Week 5: Introduction to R and Linear Models for Exponential Growth

EFB 370 - Spring 2022

Nate Wehr and Elie Gurarie

2022-02-24

Part 1: Basics

> The following examples should give you a first look at what R does and how it works.

Introduction

R is a command-line program, which means commands are entered line-by-line at the prompt. Being a programming language it is very finicky. Everything has to be entered exactly right - including case-sensitivity.

There are two ways of entering commands: either typing them out carefully into the “Console Window” (the lower-left window in Rstudio) and hitting `Enter` or writing and editing lines in the script window (upper-left window in Rstudio), and “passing” the code into the console by hitting `Ctrl+Enter` .

In general, it is better to do all of your coding in a script window, and then save the raw code file as a text document, which you can revisit and re-run at any point later.

R is a calculator

```
1+2
```

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

```
3^6
```

```
## [1] 729
```

```
sqrt((20-19)^2 + (19-19)^2 + (19-18)^2)/2
```

```
## [1] 0.7071068
```

```
12345*54312
```

```
## [1] 670481640
```

Assigning variable names

The assignment operator is `<-`. It's supposed to look like an arrow pointing left.

```
X <- 5      # sets X equal to 5
```

Using the assignment operator sets the value of `X` but doesn't print any output. To see what `X` is, you need to type:

```
X
```

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

Note that `X` now appears in the upper-right panel of Rstudio, letting you know that there is now an object in memory (also called the "Environment") called `X`.

Now, you can use `X` as if it were a number

```
X*2
```

```
## [1] 10
```

```
X^X
```

```
## [1] 3125
```

Note that you can name a variable ANYTHING, as long as it starts with a letter.

```
Fred <- 5
Nancy <- Fred*2
Fred + Nancy
```

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

Vectors

Obviously, `X` can be many things more than just a single number. The most important kind of object in R is a “vector”, which is a series of inputs (and therefore resembles “data”).

`c()` is a function - a very useful function that creates “vectors”. In all functions, arguments are passed within parentheses.

We can use the `c()` function as follows:

```
X <- c(3,4,5)   # sets X equal to the vector (3,4,5)
X
```

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

Now, let’s do some arithmetic with this vector:

```
X + 1
```

```
## [1] 4 5 6
```

```
X*2
```

```
## [1]  6  8 10
```

```
X^2
```

```
## [1]  9 16 25
```

```
((X+X^2/2)/X)^2
```

```
## [1]  6.25  9.00 12.25
```

Note that in all of these cases, the arithmetic operations are performed on a term-by-term basis.

We can very quickly look at some exponential growth (e.g. using Washington sea otter numbers: N0 = 60, 50 years gone by, and an annual growth rate of 7%:

```
years <- 0:60
lambda <- 1.07
N0 <- 50
N0*lambda^years
```

```
##  [1]   50.00000   53.50000   57.24500   61.25215   65.53980   70.12759
##  [7]   75.03652   80.28907   85.90931   91.92296   98.35757  105.24260
## [13]  112.60958  120.49225  128.92671  137.95158  147.60819  157.94076
## [19]  168.99661  180.82638  193.48422  207.02812  221.52009  237.02649
## [25]  253.61835  271.37163  290.36765  310.69338  332.44192  355.71285
## [31]  380.61275  407.25564  435.76354  466.26699  498.90568  533.82907
## [37]  571.19711  611.18091  653.96357  699.74102  748.72289  801.13349
## [43]  857.21284  917.21774  981.42298 1050.12259 1123.63117 1202.28535
## [49] 1286.44533 1376.49650 1472.85125 1575.95084 1686.26740 1804.30612
## [55] 1930.60755 2065.75007 2210.35258 2365.07726 2530.63267 2707.77695
## [61] 2897.32134
```

Exercise 1: Calculate population growth

> You can get some really quick population growth answers this way. Compute how many sea otters there will be by 2050 and 2100 (80 and 130 years after release). *HINT: you can just replace the vector with a single number*.

Multiple Vectors and Data Frames

Data is most often multiple vectors of the same length. If we create a second vector `Y` we can use it alongside our first vector `X` using the `data.frame()` command:

```
Y <- c(1,2,3)
data.frame(X,Y)
```

```
##   X Y
## 1 3 1
## 2 4 2
## 3 5 3
```

That just outputs the data, but you should save it another object:

```
mydata <- data.frame(X,Y)
```

A data frame has columns with names:

```
ncol(mydata)
```

```
## [1] 2
```

```
names(mydata)
```

```
## [1] "X" "Y"
```

A column can be extracted from a dataframe with a `$`:

```
mydata$X
```

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

```
mydata$Y
```

```
## [1] 1 2 3
```

Exercise 2: Create a data frame

> Create a data frame called `myseaotters` that contains the number of years from 0 to 130 and the predicted population, starting at 60 animals with a lambda rate of 1.07.

Part 2: Loading and Exploring Data

> The following examples should explain how to import data frames and to work with the data contained within them.

Loading Data

We will use Steller sea lion (*Eumotopias jubatus*) data as an example. These are weights, lengths, and girths (basically, under the arm/flipper pits) of sea lion pups about two months after birth as part of a tagging mark-recapture study. These data were collected (in part by Dr. Gurarie) on five islands in the Russian North Pacific.

This is what sea lion pups look like:

This dataset is available on Blackboard. Once you download it, you can use the File Explorer to determine its location and read it into R.

```
SeaLions <- read.csv("<insert the directory>/SeaLions.csv")
```

If you copy and paste the file directory in, you have to change the direction of the slashes. Note that `csv` is a **text based** file type (Comma Separated Values) - it just means that commas between entries separate columns. You can save any Excel file as a `csv` using the Save As function. CSVs are by far the the most common and convenient file type used for loading into R.

Alternatively, you can import datasets into `R` using the `RStudio` interface. To do this:

1. Click `File`
2. Hover over `Import Dataset`
3. Select `From Text` (or if you're doing this with a different file type in the future, the matching type of file)
4. Navigate to the SeaLions dataset, highlight it, and click `open`

This will automatically input the proper code into the console and save your file to the environment. Note that the file has the same name rather than a name you designate for it.

Working with data frames

Look at some properties of this data file, with the following functions:

```
is(SeaLions) # tells what type of files we have
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

```
names(SeaLions) # tells us the names of all the columns
```

```
##  [1] "Island"      "TaggingDate" "ID"          "Weight"      "Length"
##  [6] "Girth"       "BCI"         "Sex"         "Age"         "DOB"
```

```
head(SeaLions) # shows the first several rows of the dataframe
```

```
##     Island TaggingDate     ID Weight Length Girth       BCI Sex Age  DOB
## 1 Chirpoev   7/10/2005 Br800L   15.5     93  69.0 0.7419355   M  NA <NA>
## 2 Chirpoev   7/10/2005 Br801L   29.0    106  71.0 0.6698113   F  NA <NA>
## 3 Chirpoev   7/10/2005 Br802L   35.5    112  76.0 0.6785714   M  NA <NA>
## 4 Chirpoev   7/10/2005 Br803L   32.0    107  72.0 0.6728972   M  NA <NA>
## 5 Chirpoev   7/10/2005 Br804L   32.0    105  73.5 0.7000000   M  NA <NA>
## 6 Chirpoev   7/10/2005 Br805L   33.5    111  72.0 0.6486486   M  NA <NA>
```

Use a `$` to extract a given column:

```
Length <- SeaLions$Length
Weight <-SeaLions$Weight
Island <- SeaLions$Island
Sex <- SeaLions$Sex
```

Summary Statistics

Some basic summary statistics include:

```
range(Length) #range
```

```
## [1]  93 126
```

```
median(Length) #median
```

```
## [1] 110
```

```
mean(Length) #mean
```

```
## [1] 109.8434
```

```
var(Length) #variance
```

```
## [1] 34.82854
```

```
sd(Length) #standard deviation
```

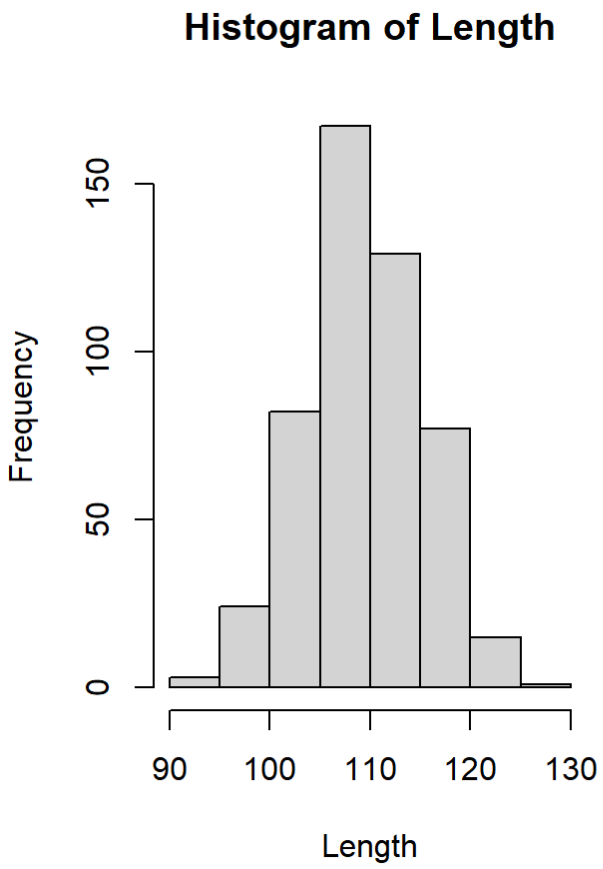
```
## [1] 5.901571
```

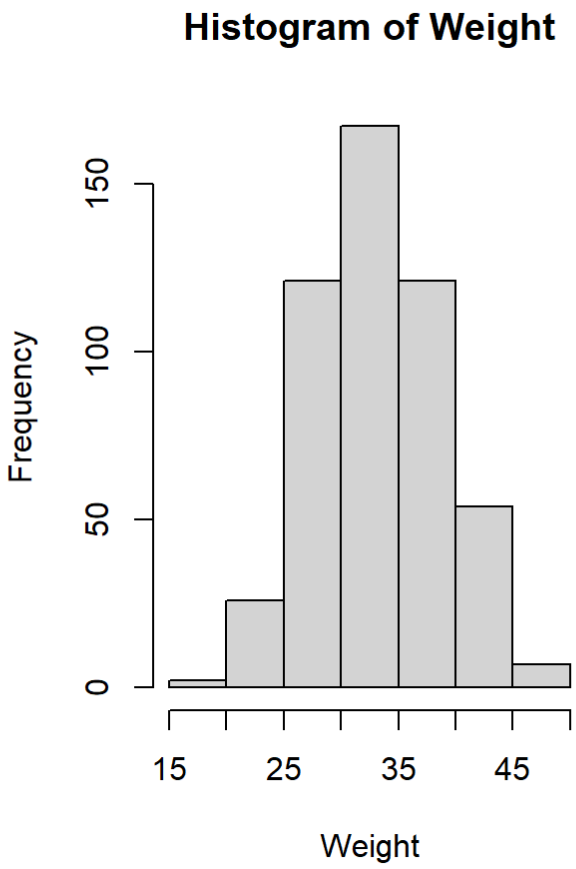
Graphical Summaries

Histogram

A histogram can show us the distribution of a single continuous variable:

```
hist(Length)
hist(Weight)
```

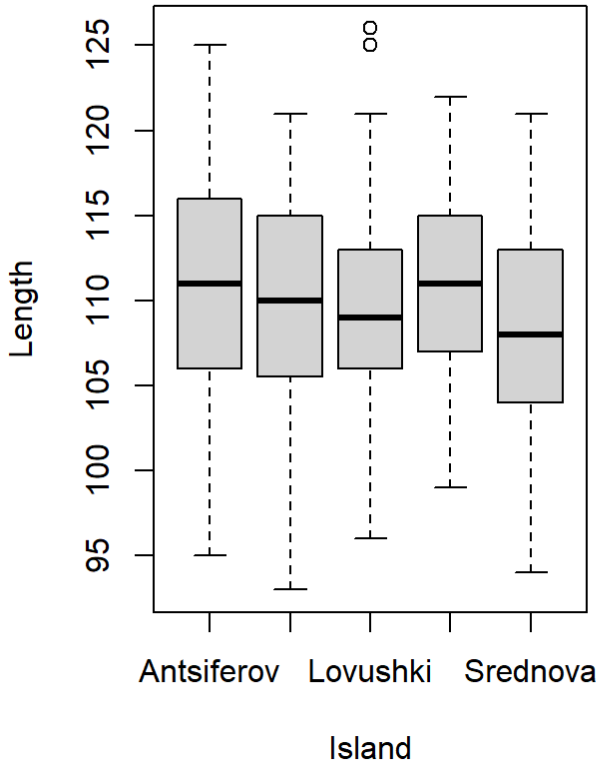


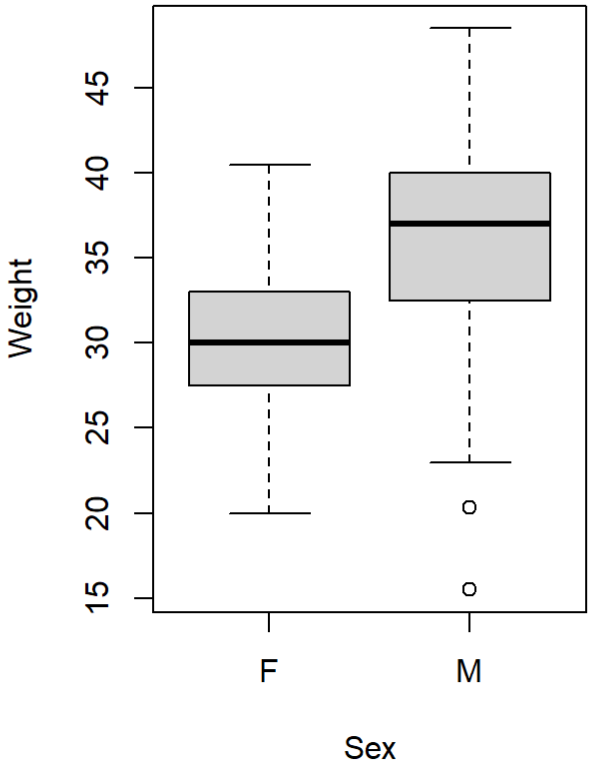


Boxplot

A boxplot shows us relationships between a continuous variable (like Length/Weight/Girth) and a discrete variable (like Island/Sex):

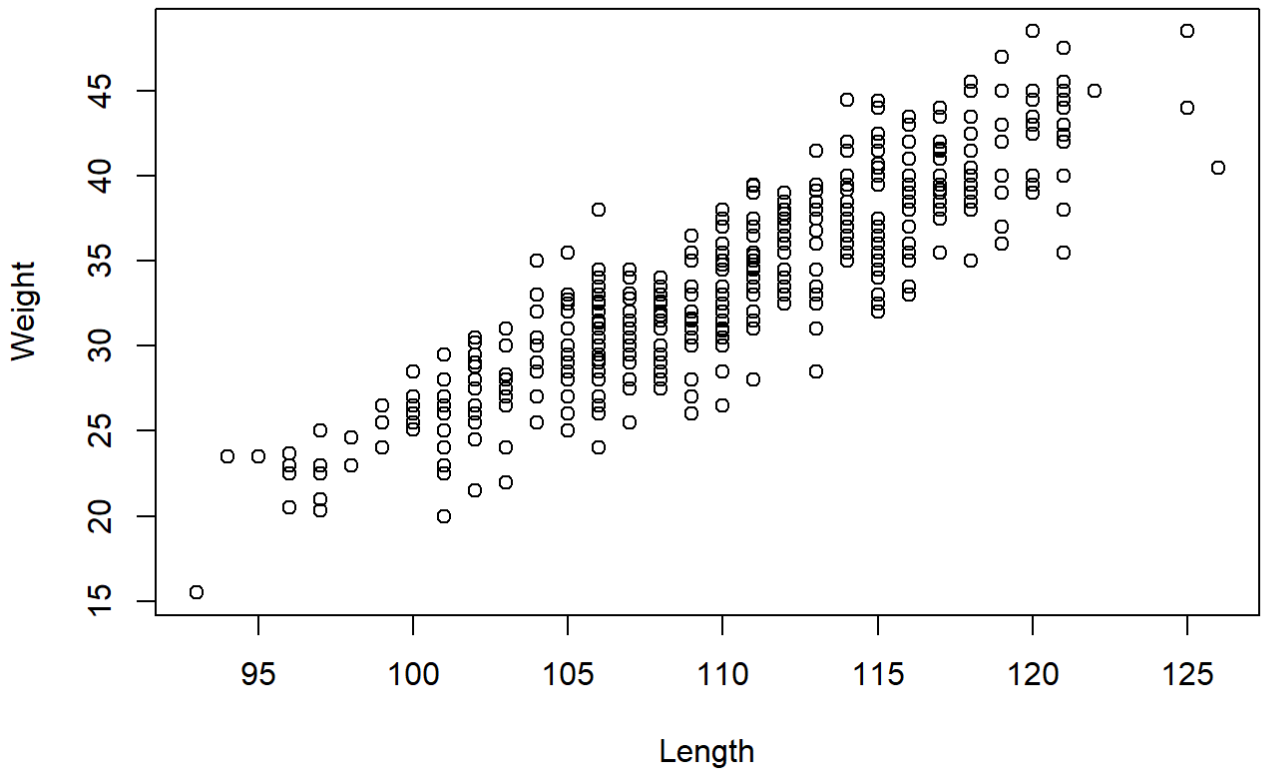
```
boxplot(Length ~ Island)
boxplot(Weight ~ Sex)
```





Scatterplot

A scatterplot shows us relationships between two continuous variables:



Exercise 3: Plot the sea otter population growth

> Plot the `myseaotter` data frame you made in Exercise 2.

Part 3: Linear Models for Exponential Growth

> The following examples show you how to do a linear regression model and a log transformation model

Importing Data

The first step is to import our new dataset. This time with sea otters.

```
SeaOtters <- read.csv("<insert your directory>/SeaOtters.csv")
```

Let's double check that the data we imported matches our Excel file.

```
head(SeaOtters)
```

```
##   year count
## 1 1970    59
## 2 1989   208
## 3 1990   212
## 4 1991   276
## 5 1992   313
## 6 1993   307
```

Extract variables and explore

```
Year <- SeaOtters$year
Count <- SeaOtters$count
```

It's usually a good idea to run some summary statistics on each of your variables. Here's a couple examples

```
range(Year)
```

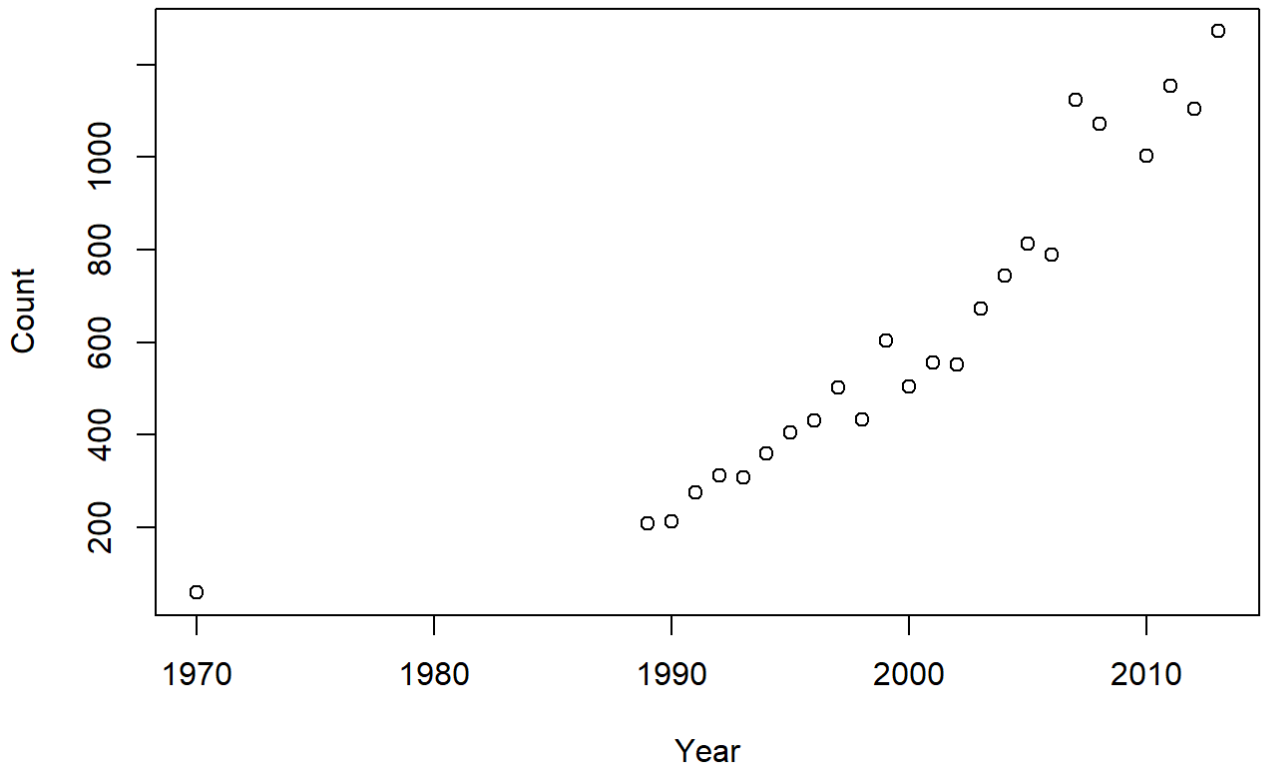
```
## [1] 1970 2013
```

```
quantile(Count)
```

```
##   0%  25%  50%  75% 100%
##   59  360  551  814 1272
```

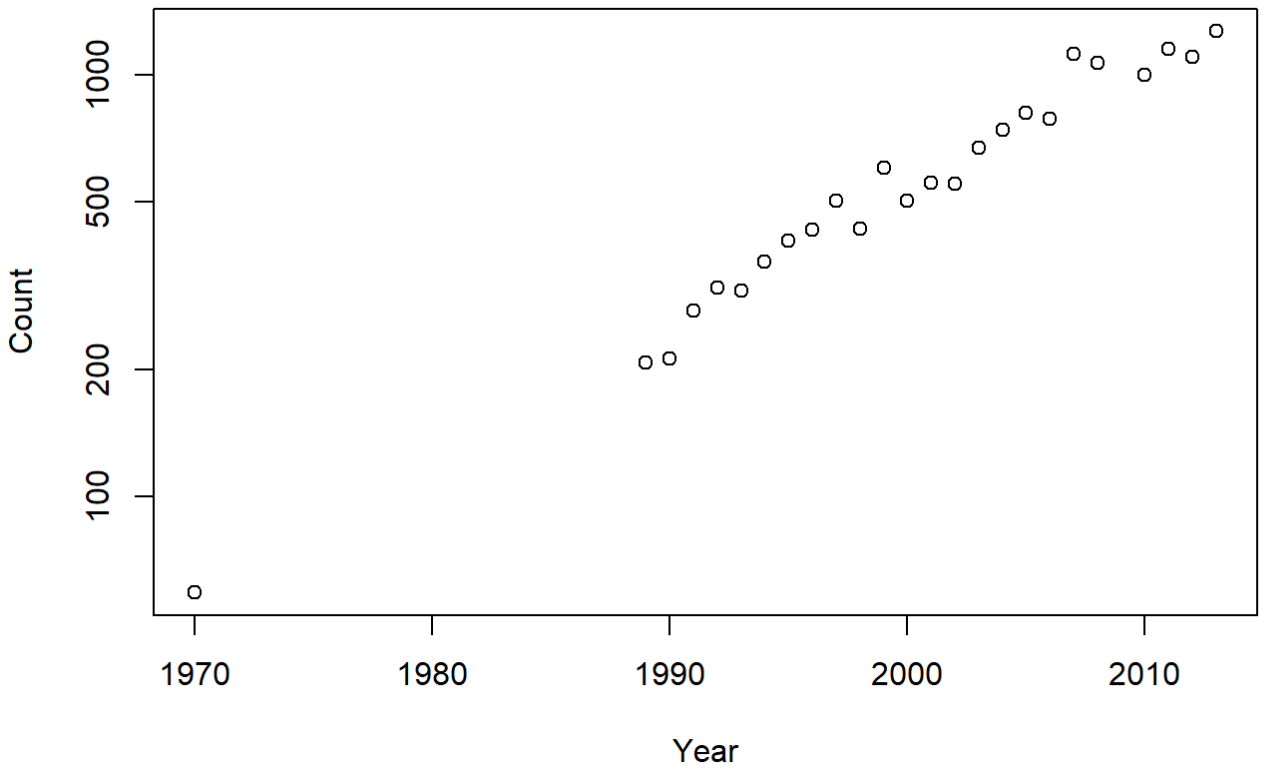
Always, always, always PLOT your data before modeling your data!

```
plot(Year, Count)
```



It is quite easy to make plots on a log scale, by the way, which can be useful:

```
plot(Year, Count, log = "y")
```



Linear Model

The `lm()` function produces a linear model. That means it finds the "best" values for a model that looks likes this:

$$Y_i = \alpha + \beta X_i + \epsilon_i$$

Where α is the intercept, β is the slope, and ϵ is the "residual" variation, i.e. a random variable: $\mathcal{N}(0, \sigma^2)$.

In R this model is fitted as follows:

```
Model1 <- lm(Count ~ Year)
```

Note, we take a "model fit" (which is a complicated thing) and put it into a new object with name `Model1`.

The output of `Model1` is just the estimated parameters for the intercept and the "Year effect" (i.e. slope of the curve).

```
Model1
```

```
##
## Call:
## lm(formula = Count ~ Year)
##
## Coefficients:
## (Intercept)         Year
##   -65751.54        33.19
```

There is a lot more information if you ask for a summary:

```
summary(Model1)
```

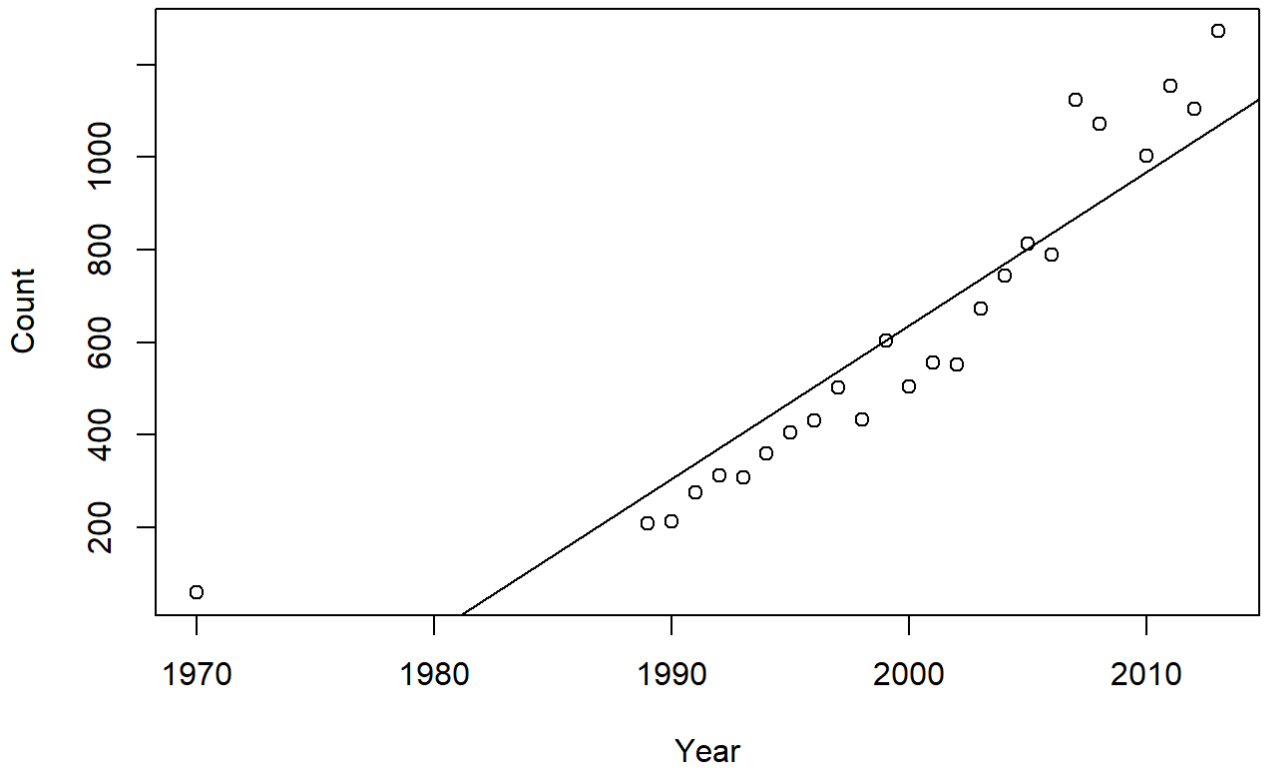
```
##
## Call:
## lm(formula = Count ~ Year)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -152.74  -78.18  -58.79   34.71  417.48
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -65751.54    6158.81  -10.68 2.19e-10 ***
## Year            33.19       3.08   10.78 1.83e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 142.4 on 23 degrees of freedom
## Multiple R-squared:  0.8347, Adjusted R-squared:  0.8275
## F-statistic: 116.1 on 1 and 23 DF,  p-value: 1.83e-10
```

Of importance to us is the `Estimate` (same as above) and the `Std. Error` which quantifies how precise our estimate is. Two standard errors is (roughly) the same as a **95% Confidence Interval**. Thus, the slope on this fit is 33.2 new sea otters / year ± 6.18. This is definitely bigger than 0! We can also see that because the *p*-value (under column `Pr(>|t|)`) is very very very very small.

Note, also that $R^2 = 0.835$. R^2 is between 0 and 1 and measures how tight the fit is, specifically it is the *proportion of the variance explained by the model*.

We can plot this linear model on our data:

```
plot(Year, Count)
abline(Model1)
```

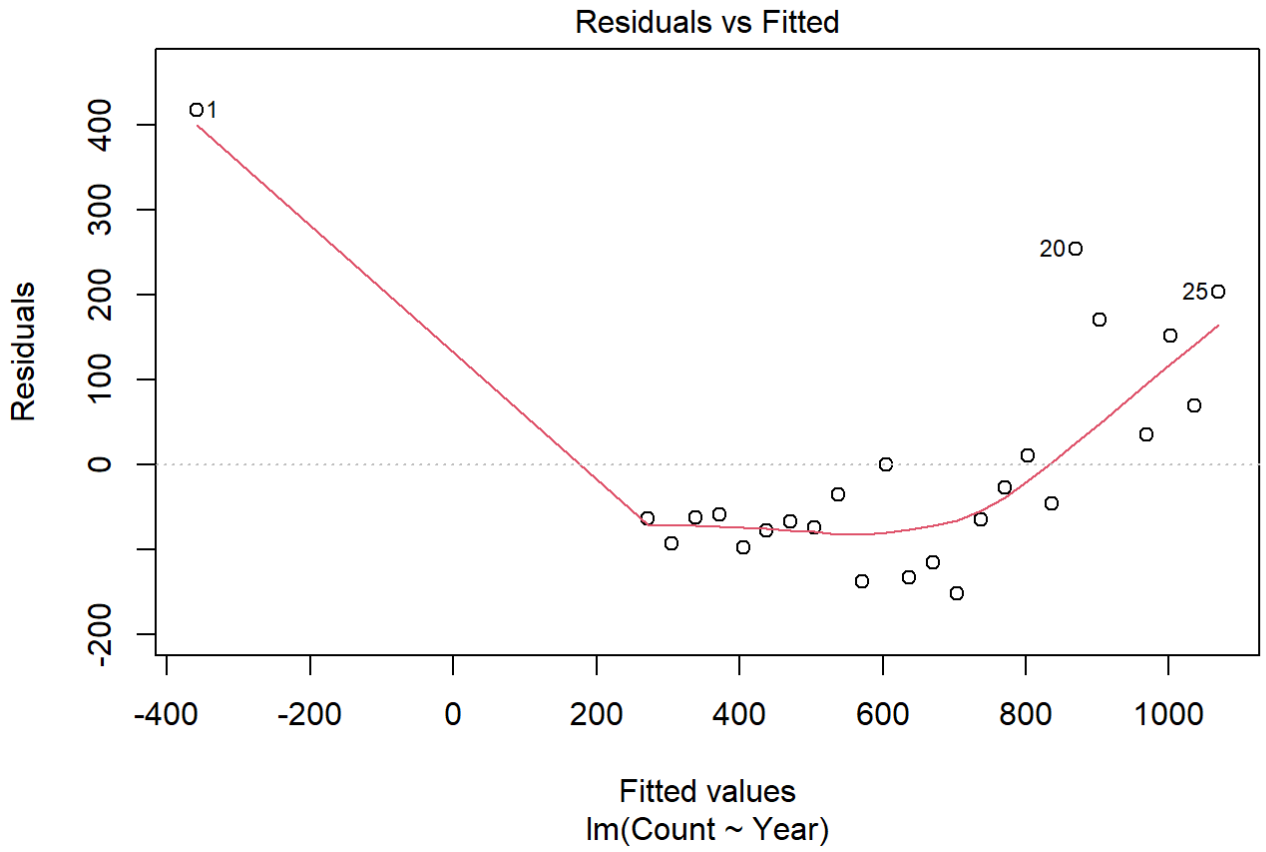


Note, `abline` is a function that adds an intercept-slope line to any existing plot.

There are some problems with this model. What are they?

You can see one issue if you plot the model residuals as follows:

```
plot(Model1,1)
```



Do those residuals look consistently normally distributed?

Log Transformation

We kind of knew the growth was not linear (otherwise this week's topic wouldn't have been called *Exponential Growth*). But how can we use a linear model to fit an exponential growth function?

With a simple transformation!

$$\log(Y_i) = \alpha + \beta X_i + \epsilon_i$$

How does this relate to an exponential growth model?

$$e^{\log(Y_i)} = e^{\alpha + \beta X_i + \epsilon_i}$$

$$Y_t = N_0 e^{\beta X_t + \epsilon_t}$$

We just replaced e^{α} with N_0, and i with t. But β is EXACTLY the growth rate r (and $e^{\beta} = \lambda$). The ϵ_i hanging on the end is a bit of stochasticity.

Anyways, doing this with our linear modeling tools is easy. We just wrap `Count` in a log.

```
Model2 <- lm(log(Count) ~ Year)
summary(Model2)
```

```
##
## Call:
## lm(formula = log(Count) ~ Year)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.191084 -0.062944 -0.005104  0.055518  0.231704
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.402e+02  4.732e+00  -29.63   <2e-16 ***
## Year         7.325e-02  2.367e-03   30.95   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1094 on 23 degrees of freedom
## Multiple R-squared:  0.9766, Adjusted R-squared:  0.9755
## F-statistic: 958.1 on 1 and 23 DF,  p-value: < 2.2e-16
```

Note how our statistics changed. Specifically: our β estimate (which is also r) is now 0.07325, which is a good estimate of the intrinsic growth rate. Even better, we know have a standard error around that estimate: 0.00237, which means our growth rate can be written as: $\hat{r} = (7.32 \pm 0.47) \times 10^{-2}$. We can convert that to an annual growth rate in R as well (*do NOT worry about understanding the following code*:

```
r.hat <- summary(Model2)$coefficients[2,1]
r.sd <- summary(Model2)$coefficients[2,2]
exp(c(r.hat, r.hat-2*r.sd, r.hat+2*r.sd))
```

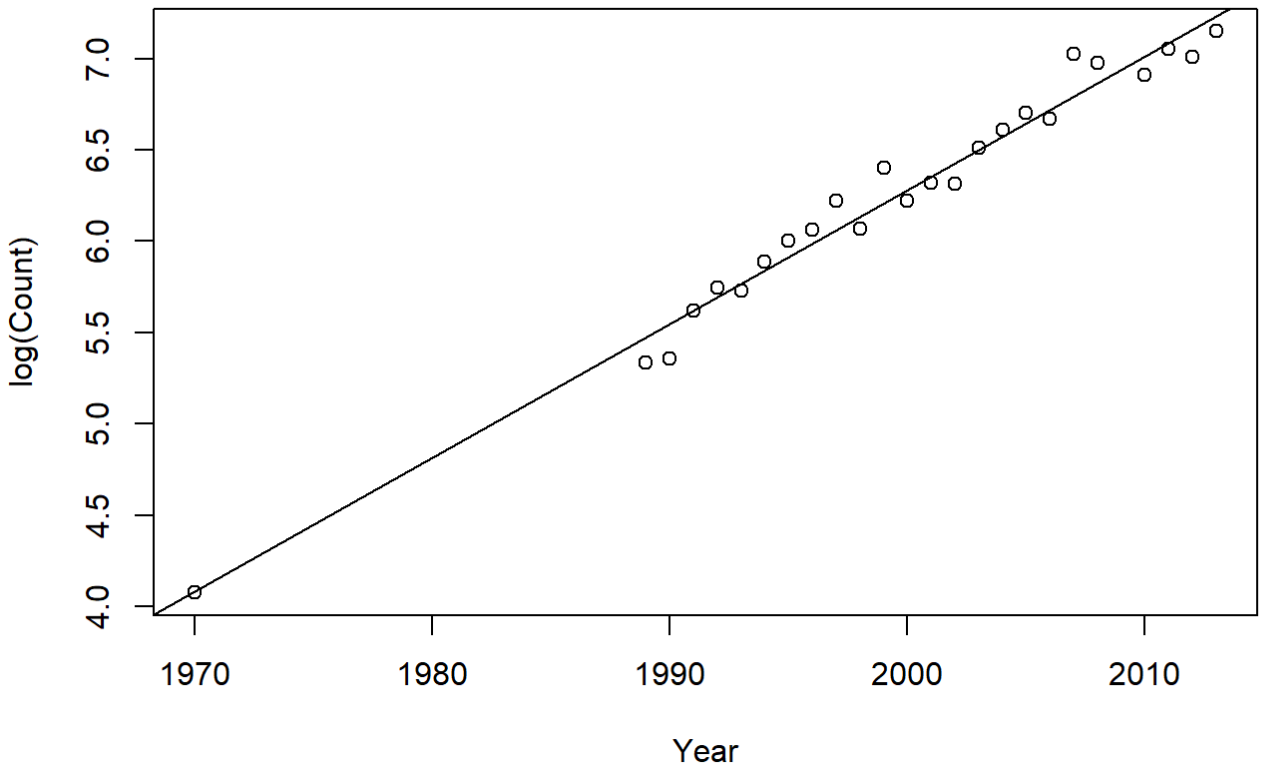
```
## [1] 1.076001 1.070920 1.081106
```

So our estimate of annual population growth is 7.6% (95% Confidence Interval 7.1%-8.1%)

Note how much right the R^2 value is! This is (by that measure also) a much better fit.

Let's visualize the fit:

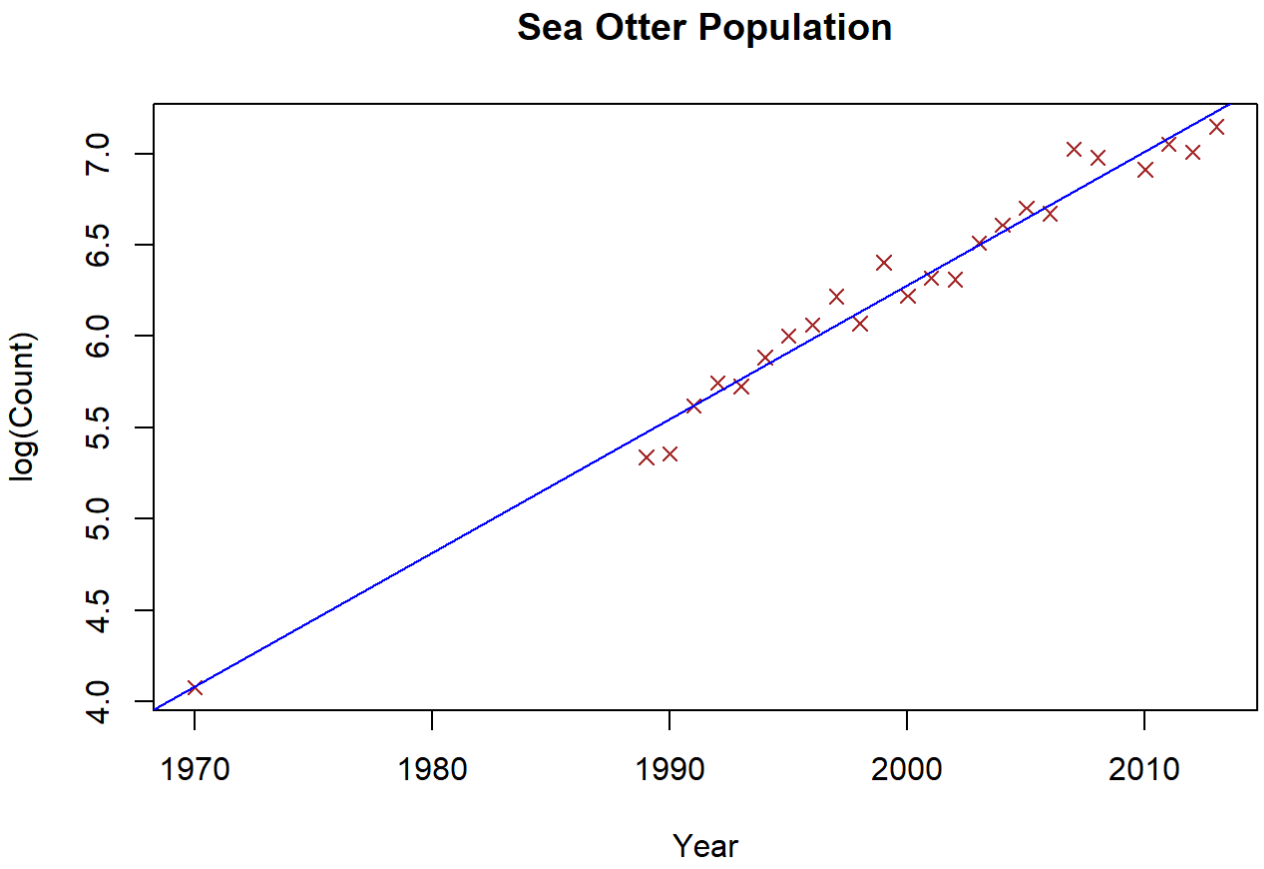
```
plot(Year, log(Count))
abline(Model2)
```



Much better!

Let’s make our final graph a little nicer

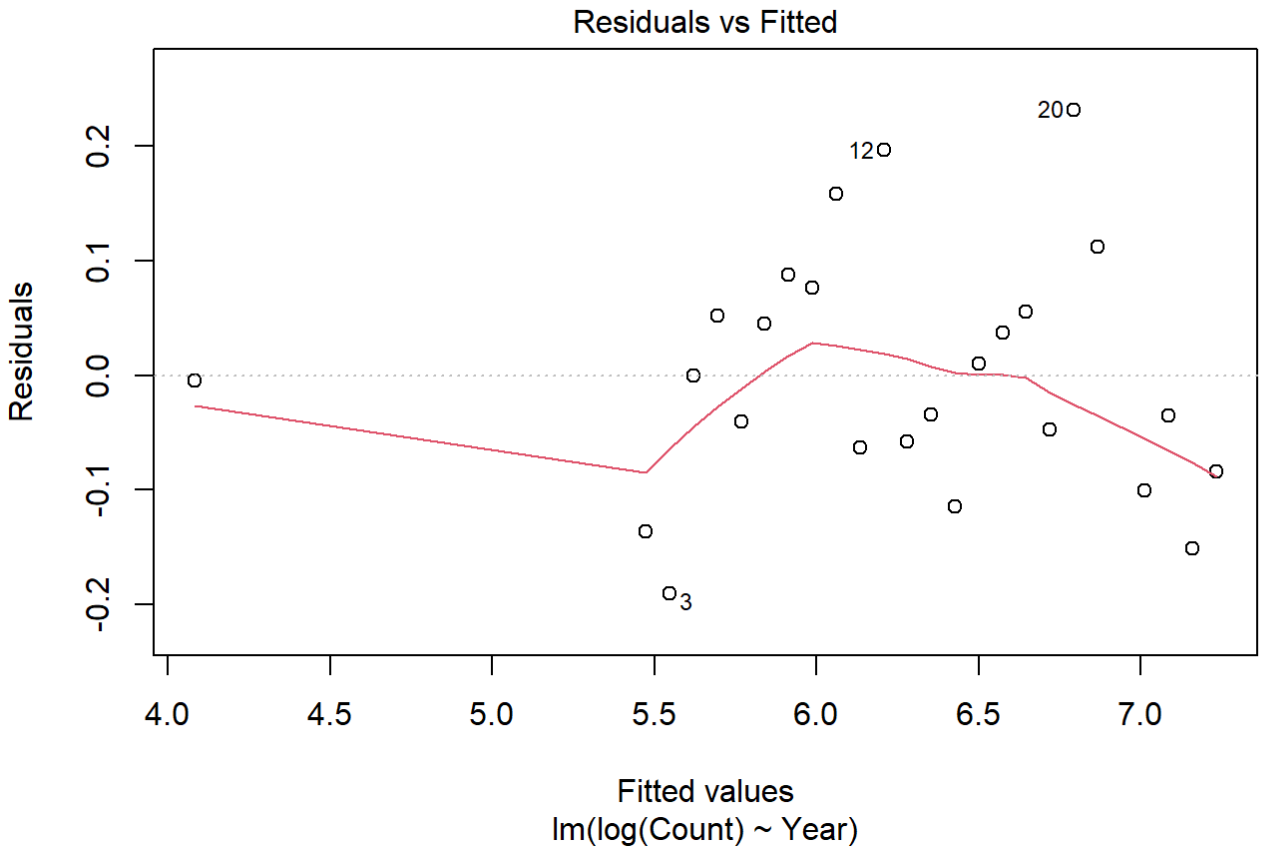
```
plot(Year, log(Count), main = 'Sea Otter Population', col = 'brown', pch = 4)
abline(Model2, col = "blue")
```



The commands we're using within the `plot` function include setting the title of the graph (`main =`), changing the color of the symbols and the line (`col =`), and changing the shape of the symbols (`pch =`). There also many more alterations you can make to get your graph perfect - but that can be a deep dark `R` abbit hole!

Check Your New Residuals

```
plot(Model2, 1)
```



These look better! (Though it takes practice to understand good vs. bad residuals.)

Extra packages (extra material)

It can be a bit fussy to plot the confidence interval around our prediction. Below, we include a little snippet of code which does just that, but you have to install a new “library” called `ggplot2`, which has a bunch of fancier graphing tools.

To install a library, you use the `install.packages()` function:

```
install.packages("ggplot2")
```

You only need to install a package once on your computer. After that, it will always be there.

Alternatively, you can use the `R` interface to install new packages. To do so:

> Click the `Packages` tab in the bottom right window of `Rstudio`

> Click `Install`

> Leave the defaults in place; in the blank “packages” line, type the name of the package you want. `RStudio` should automatically fill this in with suggestions. Use a comma to separate the names of multiple package if installing more than one simultaneously.

> Click `Install`

This will automatically put the correct code in the console for you.

From now on, whenever you need that package loaded you use the `library` function. You only have to do this once each time you open `R`, after that, a package stays in your library for the rest of your session.

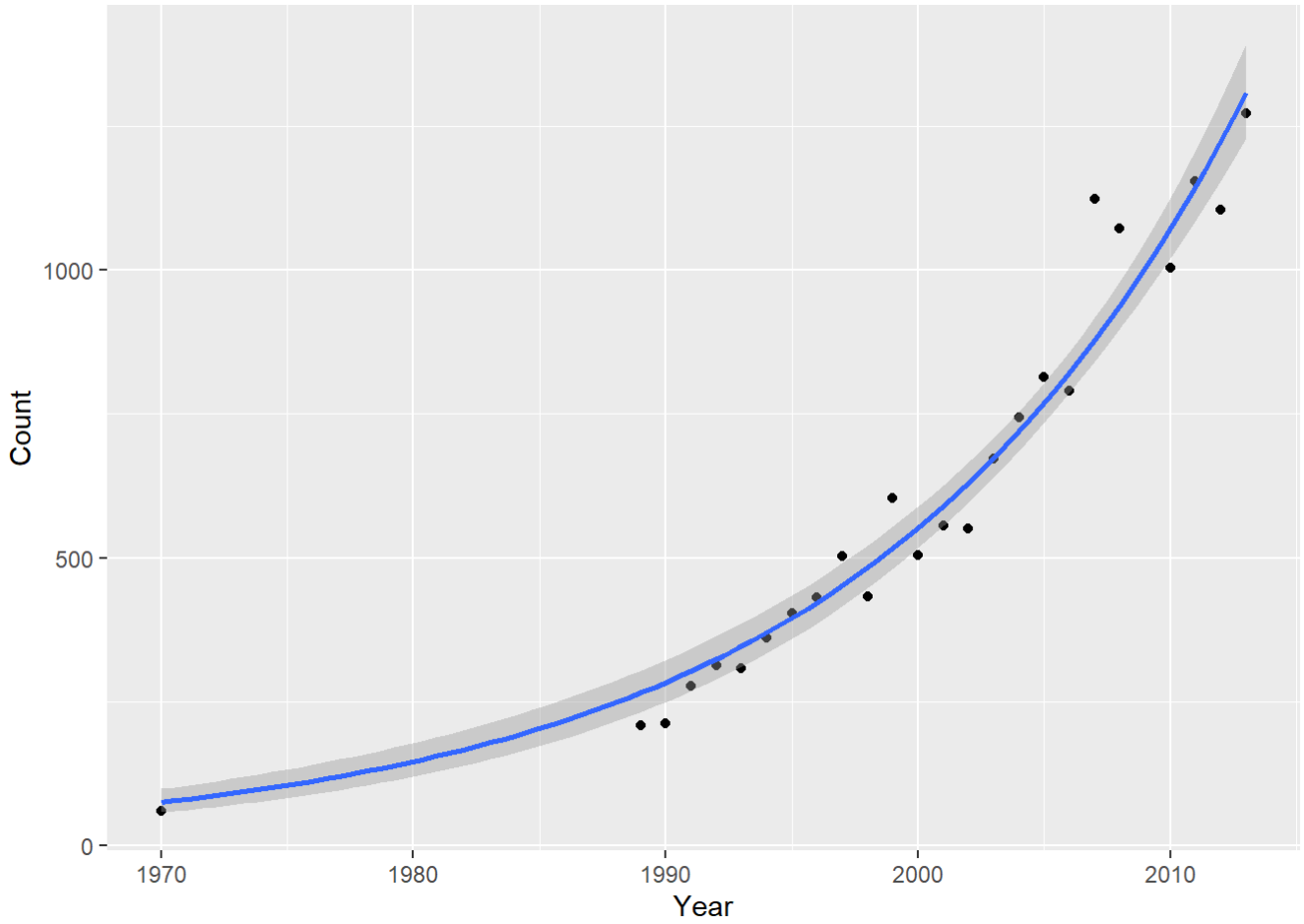
```
library(ggplot2)
```

If you receive prewritten code from someone and open the file in `RStudio`, the software will automatically identify packages called by the `library` funtion. If you do not have these packages installed, `Rstudio` will ask you if you’d like to install them, which can save you time. However, you will still need to run the `library` function on each package before you can use them.

Now - **finally** - you have access to the funky / fancy library of ggplotting functions. For example:

```
ggplot(SeaOtters, aes(Year, Count)) + geom_point() +
    stat_smooth(method = "glm",
                method.args = list(family = gaussian(link = "log")))
```

```
## `geom_smooth()` using formula 'y ~ x'
```



This really looks like a nice fit!