BIOE286 Lab C

Optimizing sample size, extracting model components, annotating plots with model results, adding fitted lines & confidence intervals

1) Optimizing sample size

The goal of this exercise is to see how sample size influences our ability to accurately measure the mean of a response variable and how the standard error of the mean changes with sample size.

We'll work with data on size of abalone. Abalone are a group of shellfish species that used to be abundant along the coast of California but are now relatively rare (due to disease, overharvesting, loss of kelp forest habitat, pollution, etc.), with several species being listed as endangered. Their size is very important because reproduction increases sharply with size. Thus it's quite important to get an accurate estimate of the size of individual abalone within a population. A huge number of size measurements have been made and we are going to use these data to learn how our estimates of mean size change as our sample size increases.

Open the file "AbaloneSizes2.csv" (and don't forget to set your working directory first!).

```
dat=read.csv("AbaloneSizes2.csv")
```

Using `str(dat)` or `head(dat)` we find that there are 6900 observations of three variables in this file. The column names `colnames(dat)` are `SampleSize`, which is the number of observations, `Group` which is the group of abalone measured for that sample size, and `AbaloneSize`, which is the size of abalone measured in millimeters. `table(dat$SampleSize,dat$Group)` shows you that there are 10 groups, a-j, for each sample size with 5, 10, 25, 50, 100 and 500 observations per group for each respective sample size. You can think of the groups as random draws from the population; the variation among groups indicates how variable a sample is with a given sample size.

```
table(dat$SampleSize,dat$Group)
```

```
##
##         a   b   c   d   e   f   g   h   i   j
##   5     5   5   5   5   5   5   5   5   5   5
##   10   10  10  10  10  10  10  10  10  10  10
##   25   25  25  25  25  25  25  25  25  25  25
##   50   50  50  50  50  50  50  50  50  50  50
##   100 100 100 100 100 100 100 100 100 100 100
##   500 500 500 500 500 500 500 500 500 500 500
```

Before we move on, let me give you a better explanation of table, using sex and colony as examples. The `rep()` function repeats something X times. If you want to repeat `"Hello"` 3 times, you would write `rep("Hello",times=3)`. If you wanted to *alternate* `"Hello"` and `"Goodbye"` 3 times each, you would write `rep(c("Hello","Goodbye"),times=3)`. If you wanted to stack 3 Goodbye and 3 Hello, you would use "each" instead of "times"... `rep(c("Hello","Goodbye"),each=3)`. This is important because we make a lot of fake data sets in this class. Let me give you an example.

Here we'll repeat the names of 3 seal colonies, each 10 times, then repeat male and female (alternating) 15 times each. Note that there are 3 colonies * 10 repetitions = 30 rows of data, and this matches the sex column (2 sexes * 15 repetitions = 30 rows). It doesn't matter that colonies are stacked (each=10) whereas sexes are alternating (times=15). If these row numbers don't match when you create a data frame, you'll get an error.

```
r=data.frame(colony=rep(c("Ano Nuevo","San Simeon","Farallones"),each=10),
             sex=(rep(c("Male","Female"),times=15)))
print(r)
```

```
##        colony    sex
## 1   Ano Nuevo   Male
## 2   Ano Nuevo Female
## 3   Ano Nuevo   Male
## 4   Ano Nuevo Female
## 5   Ano Nuevo   Male
## 6   Ano Nuevo Female
## 7   Ano Nuevo   Male
## 8   Ano Nuevo Female
## 9   Ano Nuevo   Male
## 10  Ano Nuevo Female
## 11 San Simeon   Male
## 12 San Simeon Female
## 13 San Simeon   Male
## 14 San Simeon Female
## 15 San Simeon   Male
## 16 San Simeon Female
## 17 San Simeon   Male
## 18 San Simeon Female
## 19 San Simeon   Male
## 20 San Simeon Female
## 21 Farallones   Male
## 22 Farallones Female
## 23 Farallones   Male
## 24 Farallones Female
## 25 Farallones   Male
## 26 Farallones Female
## 27 Farallones   Male
## 28 Farallones Female
## 29 Farallones   Male
## 30 Farallones Female
```

Now that we have our fake dataset, we can use the `table()` function to figure out how many datapoints we have for each colony and/or each sex...

```
table(r$colony) #how many seals are there at each colony, across sexes?
```

```
##
##  Ano Nuevo Farallones San Simeon
##         10         10         10
```

```
table(r$colony,r$sex) #how many seals are there in each colony and sex?
```

```
##
##               Female Male
##   Ano Nuevo        5    5
##   Farallones       5    5
##   San Simeon       5    5
```

Look back at the raw data and you'll see that all of this checks out. Hooray!

OK, now let's generate a matrix of **means**, **medians**, **standard deviations**, and **standard errors** for each group for each sample size.

Calculating descriptive statistics is easy in R thanks to built-in functions like `mean()`, `median()`, `sd()`, and `length()`. Notice, though, that these functions take the mean of an entire dataset (e.g., `mean(dat)`) or an entire column (e.g., `mean(dat$AbaloneSize)`). What if you want to calculate the mean of each sample size category?

Well, you could use the `aggregate()` function, like this:

```
out=aggregate(AbaloneSize~Group+SampleSize,data=dat,FUN="mean")
head(out)
```

```
##   Group SampleSize AbaloneSize
## 1     a          5          74
## 2     b          5          86
## 3     c          5          76
## 4     d          5         104
## 5     e          5          86
## 6     f          5          76
```

This says: calculate the mean abalone size for each sample size category and group.

However, the `aggregate()` function is only really useful if you want to calculate a single descriptive statistic (e.g., the mean). It's pretty clunky to add multiple descriptive statistics to a single matrix.

If you want to calculate multiple descriptive statistics, you should use a package called 'dplyr' which is party of tidyverse that includes ggplot. It's easiest to just loady tidyverse which will also load these other packages. After you load the package, you can group by a variable within a dataset and then calculate summary statistics.

```
library(tidyverse)

out=dat %>% #do the following things to the data called dat
  group_by(SampleSize,Group) %>% #group by the sample size factor, and also group
  summarise(MEAN=mean(AbaloneSize),
            MEDIAN=median(AbaloneSize),
            SD=sd(AbaloneSize),
            N=n(),
            SE=SD/sqrt(N))
#summarize mean, median, standard deviation, number of cases, and SE
```

FYI within the `summarize()` function, putting `MEAN=` in front of `mean(AbaloneSize)` changes the column name of that calculated value to `MEAN`, which is helpful for our calculations below. If you didn't set the

column names (e.g., if that line of code read `summarise(mean(AbaloneSize),median(AbaloneSize), ...)`, you would end up with columns named `mean(AbaloneSize)` and `median(AbaloneSize)`. Annoying.

Also, see how the `n()` calculation doesn't have AbaloneSize specified within it, but the rest (mean, median, and sd) do? This is a little hard to explain, but basically `n()` calculates the number of rows, which is the same across all the columns in the datasheet, so you don't need to specify the AbaloneSize column here. In fact, you can't (if you try replacing `n()` with `n(AbaloneSize)`, the code will throw an error).

One of the beautiful things about summarize() is that once you've created a variable you can use it in subsequent lines within summarize(). R doesn't have a built-in function for **standard error (SE)** so we had to write the equation out ourselves. As you can see **SE** is calculated using $SD/\sqrt{N}$. The last line that calculates **SE** uses **SD** and **N** from the same summarize() command a couple lines earlier!

Two alternative ways we could have created the SE column after calculating SD, N, etc using the code chunk above:

```
out$SE=out$SD/sqrt(out$N)
```

This one is a little annoying because we had to type the object name `out` multiple times. You can use `with` to avoid this:

```
out$SE=with(out,SD/sqrt(N))
```

`with(data,expression)` tells R to look inside `data` for things it can't find in the workspace (in this case the columns of `out`, `SD` and `N`). Note that if you ran each of these two lines you replaced the original column `SE` that the `summarize()` line created, but that's ok because you replaced it with the same thing!!!

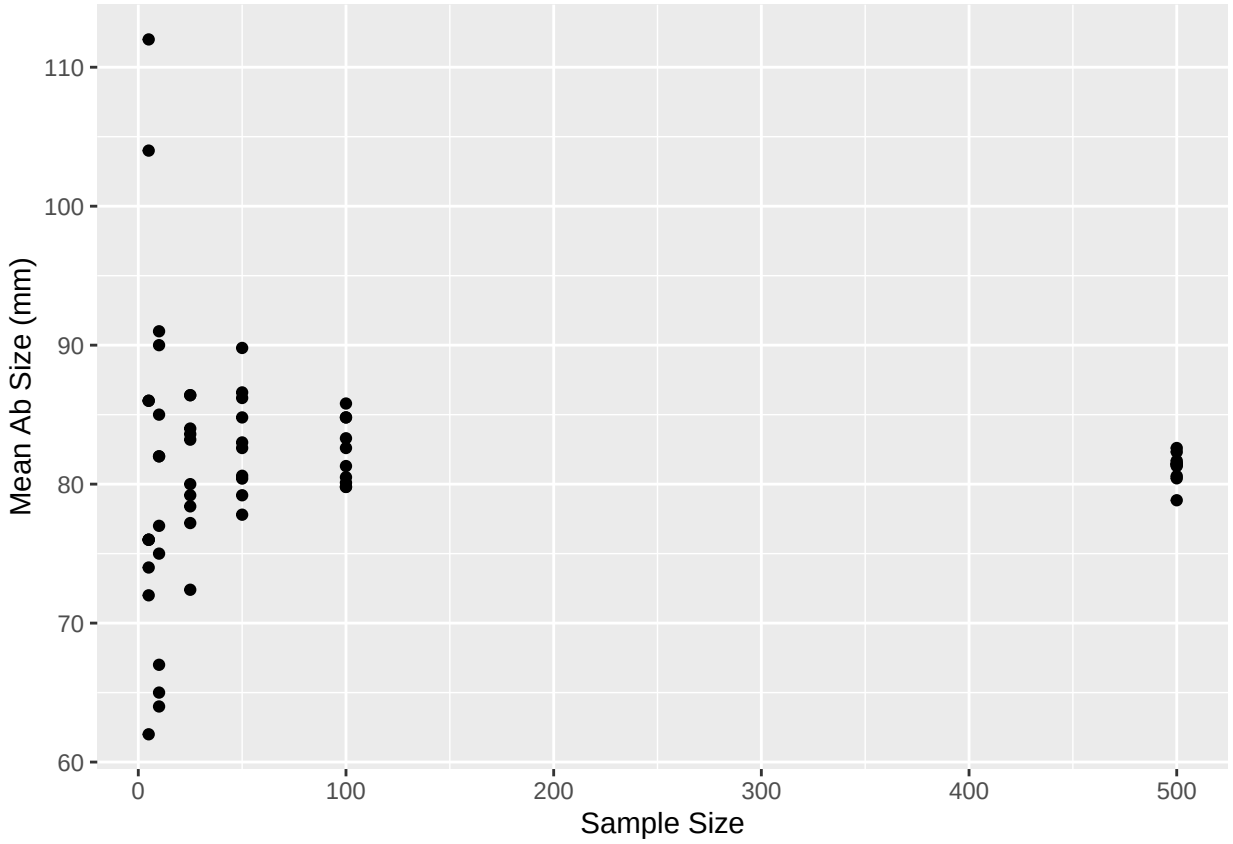
Go ahead and take a look at `out` to make sure it did what you think it did:

```
out
```

```
## # A tibble: 60 x 7
## # Groups:   SampleSize [6]
##    SampleSize Group  MEAN MEDIAN    SD     N    SE
##         <int> <chr> <dbl>  <dbl> <dbl> <int> <dbl>
##  1          5 a        74     70 28.8      5 12.9
##  2          5 b        86     80  8.94     5  4
##  3          5 c        76     70 19.5      5  8.72
##  4          5 d       104    100 27.0      5 12.1
##  5          5 e        86     90 32.1      5 14.4
##  6          5 f        76     70 20.7      5  9.27
##  7          5 g       112    100 27.7      5 12.4
##  8          5 h        62     60 19.2      5  8.60
##  9          5 i        72     60 21.7      5  9.70
## 10          5 j        76     80 38.5      5 17.2
## # i 50 more rows
```

Let's create a graph of the relationships between `SampleSize` and the summary statistics using `plot()`. Put `SampleSize` (the continuous variable) on the X axis and `mean(AbaloneSize)` on the Y axis. Note that until this point in the labs, we've just been making ggplots but not saving them as anything. Let's try to save them (e.g., save the plot below as figa). When you do this, it won't automatically show the plot unless you type `figa` into the Console or run it in your Source Code. So, I usually just put the name of the plot on the next line. This'll come in handy when we go to make a multi-panel plot below.

```
figa=ggplot(data=out,aes(x=SampleSize,y=MEAN))+
  geom_point()+
  labs(x="Sample Size",y="Mean Ab Size (mm)")
figa
```

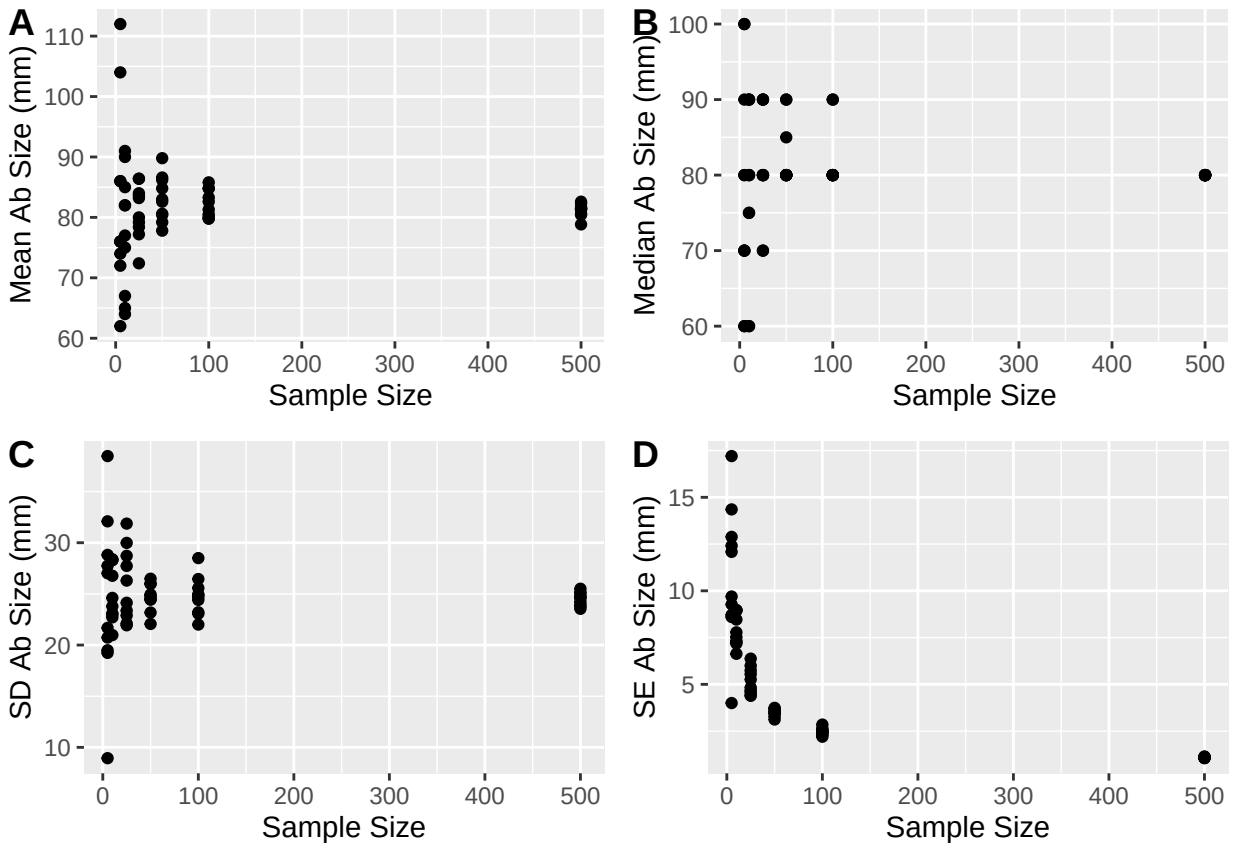


Make the other 3 plots (Median, SD, SE on the y-axis)...

```
figb=ggplot(data=out,aes(x=SampleSize,y=MEDIAN))+
  geom_point()+
  labs(x="Sample Size",y="Median Ab Size (mm)")
figc=ggplot(data=out,aes(x=SampleSize,y=SD))+
  geom_point()+
  labs(x="Sample Size",y="SD Ab Size (mm)")
figd=ggplot(data=out,aes(x=SampleSize,y=SE))+
  geom_point()+
  labs(x="Sample Size",y="SE Ab Size (mm)")
```

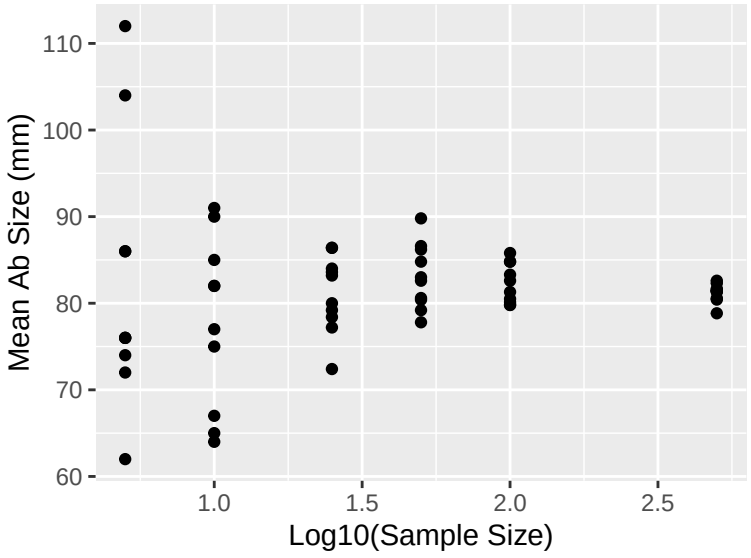
Then using the ggarrange() function within the ggpubr package, make a panelled plot:

```
library(ggpubr)
ggarrange(figa,figb,figc,figd, #good thing we named them!
          nrow=2,ncol=2,
          labels=c("A","B","C","D"))
```



That doesn't look so great because all but one point in each panel is bunched on the left. Let's try to make Panel A (mean size ~ sample size) a bit cleaner by plotting log(sample sizes) in order to see the relationships more clearly:

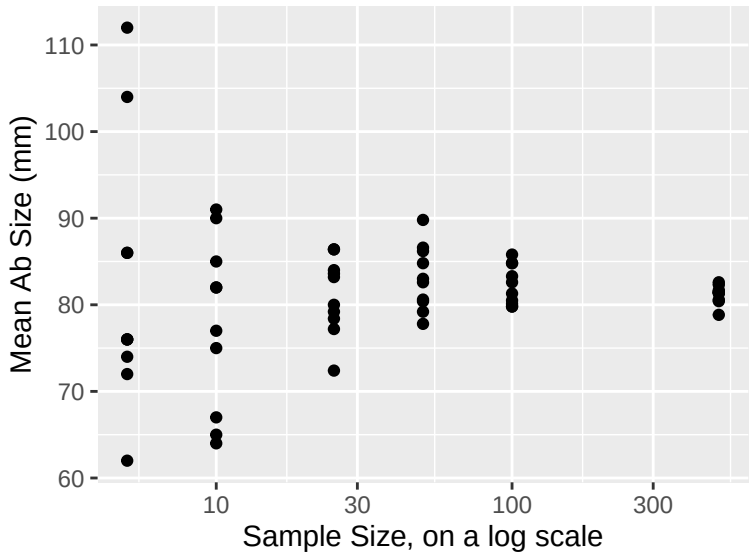
```
ggplot(data=out,aes(x=log10(SampleSize),y=MEAN))+
  geom_point()+
  labs(x="Log10(Sample Size)",y="Mean Ab Size (mm)")
```



log10(SampleSize) tells R that you want to log-transform the independent variable (SampleSize). Unfortu-

nately, most people won't be able to convert the log scale into numbers they understand. A better option than logging the data is to use a log scale x-axis like this:

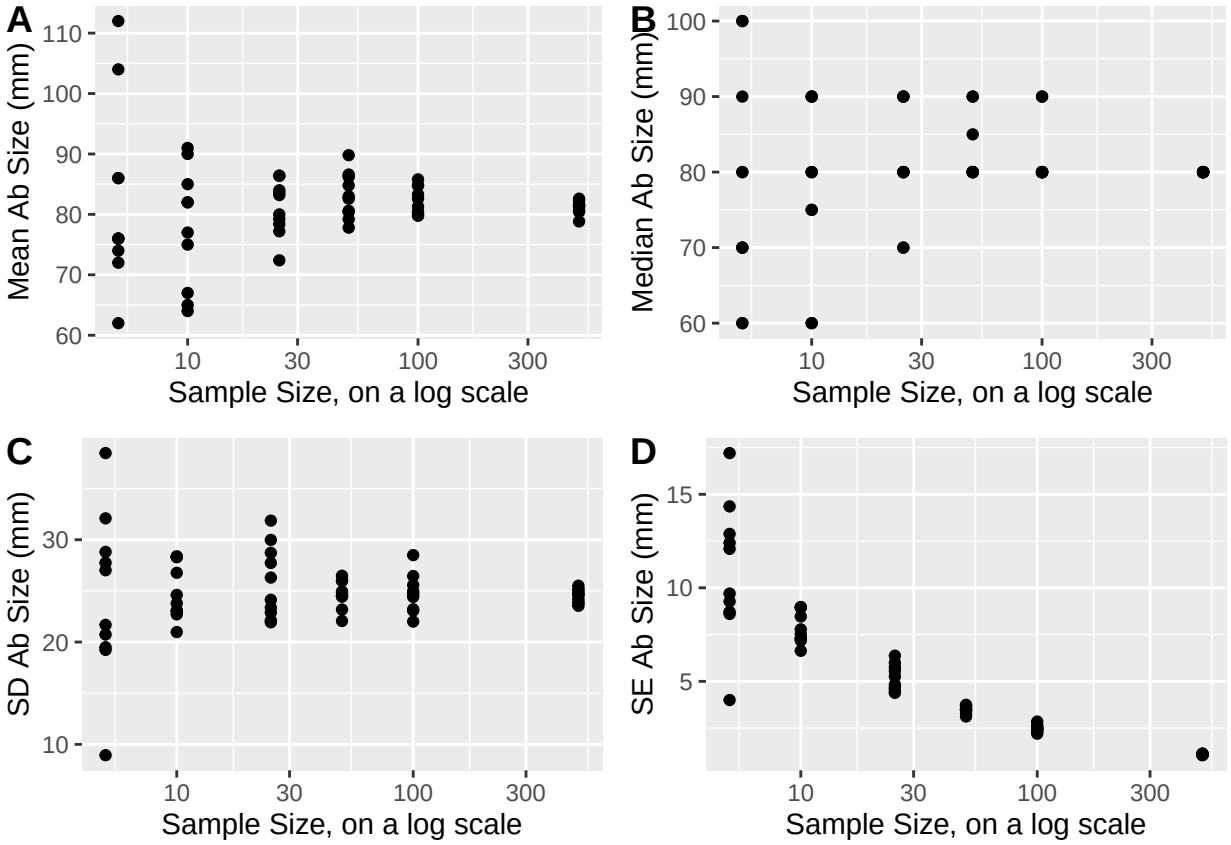
```
ggplot(data=out,aes(x=SampleSize,y=MEAN))+
  geom_point()+
  scale_x_log10()+ #this makes a log scale x axis!
  labs(x="Sample Size, on a log scale",y="Mean Ab Size (mm)")
```




```
#note we changed the x label to show that the axis, not the data, are logged
```

That's much better. Let's remake the 4 panel plot with log x-axes (show a log axis, don't log the data):

```
figalog=ggplot(data=out,aes(x=SampleSize,y=MEAN))+
  geom_point()+
    scale_x_log10()+ #this makes a log scale x axis!
  labs(x="Sample Size, on a log scale",y="Mean Ab Size (mm)")
figblog=ggplot(data=out,aes(x=SampleSize,y=MEDIAN))+
  geom_point()+
  scale_x_log10()+ #this makes a log scale x axis!
  labs(x="Sample Size, on a log scale",y="Median Ab Size (mm)")
figclog=ggplot(data=out,aes(x=SampleSize,y=SD))+
  geom_point()+
  scale_x_log10()+ #this makes a log scale x axis!
  labs(x="Sample Size, on a log scale",y="SD Ab Size (mm)")
figdlog=ggplot(data=out,aes(x=SampleSize,y=SE))+
  geom_point()+
  scale_x_log10()+ #this makes a log scale x axis!
  labs(x="Sample Size, on a log scale",y="SE Ab Size (mm)")
ggarrange(figalog,figblog,figclog,figdlog,
          nrow=2,ncol=2,
          labels=c("A","B","C","D"))
```



(Advanced exercise: The 4-panel plot here has the same x-axis on all four plots (and frankly lots of duplicate coding like the same labels and log x axes) so it's wasting space. Can you use `pivot_longer()` to put `out` into long format and then use `ggplot()` and `facet_wrap()` to make a neater plot?)

At what sample size does the mean abalone size appear to stabilize (i.e. all 10 values are close to each other)? Does this match with another panel of the figure?

Although a larger sample size is always better, it costs money and time to collect more data. Is there a sample size that might be a decent trade-off between maximize sampling efficiency and to minimizing error?

2) Extracting model components

In addition to learning how to fit and interpret models, it is really helpful to be able to extract the model output to create tables, annotate figures, and even make fancy graphical representations. Let's talk about model outputs using an example of fake data (making fake data is - weirdly - also a super important skill in coding).

Let's create a fake dataframe of two normally distributed values. Let's say the response variable (y, `ConfidenceBeg`) is confidence in R for beginners, and the explanatory variable (x, `Time`) is the number of days they've spent coding. We'll make the slope 1.5 with no intercept - you gotta spend the time if you want the skills, but if you spend the time you pick up confidence fast!

```
set.seed(1)
FakeData=data.frame(Time=rnorm(n=1000,mean=50,sd=5))
```

```
FakeData$ConfidenceBeg=0+1.5*FakeData$Time+rnorm(n=1000,mean=0,sd=5)
#add a random number to the Confidence values so they aren't perfectly correlated with Time
```

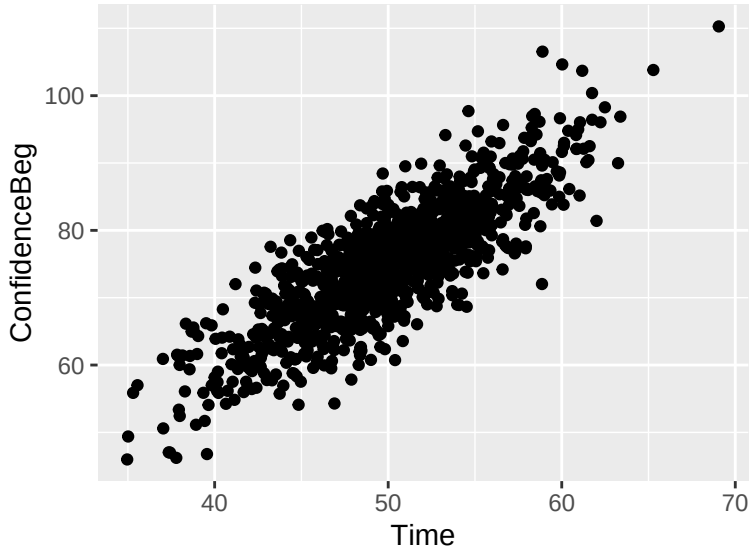
Do you know what the `set.seed()` function does? Look it up using the help file if you don't know!

Practice makes perfect! Check out the data.frame using `head()` and `ggplot()` to make sure the fake data look good (lol).

```
head(FakeData)
```

```
##       Time ConfidenceBeg
## 1 46.86773      75.97642
## 2 50.91822      81.93698
## 3 45.82186      64.37890
## 4 57.97640      88.01826
## 5 51.64754      77.81829
## 6 45.89766      60.53324
```

```
ggplot(data=FakeData,aes(x=Time,y=ConfidenceBeg))+
  geom_point()
```



Now, let's fit a linear model to these data.

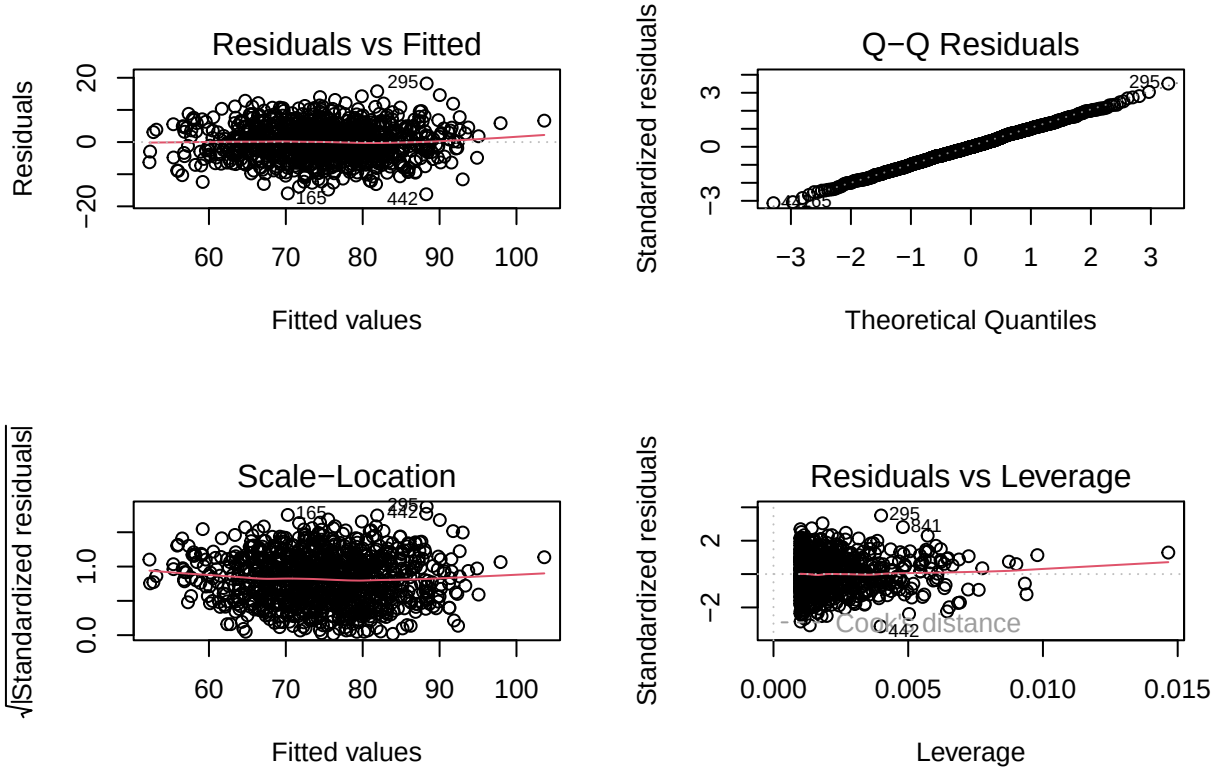
```
fit=lm(ConfidenceBeg~Time,data=FakeData)
```

A really important part of model fitting is making sure the data meet the assumptions of the model.

Testing for normality

Here's code for testing the four assumptions of general linear models all in one place. Assume you have a model called model that is something like: model=lm(Y~x1+x2,data=m) R will give you four plots to make visual assessments which are useful but you should do the actual tests below as well:

```r
par(mfrow=c(2,2)) #make a 2x2 matrix of plots
plot(fit) #make 4 diagnostic plots
```




```r
#Normality of residuals:
shapiro.test(resid(fit))
```

```
##
##  Shapiro-Wilk normality test
##
## data:  resid(fit)
## W = 0.99932, p-value = 0.9824
```

```r
#P-values <0.05 indicate non-normal residuals
```

```r
#Linearity of the relationship:
summary(lm(resid(fit)~poly(predict(fit),2)))
```

```
##
## Call:
## lm(formula = resid(fit) ~ poly(predict(fit), 2))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -16.4471  -3.3990  -0.0769   3.7820  18.0147
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)            -1.432e-16  1.645e-01   0.000    1.000
## poly(predict(fit), 2)1  3.642e-14  5.203e+00   0.000    1.000
## poly(predict(fit), 2)2  4.561e+00  5.203e+00   0.877    0.381
##
## Residual standard error: 5.203 on 997 degrees of freedom
## Multiple R-squared:  0.0007702,  Adjusted R-squared:  -0.001234
## F-statistic: 0.3842 on 2 and 997 DF,  p-value: 0.6811
```

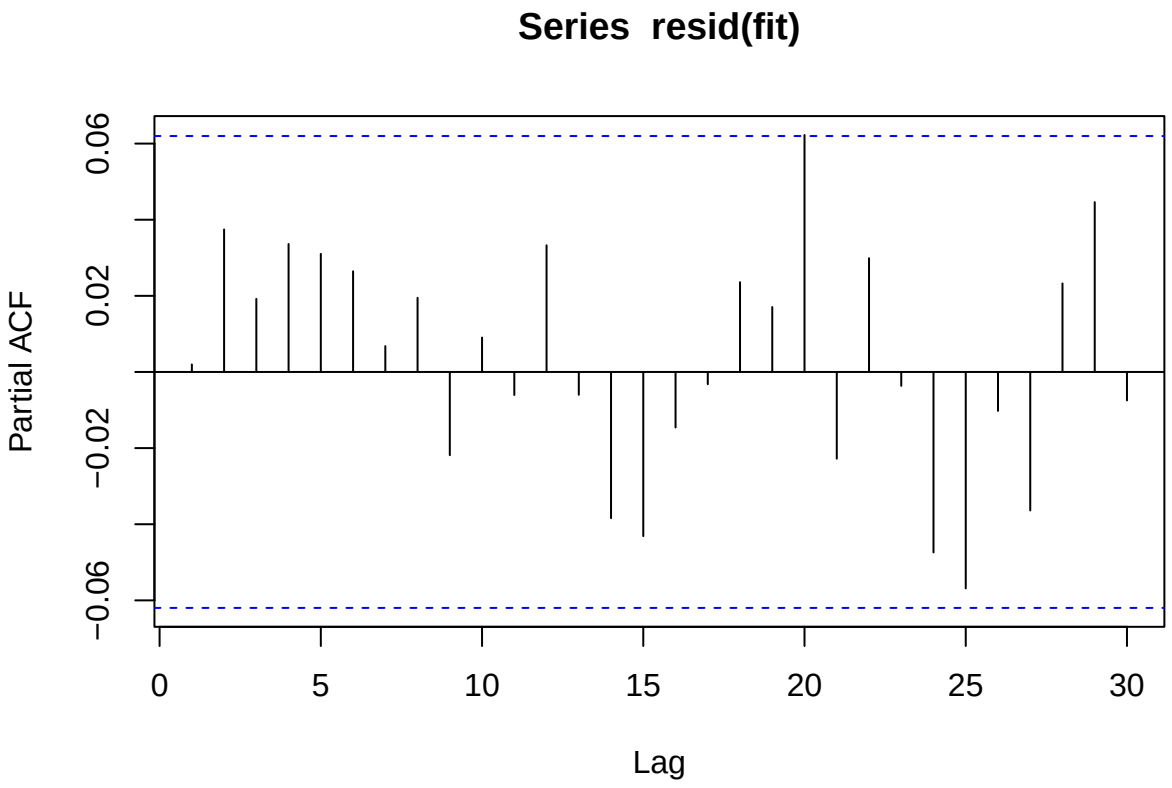
```
#This fits the residuals to a quadratic polynomial model.
#If the 2nd term of the polynomial (the ,2) is significant,
#there is non-linearity in the residuals
```

```
#Homoscedasticity (constant variance):
library(lmtest)
bptest(fit)
```

```
##
##  studentized Breusch-Pagan test
##
## data:  fit
## BP = 0.16709, df = 1, p-value = 0.6827
```

```
#P-values <0.05 indicate non-constant variance (heteroskedasticity)
```

```
#Independence of residuals for temporal data
#(i.e. data points taken on successive days/weeks/etc.);
#your response variable needs to be sorted by date:
pacf(resid(fit))
```



```
#This makes a plot of the partial autocorrelation of the residuals,
#which is the correlation between the residuals.
#If the bars extend above or below the blue dashed lines,
#this indicates significant correlation at that timestep.
```

HELL YEAH! It looks like our data does not violate any of the four assumptions of a linear model (normality, linearity, homoscedasticity, [temporal] independence of residuals).

Now, we can look at the model fit in two ways. The first is to just type `fit` into the console or source code. The second is to type `summary(fit)` into the console or source code. Try both; what is the difference?

There is a built-in function that I briefly introduced you to in the first lab called `coef()`. Check out the help file to see what it does by typing `help(coef)`. Now, try to run `coef(fit)` and `coef(summary(fit))` in your console or source code. **What is the difference?**

Spoiler alert... it is always (almost always?) better to look at the summary of the fitted model, and to extract the coefficients from the summary of the fitted model (e.g., `coef(summary(fit))`), rather than dealing with the fitted model output itself. Not sure why the R creators decided that, but it is what it is ;)

So. You can look at all the model coefficients using `coef(summary(fit))`. How do you extract individual values? Let's say you want to pull out the slope Here are two ways to do that. The first way is sloppy but faster. The second way is more accurate but longer.

```
coef(summary(fit))[2]
```

```
## [1] 1.506433
```

```
coef(summary(fit))["Time","Estimate"]
```

```
## [1] 1.506433
```

See if you can figure out how to extract the p-value of the slope. Ask us if you have questions.

Like all R objects, you can use the `str()` function to check the structure of `summary(fit)` (type `str(summary(fit))` into the console). See how the things you can extract, like `r.squared` are shown in the list with dollar signs `$` in front of them? This tells you that you can extract them using the dollar sign notation (like `summary(fit)$r.squared`). Try extracting the adjusted R squared using this method!

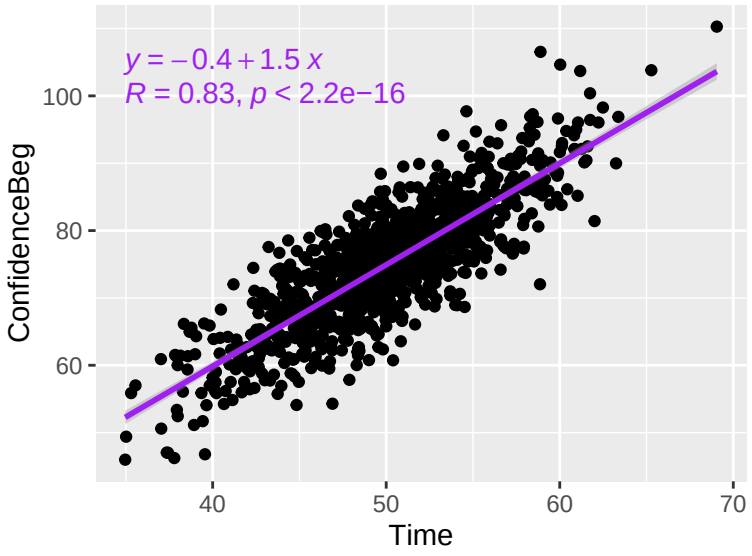
Why is all of this automatic coefficient extraction important? Well, if you want to extract the equation and R^2 and put it on the graph, you don't want want to do it manually! You'll likely make a mistake, or forget to update the value on the plot if you change the analysis!

3) Stats annotations, fitted lines, and CIs

Now, let's add the fitted line to the plot along with the equation as a text annotation. For the fitted line, ggplot has a function called `geom_smooth()` that you can use to draw a line with confidence intervals onto a plot. All you need to do is specify the function (here, `method="lm"` for a linear model), and the color if you want :)

I've also added the equation for the geom_smooth() line, the correlation, and the P-value using two commands from the ggpubr package.

```
ggplot(data=FakeData,aes(x=Time,y=ConfidenceBeg))+geom_point()+
geom_smooth(method="lm",colour="purple")+
  stat_cor(label.y = c(100),color="purple")+
  stat_regline_equation(label.y = c(105),color="purple")
```



WARNING: Before you decide to slap a line on a plot, it is best practice to fit the model(s) yourself to determine the best model to use. It is also best practice to double check that the slopes and intercepts you get from that manual model fitting (e.g., `summary(fit)`) match the ones that ggplot shows. Finally, you might want to check out the help file for `geom_smooth()` to figure out what the confidence intervals represent!

Now let's add an additional groups of students: advanced coders. The advanced group starts at a higher confidence level (75) but is in the slower part (shallower slope) of the learning curve (slope 0.5)!

```
FakeData$ConfidenceAdv=75+0.5*FakeData$Time+rnorm(n=1000,mean=0,sd=5)
```

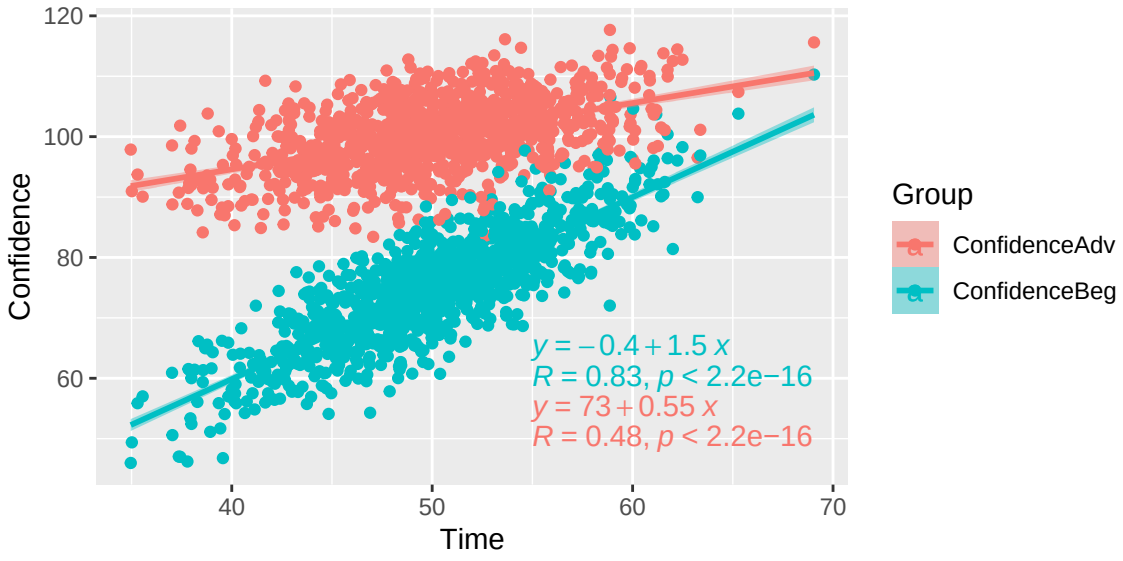
As usual, ggplot wants our data to be in long format, so let's use the `pivot_longer()` function:

```
FakeDataLong=pivot_longer(FakeData,cols=c("ConfidenceBeg","ConfidenceAdv"),
                          names_to="Group",values_to="Confidence")
```

Check out the long dataset to make sure the pivot worked as you wanted it to!

Now, we can plot the points, lines, and equation annotations separately by group.

```
ggplot(data=FakeDataLong,aes(x=Time,y=Confidence,fill=Group))+geom_point(aes(color=Group))+
  geom_smooth(aes(color=Group,fill=Group),method="lm")+
  stat_cor(label.x=c(55,55),label.y = c(50,60),aes(color=Group))+
  stat_regline_equation(label.x=c(55,55),label.y = c(55,65),aes(color=Group))
```



What do you notice about how `col=...` is specified in the first plot (just one group) versus the second (two groups)? See how it is specified within the `aes()` function instead of outside it? This tells ggplot that color is a dynamic instead of a fixed variable - ggplot essentially colors the points/lines by the groups, as they are defined at the top of the plot. Does that make sense?

Also, note that the lines (`geom_smooth()`) have both color and fill specified, whereas the points (`geom_point()`) and annotations only have color specified. That's because the lines have both confidence intervals (the "ribbon") and a mean fit. Try removing the `fill=Group` from within `geom_smooth()` and see what happens!

Like the last plot, let's double check that the stats annotations make sense. We can do it one group at a time:

```
fit2=lm(Confidence~Time,subset(FakeDataLong,Group=="ConfidenceAdv"))
summary(fit2)
```

```
##
## Call:
## lm(formula = Confidence ~ Time, data = subset(FakeDataLong, Group ==
##     "ConfidenceAdv"))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -17.9091  -3.2070  -0.0894   3.4331  14.0525
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  72.6199     1.5816   45.92   <2e-16 ***
## Time          0.5492     0.0315   17.43   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.152 on 998 degrees of freedom
## Multiple R-squared:  0.2335, Adjusted R-squared:  0.2327
## F-statistic:   304 on 1 and 998 DF,  p-value: < 2.2e-16
```

Try repeating that with the beginner coders.

What ggplot is doing is similar to fitting lines to each dataset separately, but not quite the same. It is actually fitting a model with an interaction between Group and Time:

```
fit3=lm(Confidence~Time*Group,FakeDataLong)
summary(fit3)
```

```
##
## Call:
## lm(formula = Confidence ~ Time * Group, data = FakeDataLong)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -17.9091  -3.2855  -0.0803   3.6188  18.2216
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)              72.61989    1.58934   45.69   <2e-16 ***
## Time                      0.54919    0.03165   17.35   <2e-16 ***
## GroupConfidenceBeg      -73.02245    2.24766  -32.49   <2e-16 ***
## Time:GroupConfidenceBeg   0.95724    0.04477   21.38   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.177 on 1996 degrees of freedom
## Multiple R-squared:  0.8785, Adjusted R-squared:  0.8783
## F-statistic:  4809 on 3 and 1996 DF,  p-value: < 2.2e-16
```

If we haven't talked about interactions yet, we will soon! See if you can make sense out of the values in the table and the equations on the plot!