Exploratory Data Analysis and Plotting

The purpose of this handout is to introduce you to working with and manipulating data in R, as well as how you can begin to create figures from the ground up.

1 Importing data into RStudio

In order to work with data in R, you must first import or read the data into the program. This is most easily accomplished by saving your data in .csv or .txt formats.

For this handout, you will use the data in the `tadpole.csv` file available for download on the Stats at Vassar webpage.

These data are from a hypothetical 3 X 2 factorial experiment investigating potential interacting effects of predation risk and resources on tadpole growth.

- Predator treatment (low, medium or high density)
- Food treatment (control or protein supplemented)
- N = 300 (50 per treatment combination)
- Size measured once during larval period
- Measured number of spots on tadpole tail
- At metamorphosis, recorded sex and age

In the RStudio file management window (lower right window), go to the Files tab and click on *Upload*. Click *Choose File* and find & select the 'tadpole.csv' file on your desktop (or elsewhere on your computer). You should see the file appear in the file management window.

****Check out the video on importing data into R on the StatsAtVassar webpage****

Once the data are in RStudio, you must read them into the active workspace with the command `read.csv`. Other functions exist for other types of files, such as a .txt file, but you cannot use a .xls or .xlsx file. If the data are in your working directory, which by default is your home directory on the server, you can simply do the following:

```
> tadpole<-read.csv("tadpole.csv")
```

If your data are not in your working directory, you will need to specify where to find the file on the server. Note that there are two files available to download on the Stats at Vassar webpage. The file `tadpole.csv` is what you should be using now and contains several errors (intentially!) which you will fix. The file `tadpole_clean.csv` is the file that you will have produced by the end of this handout.

****Check out the video on reading your datafile into R on the StatsAtVassar webpage****

2 A General Note About Working In R, the Vassar RStudio web-server and Using this Tutorial

You are highly encouraged to type code into the script window (upper left corner) instead of directly into the Console (lower right window). A script file is a simple text file that allows you to conveniently edit commands and then run them when you have them typed in and ready. It also allows you to save your work so you can go back to it later, and to make notes to yourself using the pound or hashtag symbol #.

To open a new script window, click on the File menu, then New File, then R Script. Once you have typed in a line of code and you want to execute it, you can either click on the "Run" button in the upper right corner of the script window or just hit Command-Enter on a Mac or Control-R (I think) on a PC. Using the script window is advantageous for many reasons. For one, you can save your script file and come back to it later (click on the icon of the little disk or use the file menu to save it). This allows you to create different script files for different uses (e.g., analyzing different datasets). Secondly, you can highlight multiple lines of text and execute them all at once. For example, by the end of this document you will have made a nice looking barplot. By just changing a few items, you can then highlight and rerun all that code and produce your new figure instantly.

As mentioned above, *you are highly encouraged to type in your commands* instead of cutting and pasting them from this document. Not only will it lead to far fewer errors, but it will get in you the habit of typing in what you need to do. When you execute commands in the console window, you can scroll through previously executed commands by hitting the up arrow.

Lastly, you are encouraged to use the Vassar Rstudio server. It only works if you are on the campus network or are using the VPN from off-campus. One advantage is that your work is automatically saved so that the next time you login everything is exactly where you left it. Secondly, you can work on your analyses using a lab computer then return to your dorm and use your personal computer or work in a different lab and everything will be exactly the same when you login.

3 Data Exploration and Error Checking

The first step whenever you import a dataset is to check it over for errors. Questions you should ask yourself include:

- Did the data import correctly?
- Are the column names correct?
- Are the types of data appropriate? (e.g., factor vs numerical)
- Are the numbers of columns and rows appropriate?
- Are there typos in the data?

If, for example, a column that is supossed to be numerical shows up as a factor, that likely indicates a typo where you accidentally have text in place of a number. Similarly, if you have a factor that should have 3 categories, but imports with 4, you likely have a typo (e.g., "predator" vs "predtaor"), and the misspelled version is showing up as a separate category. These sorts of mistakes are very common.

3.1 Data structure

Begin by examining the structure of the data frame.

```
> str(tadpole)
'data.frame':  300 obs. of  8 variables:
 $ Ind     : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Pond    : int  3 5 5 2 4 5 4 4 2 2 ...
 $ Food    : Factor w/ 2 levels "Cont","Prot": 2 1 2 2 2 2 1 1 2 2 ...
 $ Pred    : Factor w/ 3 levels "High","Low","Med": 3 1 2 2 2 3 2 3 3 1 ...
 $ Size    : num  15.2 12.7 18.9 19.6 19.5 ...
 $ Spots   : Factor w/ 128 levels "100","101","102",..: 104 39 64 67 26 94 67 87 ...
 $ MetSex  : int  0 0 0 0 0 0 1 1 0 0 ...
 $ AgeAtMet: int  56 50 79 87 80 47 47 77 49 43 ...
```

We can see that our dataset has a 300 observations of 8 different variables, some of which are integers, some are factors and some are numerical. Things to notice:

1. The tadpoles came originally from 5 different ponds. Currently these ponds are listed 1-5 and are considered numeric. Perhaps they should be factors? What would be the rationale for treating the variable one way or another?
2. We have two categorical predictors (factors): Food level and Predator treatment.
3. We have 4 response variables:
    - size at metamorphosis (length in mm)
    - number of spots on the tail. *Notice that although the values appear to be numbers, the variable is coded as a factor with 128 levels!*
    - sex at metamorphosis (coded as 0 for female and 1 for male)
    - age at metamorphosis (days from hatching to metamorphosis)

3.2 Visually looking for errors

Begin by plotting the data to check for errors. The default `plot` function will create a simple graphic based on the type of data you provide it.

```
> plot(tadpole$Size)
> plot(tadpole$Spots)
> plot(tadpole$MetSex)
> plot(tadpole$AgeAtMet)
```

What did that tell you?

1. We appear to have two errors in the Size variable.
2. Plotting Spots was not useful because it is incorrectly considered a factor.
3. Plotting MetSex by itself was not useful because all the values are 0's or 1's.
4. Age at metamorphosis does not have any obvious errors and varies from about 30 to 100 days after hatching.

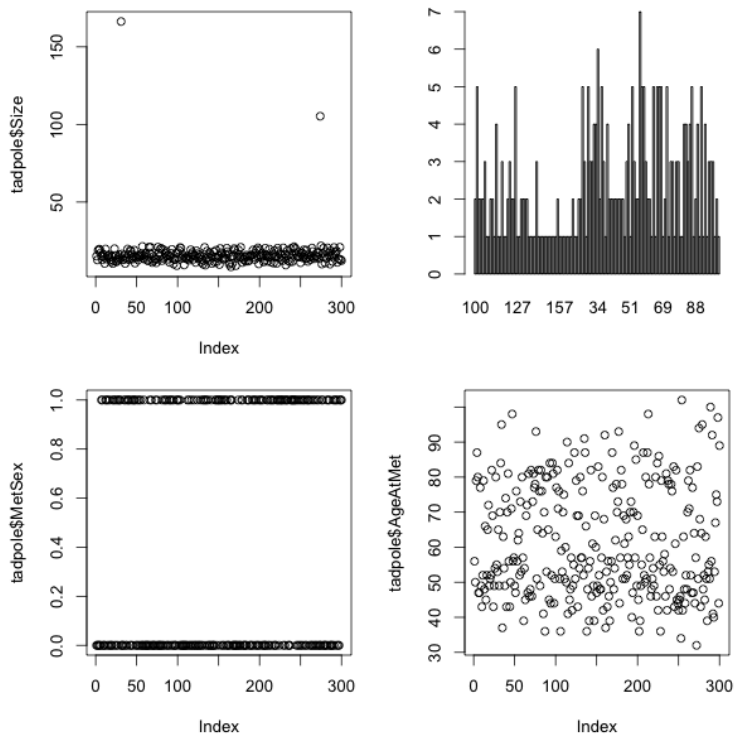

Figure 1: Plots of various variables from the uncorrected tadpole dataset

3.3 Fixing mistakes in the data

How do we go about fixing mistakes that we have found in the data file? One option would be to go back to your original .csv file or use Microsoft Excel (or whatever program you use to organize and input your data) and change it there. You should certainly do that! But in the short term, we want to fix the problems we have here and now. Plus, these exercises will help you practice using R.

We know the values for Size are generally pretty small and the two mistakes are really large. Thus, one way to identify where in the data file those mistakes are located is to search for any really big values of Size. We can do this using the []'s. One possibility would be to ask R to tell us any values larger than, say, 50. We picked 50 because when we plotted `tadpole$Size`,it was obvious that the none of the values that seem correct are greater than 50, whereas the two incorrect values are much larger than 50.

****Check out the videos on subsetting objects on the StatsAtVassar webpage****

```
> tadpole[tadpole$Size>50,]
    Ind Pond Food Pred  Size Spots MetSex AgeAtMet
31   31    4 Prot  Med 166.3    74      0       49
274 274    3 Cont High 105.3    80      0       51
```

- Remember that []'s let us access specific cells within a data frame, and coordinates of cells are

`[row,column]`.

- The line of code above said to look within the data frame `tadpole` and return any row where `tadpole$Size` was greater than 50, and to return all columns (since the part after the comma is left blank).

Here we can see our two mistakes, on lines 31 and 274. It looks like they were just typos, instead of 16.63 and 10.53, they became 166.3 and 105.3. We can easily change them, once again using the []'s. We know the rows they are in (31 and 274) and we can easily count that they are in the 5th column. Before we correct the values, we should double check we are in the correct spot.

```
> tadpole[31,5]
[1] 166.3
> tadpole[274,5]
[1] 105.3
```

Indeed, we are in the correct spot. We can write over the incorrect values with the correct ones using the <- tool.

```
> tadpole[31,5]<-16.63
> tadpole[274,5]<-10.53
```

Notice that R did not provide any warning message when we wrote over our data. *It is very easy to accidentally write over your data!* Lastly, we should confirm that it worked.

```
> tadpole[31,5]
[1] 16.63
> tadpole[274,5]
[1] 10.53
> plot(tadpole$Size)
```

Similarly, we noticed that there was a mistake in the Spots variable. It was being coded as a factor when it should be numeric (the number of spots on the tail). How do we go about finding where the mistake is? The command `levels` will tell us all the categories of Spots.

```
> levels(tadpole$Spots)
  [1] "100" "101" "102" "103" "104" "105" "106" "108" "109" "110" "111"
 [12] "112" "113" "114" "115" "117" "118" "119" "121" "123" "125" "126"
 [23] "127" "128" "129" "13"  "130" "131" "132" "136" "137" "139" "14"
 [34] "140" "141" "142" "143" "144" "146" "149" "15"  "152" "153" "155"
 [45] "157" "158" "16"  "161" "165" "17"  "179" "18"  "19"  "203" "21"
 [56] "23"  "25"  "26"  "27"  "28"  "30"  "31"  "32"  "33"  "34"  "35"
 [67] "36"  "37"  "38"  "39"  "40"  "41"  "42"  "43"  "44"  "45"  "46"
 [78] "47"  "48"  "49"  "50"  "51"  "52"  "53"  "54"  "55"  "56"  "57"
 [89] "58"  "60"  "61"  "62"  "63"  "64"  "65"  "66"  "67"  "68"  "69"
[100] "70"  "72"  "73"  "74"  "76"  "77"  "78"  "79"  "80"  "81"  "82"
[111] "83"  "84"  "85"  "86"  "87"  "88"  "89"  "90"  "91"  "92"  "93"
[122] "94"  "95"  "96"  "97"  "98"  "99"  "b"
>
```

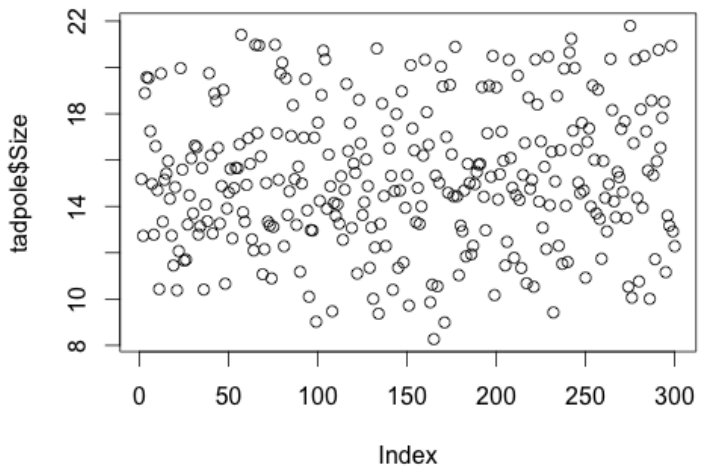

Figure 2: Scatterplot of the sizes of tadpoles at metamorphosis

A quick scan of the categories reveals that there is a category listed as `"b"`, which is clearly incorrect. How do we find where it is? Once again we can use the `[]`'s to search for any value of Spots which equals `"b"`.

```
> tadpole[tadpole$Spots=="b",]
   Ind Pond Food Pred  Size Spots MetSex AgeAtMet
91  91    3 Prot High 14.99     b      1       65
```

Since we do not know what the value should be, we should assign this cell to be `NA`. `R` will ignore the cell in all analyses.

```
> tadpole[91,6]<-NA
```

Now we have to convert Spots back into a numeric variable. Since Spots is a factor, if we convert it directly to a number it will cause a problem because the numeric values will be assigned to the factors in the *order* in which they were sorted, not based on the actual numbers present. Thus, we must first convert from a factor to a character, then from a character to a number. We will use the `as.character` and `as.numeric` commands.

```
> tadpole$Spots<-as.numeric(as.character(tadpole$Spots))
```

- Notice that we nested one function inside another! You can build as many functions together as you want.

We can verify this worked by looking at the structure of the data frame once again.

```
> str(tadpole)
'data.frame':  300 obs. of  8 variables:
 $ Ind     : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Pond    : int  3 5 5 2 4 5 4 4 2 2 ...
 $ Food    : Factor w/ 2 levels "Cont","Prot": 2 1 2 2 2 2 1 1 2 2 ...
 $ Pred    : Factor w/ 3 levels "High","Low","Med": 3 1 2 2 2 3 2 3 3 1 ...
 $ Size    : num  15.2 12.7 18.9 19.6 19.5 ...
 $ Spots   : num  76 146 33 36 13 64 36 56 67 112 ...
 $ MetSex  : int  0 0 0 0 0 0 1 1 0 0 ...
 $ AgeAtMet: int  56 50 79 87 80 47 47 77 49 43 ...
```

3.4 Further data exploration

Plotting data by itself can be useful, for example if we want to check for outliers or typos (like in the case of the Size variable). However, it is often more useful to plot response data against an explanatory variable. For example, maybe we want to get a sense of the possible sources of variation in the Age at metamorphosis data.

```
> plot(Size~Food, data=tadpole)
> plot(Spots~Pred, data=tadpole)
> plot(AgeAtMet~Food, data=tadpole)
> plot(Spots~Size, data=tadpole)
```

- Notice that we have introduced a new syntax. We can use the ~ to denote a relationship between two vectors, usually written as `response~predictor`.
- This structure will be used later for defining statistical models and can be expanded to encorporate multiple predictors. e.g., `response~predictor1 + predictor2 + etc.`.

4 Summarizing Your Data

There are many tools in R to help you summarize your data efficiently. Two important functions to know are `tapply()` and `aggregate()`.

`tapply()` is a simple function for calculating the mean or standard deviation of a group of data. The name of the function is short for 'table apply', as you are applying a function to a table of data. For example, if you want to calculate the average size of tadpoles in each of the predator treatments, you can type:

```
> tapply(tadpole$Size, tadpole$Pred, mean)
   High     Low     Med
13.1114 17.4835 14.9734
```

`tapply()` requires three arguments: 1) the data you are summarizing, 2) the category you want to summarize the data by, and 3) the function to apply to those data.

`aggregate()` is a similar function but the output is slightly different. It is better for summarizing data across multiple groups. It takes the same three arguments as `tapply()` but the second argument

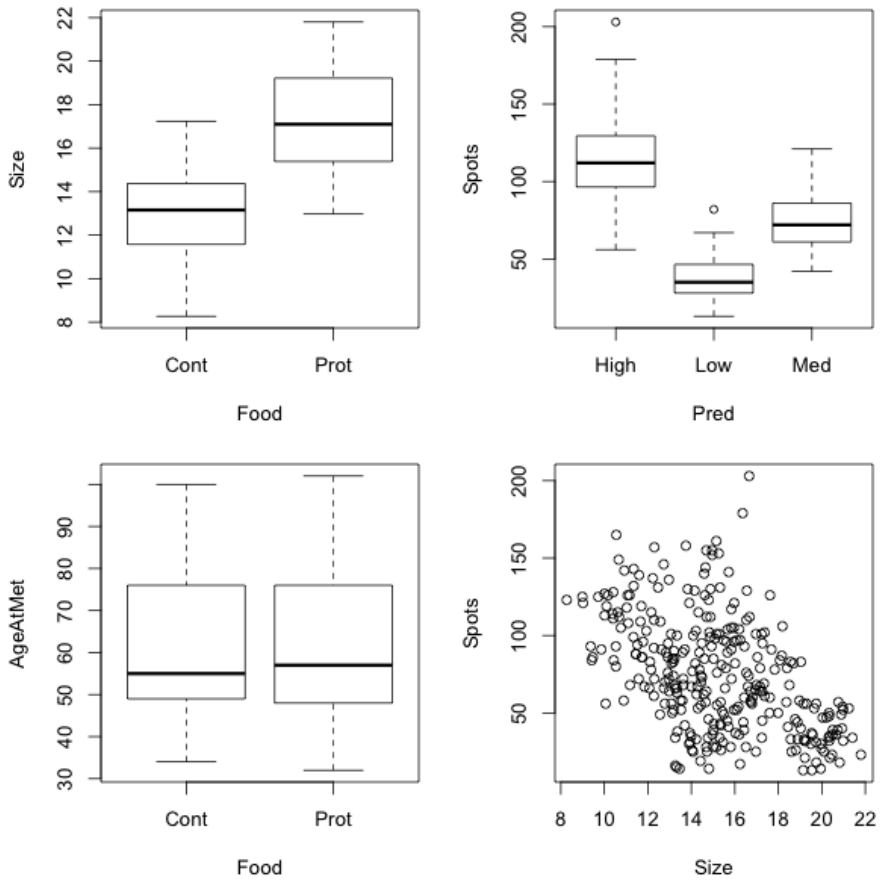

Figure 3: Various basic plots of response variables

is in the form of a list, which allows you to easily include multiple factors. Another advantage of `aggregate()` is that the output is a data frame, which makes it easy to then use the output for plotting figures or other purposes. For example, if you want to calculate the average tadpole size across all combinations of Food and Predator treatments, you can type:

```
> aggregate(tadpole$Size, list(tadpole$Pred, tadpole$Food), mean)
  Group.1 Group.2       x
1    High    Cont 11.0460
2     Low    Cont 15.0274
3     Med    Cont 12.9622
4    High    Prot 15.1768
5     Low    Prot 19.9396
6     Med    Prot 16.9846
```

4.1 Reordering Your Data

We are going to use `aggregate()` to create a bar graph of mean tadpole size. However, you might have noticed that R orders the levels of a factor alphabetically, so that "High" comes before "Low", which comes before "Med". For plotting purposes, it would probably make more sense to have them ordered logically (Low, Medium, High, or vice-versa). To see the current ordering of the factor levels, type:

```
> levels(tadpole$Pred)
[1] "High" "Low"  "Med"
```

As with most things in R, there multiple ways to reorder the factor levels. Here is one. We can use the function `factor()` to recode the factor all at once. `factor()` is a generic function which could be used, for example, to turn a vector of text or of 0's and 1's into factor levels. Here, we are essentially writing over the current factor with a new factor that is the same, but has a different order for the factor levels.

```
> tadpole$Pred<-factor(tadpole$Pred, c("Low","Med","High"))
```

4.2 Calculating Means and Standard Errors

Now that our Pred factor is in the order we want, let's use `aggregate()` to make two new objects, for the means and standard errors of the data. There is no built in function to calculate the standard error in R, so we will have to calculate it ourselves. For this exercise, we will use the definition: standard deviation/square root of N.

```
#Calculate the average length of tadpoles
> tadpole.mean<-aggregate(tadpole$Size, list(tadpole$Pred, tadpole$Food), mean)

#Calculate the standard deviation of tadpoles
> tadpole.SD<-aggregate(tadpole$Size, list(tadpole$Pred, tadpole$Food), sd)

#Calculate the sample size for each group of tadpoles
> tadpole.N<-aggregate(tadpole$Size, list(tadpole$Pred, tadpole$Food), length)

#Make a new variable which will become the standard error
> tadpole.SE<-tadpole.SD

#Since we want to leave the first two columns alone, which are the factor
#groupings, we overwrite the third column with the third column divided
#by the third column of sample size object
> tadpole.SE[,3]<-tadpole.SE[,3]/sqrt(tadpole.N[,3])

#Add column headings to make it nice and pretty
> names(tadpole.mean)<-c("Pred","Food","mean")
> names(tadpole.SE)<-c("Pred","Food","SE")

> tadpole.mean
  Pred Food    mean
```

```
1  Low Cont 15.0274
2  Med Cont 12.9622
3 High Cont 11.0460
4  Low Prot 19.9396
5  Med Prot 16.9846
6 High Prot 15.1768

> tadpole.SE
  Pred Food        SE
1  Low Cont 0.1526594
2  Med Cont 0.1279739
3 High Cont 0.1729247
4  Low Prot 0.1182778
5  Med Prot 0.1531887
6 High Prot 0.1288833
```

5 Plotting Your Data

That probably seemed like a lot of trouble just to calculate the means and SE's, but now that we have those values we can create a very nice looking bar graph. The simplest function to make a bar graph is `barplot()`. There is a slightly better version of the function in the `gplots` package, aptly named `barplot2()`. Either way, there are many, many arguments that can be passed to function, and these can be viewed in the help file. In order to use `barplot2()` you will need to go the Packages tab in the lower right window of RStudio and click on the package name. Alternatively, you can load the library by typing the following.

```
> library(gplots)
```

To see why using `barplot2()` will be useful, let's start by simply plotting the bars in their most basic form.

```
> barplot(tadpole.mean[,3])
```

We have passed to `barplot()` the column of mean tadpole sizes, which have been plotted in order from top to bottom. However, this figure by itself is quite boring. It has no axes titles, color, error bars, etc. In addition, the spacing is not very logical.

We can add a label to the y-axis with the argument `ylab="label"` (short for "y label"). Similarly, we can extend the y-axis up to 20 (since the default only plotted to 15 for some reason) with the command `ylim=c(min,max)` (short for "y limits").

Since we have two categories for our Food treatment and three categories for our Predator treatment, it would make sense for those sets of bars to be grouped together, either grouped in two groups of three (so that each group was a Food treatment) or three groups of two (so that each group was a Predator treatment). Let's organize them in the latter fashion, with three groups of two bars. This is where the functionality of `barplot2()` is very useful. We have one extra step to do, but it will save us a lot of work in the end. We need to turn our data of means and SE's into 2x3 matrices, which will enable the `barplot2()` function to know how we want the bars grouped.

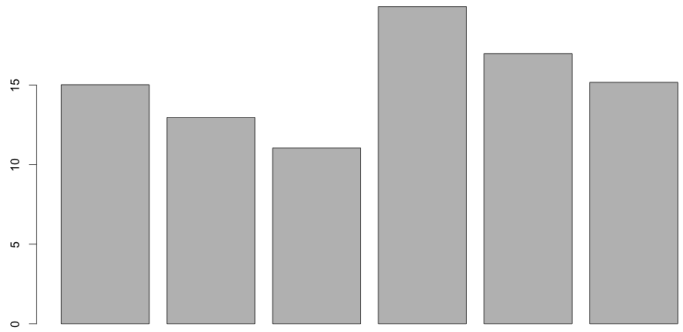

Figure 4: The most boring barplot of tadpole sizes imaginable

The function `matrix()` will turn a vector into a 2-dimensional matrix, with dimensions (numbers of rows and columns) of our choosing. We have to pass `matrix()` four things: the vector of data, the number of rows, the number of columns, and the direction that we want to data to be placed into the matrix. Our vector of data is just the third column from our objects `tadpole.mean` or `tadpole.SE`. And in this case we want it to have 2 rows and 3 columns, and the data should be entered by row.

```
> tadpole.mean.mat<-matrix(tadpole.mean[,3], nrow=2, ncol=3, byrow=T)
> tadpole.SE.mat<-matrix(tadpole.SE[,3], nrow=2, ncol=3, byrow=T)
```

Another nice feature of `barplot2()` is that if we put names on our rows and columns, the plot will automatically know what labels to put with our different bars. We can change the row and column names by assigning them new values using the `rownames()` and `colnames()` functions, respectively. We could type the names out that we want for each row and column, or we can just assign them the `levels()` from the `tadpole.mean` and `tadpole.SE` objects.

```
> rownames(tadpole.mean.mat)<-levels(tadpole.mean[,2])
> colnames(tadpole.mean.mat)<-levels(tadpole.mean[,1])
> rownames(tadpole.SE.mat)<-levels(tadpole.SE[,2])
> colnames(tadpole.SE.mat)<-levels(tadpole.SE[,1])
```

Now we have two matrices formatted and ready to plot.

```
> tadpole.mean.mat
         Low     Med    High
Cont 15.0274 12.9622 11.0460
Prot 19.9396 16.9846 15.1768
> tadpole.SE.mat
           Low       Med      High
Cont 0.1526594 0.1279739 0.1729247
Prot 0.1182778 0.1531887 0.1288833
```

By default, `barplot2()` stacks grouped bars, which is not what we want. We want our groups side-by-side. So we have to add the argument `beside=T`.

```
> barplot2(tadpole.mean.mat, beside=T, ylim=c(0,20), ylab="Mean size of tadpoles")
```

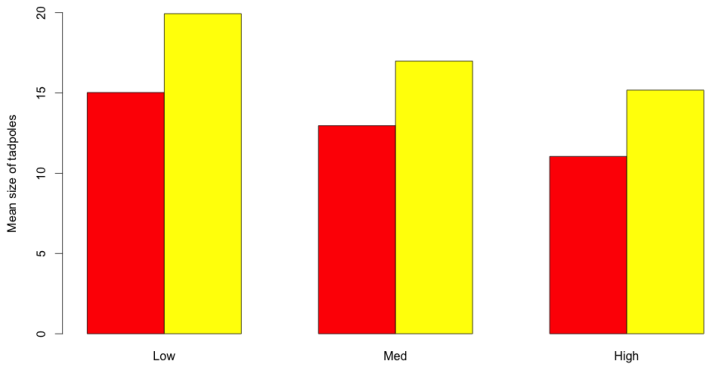

Figure 5: A barplot of tadpole sizes, with bars grouped by Pred treatment. The only problem is that it looks like ketchup and mustard.

Wow! What a great choice of default colors! The red and yellow is seriously ugly, but we can change the color by adding the `col=()` argument. Colors can be defined in many ways in `R`.

1. You can use the default colors 1-8 (in order: black, red, green, blue, cyan, magenta, yellow and grey).
2. You can define colors using the function `rgb()`.
3. You can define colors using the hex color system. One advantage of using hex colors is that you can make colors translucent, which can be useful in certain scatterplots of venn diagrams, for example.
4. You can used predefined and named colors viewable by typing `colors()`.

For this exercise, we will use prenamed colors built in to `R`. I will use two shades of green, `light green` and `dark green`. Note that since the progression of categories is the same in the first group of bars as it is in the second and third sets of bars, we only have to define our set of colors once. In addition, we should add an x-axis label, which we can do by adding the `xlab="label` argument.

```
> barplot2(tadpole.mean.mat, beside=T, ylim=c(0,20), ylab="Mean size of tadpoles",
  xlab="Predator treatment", col=c("light green","dark green"))
```

Now we should think about adding error bars. Once again, this is a benefit of using the `barplot2()` function, which makes adding error bars very easy. Since we have a matrix of SE's that is formatted the

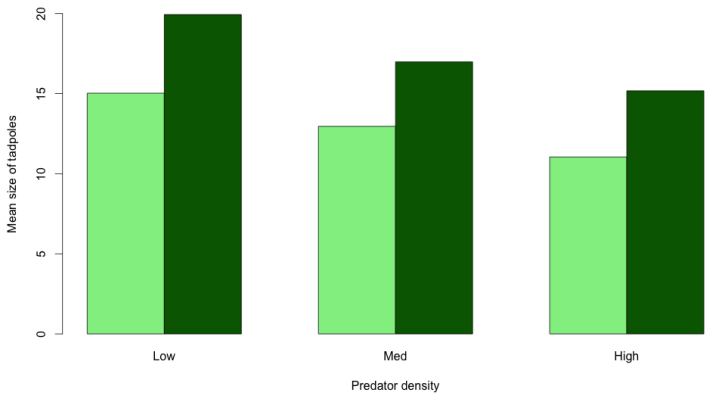

Figure 6: A barplot of tadpole sizes, with bars grouped by Pred treatment and a tasteful color scheme.

same as our matrix of means, we just have to add a few arguments to our plotting function. First, we add the argument `plot.ci=T`, which tells the function we are going to add error bars (the `ci` stands for "confidence interval"). We then have to tell it where to place the upper end of the error bar and where to place the lower end. We do this by specifying two more arguments: `ci.u=...` and `ci.l=....` I've placed the `...` after the argument to indicate that you can put anything there. Our error bars start at the mean + the SE, and they end at the mean - the SE.

```
> barplot2(tadpole.mean.mat, beside=T, ylim=c(0,20), ylab="Mean size of tadpoles",
  xlab="Predator treatment", col=c("light green","dark green"), plot.ci=T, ci.u=ta
  dpole.mean.mat+tadpole.SE.mat, ci.l=tadpole.mean.mat-tadpole.SE.mat)
```

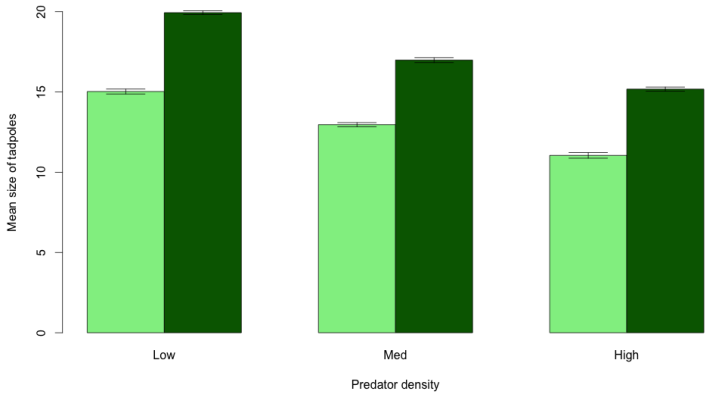

Figure 7: A barplot of tadpole sizes with standard error bars. Bars are grouped by Predator treatment and colors indicate the Food treatment

Lastly, we should add a legend. This uses a separate function `legend()`.

```
> legend(x="topright", legend=c("Control","Protein"), pch=15, col=c("light
  green","dark green"), bty="n", cex=1.5)
```

In the code for the legend, I've specified 1) that I want the legend in the top right corner of the plot (although you can also specify specific x- and y-coordinates), 2) what I want the legend to say, 3) that I want colored squares next to the text (the `pch=15` part), 4) what color to make the squares (the same as our bars), 5) that I do not want a box around the legend (the `bty="n"` part), and 6) that I want the legend 1.5 times larger than the default (the `cex=1.5` part).

What else should we do to make our figure look nice? If you want to put a box all the way around the figure, you can simply type `box()` at the prompt. If we do that, the error bar on the 2nd bar will probably be a little close to the box, so we should increase our y-limits a little. Most journals like the numbers on your y-axis to be turned so that they are more legible, which can be done by adding the argument `las=1` to any plotting function. We should also change our y-label to include the fact that we have error bars now.

Thus, in the end, we need three functions to make this figure. `barplot2()`, `legend()` and `box()`. That may seem cumbersome, but once you have made one, it is very easy to modify the code to make others. And you can easily change the data (the `tadpole.mean` and `tadpole.SE` objects) and remake the figure instantly.

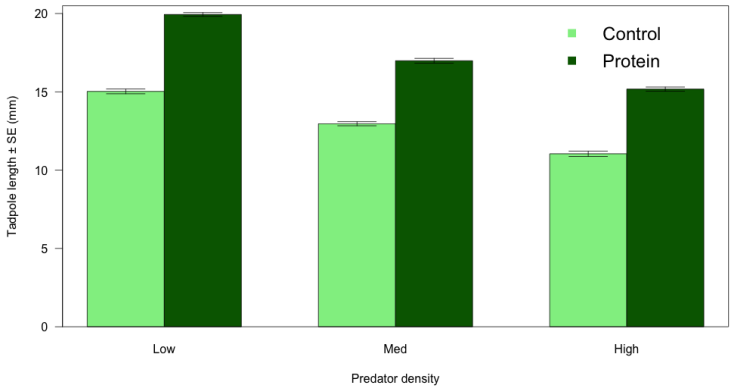

Figure 8: A very complete barplot of mean tadpole sizes with standard error bars. Bars are grouped by Food treatment and colors indicate the Predator treatment

Here is the final set of code to make the figure.

```
> barplot2(tadpole.mean.mat, beside=T, col=c("light green","dark green"), ylim=c
  (0,20.5), ylab="Tadpole length  SE (mm)", xlab="Predator density", plot.ci=T,
  ci.u=tadpole.mean.mat+tadpole.SE.mat, ci.l=tadpole.mean.mat-tadpole.SE.mat,
  las=1)
> legend(x="topright", legend=c("Control","Protein"), pch=15, col=c("light green"
  ,"dark green"), bty="n", cex=1.5)
>  box()
```

6 Exercises

Here are some exercises to try on your own, to build on the plotting skills you have just worked on.

1. Make the barplot on the next page. It is similar to the one above, but the bars are grouped by Food level instead of Predator density, so that you have two sets of three bars. Be creative with the colors you choose for your bars! Remember that you can view the list of pre-defined colors by typing `colors()`. You will need to restructure the matrix, but if your barplot doesn't come out looking like this one, make sure to look at the matrix you have made and verify the correct values are where you want them.

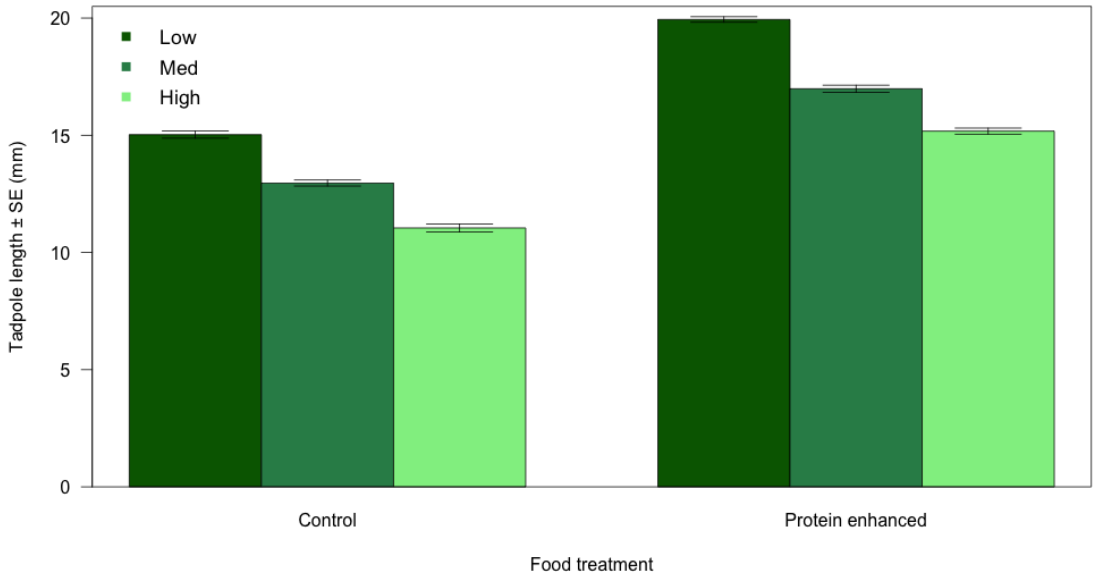

Figure 9: Make this figure for your BIOL 106 prelab assignment!

2. Make a barplot but using age at metarmophosis as the response variable.

3. Plot age at metamorphosis against final metamorph SVL, adding informative axes labels and changing the color or symbol of the points for each predator treatment.