Regression basics

Much of the content here is from Appendix A of my *Analysis of Categorical Data with R* book (`www.chrisbilder.com/categorical`).

Suppose you would like to estimate an individual's college GPA by their high school GPA through a simple linear regression model. The corresponding R program for this example is gpa.R and the data files are gpa.txt (plain text file using space delimiters) and gpa.csv (plain text file using comma delimiters).

Data management

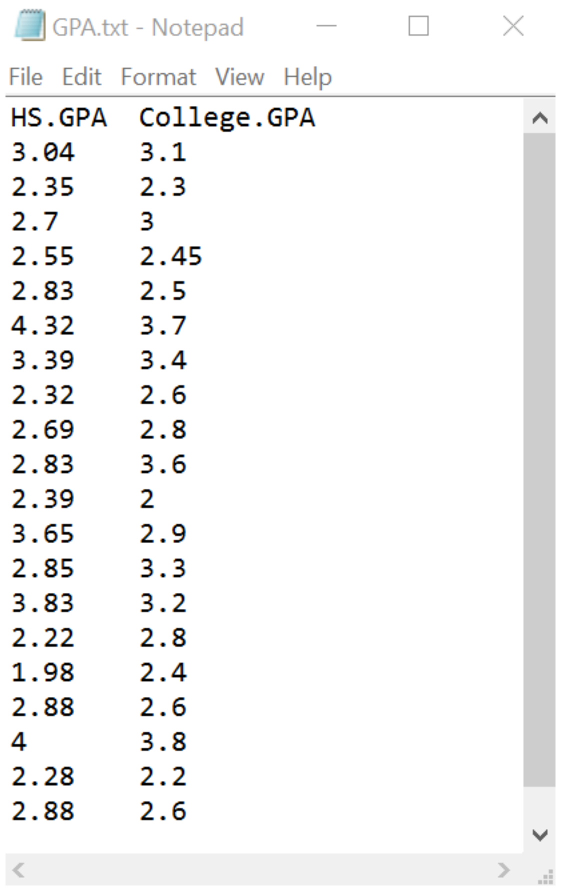
Below is how I read in the data

```
> #' AUTHOR: Chris Bilder
> #' DATE: 8-12-14
> #' PURPOSE: Regression model for GPA data
> # Note that a single quote is not needed in R for comments. I
> # included them here only to make LyX and knitR recognize
> # multiple lines of comments rather than combining them into
> # one line.
>
> # Read in the data - location is for my computer
> gpa <- read.table(file = "C:\\data\\GPA.txt", header = TRUE,
    sep = "")
> # Print the data
> gpa
  HS.GPA College.GPA
1   3.04        3.10
2   2.35        2.30
3   2.70        3.00
4   2.55        2.45
5   2.83        2.50
6   4.32        3.70
7   3.39        3.40
8   2.32        2.60
```

```
9    2.69         2.80
10   2.83         3.60
11   2.39         2.00
12   3.65         2.90
13   2.85         3.30
14   3.83         3.20
15   2.22         2.80
16   1.98         2.40
17   2.88         2.60
18   4.00         3.80
19   2.28         2.20
20   2.88         2.60
> # Print part of the data
> head(gpa)
  HS.GPA College.GPA
1   3.04         3.10
2   2.35         2.30
3   2.70         3.00
4   2.55         2.45
5   2.83         2.50
6   4.32         3.70
```

Notes:

- The # denotes a comment line in R. At the top of every program you should have some information about the author, date, and purpose of the program.
- The gpa.txt file is an ASCII text file that looks like:



```
HS.GPA  College.GPA
3.04    3.1
2.35    2.3
2.7     3
2.55    2.45
2.83    2.5
4.32    3.7
3.39    3.4
2.32    2.6
2.69    2.8
2.83    3.6
2.39    2
3.65    2.9
2.85    3.3
3.83    3.2
2.22    2.8
1.98    2.4
2.88    2.6
4       3.8
2.28    2.2
2.88    2.6
```

The `read.table()` function reads in the data and puts it into an object called `gpa` here. Notice the use of \\ between folder names. This needs to be used instead of \. Also, you can use / too. Since the variable names are at the top of the file, the `header = TRUE` option is given. The `sep = ""` option specifies white space (spaces, tabs, ...) is used to separate variable values.

- The `gpa` object is an object type called a *data frame.*
- The `head()` function is a simple way to print the first few lines of an object as a quick check. The default is to print the first 6 lines. The `n` argument can be specified to show a different number of lines; e.g., `head(gpa, n = 1)` will give the first line only. A `tail()` function also exists to print the last few lines of an object.

Alternative data file formats:

- One can use `sep = ","` for comma delimited files with `read.table()`. Alternatively, one can use `read.csv()` without the `sep` or `header` arguments.

```
> # Location is for my computer
> gpa2 <- read.csv(file = "C:\\data\\GPA.csv")
> head(gpa2)

  HSGPA CollegeGPA
1  3.04       3.10
2  2.35       2.30
3  2.70       3.00
4  2.55       2.45
5  2.83       2.50
6  4.32       3.70
```

- There are a few different ways to read in Excel files into R. However, myself and the R community generally recommend avoiding Excel formats for a number of reasons, including communication issues with 32-bit and 64-bit versions of Excel and R. When given an Excel file, I will generally save it in a comma delimitted format and read it into R from there.
- The `write.table()` and `write.csv()` functions export data out of R:

```
> # Did not execute because need this specific file location on
> # a drive
> write.csv(x = gpa, file = "C:\\data\\GPAout.csv", row.names = FALSE,
     quote = FALSE)
```

Once data is in a data frame, one variable at a time can be accessed by using the syntax `<data.frame>$<variable>`. For example,

```
> names(gpa)
[1] "HS.GPA"      "College.GPA"
```

```
> gpa$HS.GPA
 [1] 3.04 2.35 2.70 2.55 2.83 4.32 3.39 2.32 2.69 2.83 2.39 3.65 2.85 3.83
[15] 2.22 1.98 2.88 4.00 2.28 2.88
```

Notice that the `names()` function provides a list of variables included in the data frame. We will use this function again later for more complex data objects!

Parts of the data frame can also be accessed through using a matrix-like reference. For example,

```
> gpa[1, 1]
[1] 3.04
> gpa[, 1]
 [1] 3.04 2.35 2.70 2.55 2.83 4.32 3.39 2.32 2.69 2.83 2.39 3.65 2.85 3.83
[15] 2.22 1.98 2.88 4.00 2.28 2.88
> gpa[, "HS.GPA"]
 [1] 3.04 2.35 2.70 2.55 2.83 4.32 3.39 2.32 2.69 2.83 2.39 3.65 2.85 3.83
[15] 2.22 1.98 2.88 4.00 2.28 2.88
> gpa[1, 1:2]
  HS.GPA College.GPA
1   3.04         3.1
> gpa[1, c(1, 2)]
  HS.GPA College.GPA
1   3.04         3.1
> gpa[, c("HS.GPA", "College.GPA")]
   HS.GPA College.GPA
1    3.04        3.10
2    2.35        2.30
3    2.70        3.00
4    2.55        2.45
5    2.83        2.50
6    4.32        3.70
7    3.39        3.40
8    2.32        2.60
9    2.69        2.80
10   2.83        3.60
11   2.39        2.00
```

```
12   3.65        2.90
13   2.85        3.30
14   3.83        3.20
15   2.22        2.80
16   1.98        2.40
17   2.88        2.60
18   4.00        3.80
19   2.28        2.20
20   2.88        2.60
```

Questions:

- How can you access only the first row of a data frame?
- What does `gpa[,-2]` return?

There are times when you would like to access parts of a data set based on some condition. For example, suppose you would like to view observations where the high school GPA was less than 2.5:

```
> gpa$HS.GPA < 2.5
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
[12] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
> gpa[gpa$HS.GPA < 2.5, ]
   HS.GPA College.GPA
2    2.35         2.3
8    2.32         2.6
11   2.39         2.0
15   2.22         2.8
16   1.98         2.4
19   2.28         2.2
> sum(gpa$HS.GPA < 2.5)
[1] 6
> gpa$HS.GPA < 2.5 & gpa$College.GPA < 2.5  # And
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[12] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
> gpa$HS.GPA < 2.5 | gpa$College.GPA < 2.5  # Or
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
[12] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

```
> gpa$HS.GPA == 2.35  # Equal
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> gpa$HS.GPA != 2.35  # Not equal
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The `gpa$HS.GPA < 2.5` part performs the logical comparison of "Is a high school GPA < 2.5?" A `TRUE` or `FALSE` is produced for each entry. Using the resulting vector, we can pull out those rows from `gpa` that satisfy the condition. Also, note that R treats the `TRUE` and `FALSE` values as 1's and 0's, respectively, when working with a mathematical function. This is helpful to determine how often a condition is satisfied. The `ifelse()` function performs a similar logical comparison and we will discuss this further later in the course.

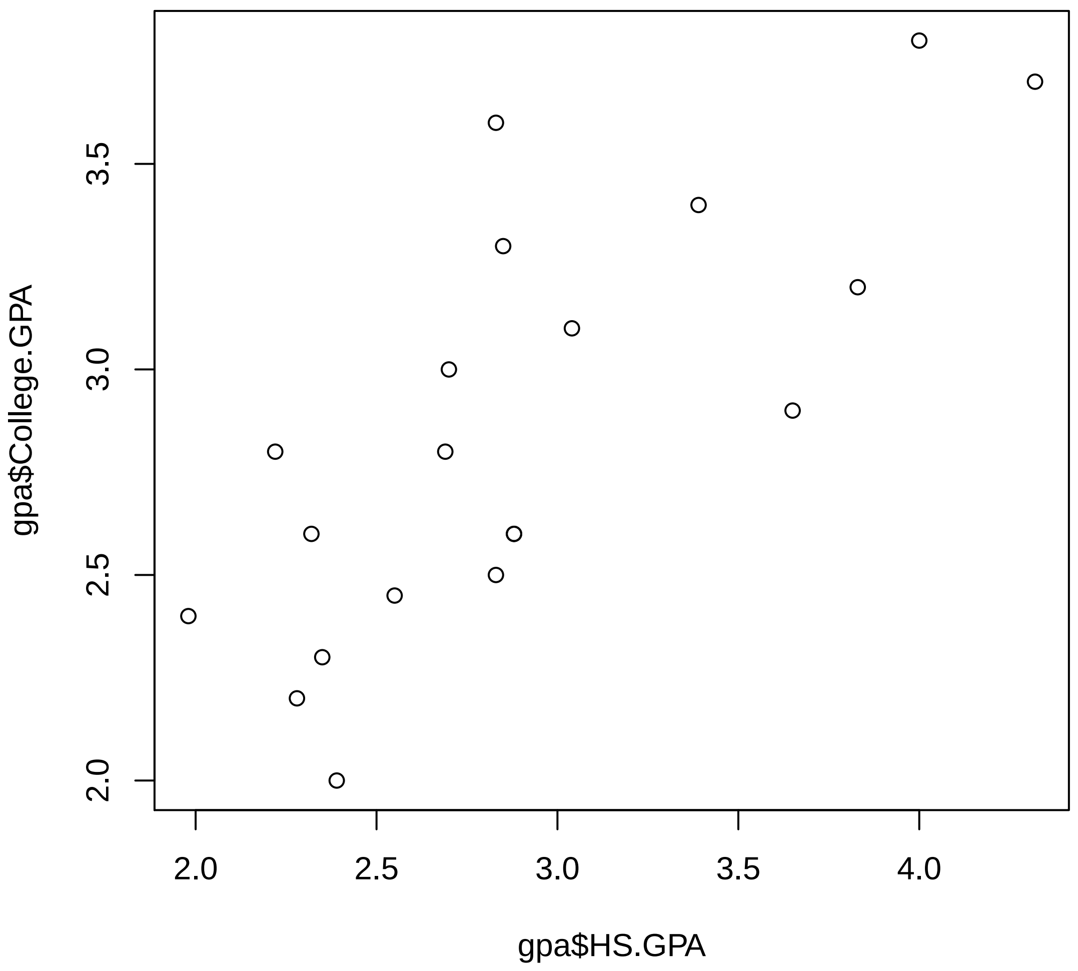
The `summary()` function provides a simple data summary:

```
> summary(object = gpa)  # Can also use summary(gpa)
     HS.GPA       College.GPA
 Min.   :1.98   Min.   :2.00
 1st Qu.:2.38   1st Qu.:2.49
 Median :2.83   Median :2.80
 Mean   :2.90   Mean   :2.86
 3rd Qu.:3.13   3rd Qu.:3.23
 Max.   :4.32   Max.   :3.80
```

Scatter plot

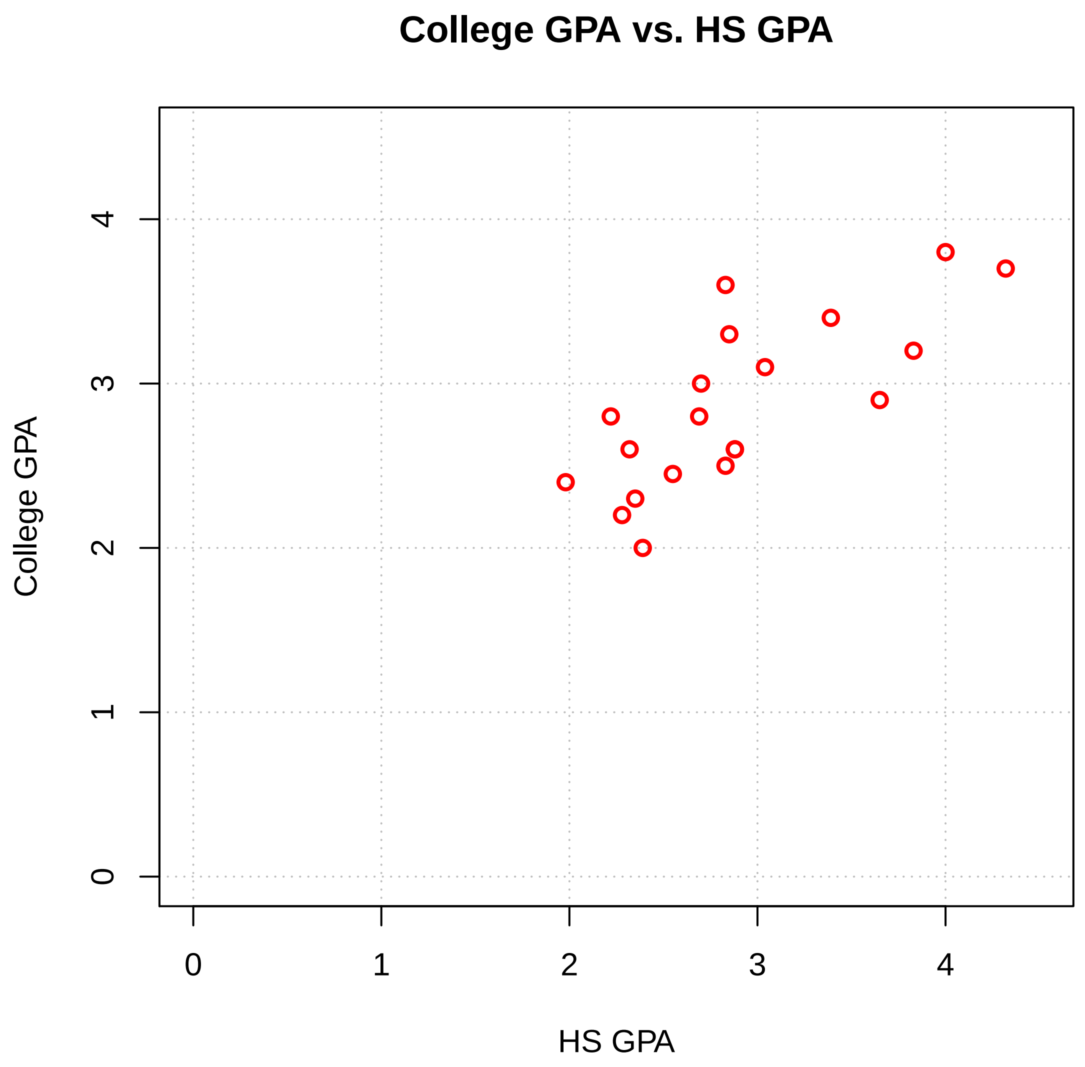
Below is a simple scatter plot of the data created by the `plot()` function.

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA)
```



Including optional arguments makes the plot look much better:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
     ylab = "College GPA", main = "College GPA vs. HS GPA",
     xlim = c(0, 4.5), ylim = c(0, 4.5), col = "red",
     pch = 1, cex = 1, lwd = 2, panel.first = grid(col = "gray",
         lty = "dotted"))
```

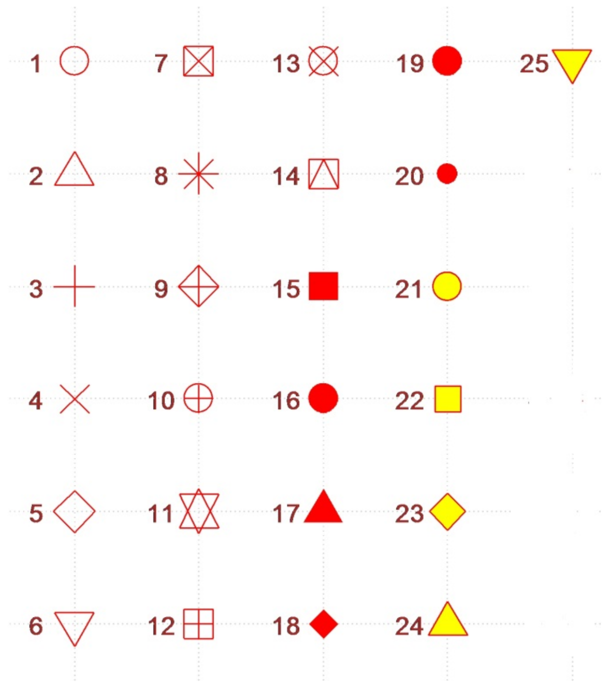


Descriptions of the optional arguments:

- `x` and `y` specify what is plotted on the x-axis and y-axis, respectively
- `xlab` and `ylab` specify the x-axis and y-axis labels, respectively
- `main` specifies the main title of the plot
- `xlim` and `ylim` specify the x-axis and y-axis limits, respectively; notice the use of the `c()` function
- `col` specifies the color of the plotting points; run the `colors()` function to see what possible colors can be used; also, you can

see these colors at `http://research.stowers-institute.org/efg/R/Color/Chart/index.htm`

- `pch` specifies the plotting characters; below is a list of possible characters



- `cex` specifies the magnification level of the plotting characters, where 1.0 is the default; a value of 1.5 means 50% larger than the default, and a value of 0.5 means 50% smaller than the default

- `lwd` specifies the thickness of plotting points or lines, where 1.0 is the default

- `panel.first = grid()` specifies that grid lines are to be drawn and they should be plotted first before any points. The line types for `lty` are 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, and 6 = twodash; the corresponding words "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash" can be given as well. The default for `grid()` is `col = "lightgray"` and `lty = "dotted"`, which will generally work well.

- These line type specifications are used in other functions too (including `plot()`) with the `lty` argument. A general way to produce any line type is to specify the number of units for a line, space, line, space, ... For example, `"1343"` gives a line of one unit (a dot), a space of 3 units, a line of 4 units, and a space of 3 units. The pattern will subsequently repeat as needed. The `"1343"` is equivalent to `"dotdash"`.

- The `par()` function's Help contains more information about the different plotting options!

Plots can easily be included in a Word document. First, make sure the R Graphics window is the current window in R and then select FILE > COPY TO THE CLIPBOARD > AS A METAFILE. Select the PASTE in Word to import it. You may need to crop the plot to limit the space it takes up.

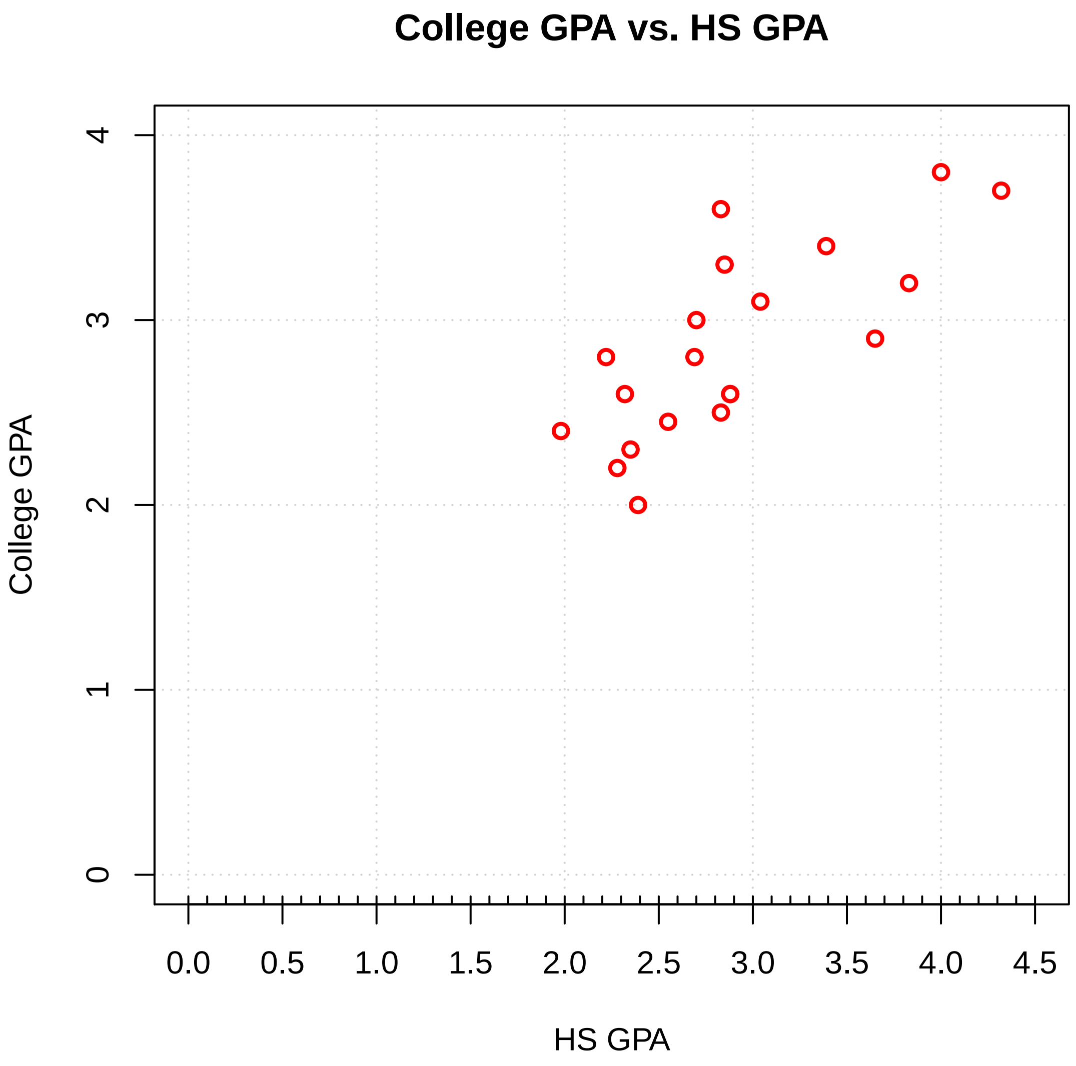
Despite this importation method being quite easy, there can be some distortion introduced through the procedure. The highest quality ways to include a plot in any type of document is to use a PDF or postscript format. Plots can be exported from R into the PDF format by using FILE > SAVE AS > PDF. Alternatively, PDF files can be automatically created by using `pdf()` and `dev.off()`:

```
> # Did not execute because need this specific file
> # location on a drive
> pdf(file = "C:\\figures\\Figure-temp.pdf", width = 6,
    height = 6, colormodel = "cmyk")
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", main = "College GPA vs. HS GPA",
    xlim = c(0, 4.5), ylim = c(0, 4.5), col = "red",
    pch = 1, cex = 1, lwd = 2, panel.first = grid())
> dev.off()
```

All graphics output within `pdf()` and `dev.off()` goes to a PDF file at the location specified rather than to a graphics window.

To obtain specific x-axis or y-axis tick marks on a plot, use the `axis()` function. For example,

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", main = "College GPA vs. HS GPA",
    xlim = c(0, 4.5), ylim = c(0, 4), col = "red",
    pch = 1, cex = 1, lwd = 2, panel.first = grid(),
    xaxt = "n")
> # Major tick marks for x-axis
> axis(side = 1, at = seq(from = 0, to = 4.5, by = 0.5))
> # Minor tick marks for x-axis
> axis(side = 1, at = seq(from = 0, to = 4.5, by = 0.1),
    tck = 0.01, labels = FALSE)
```



Notice the use of `xaxt = "n"` in the `plot()` function. This specifies that no tick marks are to be drawn on the x-axis by `plot()`.

Estimate a model

The `lm()` function estimates linear regression models:

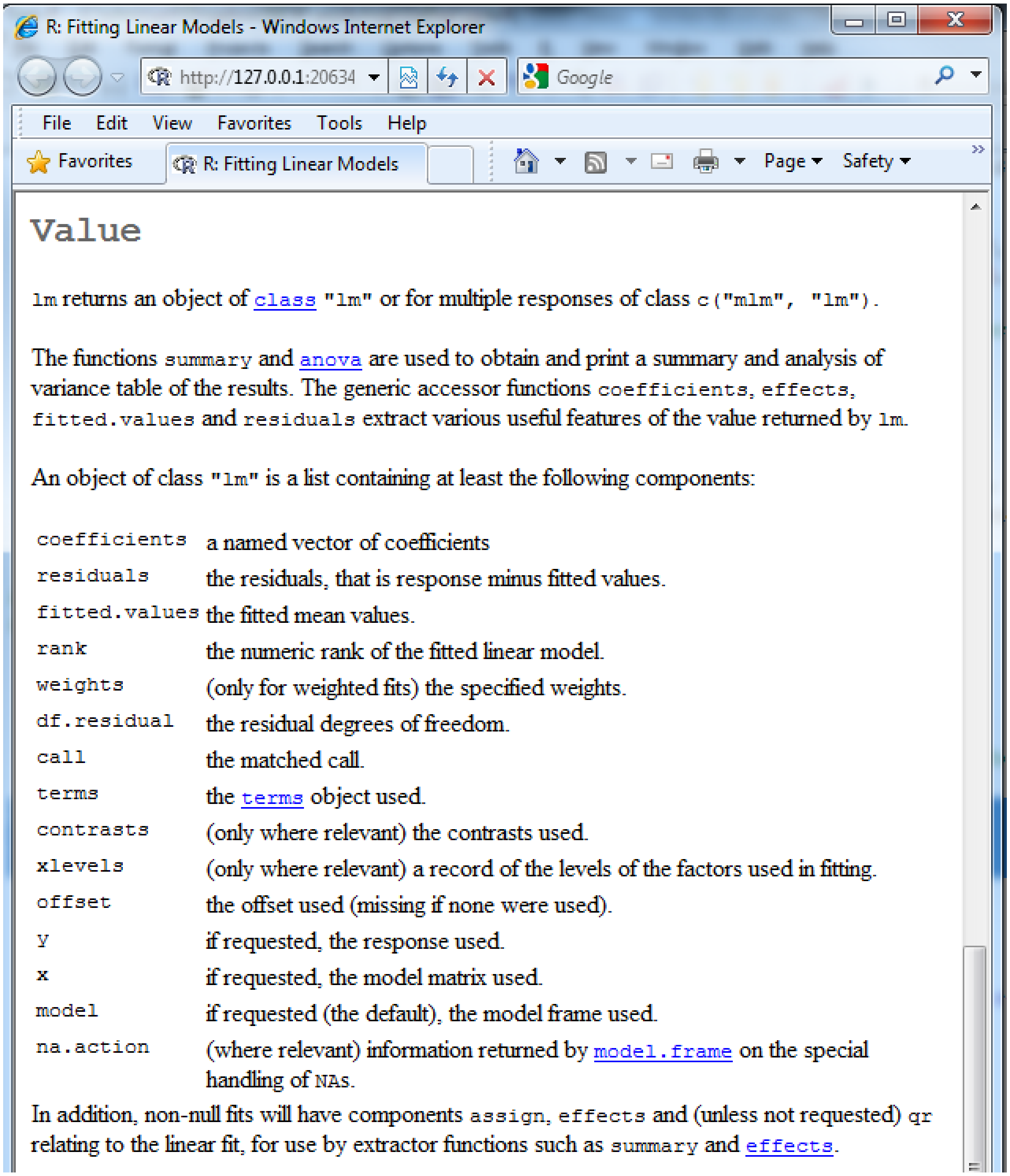
```
> mod.fit <- lm(formula = College.GPA ~ HS.GPA, data = gpa)
> # A very brief look of what is inside of mod.fit
> mod.fit

Call:
lm(formula = College.GPA ~ HS.GPA, data = gpa)

Coefficients:
(Intercept)       HS.GPA
      1.087        0.612
```

The `~` symbol separates the response (dependent) and explanatory (independent) variables within the `formula` argument. If there were more than one explanatory variable, the `+` symbol would be used to separate them.

The results are stored in an object that I decided to call `mod.fit`. By running the `mod.fit` object name only at a command prompt, R prints some information about what is inside of it. To obtain a more thorough listing, use the `names()` function:

```
> names(mod.fit)
 [1] "coefficients"  "residuals"     "effects"       "rank"         
 [5] "fitted.values" "assign"        "qr"            "df.residual"  
 [9] "xlevels"       "call"          "terms"         "model"        
```

The `mod.fit` object is referred to as a *list* in R's terminology. Lists provide a general way to link a number of items together under one object. The linked items do not need to be the same size or type, so lists are often used as the object returned from running more complex functions. A summary of what each item represents within this list is given on the help web page for `lm()`:


R: Fitting Linear Models - Windows Internet Explorer
http://127.0.0.1:20634
Google
File Edit View Favorites Tools Help
Favorites
R: Fitting Linear Models
Page
Safety

Value

`lm` returns an object of `class` `"lm"` or for multiple responses of class `c("mlm", "lm")`.

The functions `summary` and `anova` are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` extract various useful features of the value returned by `lm`.

An object of class `"lm"` is a list containing at least the following components:

`coefficients`	a named vector of coefficients
`residuals`	the residuals, that is response minus fitted values.
`fitted.values`	the fitted mean values.
`rank`	the numeric rank of the fitted linear model.
`weights`	(only for weighted fits) the specified weights.
`df.residual`	the residual degrees of freedom.
`call`	the matched call.
`terms`	the `terms` object used.
`contrasts`	(only where relevant) the contrasts used.
`xlevels`	(only where relevant) a record of the levels of the factors used in fitting.
`offset`	the offset used (missing if none were used).
`y`	if requested, the response used.
`x`	if requested, the model matrix used.
`model`	if requested (the default), the model frame used.
`na.action`	(where relevant) information returned by `model.frame` on the special handling of `NA`s.

In addition, non-null fits will have components `assign`, `effects` and (unless not requested) `qr` relating to the linear fit, for use by extractor functions such as `summary` and `effects`.

To access part of the list, use the syntax `<list>$<component>`. This is the same syntax used with a data frame, because a data frame is a special type of list (each component is a vector of the same length). Below are a couple of examples with the `mod.fit` object:

```
> mod.fit$coefficients
(Intercept)      HS.GPA
     1.0869      0.6125
> mod.fit$residuals
       1        2        3        4        5        6        7        8
 0.15114 -0.22624  0.25939 -0.19874 -0.32024 -0.03285  0.23677  0.09213
       9       10       11       12       13       14       15       16
 0.06551  0.77976 -0.55074 -0.42248  0.46751 -0.23273  0.35338  0.10038
      17       18       19       20
-0.25086  0.26314 -0.28337 -0.25086
```

We can combine some of these items together into one data frame to summarize the model's fit:

```
> save.fit <- data.frame(gpa, College.GPA.hat = round(mod.fit$fitted.values,
     2), residuals = round(mod.fit$residuals, 2))
> head(save.fit)
  HS.GPA College.GPA College.GPA.hat residuals
1   3.04        3.10            2.95      0.15
2   2.35        2.30            2.53     -0.23
3   2.70        3.00            2.74      0.26
4   2.55        2.45            2.65     -0.20
5   2.83        2.50            2.82     -0.32
6   4.32        3.70            3.73     -0.03
```

The `summary()` function can be used with the `mod.fit` object to summarize the list's contents:

```
> summary(object = mod.fit)

Call:
lm(formula = College.GPA ~ HS.GPA, data = gpa)

Residuals:
    Min      1Q  Median      3Q     Max
-0.5507 -0.2509  0.0163  0.2424  0.7798

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)    1.087      0.367    2.96   0.0083 **
HS.GPA         0.612      0.124    4.95   0.0001 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.344 on 18 degrees of freedom
Multiple R-squared:  0.577,Adjusted R-squared:  0.553
F-statistic: 24.5 on 1 and 18 DF,  p-value: 0.000103
```

Notice the different results that we received here from what we received earlier with `summary(object = gpa)`! We will discuss soon why the same function produces different results. From using the output, we can see that the estimated regression model is

$$\hat{Y} = 1.0869 + 0.6125x.$$

where x is the high school GPA and $\hat{Y}$ is the estimated college GPA. Less formally, we could have also stated the model as

$$\widehat{\text{College}} = 1.0869 + 0.6125 \times (\text{HighSchool}).$$

What if there was a categorical explanatory variable? R automatically creates indicator variables to represent it in a model, where the "set first level equal to 0" type of coding is performed (SAS does "set last level equal to 0"). Below is a short example:

```
> where.live <- c("with parents", "dorm", "off-campus")
> x <- rep(x = where.live, each = 7)
> gpa3 <- data.frame(gpa, where.live = x[-21])
> head(gpa3)
  HS.GPA College.GPA   where.live
1   3.04        3.10 with parents
2   2.35        2.30 with parents
3   2.70        3.00 with parents
4   2.55        2.45 with parents
5   2.83        2.50 with parents
6   4.32        3.70 with parents
> levels(gpa3$where.live)
```

```
[1] "dorm"         "off-campus"   "with parents"
> contrasts(gpa3$where.live)
             off-campus with parents
dorm                  0            0
off-campus            1            0
with parents          0            1
> mod.fit3 <- lm(formula = College.GPA ~ HS.GPA + where.live, data = gpa3)
> summary(mod.fit3)

Call:
lm(formula = College.GPA ~ HS.GPA + where.live, data = gpa3)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.5795 -0.2447  0.0118  0.2605  0.7513 

Coefficients:
                       Estimate Std. Error t value Pr(>|t|)    
(Intercept)              1.1169     0.4166    2.68  0.01640 *  
HS.GPA                   0.6119     0.1339    4.57  0.00031 ***
where.liveoff-campus    -0.0399     0.2047   -0.19  0.84787    
where.livewith parents  -0.0471     0.1948   -0.24  0.81219    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 0.364 on 16 degrees of freedom
Multiple R-squared:  0.579,Adjusted R-squared:   0.5 
F-statistic: 7.32 on 3 and 16 DF,  p-value: 0.00262
```

R uses the ordering given by `levels()` to determine the indicator variables. This ordering will be alphabetical (lowercase before uppercase) unless specified otherwise. Thus, the base level is "dorm". The estimated regression model is

$$\begin{aligned}\widehat{\text{College}} &= 1.1169 + 0.6119(\text{HighSchool}) - 0.0399\text{OffCampus}\\ &\quad -0.0471\text{Parents},\end{aligned}$$

where OffCampus $= 1$ for living off-campus and $= 0$ otherwise and Parents $= 1$ for living with parents and $= 0$ otherwise.

If a categorical explanatory variable is coded as a number, you need to specify it is categorical within `lm()`. This is done by using `factor(<variable>)` in the `formula` argument as well. For example, suppose `gpa$where.live` had the levels of 1, 2 and 3. The formula argument would be

```
College.GPA ~ HS.GPA + factor(where.live).
```

The gpa.R program provides an example. Alternatively, one could create a new variable in the data frame with

```
gpa3$where.live.new <- factor(x[-21])
```

and use

```
College.GPA ~ HS.GPA + where.live.new
```

for the `formula` argument.

Transformations of explanatory variables can be included within the `formula` argument. For some transformations, the `I()` function needs to be used to tell R how to interpret the transformation. For example, suppose we would like the main effect and quadratic term in the model. The `formula` argument would be:

```
formula = College.GPA ~ HS.GPA + I(HS.GPA^2)
```

The reason for this extra function is because a formula argument like

```
formula = Y ~ (X1 + X2)^2
```

is the syntax for R to estimate a model with main effects and an interaction term for the model

$$E(Y) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2.$$

Alternative ways to estimate this same model include:

```
formula = Y ~ X1 + X2 + X1:X2
```

and

```
formula = Y ~ X1*X2
```

Objected-oriented language

Information (i.e., model specifications, estimates, test results) created by functions is stored within an object. Different collections of information are created by functions depending on the types of calculations that are performed. To distinguish objects that contain different collections of information, R assigns each object an attribute called a *class*. You can view them by using the `attributes()` or `class()` functions:

```
> class(gpa)
[1] "data.frame"
> class(gpa$HS.GPA)
[1] "numeric"
> class(lm)
[1] "function"
> class(mod.fit)
[1] "lm"
```

Functions are typically designed to operate on only one or very few classes of objects. However, some functions, like `summary()`, are *generic*, in the sense that essentially different versions of them have been constructed to work with different classes of objects

When a generic function is run with an object, R first checks the object's class type and then looks to find a *method* function with the name format `<generic function>.<class name>`. Below are examples for `summary()`:

- `summary(mod.fit)` – The function `summary.lm()` summarizes the regression model
- `summary(gpa)` – The function `summary.data.frame()` summarizes the data frame's contents
- `summary.default()` – R attempts to run this function if there is no method function for a class

There are many generic functions! For example, `plot()` is a generic function (try `plot(mod.fit)` to see what happens!). We will also see other generic functions like `predict()` later in the notes.

Why is R set-up like this? The purpose of generic functions is to use a familiar language set with any object. For example, we frequently want to summarize data or a model fit (`summary()`), plot data (`plot()`), and find predictions (`predict()`), so it is convenient to use the same language set no matter the application. This is why R is referred to as an object-oriented language. The object class type determines the function action. Understanding generic functions may be one of the most difficult topics for new R users!

To see a list of all method functions associated with a class, use `methods(class = <class name>)`. For the regression example, the method functions associated with the `lm` class are:

```
> methods(class = "lm")
 [1] add1.lm*           alias.lm*          anova.lm
 [4] case.names.lm*     confint.lm*        cooks.distance.lm*
 [7] deviance.lm*       dfbeta.lm*         dfbetas.lm*
[10] drop1.lm*          dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*
[16] hatvalues.lm       influence.lm*      kappa.lm
[19] labels.lm*         logLik.lm*         model.frame.lm
[22] model.matrix.lm    nobs.lm*           plot.lm
[25] predict.lm         print.lm           proj.lm*
[28] qr.lm*             residuals.lm       rstandard.lm
[31] rstudent.lm        simulate.lm*       summary.lm
[34] variable.names.lm* vcov.lm*

   Non-visible functions are asterisked
```

To see a list of all method functions for a generic function, use `methods(generic.function = <generic function name>)`. Below are the method functions associated with

`summary()`:

```
> methods(generic.function = "summary")
 [1] summary.aov             summary.aovlist
 [3] summary.aspell*         summary.connection
 [5] summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*
 [9] summary.factor          summary.glm
[11] summary.infl            summary.lm
[13] summary.loess*          summary.manova
[15] summary.matrix          summary.mlm
[17] summary.nls*            summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*
[21] summary.POSIXct         summary.POSIXlt
[23] summary.ppr*            summary.prcomp*
[25] summary.princomp*       summary.proc_time
[27] summary.srcfile         summary.srcref
[29] summary.stepfun         summary.stl*
[31] summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
```

Note that one advantage of using RStudio is that you can type "summary" in its help search box to show a list of all functions that start with this word (and thus obtain the method functions).

Knowing what a name of a particular method function can be helpful to find help on it. For example, the help for `summary()` alone is not very helpful! However, the help for `summary.lm()` provides a lot of useful information about what is summarized for a regression model.

Below are a few examples of using generic functions with `mod.fit`:

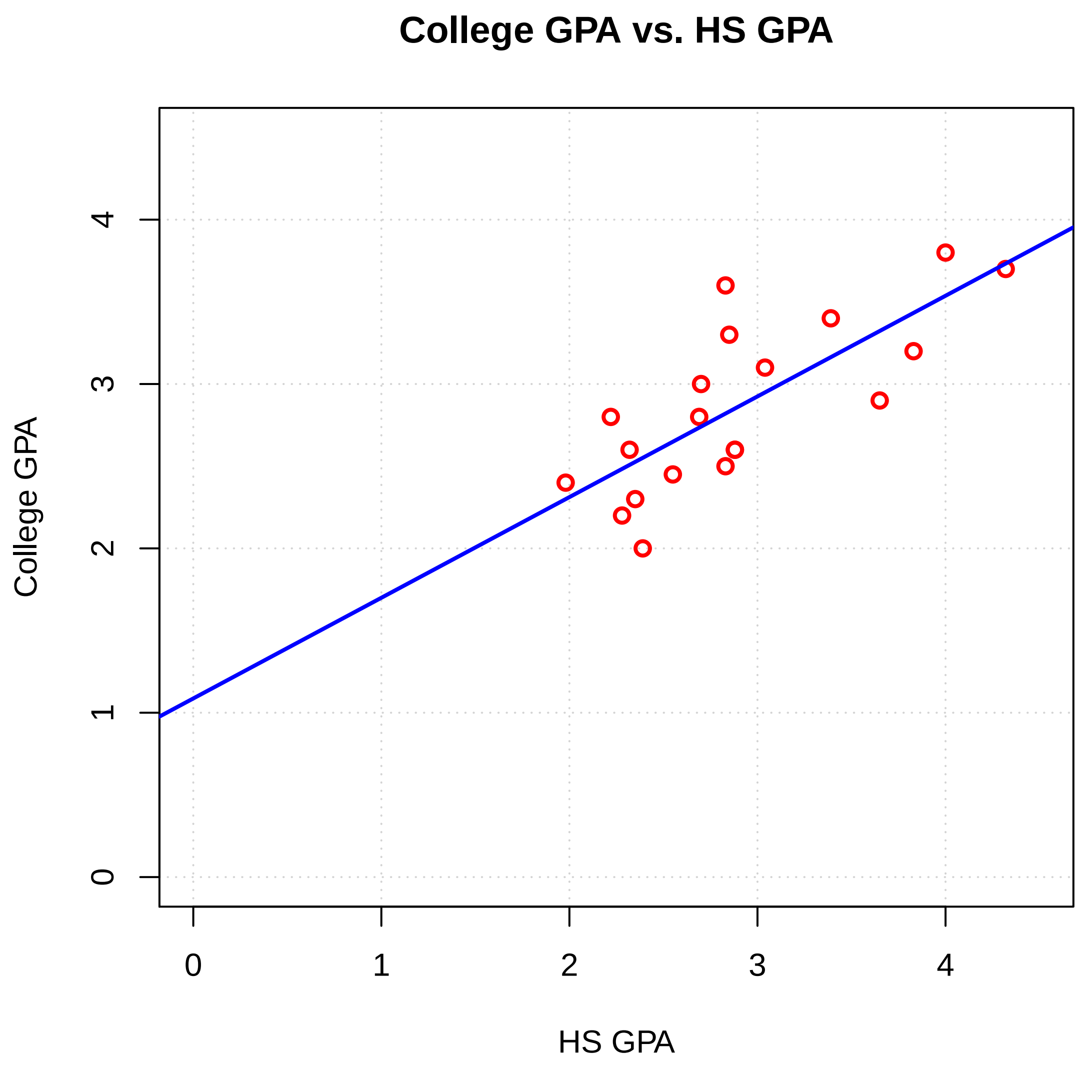
```
> anova(object = mod.fit)
Analysis of Variance Table

Response: College.GPA
          Df Sum Sq Mean Sq F value Pr(>F)
```

```
HS.GPA     1   2.90   2.898    24.5  1e-04 ***
Residuals 18   2.13   0.118
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> vcov(object = mod.fit)
            (Intercept)   HS.GPA
(Intercept)     0.13441 -0.04433
HS.GPA         -0.04433  0.01529
> confint(object = mod.fit, level = 0.95)
             2.5 % 97.5 %
(Intercept) 0.3166 1.8571
HS.GPA      0.3527 0.8723
> AIC(object = mod.fit)
[1] 17.93
> residuals(object = mod.fit)
       1        2        3        4        5        6        7        8
 0.15114 -0.22624  0.25939 -0.19874 -0.32024 -0.03285  0.23677  0.09213
       9       10       11       12       13       14       15       16
 0.06551  0.77976 -0.55074 -0.42248  0.46751 -0.23273  0.35338  0.10038
      17       18       19       20
-0.25086  0.26314 -0.28337 -0.25086
> rstudent(model = mod.fit)
      1       2       3       4       5       6       7       8       9
 0.4416 -0.6793  0.7675 -0.5873 -0.9539 -0.1120  0.7087  0.2742  0.1908
     10      11      12      13      14      15      16      17      18
 2.7070 -1.7703 -1.3415  1.4365 -0.7301  1.0958  0.3104 -0.7394  0.8532
     19      20
-0.8627 -0.7394
```

Estimating the response

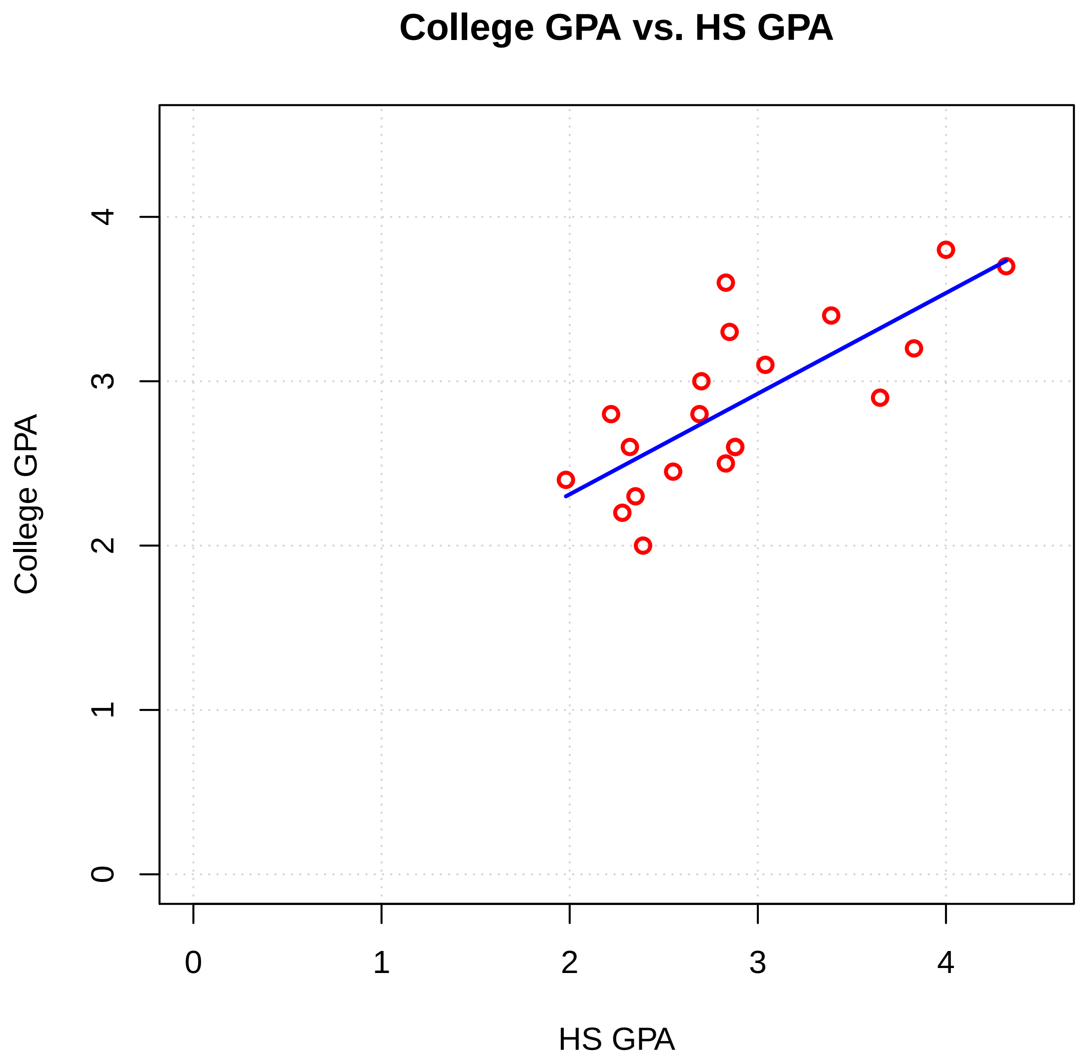
Once a simple linear regression model is found, a common next step is to plot it:

```
> #' While not necessary, new graphics windows can be opened with the
> #'   following functions:
> #' x11(width = 6, height = 6, pointsize = 10)  # General way
> #' win.graph(width = 6, height = 6, pointsize = 10)  # Windows computers only
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", main = "College GPA vs. HS GPA",
    xlim = c(0, 4.5), ylim = c(0, 4.5), col = "red",
    pch = 1, cex = 1, lwd = 2, panel.first = grid())
> # Puts the line y = a + bx on the plot
> abline(a = mod.fit$coefficients[1], b = mod.fit$coefficients[2],
    lty = "solid", col = "blue", lwd = 2)
```



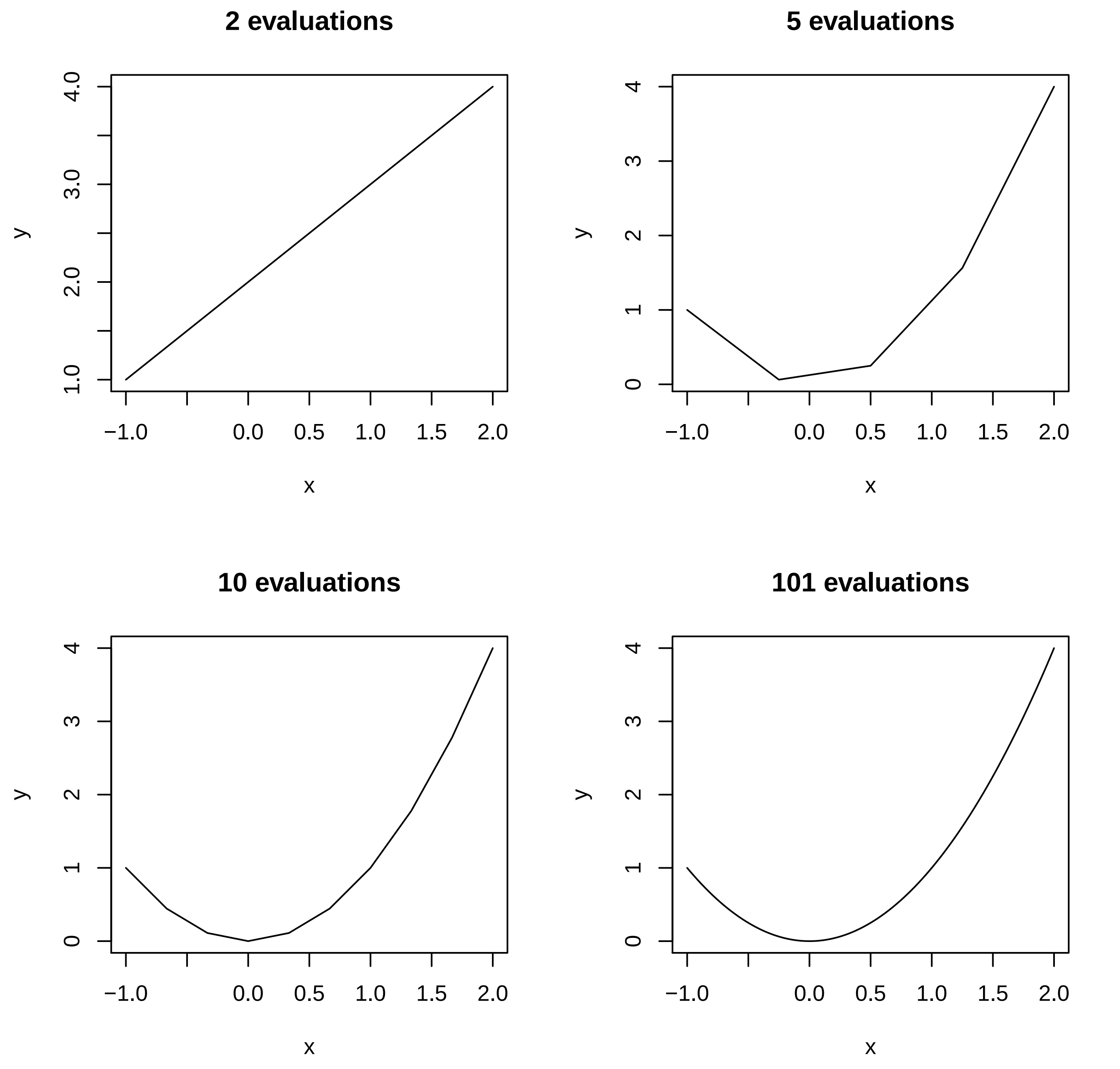
What is a problem with this plot?
Here's a better plot:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", main = "College GPA vs. HS GPA",
    xlim = c(0, 4.5), ylim = c(0, 4.5), col = "red",
    pch = 1, cex = 1, lwd = 2, panel.first = grid())
> curve(expr = mod.fit$coefficients[1] + mod.fit$coefficients[2] *
    x, xlim = c(min(gpa$HS.GPA), max(gpa$HS.GPA)),
    col = "blue", add = TRUE, lwd = 2)
```



The `curve()` function simply draws a mathematical function of "x" by evaluating it a large number of times (default is 101 evenly space values of x), plotting these values of "y", and then connecting the plotted "points" with straight lines. Below is a simple example illustrating the process for the mathematical function $y = x^2$ for $-1 \leq x \leq 2$:

```
> par(mfrow = c(2, 2))  # 2x2 grid of plots
> curve(expr = x^2, xlim = c(-1, 2), n = 2, main = "2 evaluations",
    ylab = "y")
> curve(expr = x^2, xlim = c(-1, 2), n = 5, main = "5 evaluations",
    ylab = "y")
> curve(expr = x^2, xlim = c(-1, 2), n = 10, main = "10 evaluations",
    ylab = "y")
> curve(expr = x^2, xlim = c(-1, 2), n = 101, main = "101 evaluations",
    ylab = "y")
```



```
> par(mfrow = c(1, 1))
```

Another way to draw the estimated regression model is through using the `segments()` function:

```
> # Code not executed; Draw a straight line between (x0, y0)
> # and (x1, y1)
> segments(x0 = min(gpa$HS.GPA), y0 = mod.fit$coefficients[1] +
     mod.fit$coefficients[2] * min(gpa$HS.GPA), x1 = max(gpa$HS.GPA),
     y1 = mod.fit$coefficients[1] + mod.fit$coefficients[2] *
         max(gpa$HS.GPA), lty = "solid", col = "blue", lwd = 2)
```

A more automated way to find estimates of the response is through the generic `predict()` function:

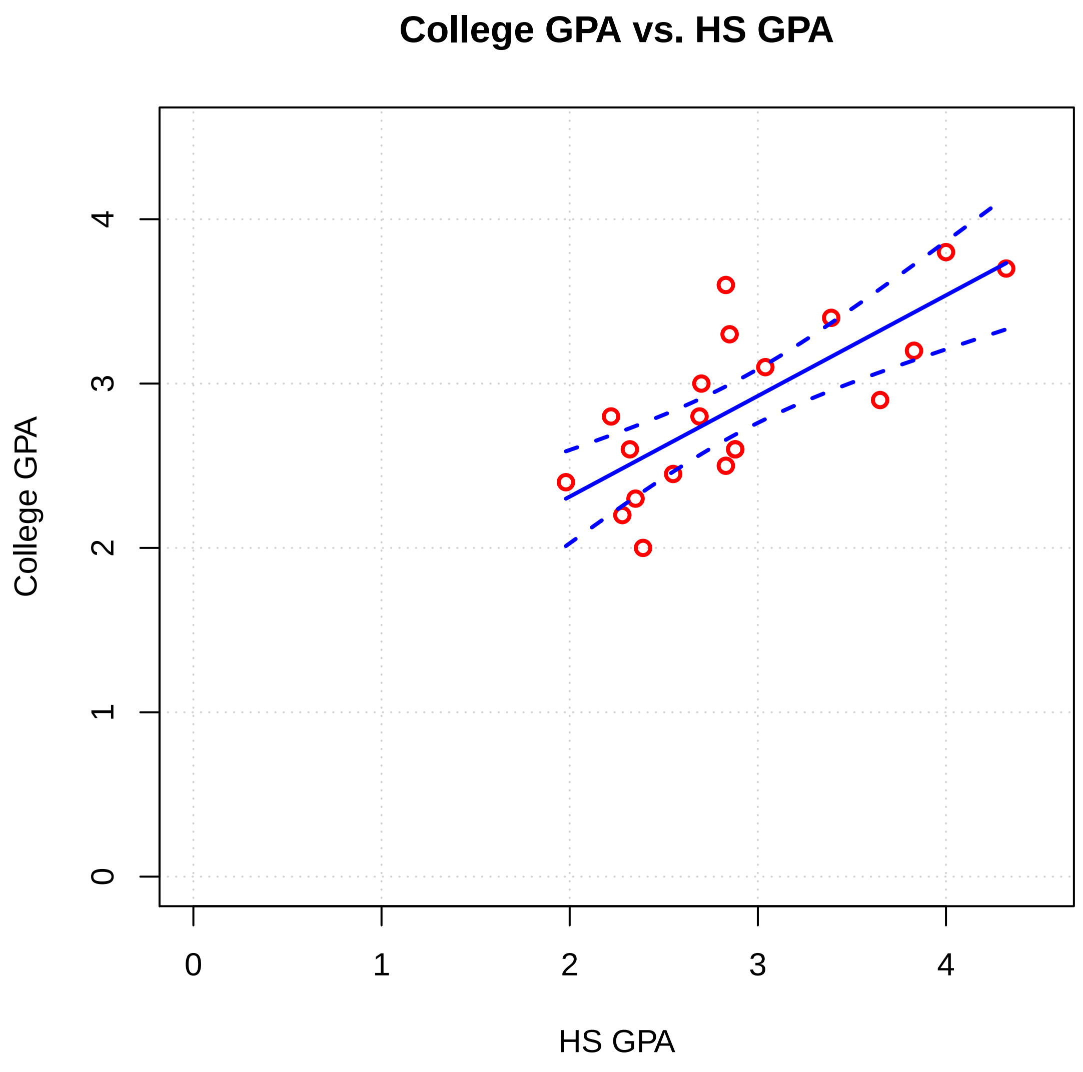
```
> pred.data <- data.frame(HS.GPA = c(2, 3, 4))
> pred.data
  HS.GPA
1      2
2      3
3      4
> predict(object = mod.fit, newdata = pred.data)
    1     2     3
2.312 2.924 3.537
> predict(object = mod.fit, newdata = pred.data, se.fit = TRUE,
     interval = "confidence", level = 0.95)
$fit
    fit   lwr   upr
1 2.312 2.028 2.596
2 2.924 2.761 3.088
3 3.537 3.208 3.865

$se.fit
      1       2       3
0.13514 0.07786 0.15634

$df
[1] 18
```

```
$residual.scale
[1] 0.3437
> save.pred1 <- predict(object = mod.fit, newdata = pred.data,
     interval = "confidence", level = 0.95)
> save.pred1
    fit   lwr   upr
1 2.312 2.028 2.596
2 2.924 2.761 3.088
3 3.537 3.208 3.865
> names(save.pred1)
NULL
> class(save.pred1)  # Not a data frame or list
[1] "matrix"
> save.pred2 <- predict(object = mod.fit, newdata = pred.data,
     se.fit = TRUE, interval = "confidence", level = 0.95)
> names(save.pred2)
[1] "fit"            "se.fit"         "df"             "residual.scale"
> class(save.pred2)
[1] "list"
> save.pred2$fit
    fit   lwr   upr
1 2.312 2.028 2.596
2 2.924 2.761 3.088
3 3.537 3.208 3.865
```

Therefore, the estimated college GPA for a student with a high school GPA of 3 is 2.9244. The 95% confidence interval for the mean college GPA is $2.76 < E(Y) < 3.09$.

The use of the `predict()` function can then be combined with `curve()`:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
     ylab = "College GPA", main = "College GPA vs. HS GPA",
     xlim = c(0, 4.5), ylim = c(0, 4.5), col = "red",
     pch = 1, cex = 1, lwd = 2, panel.first = grid())
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x)),
     col = "blue", add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
         max(gpa$HS.GPA)))
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x),
     interval = "confidence", level = 0.95)[, 2], col = "blue",
     add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
         max(gpa$HS.GPA)), lty = "dashed")
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x),
     interval = "confidence", level = 0.95)[, 3], col = "blue",
     add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
         max(gpa$HS.GPA)), lty = "dashed")
```



The dashed lines are the 95% confidence interval bands for $E(Y)$.

Viewing function code

Typing a function name, like `lm`, and invoking it at a command prompt gives the actual code in the function itself! This is useful when you want to know more about how a function works or if you want to create your own function by modifying the original version. Sometimes, there will be code within the function like .C or .Fortran provided with the R installation. These are calls outside of R to a C or Fortran program. The code within these programs can still be viewed, but they need to be obtained from CRAN.

For new R users, the code within functions can be difficult to understand. The following steps are helpful to interpret the code:

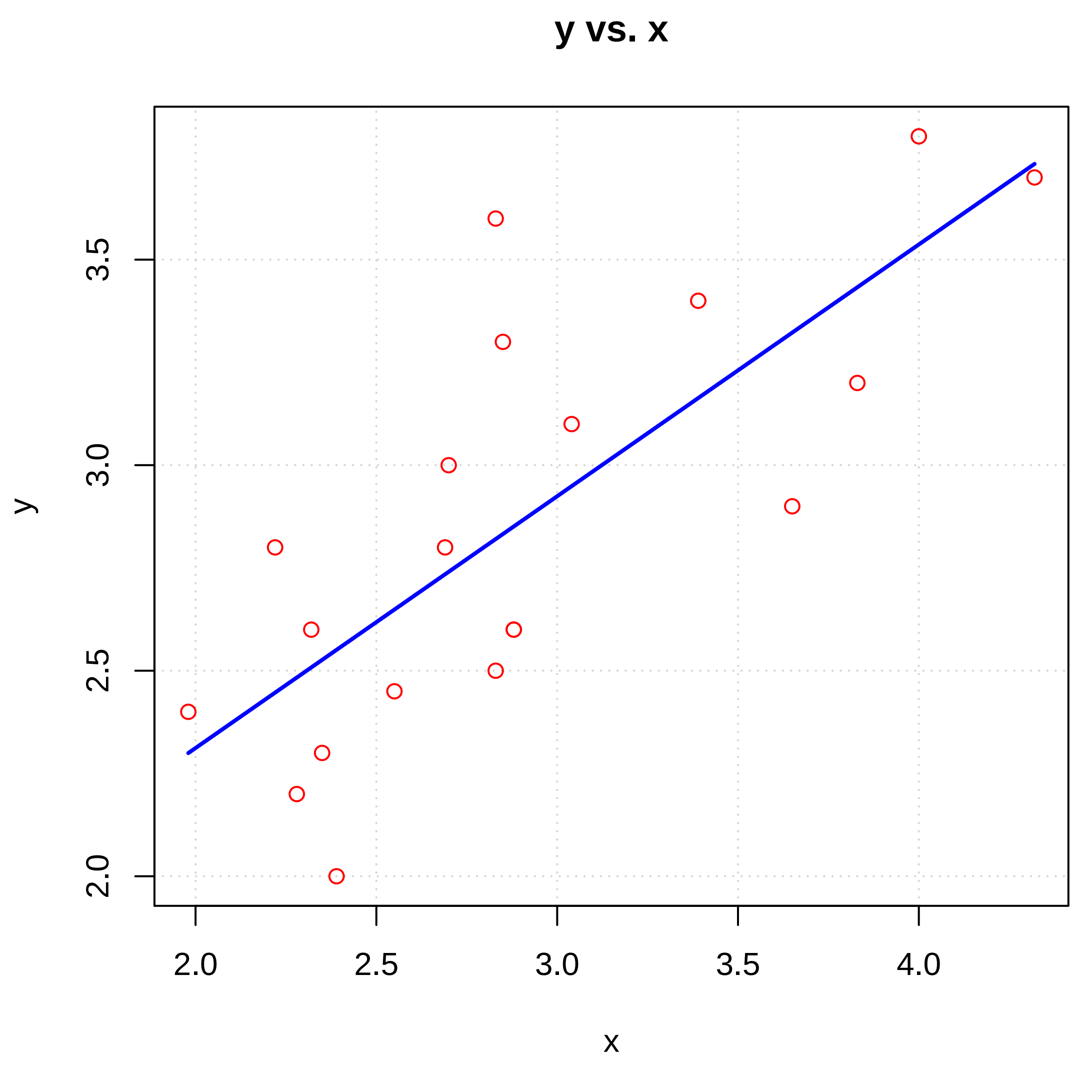
1. Copy and paste the function code into a program editor to view it with syntax highlighting.
2. Set values for the function's arguments.
3. Run the code line-by-line to see what it does!

We will see an example of this soon.

Writing your own functions

When the same code is run for different analyses, it is helpful to write a function for it. For example, below is a simple function written to estimate a regression model and construct a scatter plot with the estimated model:

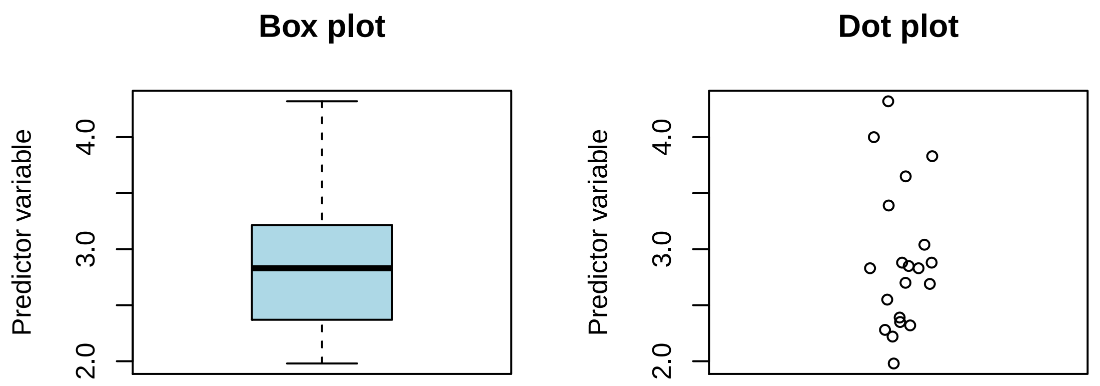
```
> my.reg.func <- function(x, y, data) {
     mod.fit <- lm(formula = y ~ x, data = data)
     plot(x = x, y = y, xlab = "x", ylab = "y", main = "y vs. x",
         col = "red", pch = 1, panel.first = grid())
     curve(expr = mod.fit$coefficients[1] + mod.fit$coefficients[2] *
         x, xlim = c(min(gpa$HS.GPA), max(gpa$HS.GPA)), col = "blue",
         add = TRUE, lwd = 2)
     mod.fit
 }
> # Run the function and save the results
> save.it <- my.reg.func(x = gpa$HS.GPA, y = gpa$College.GPA, data = gpa)
```

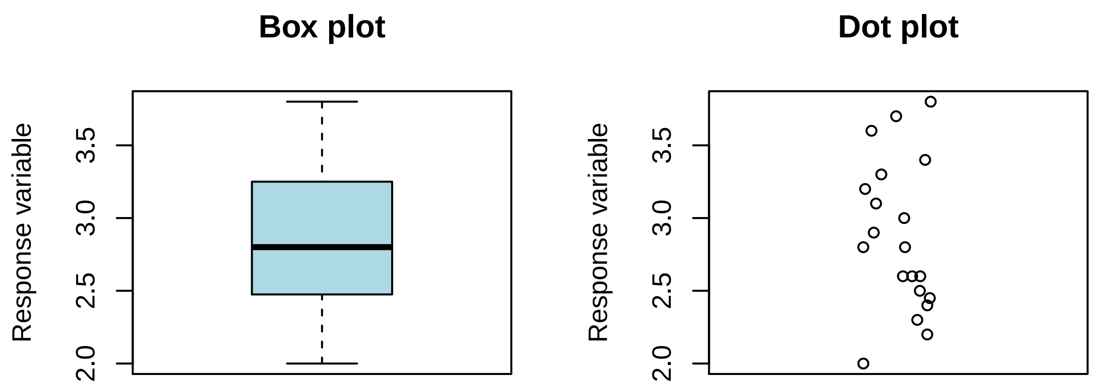


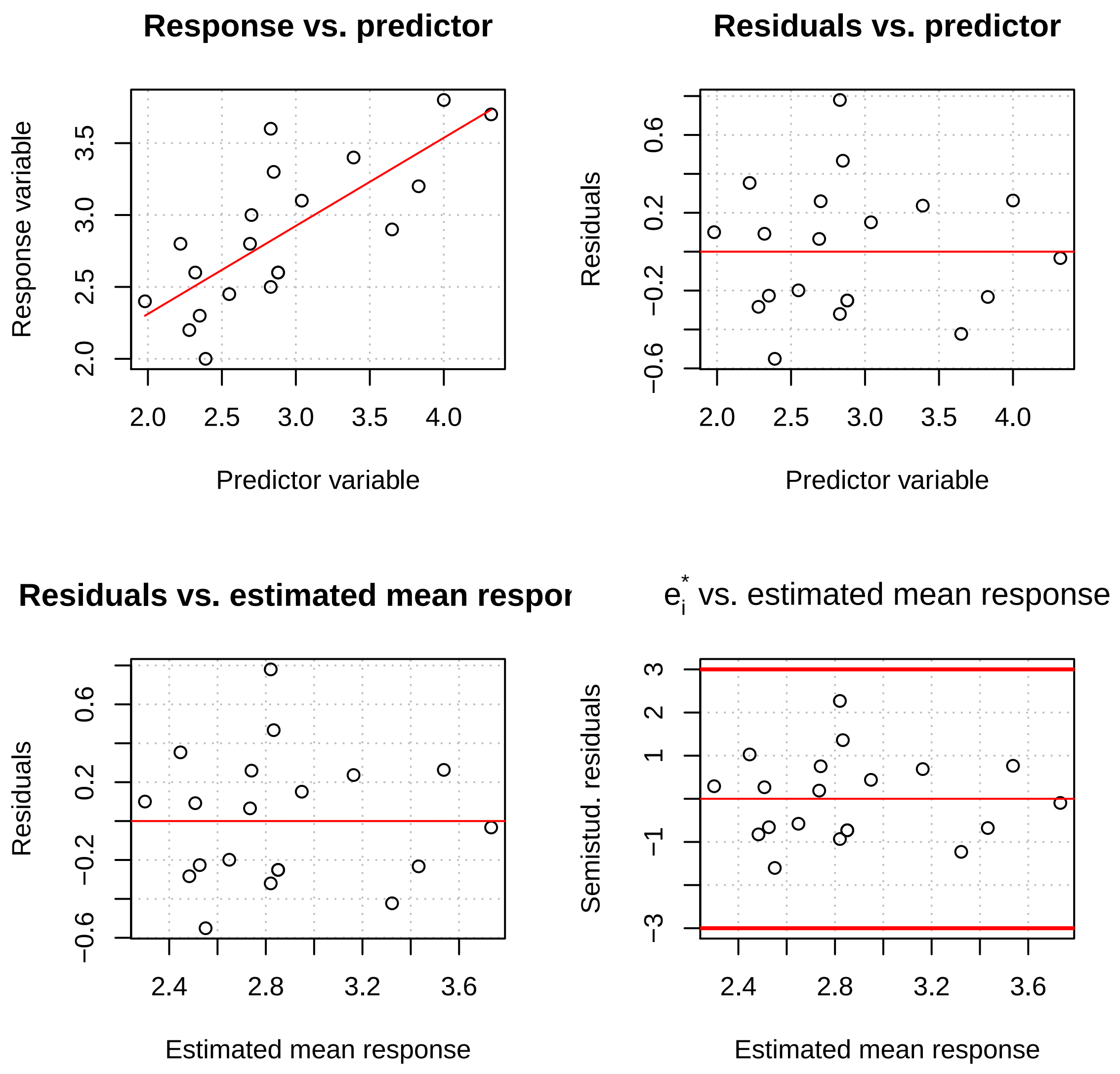
```
> names(save.it)
 [1] "coefficients" "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"       "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> summary(save.it)

Call:
lm(formula = y ~ x, data = data)

Residuals:
    Min      1Q  Median      3Q     Max
-0.5507 -0.2509  0.0163  0.2424  0.7798

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.087      0.367    2.96   0.0083 **
x              0.612      0.124    4.95   0.0001 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.344 on 18 degrees of freedom
Multiple R-squared:  0.577,Adjusted R-squared:  0.553
F-statistic: 24.5 on 1 and 18 DF,  p-value: 0.000103
```

As another example, consider the `examine.mod.simple()` function that I created for a regression course. This function automates the process of examining diagnostic tools for a simple linear regression model. You can see its code in the file examine.mod.simple.R. This code can be run as before or the `source()` function can be used to run it from the program file. Below is an example:

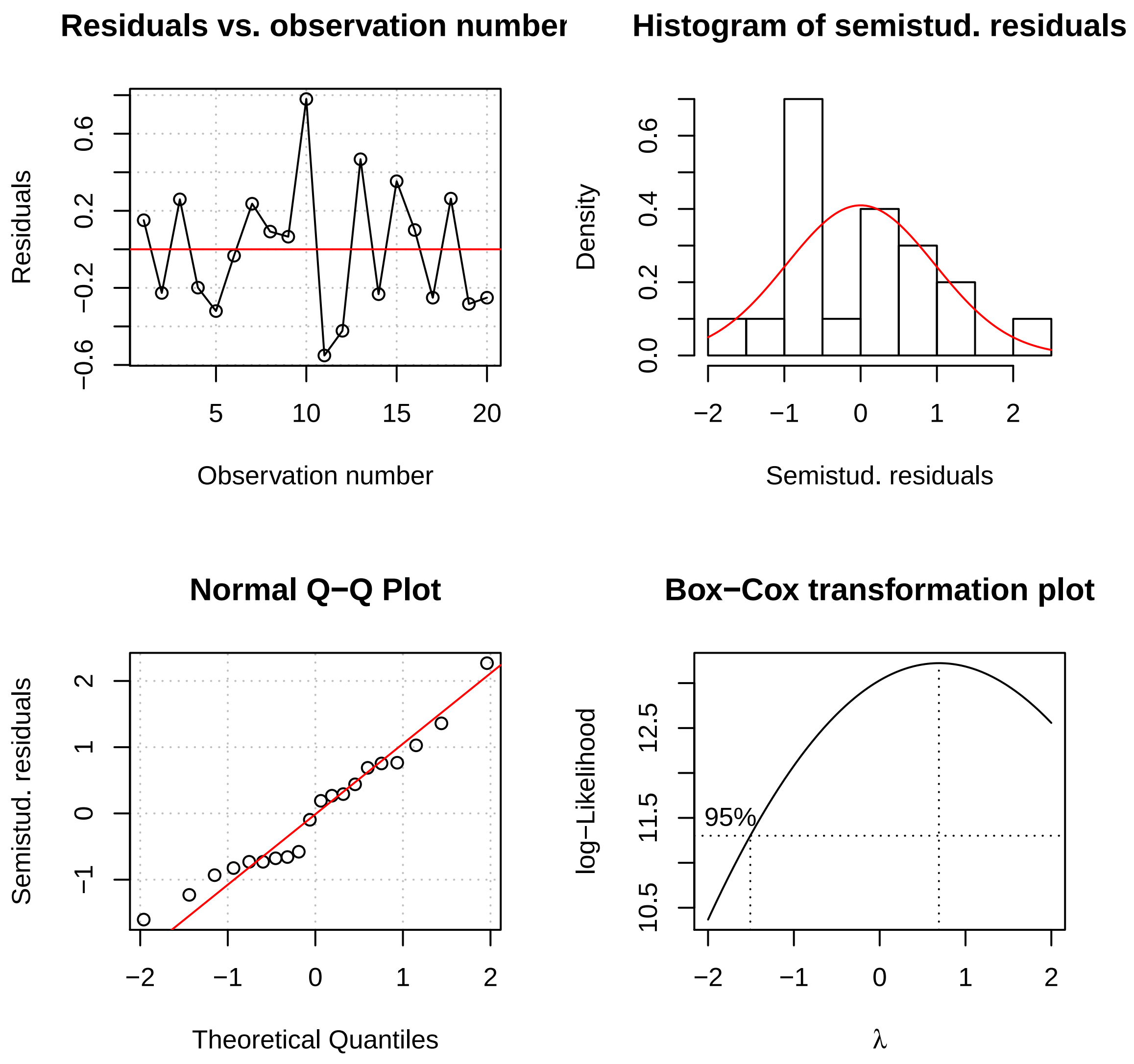
```
> #' Show current folder (directory) R looks for files
> #'   Can change with setwd()
> getwd()
[1] "C:/chris/unl/Dropbox/NEW/STAT850/R/Regression"
> source("examine.mod.simple.R")
> save.it <- examine.mod.simple(mod.fit.obj = mod.fit, const.var.test = TRUE,
     boxcox.find = TRUE)
```


Box plot
Predictor variable
2.0
3.0
4.0
Dot plot
Predictor variable
2.0
3.0
4.0


Box plot
Response variable
2.0
2.5
3.0
3.5
Dot plot
Response variable
2.0
2.5
3.0
3.5



```
Loading required package:  zoo
   Attaching package:  'zoo'
   The following objects are masked from 'package:base':
        as.Date, as.Date.numeric
```



```
> names(save.it)
[1] "sum.data"        "semi.stud.resid" "levene"          "bp"
[5] "lambda.hat"
> save.it$sum.data
       Y               X
 Min.   :2.00   Min.   :1.98
 1st Qu.:2.49   1st Qu.:2.38
 Median :2.80   Median :2.83
 Mean   :2.86   Mean   :2.90
 3rd Qu.:3.23   3rd Qu.:3.13
 Max.   :3.80   Max.   :4.32
```

The `examine.mod.simple()` function was modified in the program here so that separate graphics windows would not open on their own (helps when using LyX with the `knitR` package for creating lecture notes). One can remove the `#` in front of `win.graph()` lines of code in `examine.mod.simple()` to obtain the original version of the function.

Trust in R

Can you trust that R will produce numerically correct results? After all, R is completely open source and all of its underlying code and packages have been written by users.

This was the primary concern by non-R users early on. For example, refer to the quote by a SAS employee given in the Introduction to R section. Also, individuals used to say that the Food and Drug Administration REQUIRES the use of SAS for new drug applications, but this is not true.[1] The R Foundation has a whole document regarding this issue at `https://www.r-project.org/doc/R-FDA.pdf`. Overall, the correctness of results from any statistical software package need to be validated. The previous document talks about this, and there have been journal articles about this issue as well.

Yes, you can trust R, with some caveats. All software packages can have bugs, including SAS. Fortunately, R has now been available for a sufficient period of time with millions of users, so any bugs remaining in its default installation will be extremely minor. You can follow information regarding bug fixes by subscribing to the R Announcements listserv at `https://www.r-project.org/mail.html`. Overall, I trust R's default installation.

Can you trust user-contributed packages that are not in the default installation of R? Here is a summary of my levels of trust with these packages:

- Packages written by leaders in the area of interest: Most likely,

[1] `http://blog.revolutionanalytics.com/2012/06/fda-r-ok.html`

yes

- Packages written by people you trust: Most likely, yes
- Packages that have been peer-reviewed for the *R Journal* or the *Journal of Statistical Software*: Most likely, yes
- Packages from unknown authors: Hopefully
- Packages with version numbers beginning with a 0: Hopefully
- Packages created for a student's dissertation: Hopefully
- Packages just recently created: Hopefully

A higher level of caution should be used with packages falling in the "Hopefully" group. This is why I always focus on using those packages in the R default installation when I teach or perform research. If the default installation does not provide the tools that I need or if tools in other packages are much better, I will then use these other user-contributed packages.

Another concern about user-contributed packages not in the default installation of R is whether these packages will be available a few years from now. Due to changes in R, authors are expected to maintain their package. If a package is no longer maintained sufficiently, it becomes archived or orphaned. Again, this is why when I teach or perform research, I try to at least initially use packages in the default installation of R.

My comments here are not meant to alarm you about the correctness of R. Rather, there are a vast number of R packages contributed by users, and many of them can perform calculations that no other software can; however, these contributed packages need to be used in a judicious manner:

- If possible, initial comparisons should be made between calculations performed by a contributed package and those computed in some other trustworthy way to make sure they agree.

- Because R is open source, the code for all functions is available for users to examine if needed. Line-by-line implementations of code within a function can provide the needed assurances that the code works as desired. No other statistical software that is in wide use offers this opportunity for verification.