Module 5: Building Better R Skills

The purpose of this handout is to reinforce the data manipulation and graphing skills you have been developing in R. This module assumes you are using the `RxP.byTank` dataset that was created in the Module 3.

In the first 4 modules, you have learned a considerable number of functions including (but probably not limited to) the following:

Basic functions: length(); ls(); str(); rm(); c(); mean(); rep(); seq(); names(); head(); tail(); factor(); cbind(); round(); exp(); paste()

More advanced data manipulation commands: aggregate(); relevel(); summarise()

Graphing functions: plot(); hist(); lines(); qplot(); barplot2(); arrows(); axis(); legend(); abline()

Data analysis functions: t.test(); shapiro.test(); fitdistr(); AIC(); lm(); summary(); anova(); Anova(); glht(); cld(); predict()

That's a lot of new material to take in very quickly! Hooray! Now let's continue to work with some of these in ways that will reinforce the skills you have been developing. Below are a series of exercises to challenge you.

1 Your assignment!

The tadpoles in our experiment metamorphosed over a huge time period, from 38 days after being laid as an egg to 141 days after being laid (and that's on average! The last metamorph crawled out of the water 202 days after being laid!). We also measured lots of things when the froglets finally did crawl out of their tanks. There are tons of questions one can ask with a dataset like this.

Here, we will ask a new question. Did the amount of tail that froglets had remaining when they crawled out of the water affect how long it took them to resorb it into their body, and was that rate affected by Predator or Resource treatments?

Here is some background information for those of you not well-versed in amphibian biology. When a tadpole goes through metamorphosis and turns in to a frog, it absorbs the tail into the body. Essentially, they digest the tail and metabolize it in order to complete the process of metamorphosis, which involves major rearrangements of the digestive tract (they change from herbivore to carnivore), the mouthparts (same reason), they have limbs all of a sudden, and they so on. Being on land with a tail is also makes you very bad at hopping, so they are are quite vulnerable during this period of time. It is a good idea to complete the process as quickly as possile, but the whole thing naturally takes some time.

Here are the basic steps you will need to do to complete this assignment.

1. Check the normality of your continuous variables. If they are not normal, does log-transformation help?

2. Define the model. You have three predictor variables, one of which is continuous, so remember to code the continuous variable first (if not mentioned before, you should always include your continuous variable, also known as a covariate, first). If one of them is never significant, you can probably remove it from the model entirely and re-run it.

3. Plot the data. Make sure to use colors or symbols to differentiate any different treatments in your figure. Make sure the data are plotted back on the original axes (in case your model actually uses log-transformed data).

4. Use `predict()` to generate the regression lines and standard errors. You will most likely want to use `expand.grid()` to generate the blank dataframe. See below for information about `expand.grid()`.
5. Lastly, plot the figure including regressions for each predictor or combination of predictors with confidence intervals.

2 A brief word about `expand.grid()` and `predict()`

What does the `predict()` function actually do? It takes your model and feeds into a set or sets of values and calculates the predicted response variable value for you. In Module 4 we saw how to do the math for a simple linear regression. `predict()` just does that, but it can do it for much more complicated models that would be very difficult to code by hand, and it can calculate lots of things very quickly.

In Module 4, we made model `lm4` and used it to explore the math of the linear regression.

```
> lm4<-lm(log.SVL.final~log.Age.DPO, data=RxP.byTank)
> summary(lm4)

Call:
lm(formula = log.SVL.final ~ log.Age.DPO, data = RxP.byTank)

Residuals:
      Min        1Q    Median        3Q       Max
-0.182298 -0.049070  0.000516  0.038342  0.159057

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.44001    0.09211  37.345  < 2e-16 ***
log.Age.DPO  -0.11624    0.02232  -5.208 1.58e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.06244 on 76 degrees of freedom
Multiple R-squared:  0.263,  Adjusted R-squared:  0.2533
F-statistic: 27.12 on 1 and 76 DF,  p-value: 1.581e-06
```

Using this output, we saw that we could calculate the predicted log of SVL.final by doing the following math: `log.SVL.final = 3.44001 - 0.11624 * log.Age.DPO`. Then, we saw that we could put any value we want into the place of `log.Age.DPO` and predict the logged final SVL. For example, to calculate the predicted size of metamorphs emerging at 60, 80 or 100 days post-oviposition, we typed the following:

```
> exp(3.4401 - 0.11624 * log(60))
[1] 19.37868
> exp(3.4401 - 0.11624 * log(80))
[1] 18.74137
> exp(3.4401 - 0.11624 * log(100))
```

```
[1] 18.2615
```

Okay, so how does this relate to `predict()`? In this simple case, we can explore how predict does the exact same thing, but hopefully that will show how in more complex cases `predict()` is very useful.

`predict()` requires really only two things: 1) a model to predict from and 2) a set of values to feed into the model. Those values have to be in the form of a data frame. So, let's make a data frame of just 3 values, the logs of 60, 80 and 100.

```
> df1<-data.frame(log.Age.DPO = c(log(60),log(80),log(100)))
> df1
  log.Age.DPO
1    4.094345
2    4.382027
3    4.605170
```

Now, we can use our object `df1` to generate predicted values from the model.

```
> temp1<-predict(lm4, newdata=df1)
> temp1
       1        2        3
2.964061 2.930619 2.904680
> exp(temp1)
       1        2        3
19.37650 18.73924 18.25940
```

Here, we can see that the values generated by `predict()` are *essentially* the same as what we calculated by hand. The small differences between the two values are due to the fact that when we calculated by hand we stopped at the 4th or so decimal place.

Okay, so `predict()` does the math for us, essentially. If we make our data frame `df1` bigger, we can see what happens.

```
> df1<-data.frame(log.Age.DPO = log(35:145))
> temp1<-predict(lm4, newdata=df1)
> exp(temp1)
       1        2        3        4        5        6        7        8        9       10
20.62939 20.56194 20.49656 20.43311 20.37151 20.31164 20.25342 20.19677 20.14160 20.08785
      11       12       13       14       15       16       17       18       19       20
20.03544 19.98431 19.93442 19.88569 19.83808 19.79155 19.74604 19.70152 19.65794 19.61528
      21       22       23       24       25       26       27       28       29       30
19.57348 19.53253 19.49238 19.45301 19.41439 19.37650 19.33931 19.30278 19.26692 19.23168
      31       32       33       34       35       36       37       38       39       40
19.19705 19.16301 19.12954 19.09662 19.06424 19.03238 19.00103 18.97016 18.93977 18.90983
      41       42       43       44       45       46       47       48       49       50
18.88035 18.85130 18.82268 18.79447 18.76666 18.73924 18.71219 18.68552 18.65921 18.63326
      51       52       53       54       55       56       57       58       59       60
18.60764 18.58236 18.55740 18.53276 18.50844 18.48441 18.46069 18.43725 18.41409 18.39121
      61       62       63       64       65       66       67       68       69       70
18.36860 18.34626 18.32417 18.30234 18.28075 18.25940 18.23830 18.21742 18.19677 18.17635
```

```
      71       72       73       74       75       76       77       78       79       80
18.15614 18.13614 18.11636 18.09678 18.07740 18.05822 18.03923 18.02043 18.00182 17.98340
      81       82       83       84       85       86       87       88       89       90
17.96515 17.94708 17.92918 17.91145 17.89389 17.87649 17.85925 17.84217 17.82525 17.80848
      91       92       93       94       95       96       97       98       99      100
17.79186 17.77539 17.75906 17.74288 17.72683 17.71093 17.69516 17.67952 17.66402 17.64864
     101      102      103      104      105      106      107      108      109      110
17.63340 17.61828 17.60328 17.58840 17.57365 17.55901 17.54449 17.53008 17.51578 17.50160
     111
17.48753
```

So `predict()` calculated the predicted log.SVL.final for metamorphs emerging from 35 to 140 days after oviposition. What should we do with these numbers? The best thing to do with them is to put them in our data frame with our values from log.Age.DPO. Recall that if you assign a vector to a column that doesn't exist in a data frame, `R` will automatically make that column.

```
> df1$predicted<-exp(temp1)
> str(df1)
'data.frame': 111 obs. of  2 variables:
 $ log.Age.DPO: num  3.56 3.58 3.61 3.64 3.66 ...
 $ predicted  : num  20.6 20.6 20.5 20.4 20.4 ...
```

2.1 `expand.grid()` is your friend!

If we make a more complex model, such as `lm5` from Module 4, making the blank data frame to feed into `predict()` can be very cumbersome. This is where the `expand.grid()` function comes into play. `expand.grid()` is a function that just makes every possible combination of whatever variables you give it and spits them out in the format of a data frame. For example:

```
> expand.grid(c("a","b","c","d","e"), 1:5)
   Var1 Var2
1     a    1
2     b    1
3     c    1
4     d    1
5     e    1
6     a    2
7     b    2
8     c    2
9     d    2
10    e    2
11    a    3
12    b    3
13    c    3
14    d    3
15    e    3
16    a    4
```

```
17    b    4
18    c    4
19    d    4
20    e    4
21    a    5
22    b    5
23    c    5
24    d    5
25    e    5
```

In the code above, we gave the function 5 letters and 5 numbers and it made all possible combinations of them. You can make these sorts of things as complex as you want. You can also provide column names.

```
> expand.grid("Letters"=c("a","b","c"), "Numbers"=1:2, "More_letters"=c("x","y"),
  "Folks"=c("George","Bob"))
   Letters Numbers More_letters  Folks
1        a       1            x George
2        b       1            x George
3        c       1            x George
4        a       2            x George
5        b       2            x George
6        c       2            x George
7        a       1            y George
8        b       1            y George
9        c       1            y George
10       a       2            y George
11       b       2            y George
12       c       2            y George
13       a       1            x    Bob
14       b       1            x    Bob
15       c       1            x    Bob
16       a       2            x    Bob
17       b       2            x    Bob
18       c       2            x    Bob
19       a       1            y    Bob
20       b       1            y    Bob
21       c       1            y    Bob
22       a       2            y    Bob
23       b       2            y    Bob
24       c       2            y    Bob
```

Okay, so in the case of `lm5` we can make a data frame that is our range of `log.Age.DPO` and contains all of the levels of `Pred`.

```
> lm5<-lm(log.SVL.final~log.Age.DPO*Pred, data=RxP.byTank)
> predicted.data<-expand.grid("log.Age.DPO"=log(35:140), "Pred"=levels(RxP.byTank$Pred))
```

```
> head(predicted.data)
  log.Age.DPO Pred
1    3.555348    C
2    3.583519    C
3    3.610918    C
4    3.637586    C
5    3.663562    C
6    3.688879    C
...
```

We can then use this data frame in conjunction with `predict()` to get all the values necessary for plotting our ANCOVA regression lines, especially if we want to include the confidence intervals for our lines.

```
> temp2<-predict(lm5, newdata=predicted.data, se.fit=T)
> str(temp2)
List of 4
 $ fit           : Named num [1:318] 2.94 2.94 2.94 2.94 2.94 ...
  ..- attr(*, "names")= chr [1:318] "1" "2" "3" "4" ...
 $ se.fit        : Named num [1:318] 0.0227 0.022 0.0213 0.0206 0.0199 ...
  ..- attr(*, "names")= chr [1:318] "1" "2" "3" "4" ...
 $ df            : int 72
 $ residual.scale: num 0.0523
```

Lastly, we can easily move the `fit` and `se.fit` vectors into our `predicted.data` object.

```
> predicted.data$fit<-temp2$fit
> predicted.data$se.fit<-temp2$se.fit
> head(predicted.data)
  log.Age.DPO Pred      fit     se.fit
1    3.555348    C 2.943380 0.02274332
2    3.583519    C 2.942260 0.02200193
3    3.610918    C 2.941172 0.02128596
4    3.637586    C 2.940112 0.02059441
5    3.663562    C 2.939079 0.01992641
6    3.688879    C 2.938073 0.01928117
```

Once we have these data all together, it is straightforward to subset the data by Predator treatment and then plot each line and CI.

```
> plot(log.SVL.final~log.Age.DPO, col=Pred, data=RxP.byTank, pch=20)
> #Subset the Control data and plot htem
> tempC<-predicted.data[predicted.data$Pred=="C",]
> lines(x=tempC$log.Age.DPO, y=tempC$fit, col=1, lwd=2)
> lines(x=tempC$log.Age.DPO, y=tempC$fit+tempC$se.fit, col=1, lwd=1, lty=2)
> lines(x=tempC$log.Age.DPO, y=tempC$fit-tempC$se.fit, col=1, lwd=1, lty=2)
> #Subset the Nonlethal data and plot htem
> tempNL<-predicted.data[predicted.data$Pred=="NL",]
```

```
> lines(x=tempNL$log.Age.DPO, y=tempNL$fit, col=2, lwd=2)
> lines(x=tempNL$log.Age.DPO, y=tempNL$fit+tempNL$se.fit, col=2, lwd=1, lty=2)
> lines(x=tempNL$log.Age.DPO, y=tempNL$fit-tempNL$se.fit, col=2, lwd=1, lty=2)
> #Subset the Lethal data and plot htem
> tempL<-predicted.data[predicted.data$Pred=="L",]
> lines(x=tempL$log.Age.DPO, y=tempL$fit, col=3, lwd=2)
> lines(x=tempL$log.Age.DPO, y=tempL$fit+tempL$se.fit, col=3, lwd=1, lty=2)
> lines(x=tempL$log.Age.DPO, y=tempL$fit-tempL$se.fit, col=3, lwd=1, lty=2)
> legend(x="topright", legend=c("Control","Nonlethal","Lethal"), text.col=1:3, cex=1.6,
  bty="n")
```

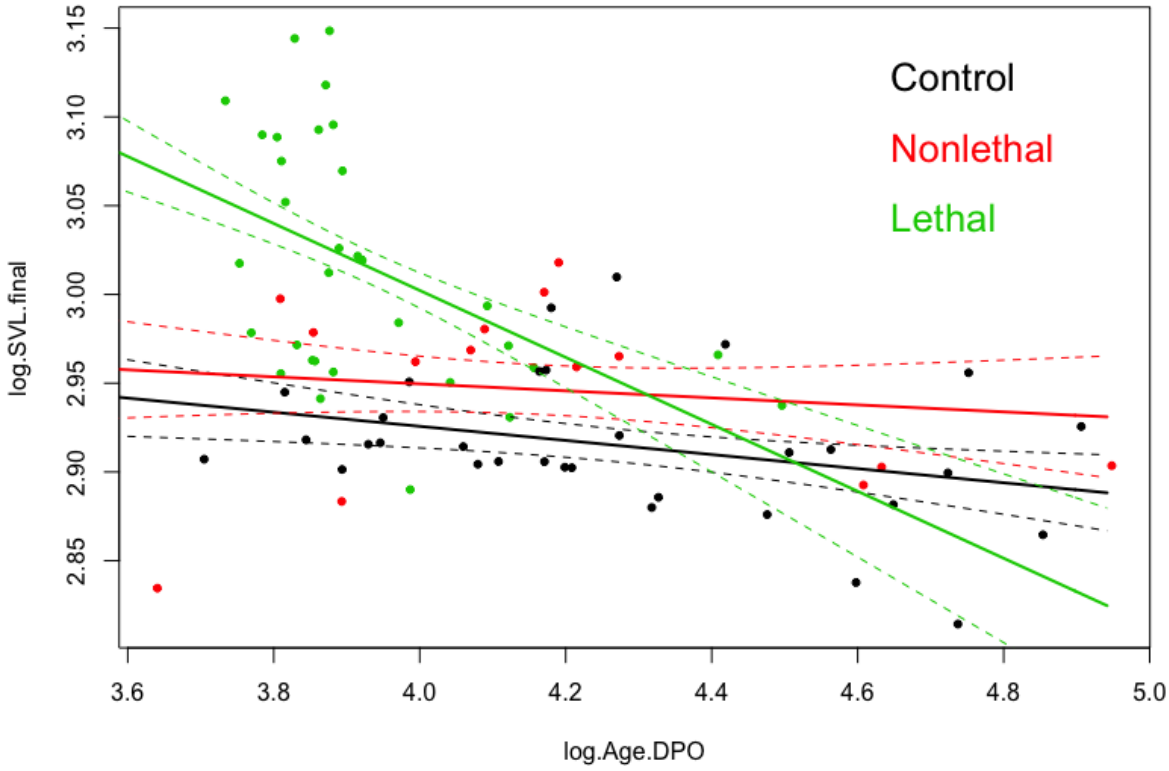

Figure 1: A scatterplot of the ANCOVA in lm5, plotted with regression lines and confidence intervals.