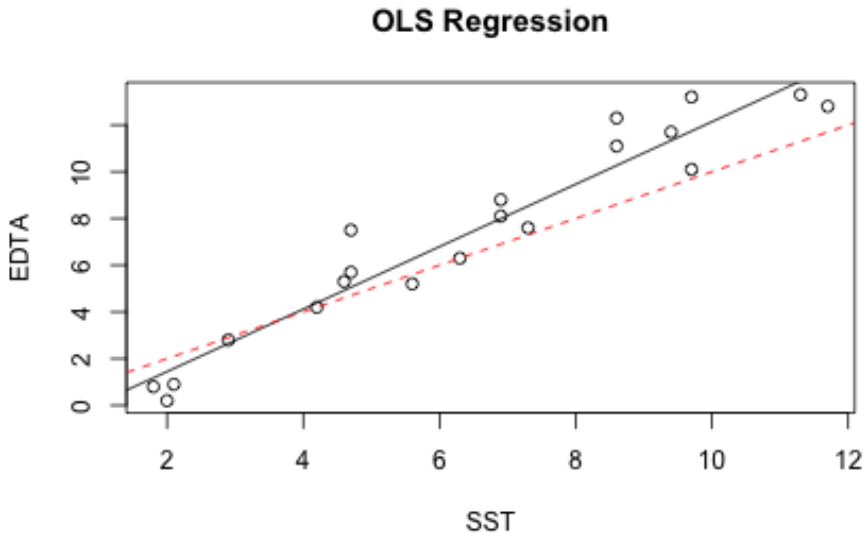
```
## List of 12
##  $ coefficients : Named num [1:2] -1.23 1.34
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "SST"
##  $ residuals    : Named num [1:20] 1.459 0.379 0.803 0.36 0.83 ...
##   ..- attr(*, "names")= chr [1:20] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:20] -33.071 17.919 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

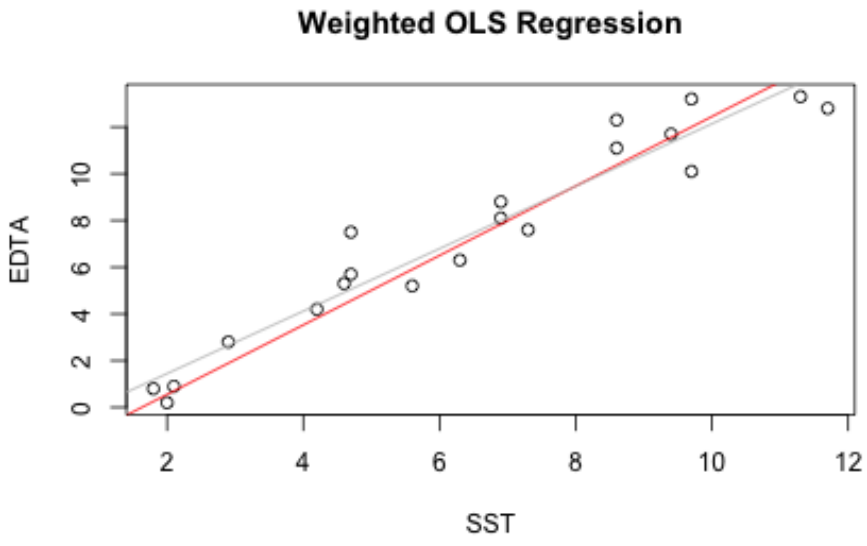
Now, sometimes you want to perform weighted regression. This is what we do on the mass spectrometer when we want to improve the recoveries of the low-level calibrators and the expense of the high level calibrators. In other words, when accuracy at the low end is clinically required, we weight the regression. 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")
```

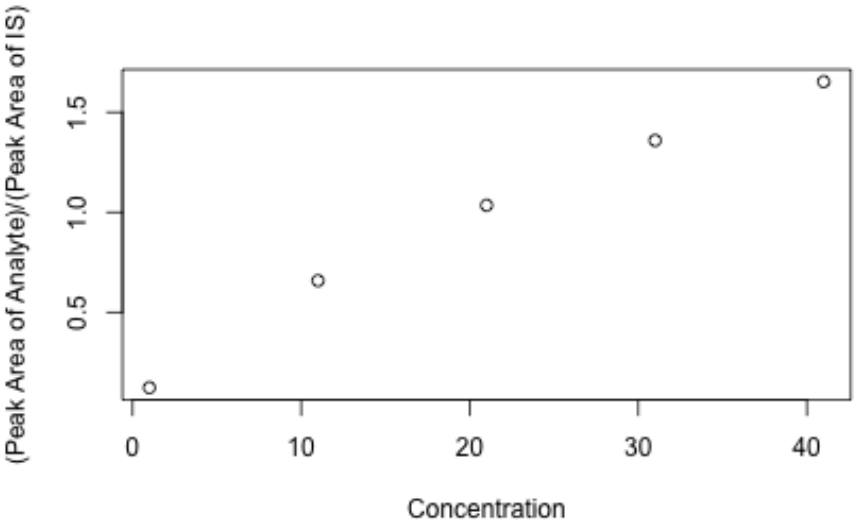


```
#What do you notice?
```

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?

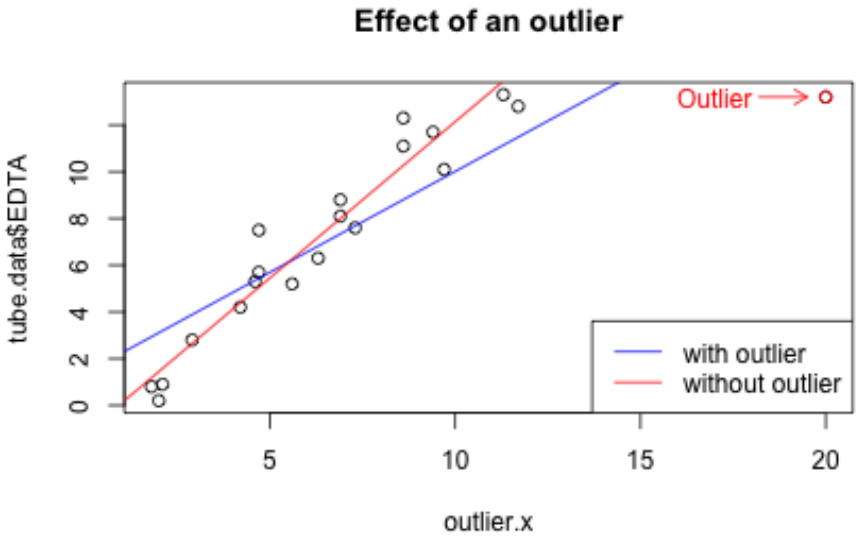
1.5 Outlier Effects in OLS

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
```

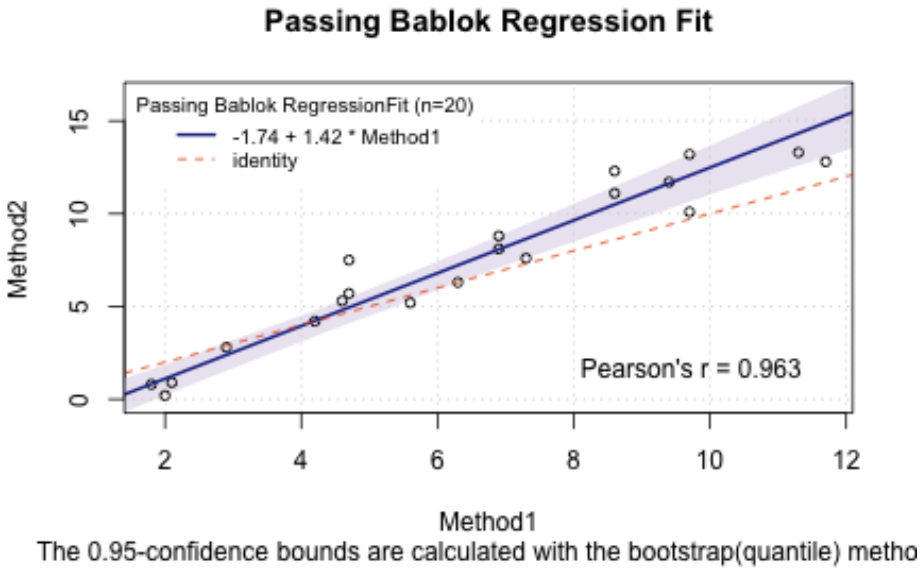
```r
plot(outlier.x, tube.data$EDTA, main = "Effect of an outlier")
abline(lm(tube.data$EDTA ~ outlier.x), col = "blue") #regression with outlier
abline(OLS.reg, col = "red") #regression without outlier
legend("bottomright",c("with outlier","without outlier"), lty = c(1,1), col = c("blue","red"
points(20, 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:

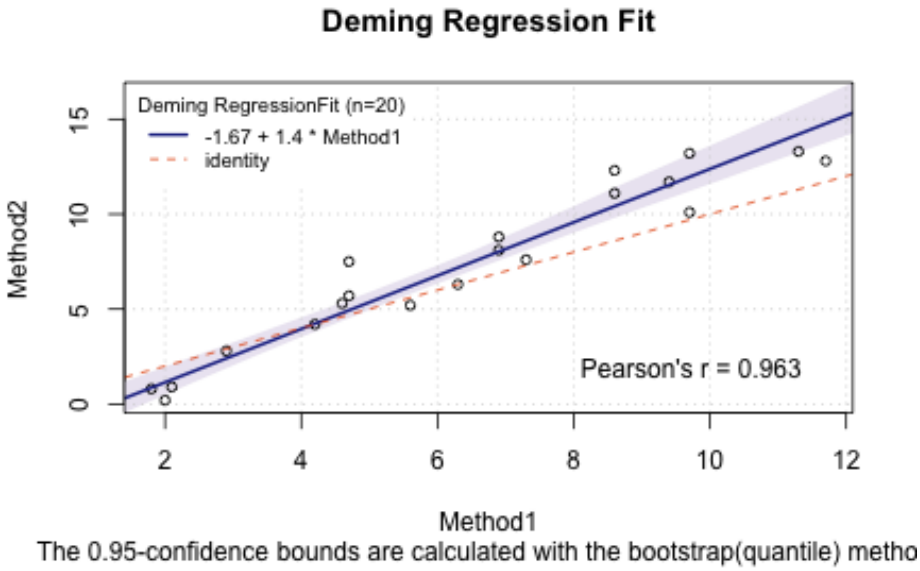
```
PB.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "PaBaLarge")
```

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

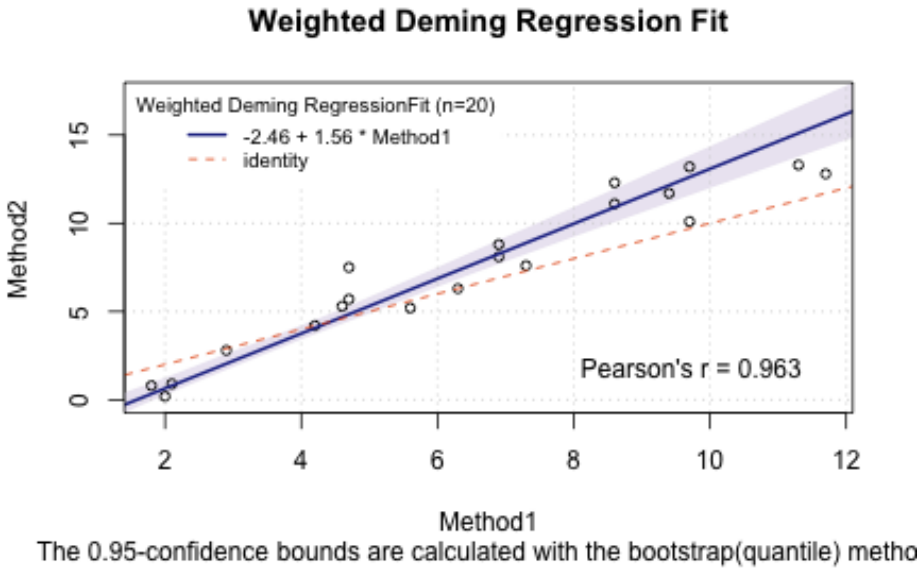
behavior of the two methods is approximately the same, then this value, called δ, is taken to be its default value of 1. The "mcr" package has both a Deming and a weighted Deming regression.

```
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.

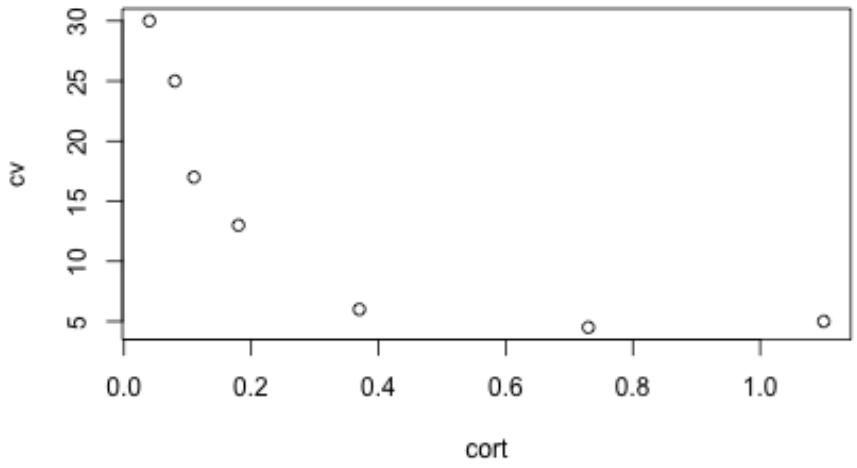
1.8 Non-Linear Regression

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)
```

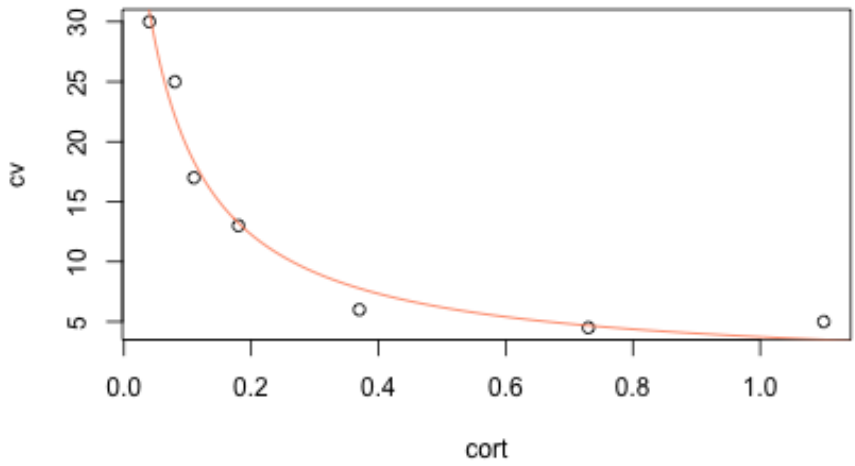
```
## 2064.25 :  0
## 149.5393 :  1.454733
```

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%:

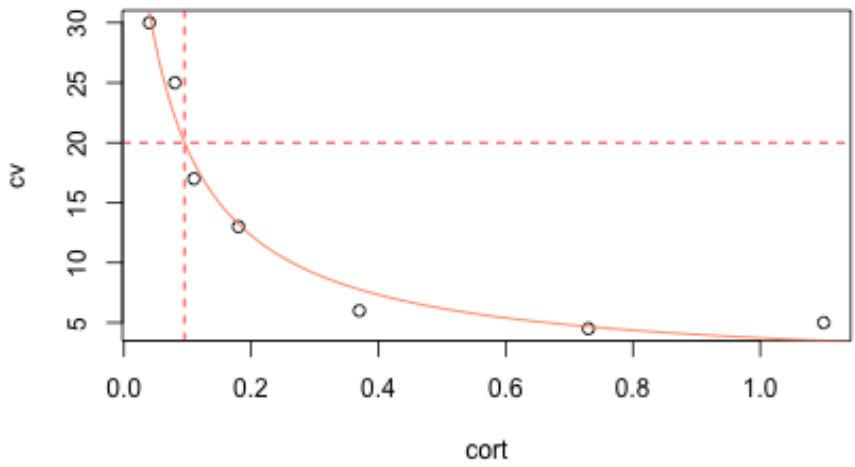
```
which((abs(cv.fit-20))<0.1) #tells us if there are any fitted values of the cv that are wit
```

```
## [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
```

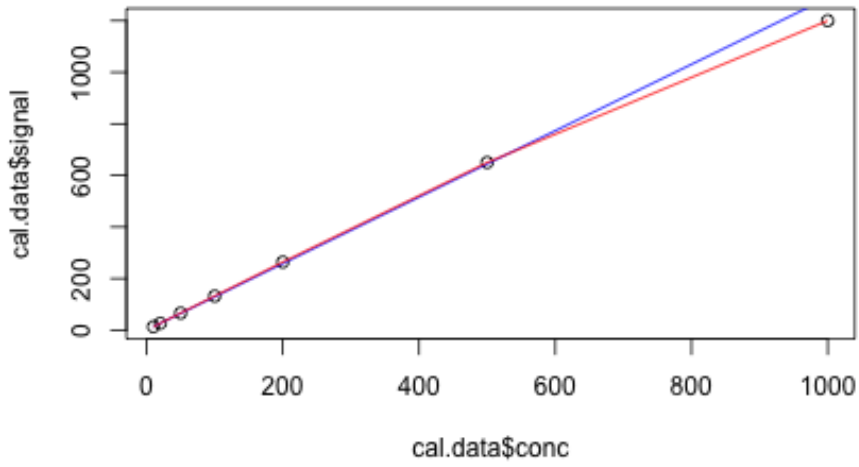
```
summary(cube.mod)
```

```
##
## 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
##
```

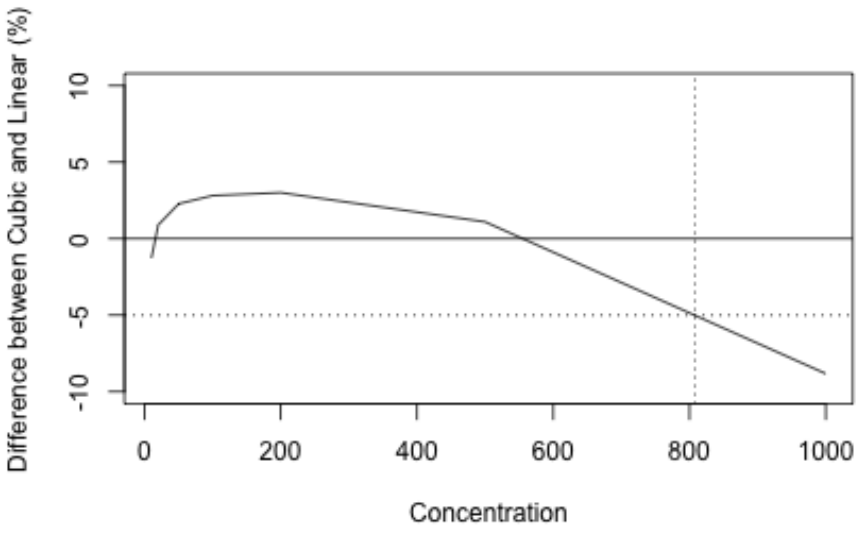
```
## Number of iterations to convergence: 1
## Achieved convergence tolerance: 4.03e-06
```

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!