Graphing Useful for Regression

Graphing regressions allows you to visually see what is happening in your data much more clearly than simply looking at numbers on a page. It is possible to graph every aspect of a regression that might serve of interest in your analysis. This guide is organized by type of graph.

If you would like more details about something, remember that you can use the ? to get more information. Example: for histograms use ?hist. Another command that will be very helpful throughout this manual is ?par. It will show you graphical parameters you can manipulate.

Some of the datasets used in this manual should be already installed within R. Some will be found in packages that you will need to load into R first. Make sure to load the appropriate package prior to trying to graph a dataset The library() command will take care of this for you.

Remember: set your working directory to whatever you want it to be prior to starting this.

In addition, the Courier New font will represent all R code in this manual and > will represent the start of a line of code.

I. Single Predictor Line Graph

We will start with a simple model wherein we regress one variable upon another.

> data()

The above brings up all datasets currently loaded into R. We will use “pressure” for this example and it should already be installed into R. It uses the temperature of Mercury to predict vapor pressure. If the dataset pressure is not listed, use library(datasets).

> pressure

The above command allows us to look at the full dataset.
a. Initial graphing of data

```r
> plot(pressure~temperature, data = pressure)
```

Pressure is our dependent variable and temperature is our independent variable. This plots temperature on the x-axis and pressure on the y-axis. Looking at this graph we can get a sense of the relationship between the two.

Within the command `plot()` are several different options that may help us visualize our data. For example, say we would like a line graph rather than a bunch of points. By simply adding the argument: `type = "l"` we can get a line graph of the data.

```r
> plot(pressure~temperature, data = pressure, type = "l")
```
b. Making the graph look the way you want it to

Other arguments that may be used to format the graph to be more attractive (you can use `?par` or `?plot` for more options):

```
type = "b"  gives us both points and lines

lty =   this changes the style of line in graph. If = 0, the line is blank. If 
       =1, line is solid. If = 2, line is dashed. If = 3, line is dotted. If = 4, line is dotdash. 
       If = 5, line is longdash. If = 6, line is twodash.

main = "....." fill in the “......” with whatever the title of the graph should be. 
       Make sure to put title in quotes.

sub = "....." fill in the “......” with whatever the subtitle of the graph should be. 
       Make sure to put title in quotes.

xlab = "....." fill in the “......” with whatever the x axis should be labeled. 
       Again- put the label in quotes.

ylab = "....." fill in the “......” with whatever the x axis should be labeled. Put 
       the label in quotes.

col = "....." this will change the color of the line to blue. If you are using 
       points rather than a line, the points will change to blue. For all possible colors, use 
       color() to see all of the color names.

col.axis = "....." this will change the color of the annotations for both x 
           and y axes.

col.lab = "....." this will change the color of the x and y labels.

font.axis =   this will change the style of font for the axes. If = 1, the 
              style will be plain. If = 2, the font will be bolded. If = 3, the font will be italicized. 
              If = 4, the font will be bolded and italicized.

font.lab =   this does the exact same thing as in font.axis, just for the 
           labels instead.

font.main =   this does the exact same thing as in font.axis, just for the 
           main title instead.

font.sub =   this does the exact same thing as in font.axis, just for the 
           subtitle instead.
```
Here is an example of this code all put together followed by the graph that results:

```r
> plot(pressure~temperature, data=pressure, type="l", lty = 2, main="Effects of Temperature on Pressure", xlab="Temp", ylab="Pressure", col = "blue", col.axis = "yellow", col.lab = "violetred", font.main = 3)
```

![Graph of Effects of Temperature on Pressure](image)

Hint: If you have data with several observations clustered together, it can be hard to see exactly what is going on. The jitter() function will add a small random amount of noise to each point and make it clear where any clusters might be. Instead of saying `plot(pressure~temperature, data = pressure)`, change it to `plot(jitter(pressure)~jitter(temperature), data=pressure)`.

II. Histograms

The dataset lynx contains information on the number of trappings of Canadian lynxes each year from 1821-1934. We want to see the frequency of trappings (not sure why we would want to know this, but it will show us how to plot histograms).

For this example, we will first need to turn the dataset lynx from a time series set of data into a dataframe. Then we can plot the number of trappings that occurred each year.

```r
> lynx.data <- as.data.frame(lynx)
> years <- seq(1821,1934, by = 1)
> lynx.data <- cbind(years, lynx.data)
> names(lynx.data) <- c("Year", "Number of Trappings")
```
a. Initial graphing of data

Now we can look at a histogram of the annual number of trappings.

> hist(lynx.data$"Number of Trappings")

This works, but is not very attractive.

b. Making the graph look the way you want it to

Other arguments that may be used to format the graph to be more attractive (or use \texttt{?par} and \texttt{?hist} to get more information):

- \texttt{breaks} = this will allow you to change how the bins are broken up. See \texttt{?hist} for more details on this.

- \texttt{density} = this will change the shading of the bars. The higher the number, the more densely shaded the bars will be.

- \texttt{labels} = if equal to TRUE, the counts for each bin will be put onto the graph.

- \texttt{xlim} = \texttt{c(...,...)} this will allow you to define the limits of x. The first … will be the number for the lower limit and the second … will be the upper limit.

- \texttt{ylim} = \texttt{c(...,...)} this will allow you to define the limits of y. The first … will be the number for the lower limit and the second … will be the upper limit.
main = "....." fill in the “.....” with whatever the title of the graph should be. Make sure to put title in quotes.

sub = "....." fill in the “.....” with whatever the subtitle of the graph should be. Make sure to put title in quotes.

xlab = "....." fill in the “.....” with whatever the x axis should be labeled. Again put the label in quotes.

ylab = "....." fill in the “.....” with whatever the x axis should be labeled. Put the label in quotes.

col = "....." this will change the color of fill of the bars. For all possible colors, use color() to see all of the color names.

col.axis = "....." this will change the color of the annotations for both x and y axes.

col.lab = "....." this will change the color of the x and y labels.

font.axis = this will change the style of font for the axes. If = 1, the style will be plain. If = 2, the font will be bolded. If = 3, the font will be italicized. If = 4, the font will be bolded and italicized.

font.lab = this does the exact same thing as in font.axis, just for the labels instead.

font.main = this does the exact same thing as in font.axis, just for the main title instead.

font.sub = this does the exact same thing as in font.axis, just for the subtitle instead.

Here is an example of some of this code all put together followed by the graph that results:

> hist(lynx.data$"Number of Trappings", density = 5, col = "blue", sub = "Years 1821-1934", main = "Canadian Lynx Trappings", xlab = "Number Trapped Annually", labels = TRUE)
c. Histograms with density instead of frequency

It is easy to look at the density of a particular variable using a histogram. Plot a histogram like you normally would, but add in the argument “probability = T.” Then just add in a density line and you will be good to go.

```r
> hist(lynx.data$"Number of Trappings", probability=T, xlab="Numbers Trapped Annually", main="Canadian Lynx")
> lines(density(lynx.data$"Number of Trappings")
```
III. Box Plots

You can use this type of graph to look at the overall range of a distribution of a variable. The graph will show the lower extreme, lower quartile, median, upper quartile, and upper extreme values.

a. Looking at one variable

If histograms are not your favorite, you can use box plots to look at the same type of information. For example, we can show our lynx trapping data as a box plot instead.

> boxplot(lynx.data$"Number of Trappings", main="Canadian Lynx Trappings", ylab="Number Trapped Annually")

b. Looking at Multi-Level Variables

If you have data with different levels of factors, you can use box plots to see how the data varies between the different levels. In the dataset ChickWeight (should be already installed on your computer in package “datasets”), chicks were assigned to different diets and their weights were measured across several days. We want to see how distributions of weights varied between the different diets.

> split.chick <- split(ChickWeight, ChickWeight$Diet)

The above splits the dataset ChickWeight into four levels, each corresponding to a different type of Diet.
> boxplot(split.chick$"1"$weight ~ split.chick$"1"$Time)

The above will give the weight of chicks assigned to the first diet across the days of the experiment. We could also look at box plots for the other three diets at the same time. First we will set our graphing options to allow for four graphs on one screen.

> op <- par(mfrow = c(2,2))

Now graph all four diets. Notice that we can use the same arguments as in line and histograms for main, xlab, and ylab. You could also add color, change the style of the fonts, etc.

> boxplot(split.chick$"1"$weight~split.chick$"1"$Time, main = "Diet 1", xlab= "Day", ylab="Weight")

> boxplot(split.chick$"2"$weight~split.chick$"2"$Time, main = "Diet 2", xlab= "Day", ylab="Weight")

> boxplot(split.chick$"3"$weight~split.chick$"3"$Time, main = "Diet 3", xlab= "Day", ylab="Weight")

> boxplot(split.chick$"4"$weight~split.chick$"4"$Time, main = "Diet 4", xlab= "Day", ylab="Weight")
IV. Checking regression assumptions (outliers and influence):

The following example will use the dataset Vocab from the package car. You will need to load it into your R session.

```r
> library(car)
> vocab.reg <- lm(vocabulary~education, data = Vocab)
> summary(vocab.reg)
```

The above gives us the output for the regression.

```r
> plot(vocab.reg)
```

The above will give us four different regression diagnostic graphs one at a time.

1) Residuals vs. Fitted Values
2) Normal Q-Q Plot

3) Scale-Location Plot
4) Cook’s Distance Plot

![Residuals vs Leverage Graph]

We can also test for outliers using function outlier.test from package car.

```r
> library(car)
> outlier.test(vocab.reg)
```

From this we can get the largest absolute studentized residual as well as the Bonferroni p-value for the most extreme observation.

V. More Complex Regression Graphs

We will use the dataset ToothGrowth from package “datasets” that should be already loaded into R. It measures how different doses of different supplements change how long guinea pigs’ teeth grow. Admit it- this is a question that keeps you up at night.

First make sure you have access to the dataset:

```r
> ToothGrowth
```

Now we will set up the regression for ToothGrowth.

```r
> names(ToothGrowth) <- c("Tooth.Length", "Supplement", "Dose")
> contrasts(ToothGrowth$Supplement) <- c(-.5,.5)
```
```r
> model <- lm(Tooth.Length~Supplement*Dose, data=ToothGrowth)
> summary(model)

Now that we know we have a significant interaction, we can plot it in order to see exactly what is going on.

a. Graphing interactions with `persp()`

We will first create a matrix of predicted values for model. Then we can use `persp()` on that matrix to see the interaction in 3D.

> a <- vector(length=42)
> j <- 0

> for (i in c(-.5,.5)) {
>   for (k in seq(0, 2, by=.1)){
>     j <- j+1
>   }
> }

The above loop gets us a vector called `a` with predicted values for our model. The first half of the values is when the supplement given is OJ (coded -.5) and the dose of the supplement ranged from 0 to 2. The second half is when the supplement given is VC (coded .5) and the dose ranged from 0 to 2.

> predicted <- matrix(a, ncol=2, byrow=FALSE)

The above puts our values into a matrix with the first column for values for supplement OJ and the second column for values for supplement VC. We can now use `persp()` to see what the interaction looks like.
Now that we have a graph, we can play around with the perspective to really see what is going on:

```r
> persp(predicted)
```

And if playing the perspective is not enough, you can use all of the arguments used for histograms and line graphs here to change the appearance of the graph.

```r
> persp(predicted, phi=10, theta=60)
```
b. More complex options

While the stripes are pretty, they do not make it obvious where the highest values of predicted are. We can instead color the graph in such a way that the colors will be on a gradient so that the colors will change as the predicted values decrease. This will be very useful if you have data with dips in it.

The first step is to find facets for the predicted values matrix. Facets are averages of four predicted values. For example, if we have a matrix that looks like this:

```
  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15 16
```

We have 16 total predicted values (4 x 4). To get our facets, we will take four values that are next to each other in a square and average them and then do this for all possible values. For example, we could first do (1+2+5+6)/4. That will give us a facet value of 3.5 for that square. By the time all facets are computed, we will end up with 9 total (4-1 x 4-1).
Lucky for us, we can use a function to get the values for us.

```r
> make.facets <- function(x) {
  if (class(x) != "matrix" & class(x) != "data.frame")
    stop("x must be matrix or data.frame")
  facets <- (x[-1,-1]+x[-1,ncol(x)]+x[-nrow(x),-1]+x[-nrow(x),-ncol(x)])/4
  return(facets)
}
```

We will now use this function on our matrix of predicted values for ToothGrowth.

Here is our original matrix for predicted values.

```r
> Predicted
     [,1]      [,2]
  [1,] 11.550  3.2950
  [2,] 12.331  4.4667
  [3,] 13.112  5.6381
  [4,] 13.893  6.8097
  [5,] 14.675  7.9813
  [6,] 15.456  9.1529
  [7,] 16.237 10.3244
  [8,] 17.018 11.4960
  [9,] 17.799 12.6676
 [10,] 18.581 13.8391
 [11,] 19.361 15.0107
 [12,] 20.143 16.1823
 [13,] 20.924 17.3539
 [14,] 21.705 18.5254
 [15,] 22.486 19.6970
 [16,] 23.267 20.8686
 [17,] 24.048 22.0401
 [18,] 24.829 23.2117
 [19,] 25.611 24.3833
 [20,] 26.392 25.5486
 [21,] 27.173 26.7264
```

```r
> facets <- make.facets(predicted)
```

Here are the values for facets:

```r
```
Now we need to use `cut()` to divide up the values of facets into equal intervals. We will do 20 intervals as that is the length of our vector. With multiple columns, you can use the length of one column as the number of cuts:

```r
> cuts <- cut(facets, seq(min(facets), max(facets), 20), include.lowest=TRUE)
```

The next step is to get a matrix wherein we take a predefined set of colors such as `terrain.colors()` or `rainbow()` and index the values from cuts. This matrix will have our highest value of cut assigned to one color, the next highest to a different color, etc:

```r
> col.predict <- matrix(terrain.colors(10)[cuts])
```

We are almost done. Now that we have a matrix for the colors we want, all that remains is to use `persp()` to see the interaction.

```r
> persp(predicted, col=col.predict)
```

To make it clearer what is happening, we can look at several different views of this:

```r
> op <- par(mfcol=c(2,3))

> for (i in seq(0, 270, by=90)) {
    for (k in seq(0, 90, by=45)){
        persp(z=predicted, col=col.predict, phi=k, theta=i, main=paste("Theta = ", i, "Phi = ", k))
    }
}
```
V. Graphing Multiple Levels of a Factor

In the dataset Davis (located in package car), weight is predicted using height and gender as parameters. It is easy to graph the relationship between height and weight using `plot(Davis$weight~Davis$height)`, but it is impossible to tell which points are for males and which are for females. The following code will give the points for males in one color and the points for females in another color.

```r
> library(car)
> Davis

Look carefully at the data and see if you notice anything that is off. Hint: I do not think there are a lot of people who are 182 inches tall.

```r
> names(Davis) <- c("sex","height","weight","repwt","repht")
```

Now we that we have the data name correctly, we can set up our regression model.

```r
> plot(Davis$weight~Davis$height)
> split.davis.1 <- split(Davis$weight, Davis$sex)
```

The above will separate the values for weight by the different levels of factor sex.

```r
> split.davis.2 <- split(Davis$height, Davis$sex)
```

The above will separate the values for height by the different levels of factor sex.
Using the above code, we will get a plot with the right dimensions but no lines or points. The x-axis and y-axis are labeled appropriately at 1.25 times the normal size and the graph has been titled. Now we can put in the points we want.

The above puts in the points corresponding to the males in Davis. pch=22 will make the points square shaped, the squares will be blue, and cex=1.5 will make the points 1.5 times larger than normal.

The above does the exact same thing for females, but the points will be magenta diamonds.

The above two commands put in best-fit lines- one for male values and one for female values.

The above command will add a legend at coordinates (x=170, y=180). It will be 1.5 times the normal size and right justified on the x-axis. See ?legend for all options.

After all of this code, the following graph results:
Note: Now we can see that there is a HUGE outlier and it is for one of the females. We should take care of that before doing anything else for this model.

FYI: There are many different types of points we can use. If you run the following code, you will create a function called “pchShow.”

```r
pchShow <- function(extras = c("*",".", "o","O","0","+","- ", ",","%","#"),
                    cex = 3, ## good for both .Device="postscript" and "x11"
                    col = "red3", bg = "gold", coltext = "brown", cextext = 1.2,
                    main = paste("plot symbols : points (... pch = *", cex =",
                    cex,")")
                    {
                      nex <- length(extras)
                      np <- 26 + nex
                      ipch <- 0:(np-1)
                      k <- floor(sqrt(np))
                      dd <- c(-1,1)/2
                      rx <- dd + range(ix <- ipch %% k)
                      ry <- dd + range(iy <- 3 + (k-1)- ipch %% k)
                      pch <- as.list(ipch) # list with integers & strings
                      if(nex > 0) pch[26+ 1:nex] <- as.list(extras)
                      plot(rx, ry, type="n", axes = FALSE, xlab = "", ylab = ", main =
                      abline(v = ix, h = iy, col = "lightgray", lty = "dotted")
                      for(i in 1:np) {
                        pc <- pch[[i]]
                        ## 'col' symbols with a 'bg'-colored interior (where available) :
                        points(ix[i], iy[i], pch = pc, col = col, bg = bg, cex = cex)
                        if(cextext > 0)
                          text(ix[i] - .3, iy[i], pc, col = coltext, cex = cextext)
                      }
                    }
```

```r
# Create function pchShow
pchShow()
```
> pchShow()

This will give us a graphic showing all of the different possible values for pch. Each value corresponds to a different shape. For example, `pch=19` (solid circle), `pch=22` (square), and `pch=25` (triangle point down).

VI. Saving the Graph to Use Later

There is a good chance you will want to save your graphs so that you can use it later on. It is possible to save the graphs as PDFs or jpegs using functions found in the package `grDevices`. Make sure you know what your working directory is, as that is where the graph will be saved.

> pdf("Title of graph.pdf")

> Insert whatever code you used to create the graph.

> It can be several commands long.

> dev.off()

The final command tells R that the graph is complete. It is critical that you use it. After that, you can look in your working directory and the graph should be there ready to go.