Graphics

“Traditional” R plots are created using functions from the `graphics` package. This package is installed in R by default, and its functions are always available for use. The functions within it should be able to satisfy the majority of your needs. The best way to start learning about R graphics is with this package, because many of its basics can be applied to other packages. These other packages, like `lattice` and `ggplot2`, produce most of the same plots, but they can also produce more sophisticated plots.

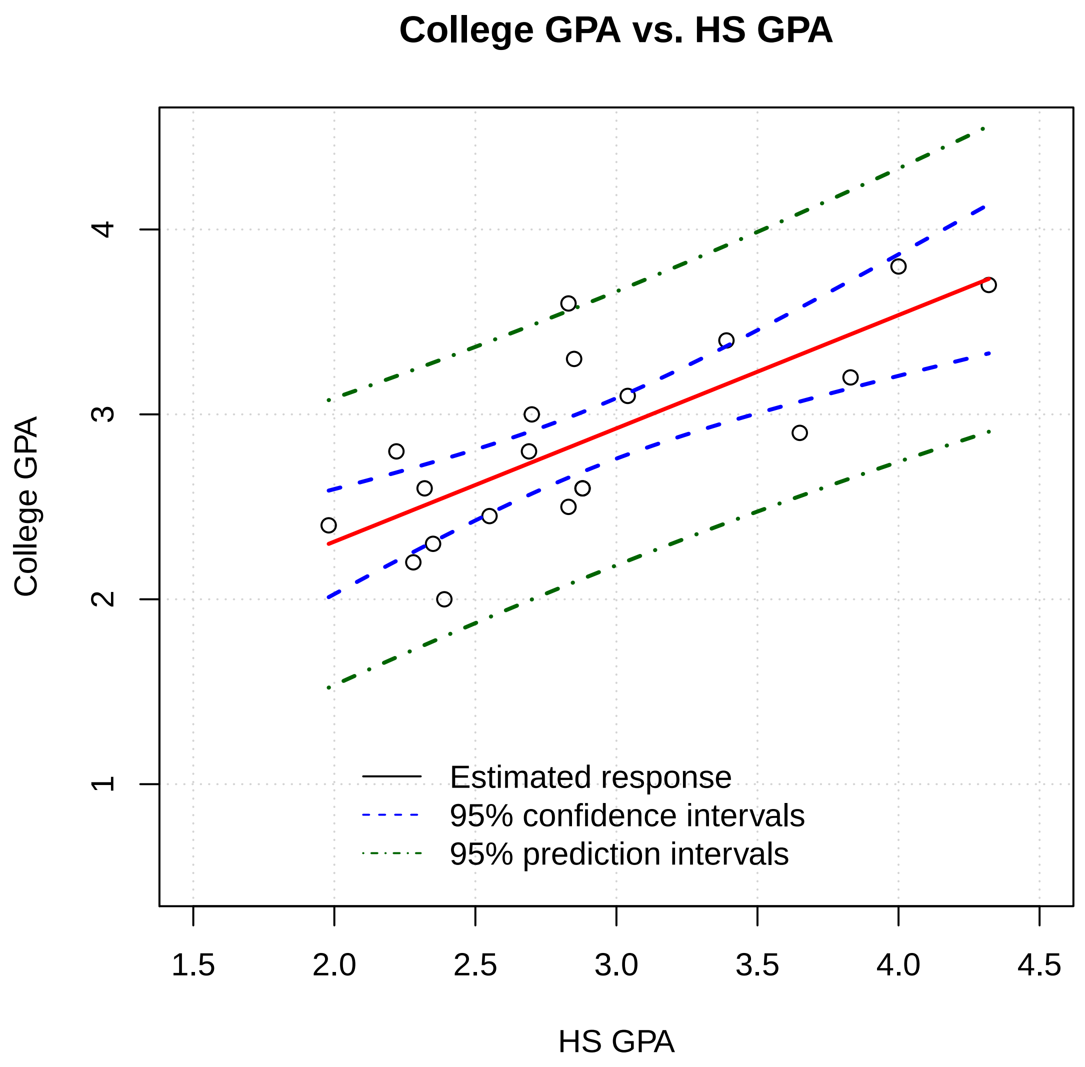
The code used in this section is in gpa_graphics.R

Graphics package

Adding content to plots

Whenever there is more than one type of item being plotted, a legend is needed to differentiate among these items. The `legend()` function provides a general way to add the legend. For example, suppose I would like to construct a plot of the regression model used with the gpa data set and include 95% confidence and prediction interval bands.

```
> # Location is for my computer
> gpa <- read.table(file = "C:\\data\\GPA.txt", header = TRUE,
    sep = "")
> # head(gpa)
> mod.fit <- lm(formula = College.GPA ~ HS.GPA, data = gpa)
> # summary(object = mod.fit)
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", main = "College GPA vs. HS GPA",
    xlim = c(1.5, 4.5), ylim = c(0.5, 4.5), col = "black",
    lwd = 1, panel.first = grid())
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x)),
    col = "red", 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")
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x),
    interval = "prediction", level = 0.95)[, 2], col = "darkgreen",
    add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
       max(gpa$HS.GPA)), lty = "dotdash")
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x),
    interval = "prediction", level = 0.95)[, 3], col = "darkgreen",
    add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
       max(gpa$HS.GPA)), lty = "dotdash")
> legend(x = 2, y = 1.25, legend = c("Estimated response",
    "95% confidence intervals", "95% prediction intervals"),
    bty = "n", col = c("black", "blue", "darkgreen"),
    lty = c("solid", "dashed", "dotdash"))
```



Useful arguments to use with `legend()`:

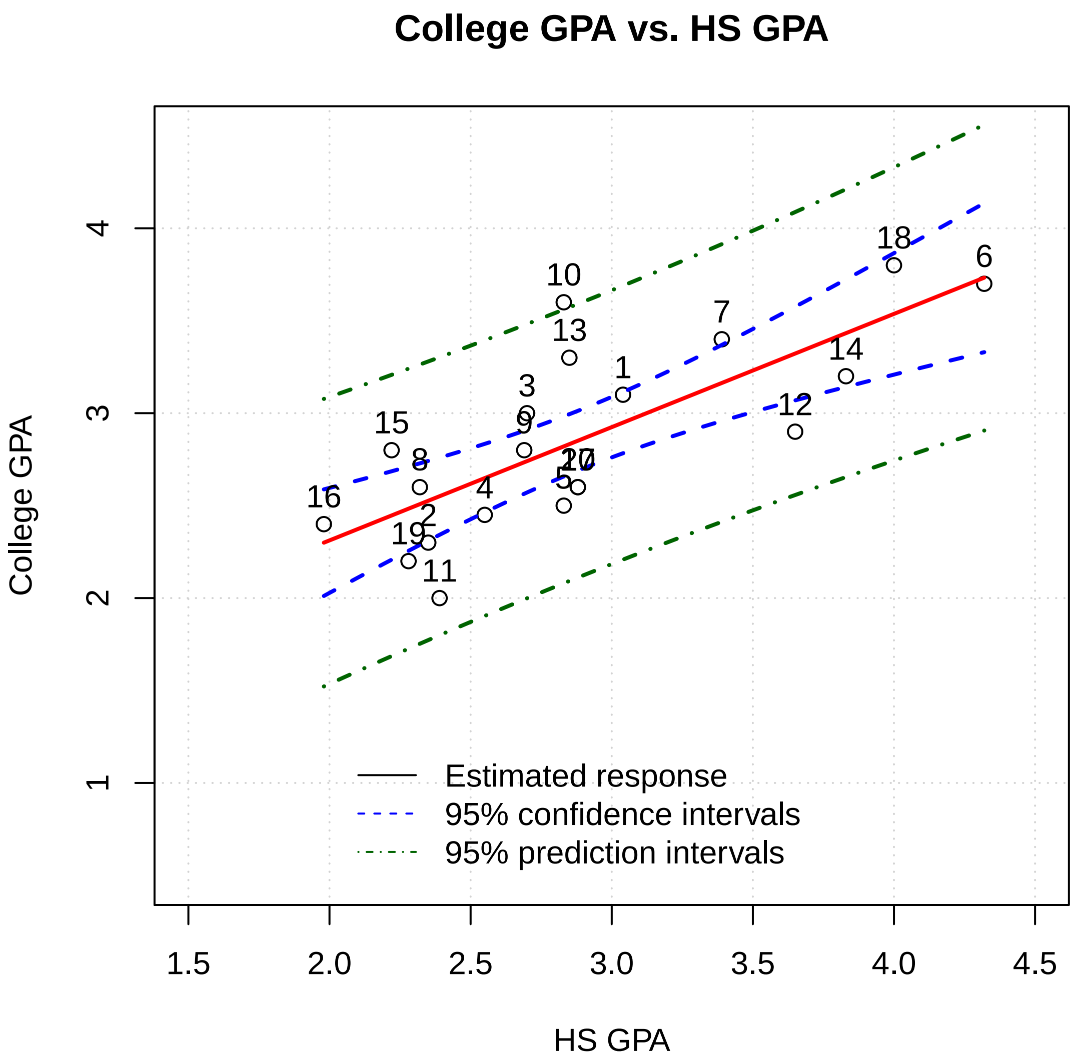
- Rather than specifying a location for the legend with code, one can interactively place it by using the `locator(1)` function. After the `legend()` function is run with this new argument, the mouse cursor will change to a `+` symbol. This is an indication that you need to “add” the legend to the plot by clicking on a location with the mouse.
- Especially when different line types are used on a plot, the `legend()` function may not provide a long enough “sample” of the line types when using its default values. The `seg.len` argument can be used to increase the length of the line segment.

- The `pch` argument needs to be used when you want to include symbols. The syntax is similar to how `lty` was used in the previous code.

If you would like the legend to be in a margin outside of the main plotting area, the `xpd` argument in `par()` can be helpful for this purpose. Please see the program for an example.

The `text()` function can be useful for identifying particular observations on a plot and/or plotting observations with text alone. For example, below is how I identify each observation in the last plot:

```
> text(x = gpa$HS.GPA, y = gpa$College.GPA + 0.15, labels = 1:nrow(gpa))
```



The `identify()` function can perform a similar identification, but in an interactive manner like how `locator(1)` works. For example,

```
> # Code not executed
> identify(x = gpa$HS.GPA, y = gpa$College.GPA, labels = 1:nrow(gpa))
```

leads the mouse cursor to change to a `+` symbol so that one can click on particular observations in the plot to identify with the observation number. When identifying is completed, one can right click on the plot and select STOP from a short-cut menu.

Additional useful functions:

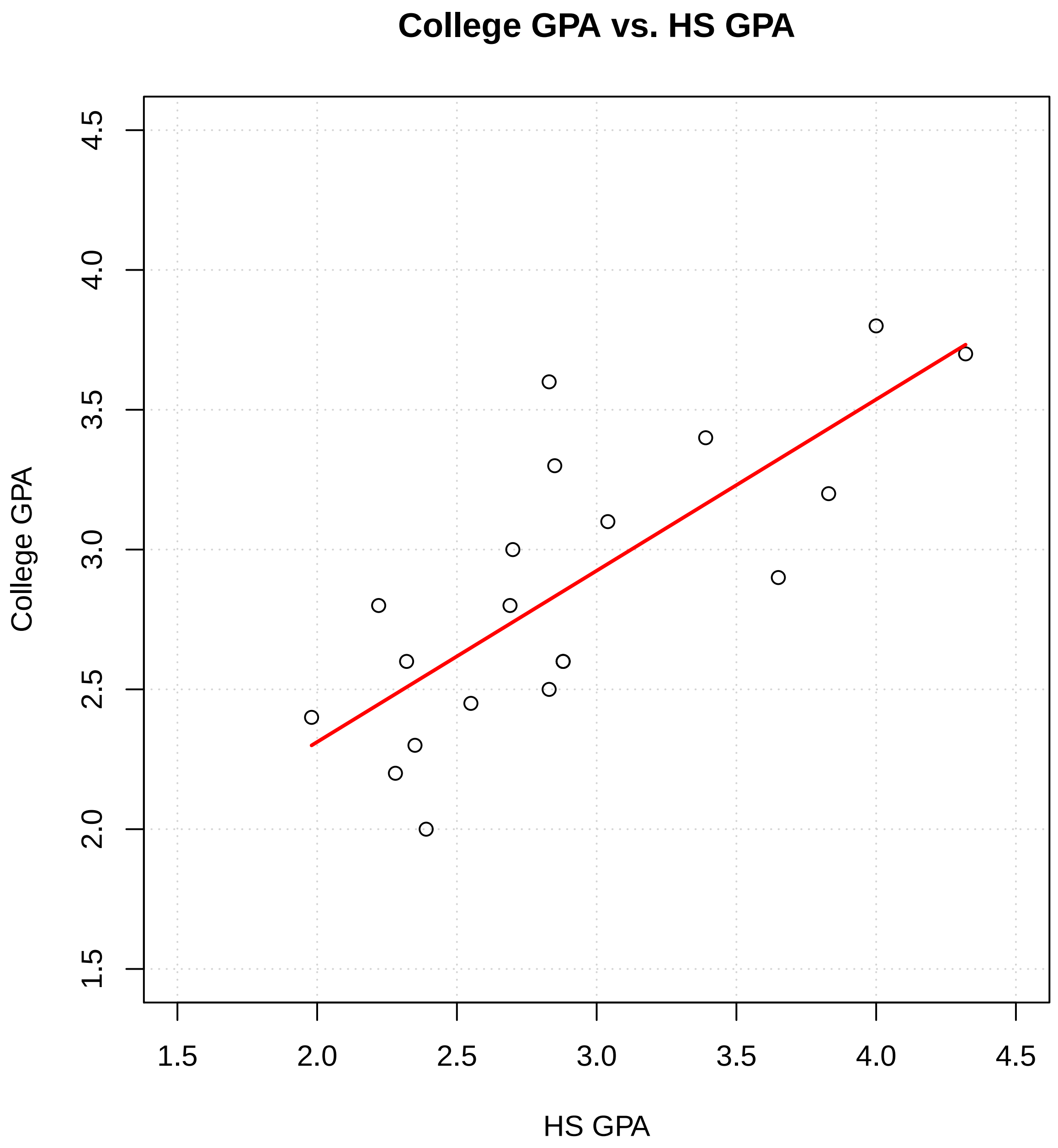
- `points()`: This function adds additional points to a plot. For example, suppose the previous plot involved a sample of GPAs for one year. A second year of data from a new data frame could be added with different plotting symbols by using `points()`.
- `abline()`: In addition to drawing the line $y = a + bx$ on a plot, this function can be used to draw horizontal and vertical lines using either the `h` or `v` arguments in place of `a` and `b`.
- `axis()`: This function allows for finer control of axis tick marks and placement. For example, notice in the previous scatter plots that the axes have different locations of major tick marks (y-axis: 1, 2, 3, 4; x-axis: 1.5, 2.0, ..., 4.5). These tick mark locations are chosen by R. Instead, we can change the y-axis to have the same tick mark locations as with the x-axis. Also, we can take advantage of the somewhat same numerical scale for college and high school GPA by forcing the plot to be square (helps for interpretation of distances between points and lines in the plot).

  ```
  > par(pty = "s")  # Sqaure plot
  > plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
      ylab = "College GPA", main = "College GPA vs. HS GPA",
  ```

```
     xlim = c(1.5, 4.5), ylim = c(1.5, 4.5), col = "black",
     lwd = 1, panel.first = grid(), yaxt = "n")
> seq(from = 1.5, to = 4.5, by = 0.5)  # Create a sequence of numbers

[1] 1.5 2.0 2.5 3.0 3.5 4.0 4.5

> axis(side = 2, at = seq(from = 1.5, to = 4.5, by = 0.5))
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x)),
     col = "red", add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
         max(gpa$HS.GPA)))
```



```
> par(pty = "m")  # Default, maximize area used to draw plot
```

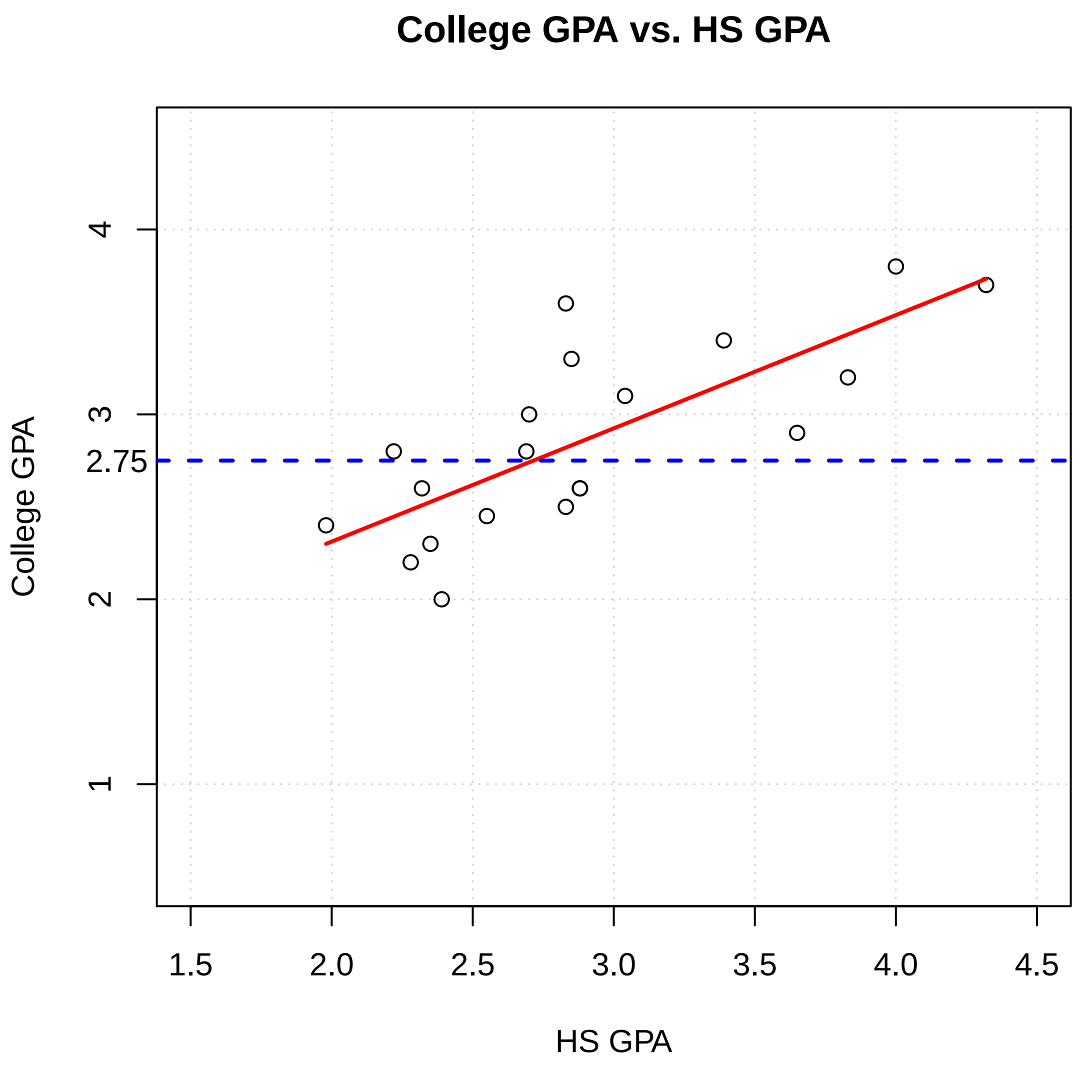
Minor tick marks (smaller ticks without numerical labels) can be created as well using `axis()` again. For example, we could use

```
> axis(side = 2, at = seq(from = 1, to = 5, by = 0.1), tck = -0.01,
    labels = FALSE)
> axis(side = 1, at = seq(from = 1, to = 5, by = 0.1), tck = -0.01,
    labels = FALSE)
```

to add tick marks for both the x and y-axes at 0.1 increments.

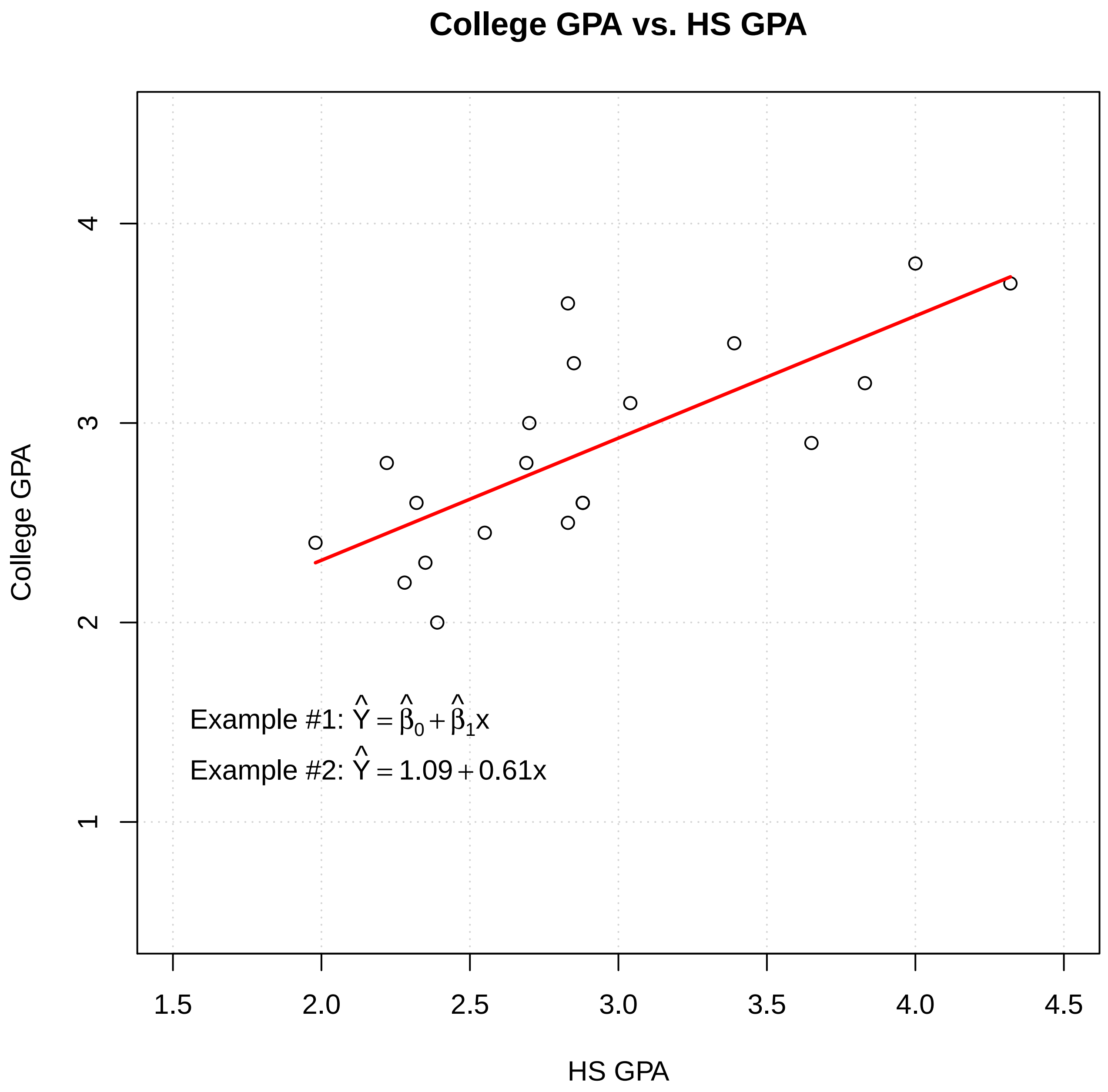
- `mtext()`: This function is used to draw text in the margin of a plot. For example, suppose an estimated college GPA threshold of 2.75 is used by an admission office. A horizontal line drawn and labeled at 2.75 can then be useful.

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", main = "College GPA vs. HS GPA",
    xlim = c(1.5, 4.5), ylim = c(0.5, 4.5), col = "black",
    lwd = 1, panel.first = grid())
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x)),
    col = "red", add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
        max(gpa$HS.GPA)))
> abline(h = 2.75, lty = "dashed", col = "blue", lwd = 2)
> mtext(text = "2.75 ", side = 2, at = 2.75, las = 1)
```



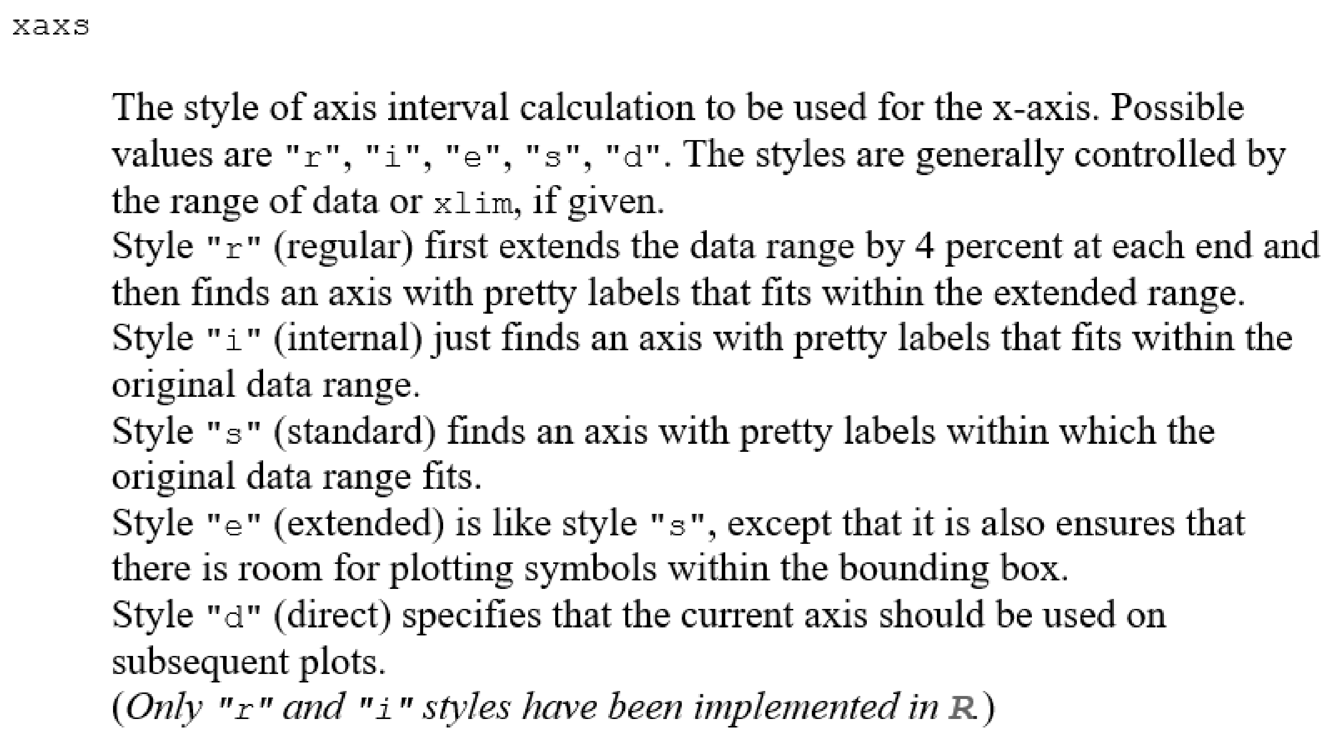
- `expression()` and `substitute()`: Mathematical equations and Greek letters are included on a plot using these functions. The `paste()` function is used to include regular text in these items as well. Below is a simple example:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
     ylab = "College GPA", main = "College GPA vs. HS GPA",
     xlim = c(1.5, 4.5), ylim = c(0.5, 4.5), col = "black",
     lwd = 1, panel.first = grid())
> curve(expr = predict(object = mod.fit, newdata = data.frame(HS.GPA = x)),
     col = "red", add = TRUE, lwd = 2, xlim = c(min(gpa$HS.GPA),
         max(gpa$HS.GPA)))
> # Example of combining text and math expressions:
> text(x = 1.5, y = 1.5, label = expression(paste("Example #1: ",
     hat(Y) == hat(beta)[0] + hat(beta)[1] * x)), pos = 4)
> # Example of including numerical values directly
> # from mod.fit:
> text(x = 1.5, y = 1.25, label = substitute(paste("Example #2: ",
     hat(Y) == betahat0 + betahat1 * x), list(betahat0 = round(mod.fit$coef
     2), betahat1 = round(mod.fit$coefficients[2], 2))),
     pos = 4)
```



Run `demo(plotmath)` at an R Console prompt for help on the code to use with `expression()` and `substitute()`.

- `par()`: As remarked before, the `par()` function contains a plethora of options that control how a plot looks. I strongly encourage students to go through these options and test out what they do! For example, the previous plots in this section have specified x-axis limits with `xlim = c(1.5,4.5)` in `plot()`. However, the x-axis limits actually start a little before 1.5 and a little after 4.5. The reason is due to the default behavior of R. From R's help for `par()`:

  `xaxs`

  The style of axis interval calculation to be used for the x-axis. Possible values are `"r"`, `"i"`, `"e"`, `"s"`, `"d"`. The styles are generally controlled by the range of data or `xlim`, if given.
  Style `"r"` (regular) first extends the data range by 4 percent at each end and then finds an axis with pretty labels that fits within the extended range.
  Style `"i"` (internal) just finds an axis with pretty labels that fits within the original data range.
  Style `"s"` (standard) finds an axis with pretty labels within which the original data range fits.
  Style `"e"` (extended) is like style `"s"`, except that it is also ensures that there is room for plotting symbols within the bounding box.
  Style `"d"` (direct) specifies that the current axis should be used on subsequent plots.
  (*Only `"r"` and `"i"` styles have been implemented in R*)

  By default, R extends the axis limits out by 4% of the data range. To change this behavior to an exact specification or the exact data range, use `par(xaxs = "i")`. Note that one can run `par()` at a command prompt to see the current values set for arguments in the function.

- The escape character `\n` is used to include text onto a second line. For example, use `main = "College GPA vs. \n HS GPA"` in one of the previous `plot()` function calls to see an example of its use.

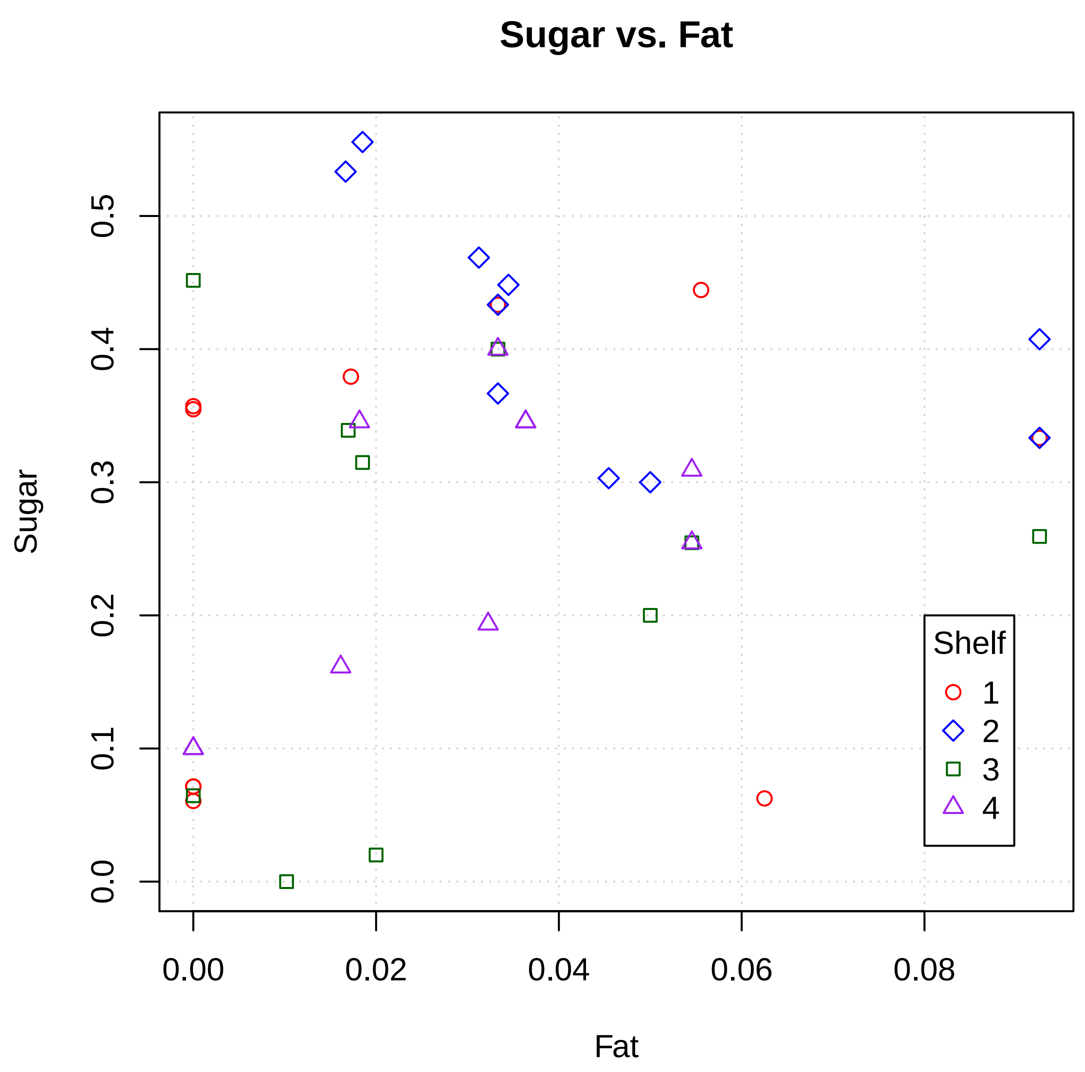
- To change the font, use `par(family = <font name>)`. For example, for the Arial font, use `par(family =`

`windowsFont("Arial"))` on a Windows computer.

Varying symbols within a plot

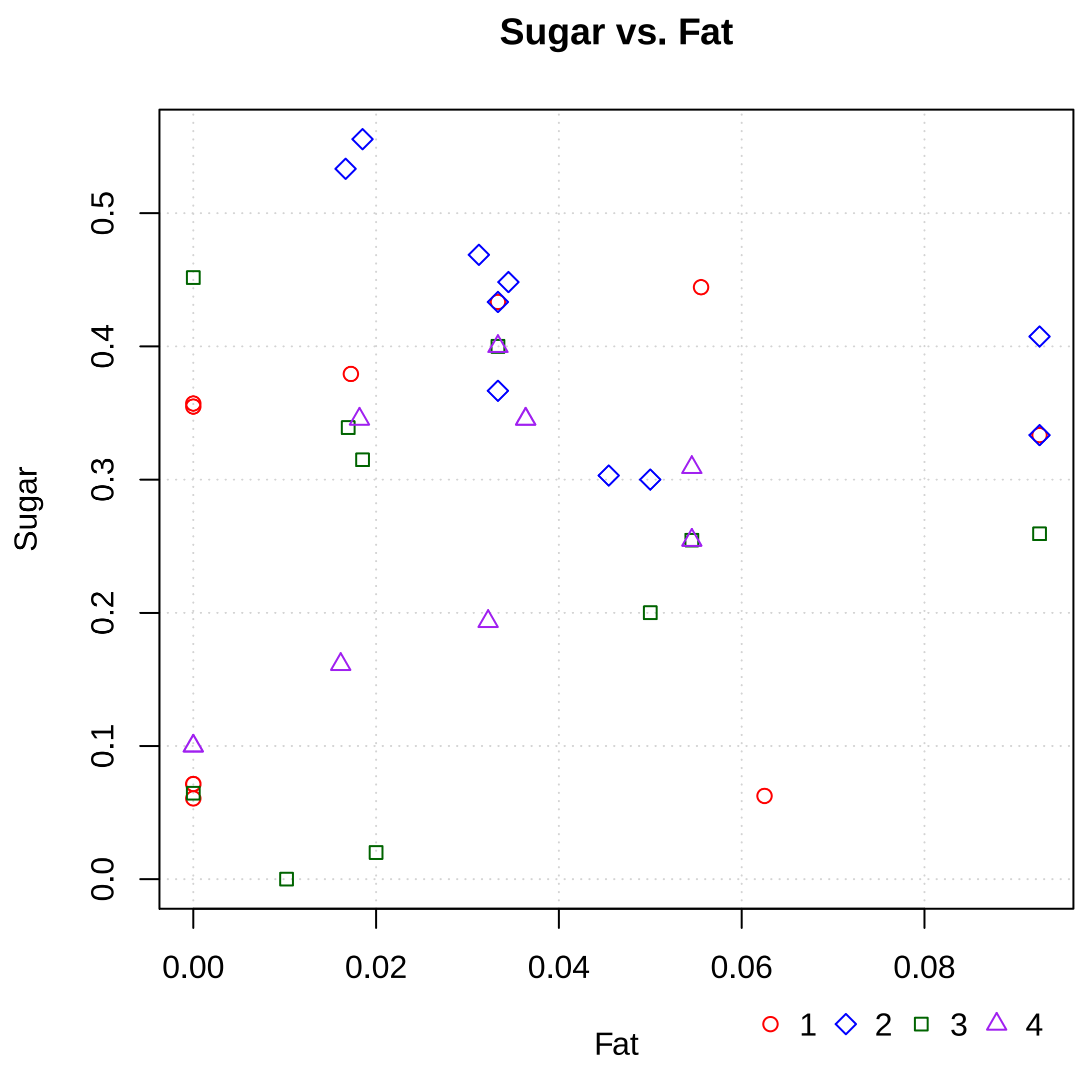
The `col` and `pch` arguments of `plot()` and other functions can have vector values rather than single quantities. Thus, to vary a plotting symbol's color (say, by another variable), provide a value to the `col` argument that is a vector with each element having the color for each plotted point. Below is an example with the cereal data set:

```
> cereal <- read.csv(file = "C:\\data\\cereal.csv")
> cereal$sugar <- cereal$sugar_g/cereal$size_g
> cereal$fat <- cereal$fat_g/cereal$size_g
> cereal$sodium <- cereal$sodium_mg/cereal$size_g
> shelf.color <- rep(x = c("red", "blue", "darkgreen", "purple"),
    each = 10)
> shelf.symbol <- rep(x = c(1, 5, 22, 2), each = 10)
> plot(x = cereal$fat, y = cereal$sugar, xlab = "Fat", ylab = "Sugar",
    main = "Sugar vs. Fat", panel.first = grid(), pch = shelf.symbol,
    col = shelf.color)
> legend(x = 0.08, y = 0.2, legend = 1:4, bty = "y", title = "Shelf",
    col = c("red", "blue", "darkgreen", "purple"), pch = c(1,
        5, 22, 2), bg = "white")
```



Assigning colors and symbols to the levels of `cereal$Shelf` is relatively easy here because the data is sorted by shelf. When the data is not sorted in this manner and/or there are more than a few levels with an unknown quantity of observations per level, this assignment can be more difficult. Below is how the assignment can be done in a more general manner.

```
> # Suppose Shelf is actually a factor
> cereal$Shelf.factor <- as.factor(cereal$Shelf)
> levels(cereal$Shelf.factor)
[1] "1" "2" "3" "4"
> library(plyr)  # Need to install for revalue()
> shelf.color2 <- as.character(revalue(x = cereal$Shelf.factor,
    replace = c(`1` = "red", `2` = "blue", `3` = "darkgreen",
         `4` = "purple")))  # Need character value
> # Even more general to get different colors for each level:
> # shelf.color2 <-
> # palette()[1:length(levels(cereal$Shelf.factor))]
> shelf.symbol2 <- as.integer(as.character(revalue(x = cereal$Shelf.factor,
    replace = c(`1` = 1, `2` = 5, `3` = 22, `4` = 2))))  # Need integer value
> plot(x = cereal$fat, y = cereal$sugar, xlab = "Fat", ylab = "Sugar",
    main = "Sugar vs. Fat", panel.first = grid(), pch = shelf.symbol2,
    col = shelf.color2)
> # Shows how to position legend outside of main plot area
> par(xpd = TRUE)
> legend(x = 0.06, y = -0.08, legend = levels(cereal$Shelf.factor),
    bty = "n", col = c("red", "blue", "darkgreen", "purple"),
    pch = c(1, 5, 22, 2), horiz = TRUE)
```



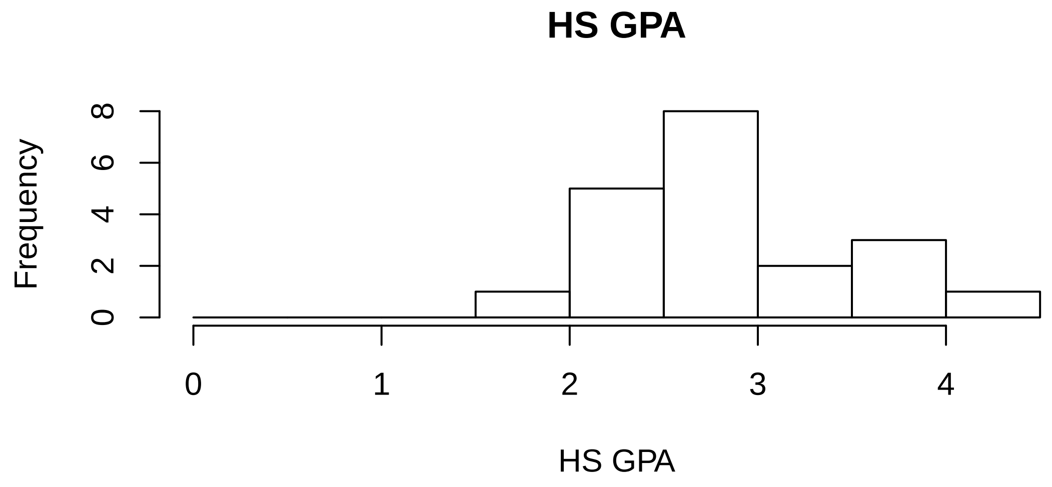
```
> par(xpd = FALSE)
```

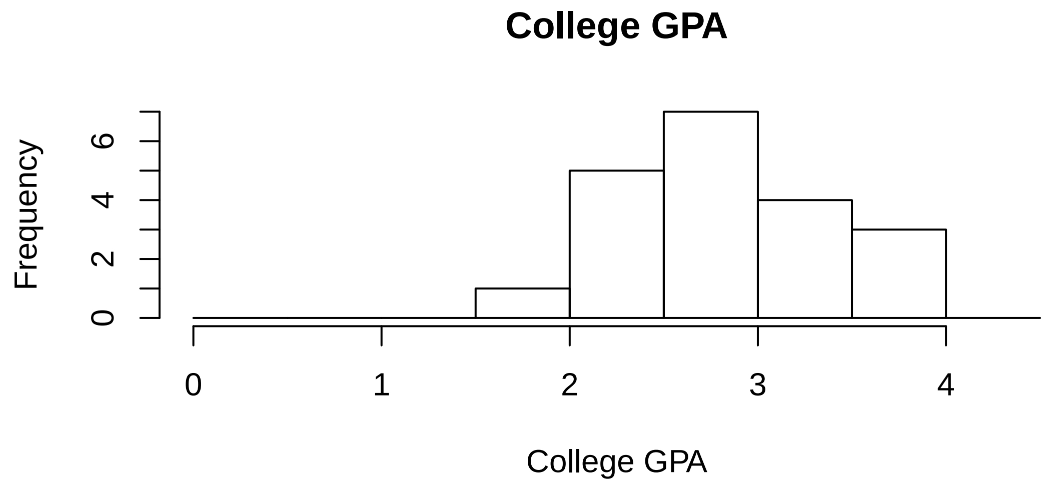
Working with the `revalue()` function in combination with `plot()` was surprisingly difficult. I had to be very careful what class type R needed for the `pch` and `col` arguments of `plot()`. For example, a factor class type used with the `pch` argument did not produce the correct symbols (symbol numbers in a factor were interpreted by their ordering; thus, the "fourth" symbol given by `levels()` was interpreted as symbol style #4)

Useful single variable summary plots

The `hist()` function plots histograms. The code below shows how to include two histograms in one R Graphics window:

```
> par(mfrow = c(2, 1))
> hist(x = gpa$HS.GPA, xlab = "HS GPA", main = "HS GPA", breaks = c(0,
    0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5))
> hist(x = gpa$College.GPA, xlab = "College GPA", main = "College GPA",
    breaks = seq(from = 0, to = 4.5, by = 0.5))
```



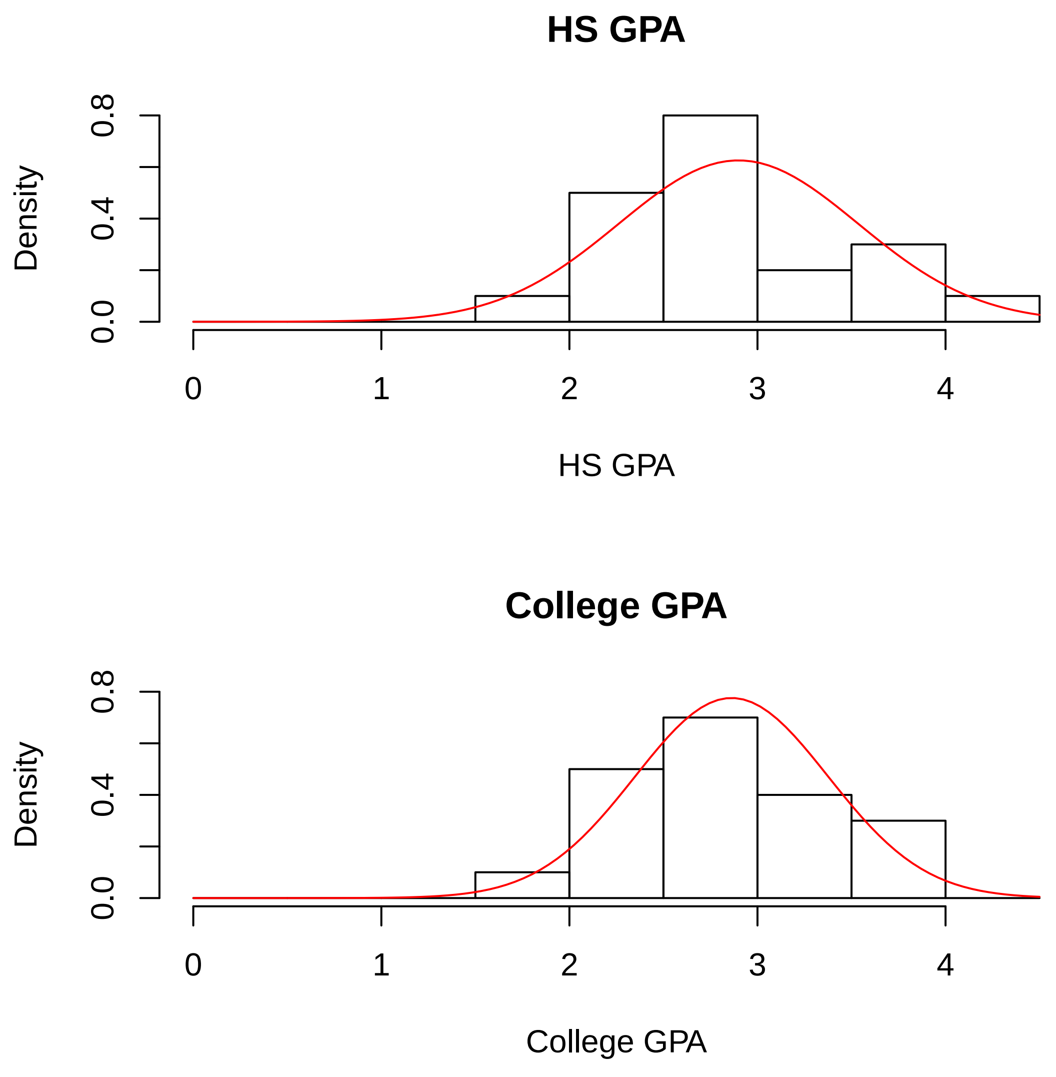


If you do not specify the `breaks` argument, R chooses the histogram classes for you. Usually, R's choice will work well. I chose the classes here to make sure that each histogram has the same classes. The use of both the `c()` function and the `seq()` function was done only for demonstration purposes.

If you want to specify the number of classes for the histogram (rather than breaks), use the `nclass` argument. If the x-axis for both plots did not have the same limits, I would have used the `xlim` argument to make sure they were the same.

We can combine the `hist()` function with the `curve()` function to produce a histogram with a probability density function overlay. Below is an example with a normal distribution approximation:

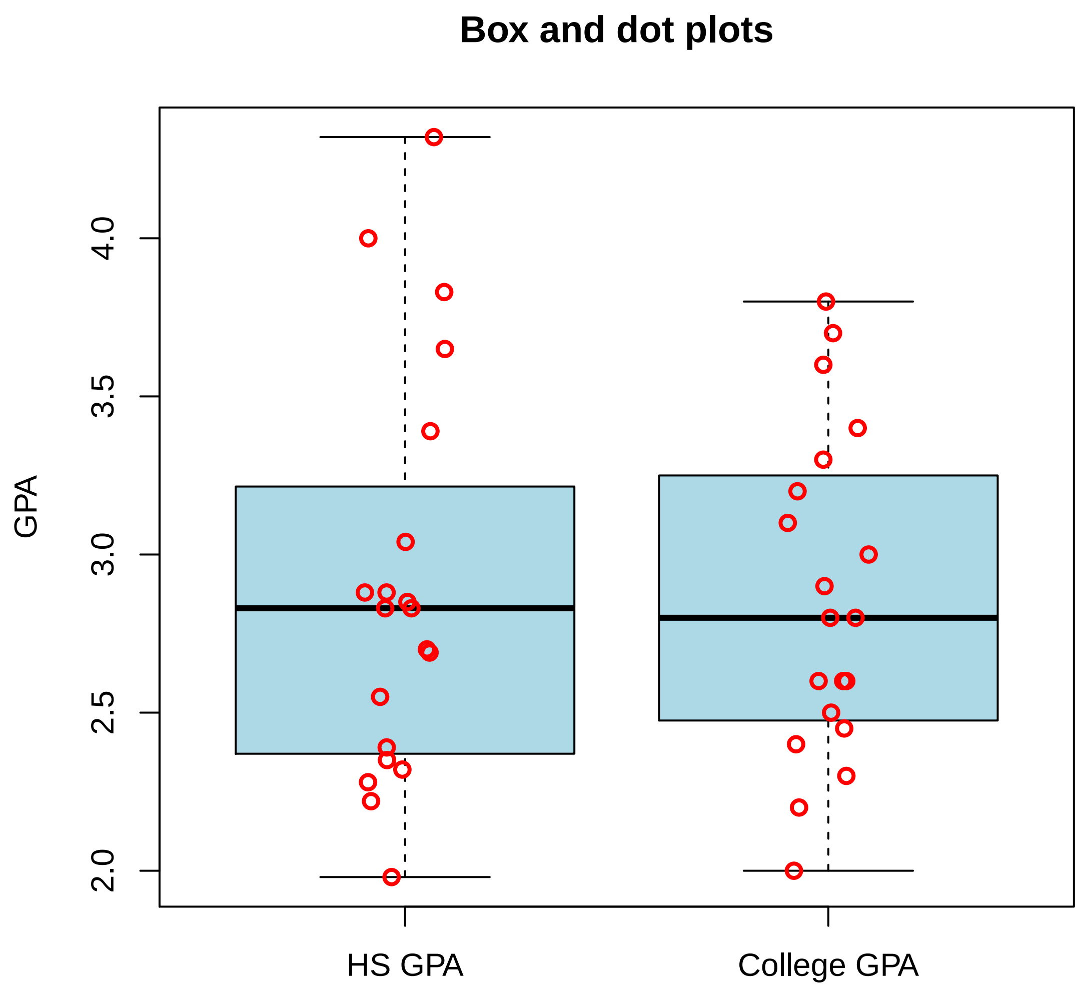
```
> par(mfrow = c(2, 1))
> hist(x = gpa$HS.GPA, xlab = "HS GPA", main = "HS GPA",
     breaks = c(0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5),
     ylim = c(0, 0.8), freq = FALSE)
> curve(expr = dnorm(x = x, mean = mean(gpa$HS.GPA),
     sd = sd(gpa$HS.GPA)), col = "red", add = TRUE)
> hist(x = gpa$College.GPA, xlab = "College GPA", main = "College GPA",
     breaks = seq(from = 0, to = 4.5, by = 0.5), ylim = c(0,
         0.8), freq = FALSE)
> curve(expr = dnorm(x = x, mean = mean(gpa$College.GPA),
     sd = sd(gpa$College.GPA)), col = "red", add = TRUE)
```



The `freq = FALSE` argument value in `hist()` leads to a rescaling of the y-axis for the histogram bars so that the density overlay can be performed. Note that after completing a plot like the previous one with multiple plots in one graphics window, it can be helpful to reset the number of plots per graphics window to 1 by using `par(mfrow = c(1,1))` after plotting. Alternatively, one can simply close the graphics window to have the same effect (all default values for `par()` go back into effect).

Box plots are produced by `boxplot()`, and dot plots are produced by `stripchart()`:

```
> #' Box plots are drawn in order of variables in data frame
> # Only use outpch = NA if following up with a dot plot that
> # draws the outliers
> boxplot(x = gpa, col = "lightblue", main = "Box and dot plots",
    ylab = "GPA", xlab = "", names = c("HS GPA", "College GPA"),
    pars = list(outpch = NA))
> set.seed(7128)  # Reproduce same jittering each time
> stripchart(x = gpa, lwd = 2, col = "red", method = "jitter",
    vertical = TRUE, pch = 1, add = TRUE)
```



There are many definitions of box plots so there can be some uncertainty regarding exactly what is being displayed! The function here follows what most definitions provide with respect to Q_1, the 0.25 quantile (i.e., 25th percentile), the median, and Q_3, the 0.75 quantile. With respect to how far whiskers are extended, the `range` argument controls the amount with a default value of 1.5. Thus, the upper whisker is draw out to $Q_3 + 1.5(Q_3 - Q_1)$ or the largest observation (whichever is the smallest). Because there are different definitions for quantiles (`quantile()` provides 9 different!), this can lead to further differences among plots. With `range = 0`, the whiskers will always extend out to the most extreme observations.

The data for the previous example was organized so that each column in the data frame was a variable included with a box plot. In other cases, the data may be in the following form:

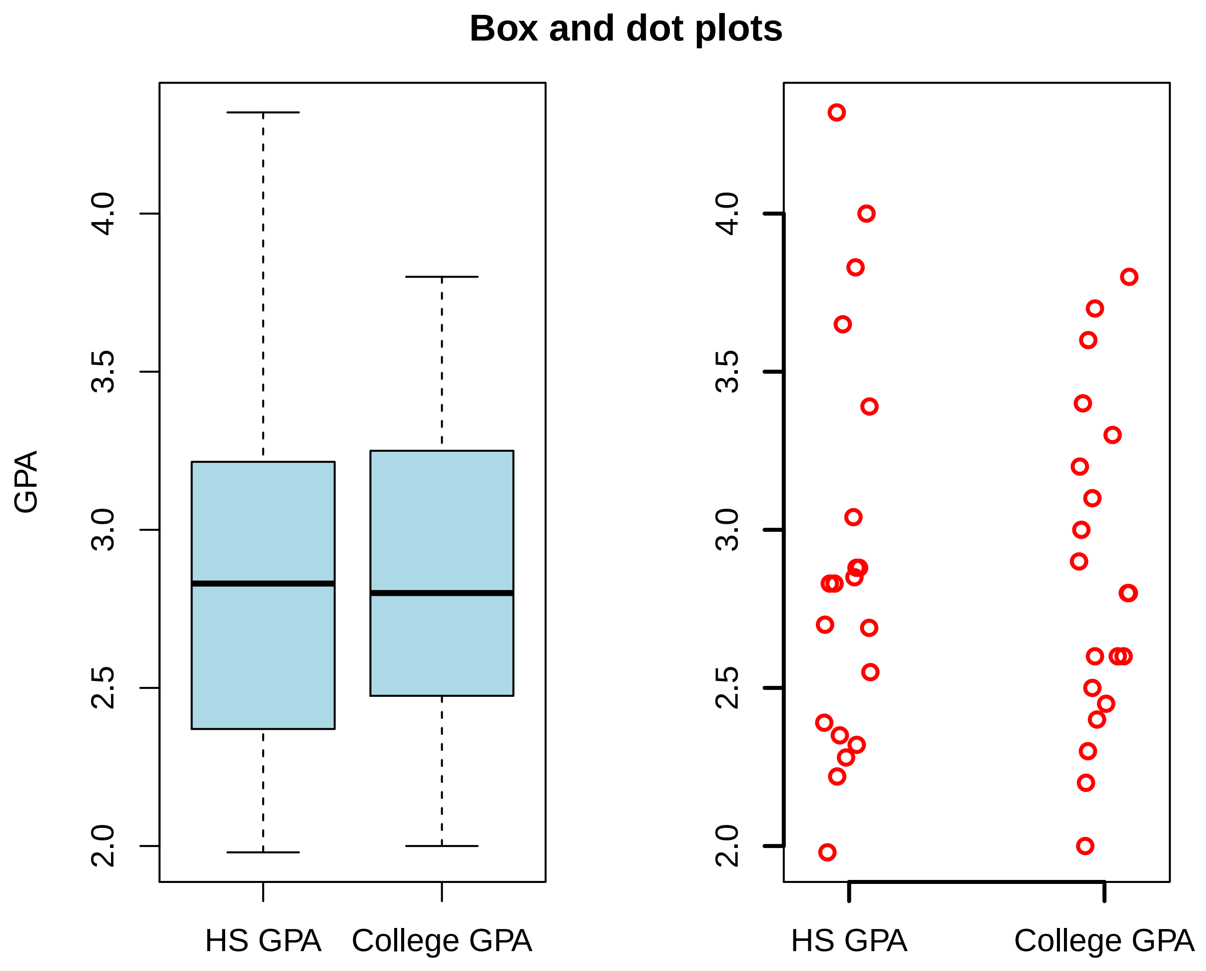
```
> HS.only <- data.frame(school = "HS", gpa = gpa$HS.GPA)
> College.only <- data.frame(school = "College", gpa = gpa$College.GPA)
> HS.college <- rbind(HS.only, College.only)
> head(HS.college, n = 3)
  school  gpa
1     HS 3.04
2     HS 2.35
3     HS 2.70
> tail(HS.college, n = 3)
    school gpa
38 College 3.8
39 College 2.2
40 College 2.6
```

When this occurs, a `formula` argument can be used to specify what to plot. Please see my program for an example.

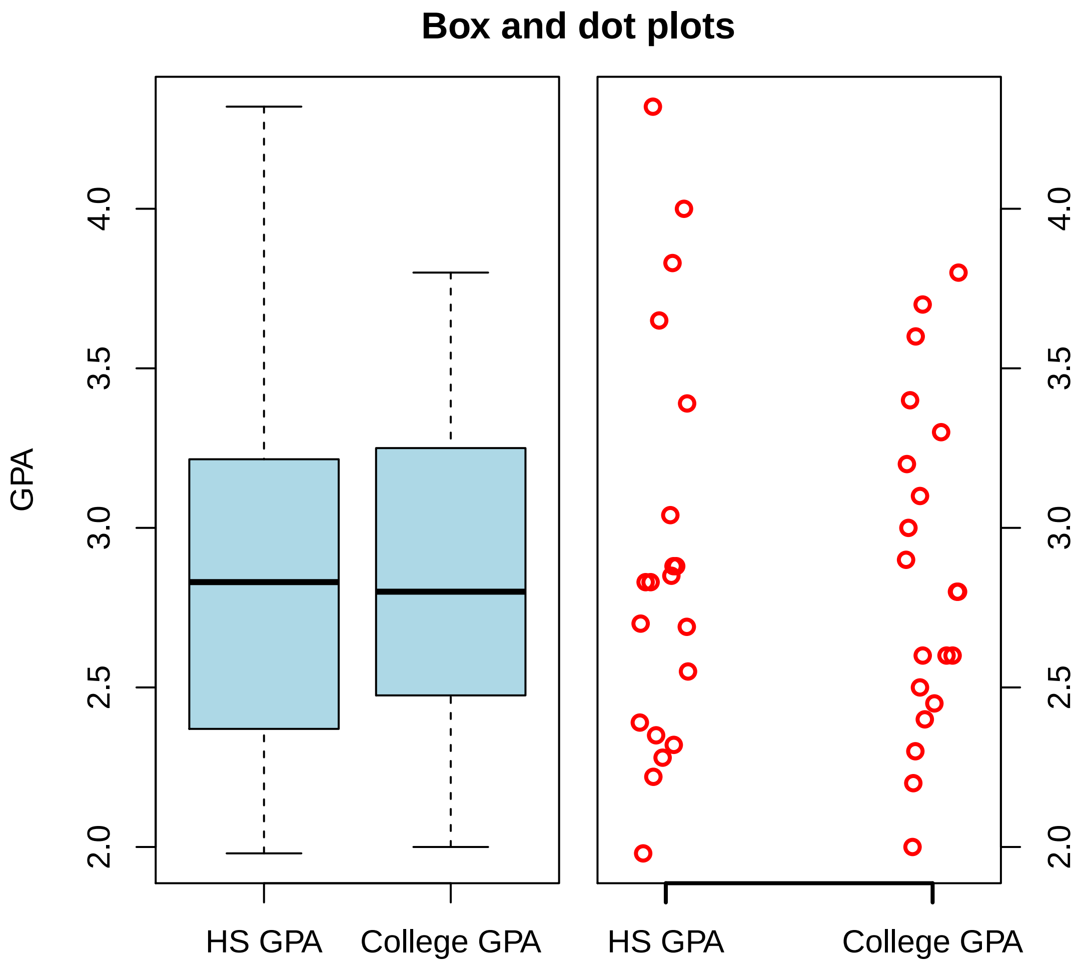
Layout of plots

Using the `mfrow` (or `mfcol`) argument value in `par()` is the most convenient way to include more than one plot in a single graphics window. Sometimes, the spacing between the plots can be larger than desired. A simple way to control the spacing is to reduce the margins with the `mar` argument of `par()`. The `mar` argument accepts a vector of length four controling the four margins in the order of bottom, left, top, and right. For example, suppose it was not desired to overlay the box and dot plots, but rather put separate plots side-by-side. Below are two separate graphics with the second one shrinking the margins between the two plots:

```
> # Default
> par(mfrow = c(1, 2))
> # Make sure plots have same y-axis scales
> min.y <- min(gpa$HS.GPA, gpa$College.GPA)
> max.y <- max(gpa$HS.GPA, gpa$College.GPA)
> # Note: removed pars = list(outpch = NA)
> boxplot(x = gpa, col = "lightblue", ylab = "GPA", xlab = "",
    ylim = c(min.y, max.y), names = c("HS GPA", "College GPA"))
> title(main = "Box and dot plots", outer = TRUE, line = -3)
> set.seed(6162)  # Reproduce same jittering each time
> stripchart(x = gpa, lwd = 2, col = "red", method = "jitter",
    vertical = TRUE, pch = 1, group.names = c("HS GPA", "College GPA"),
    ylim = c(min.y, max.y))
```



```
> # Smaller margins
> par(mfrow = c(1, 2))
> par(mar = c(5, 4, 4, 0.5))  # Default is mar = c(5, 4, 4, 2) + 0.1 inches
> boxplot(x = gpa, col = "lightblue", ylab = "GPA", xlab = "",
   ylim = c(min.y, max.y), names = c("HS GPA", "College GPA"))
> set.seed(6162)  # Reproduce same jittering each time
> title(main = "Box and dot plots", outer = TRUE, line = -3)
> par(mar = c(5, 0.5, 4, 4))
> stripchart(x = gpa, lwd = 2, col = "red", method = "jitter",
   vertical = TRUE, pch = 1, group.names = c("HS GPA", "College GPA"),
   ylim = c(min.y, max.y), yaxt = "n")
> axis(side = 4)  # Right axis
```



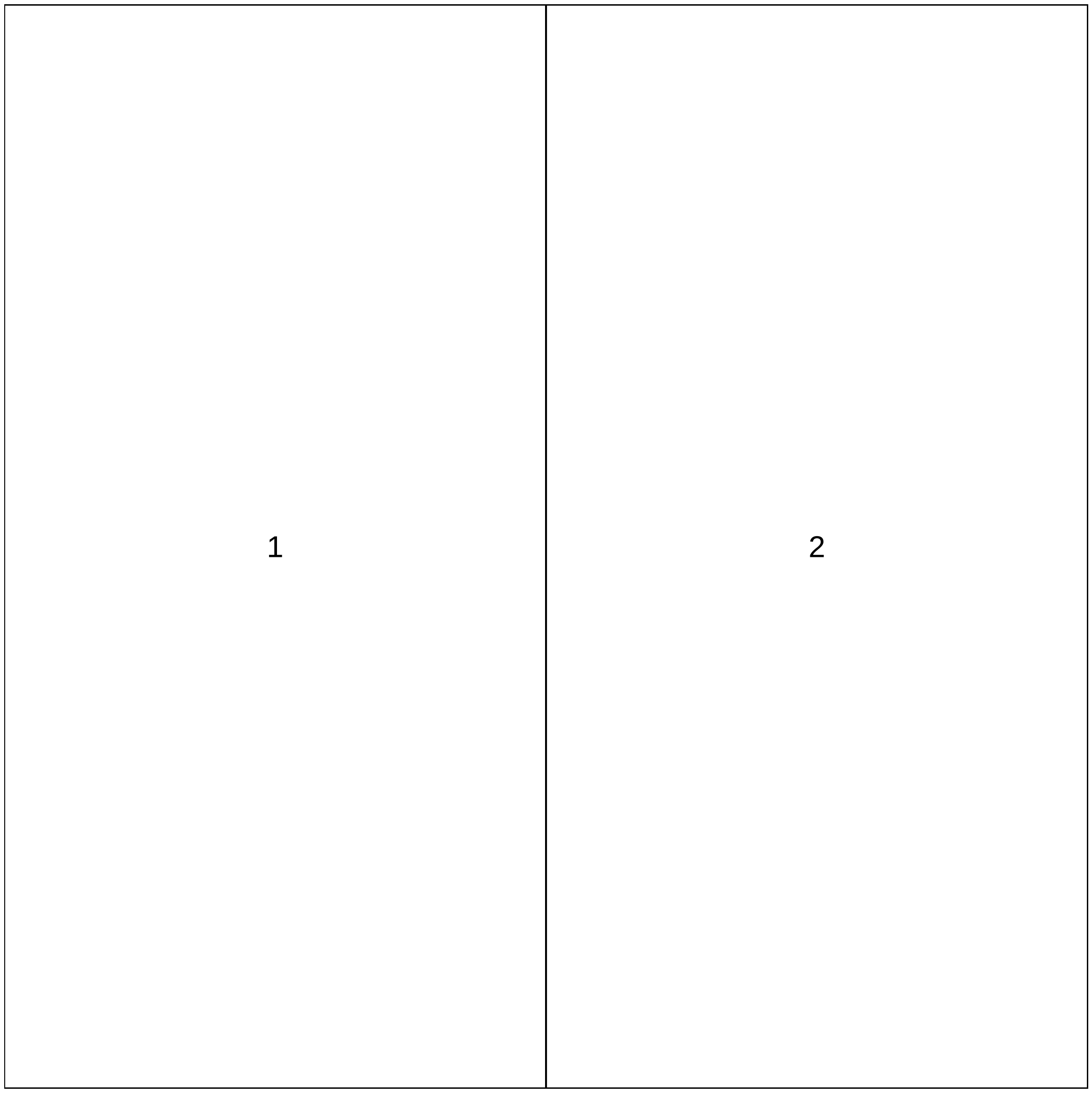
Comments:

- `title()` is an alternative way to put a title on a plot. I used the function here to help put a title across all of the plots. The `outer` argument specifies that the title is placed outside of the normal plotting area and the `line = -3` value specifies that the title should be moved three lines down from its default placement.

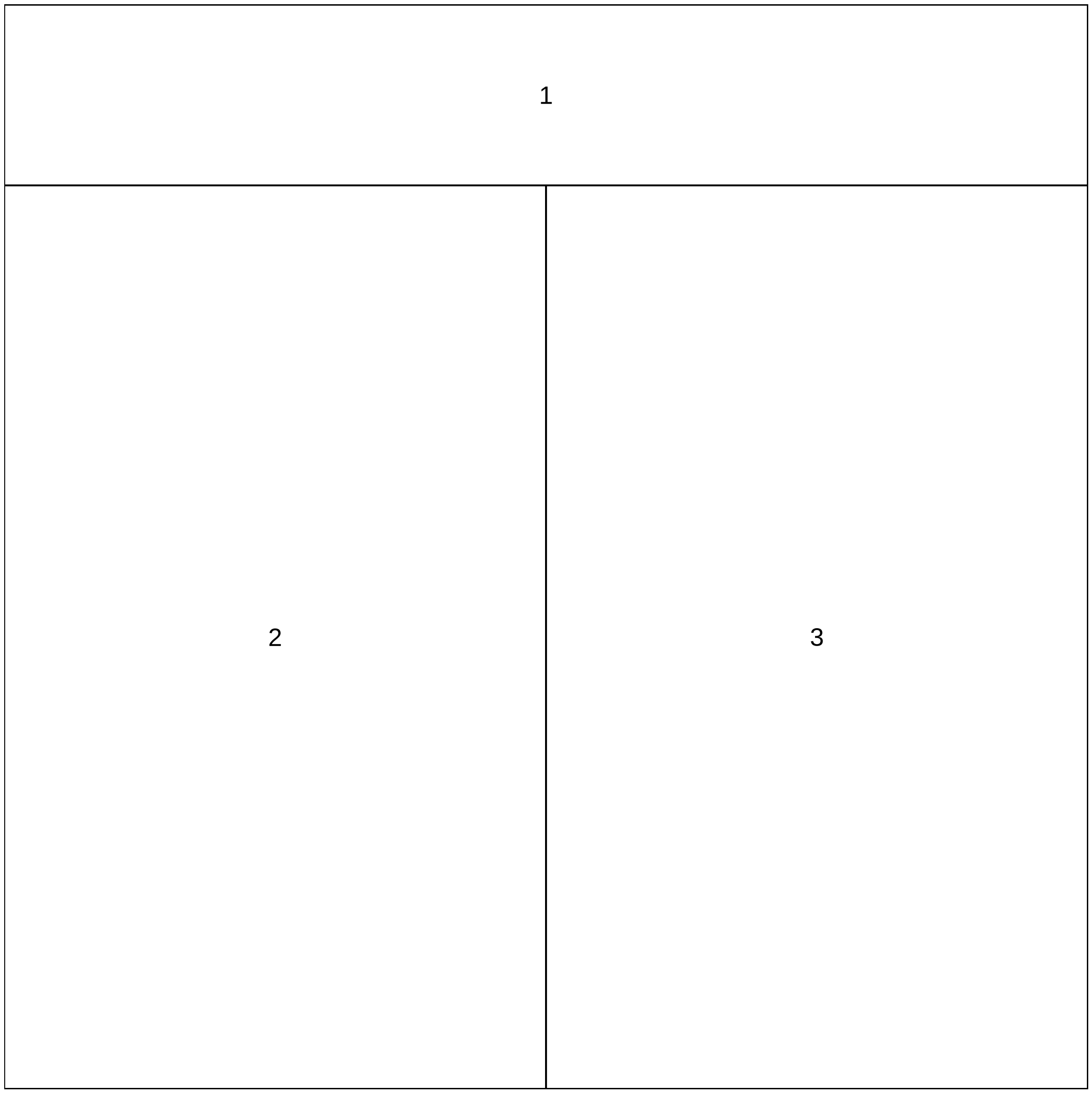
- When the first plot was originally constructed, the "College GPA" x-axis label for the box plot was omitted. R does this to prevent the labels from overlapping. A quick fix to the problem is to make the graphics window wider. Alternative fixes include changing the font size for the labels using the `cex.axis` argument in the function (see `help(axis)` for more information).

A more general way to control the layout of plots in one graphics window is through the `layout()` function. Below are two simple examples with the first one providing the same results as `par(mfrow = c(1,2))`:

```
> save.layout1 <- layout(mat = matrix(data = c(1, 2), nrow = 1,
     ncol = 2, byrow = TRUE))
> layout.show(save.layout1)
```



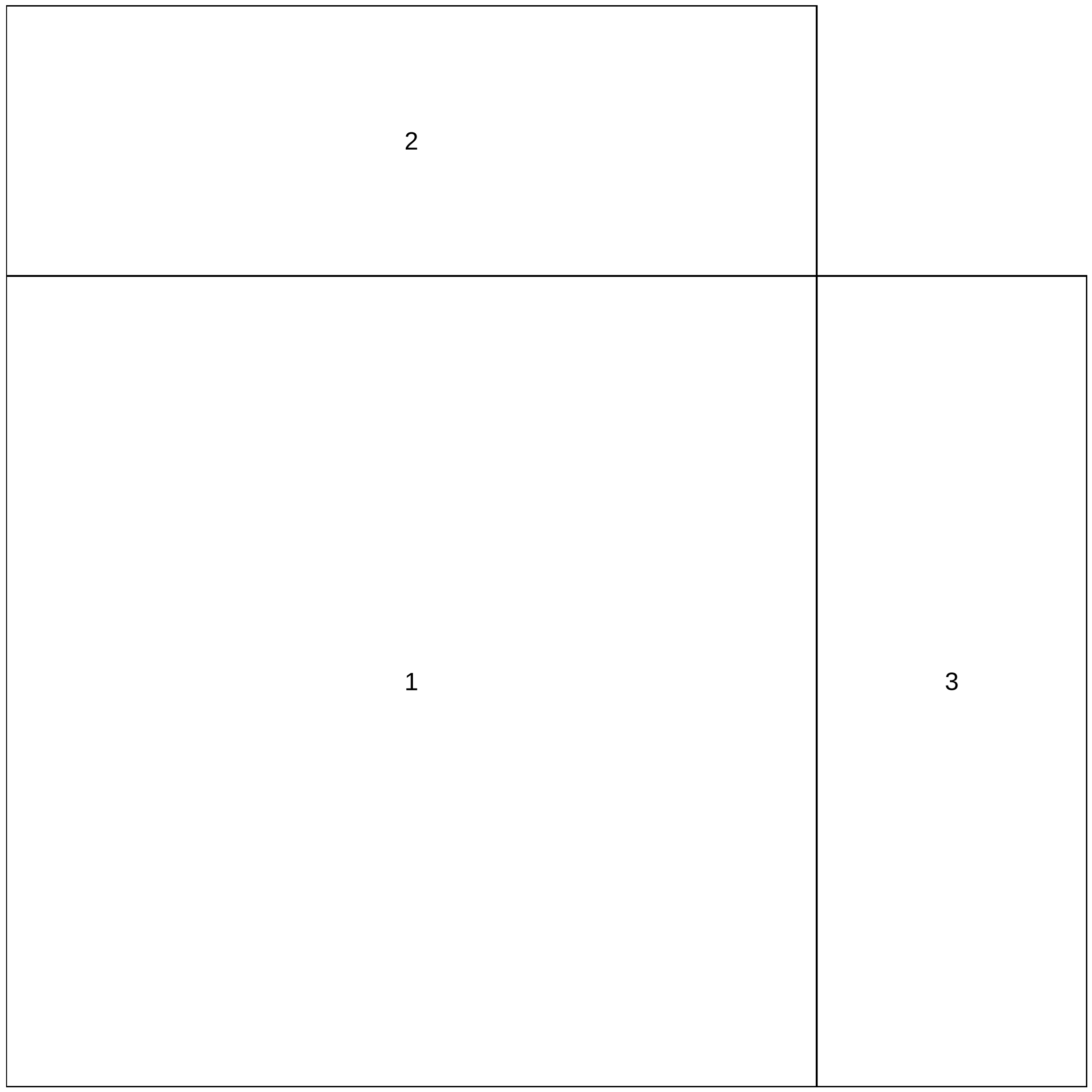
```
> save.layout2 <- layout(mat = matrix(data = c(1, 1, 2, 3), nrow = 2,
     ncol = 2, byrow = TRUE), heights = c(1, 5))
> layout.show(save.layout2)
```



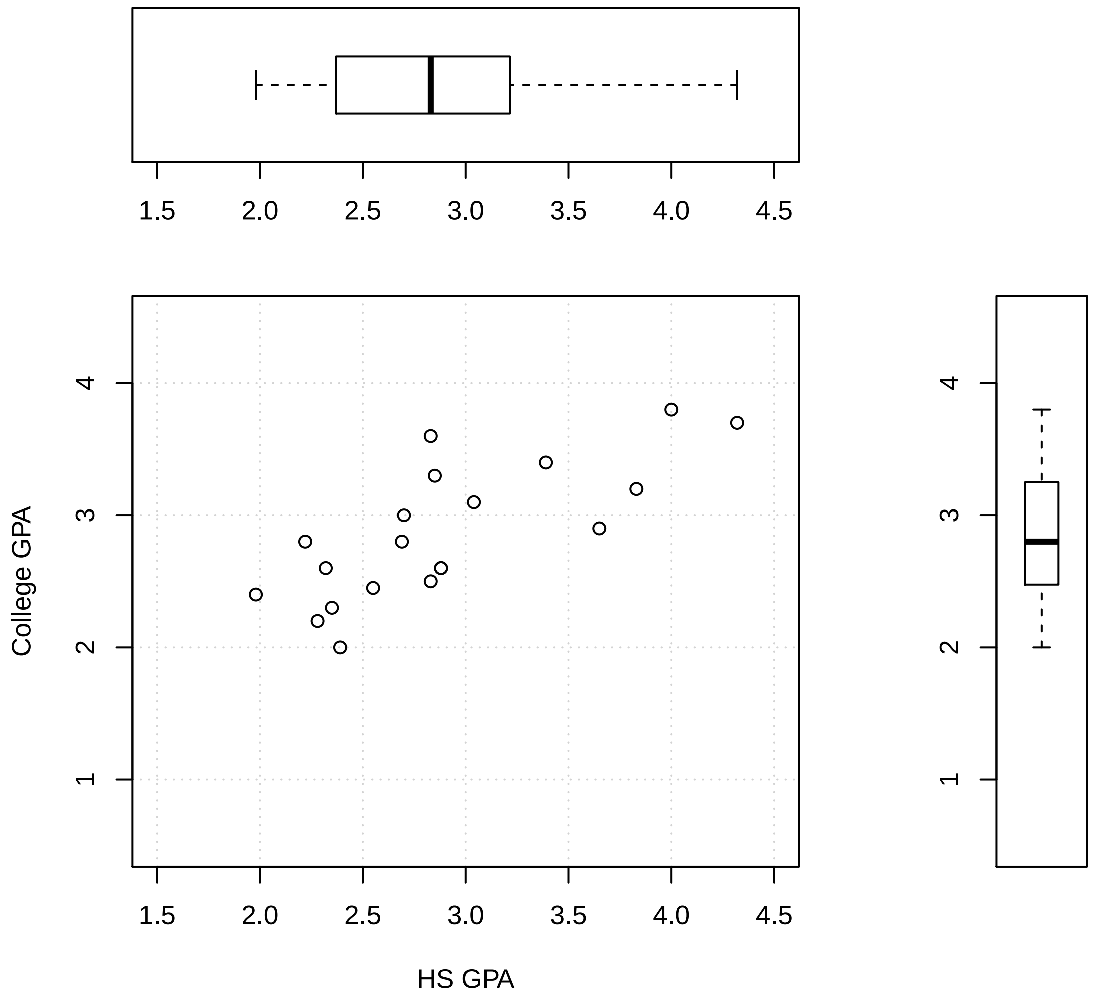
These layouts are in effect until a new `layout()` is specified or the graphics window is closed.

As a little more complicate example, below is how I construct a scatter plot with box plots.

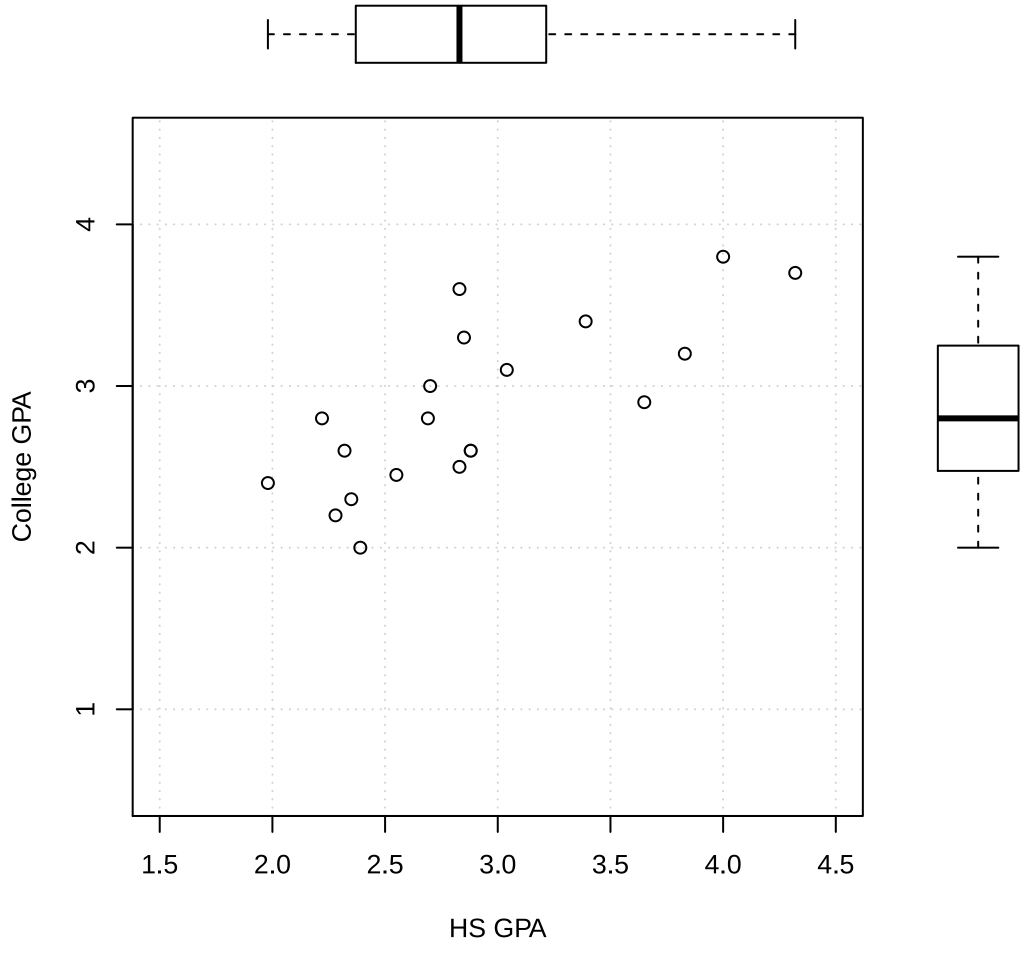
```
> save.layout3 <- layout(mat = matrix(data = c(2, 0,
     1, 3), nrow = 2, ncol = 2, byrow = TRUE), heights = c(1,
     3), widths = c(3, 1))
> layout.show(save.layout3)
```



```
> par(mar = c(5, 4, 4, 2) + 0.1)  # Default values
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
    ylab = "College GPA", xlim = c(1.5, 4.5), ylim = c(0.5,
        4.5), col = "black", lwd = 1, panel.first = grid())
> par(mar = c(0, 4, 4, 2) + 0.1)
> # Notice ylim is really for x-axis here due to
> # horizontal
> boxplot(x = gpa$HS.GPA, xlab = NA, ylab = NA, main = NA,
    horizontal = TRUE, ylim = c(1.5, 4.5))
> par(mar = c(5, 4, 4, 2) + 0.1)
> boxplot(x = gpa$College.GPA, xlab = NA, ylab = NA,
    main = NA, horizontal = FALSE, ylim = c(0.5, 4.5))
```



```
> # Without axes for box plots - should only do this
> # when sure everything lines up correctly with the
> # scatter plot
> par(mar = c(5, 4, 0, 0) + 0.1)
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",
     ylab = "College GPA", xlim = c(1.5, 4.5), ylim = c(0.5,
         4.5), col = "black", lwd = 1, panel.first = grid())
> par(mar = c(0, 4, 4, 0) + 0.1)
> boxplot(x = gpa$HS.GPA, xlab = NA, ylab = NA, main = NA,
     horizontal = TRUE, ylim = c(1.5, 4.5), axes = FALSE)
> par(mar = c(5, 0, 0, 2) + 0.1)
> boxplot(x = gpa$College.GPA, xlab = NA, ylab = NA,
     main = NA, horizontal = FALSE, ylim = c(0.5, 4.5),
     axes = FALSE)
```



When multiple plots share the same x and y-axis values, it is VERY important to get the values to correspond correctly across plots. R will not necessarily do this so it is essential to pay close attention!

`ggplot2` package

Basics

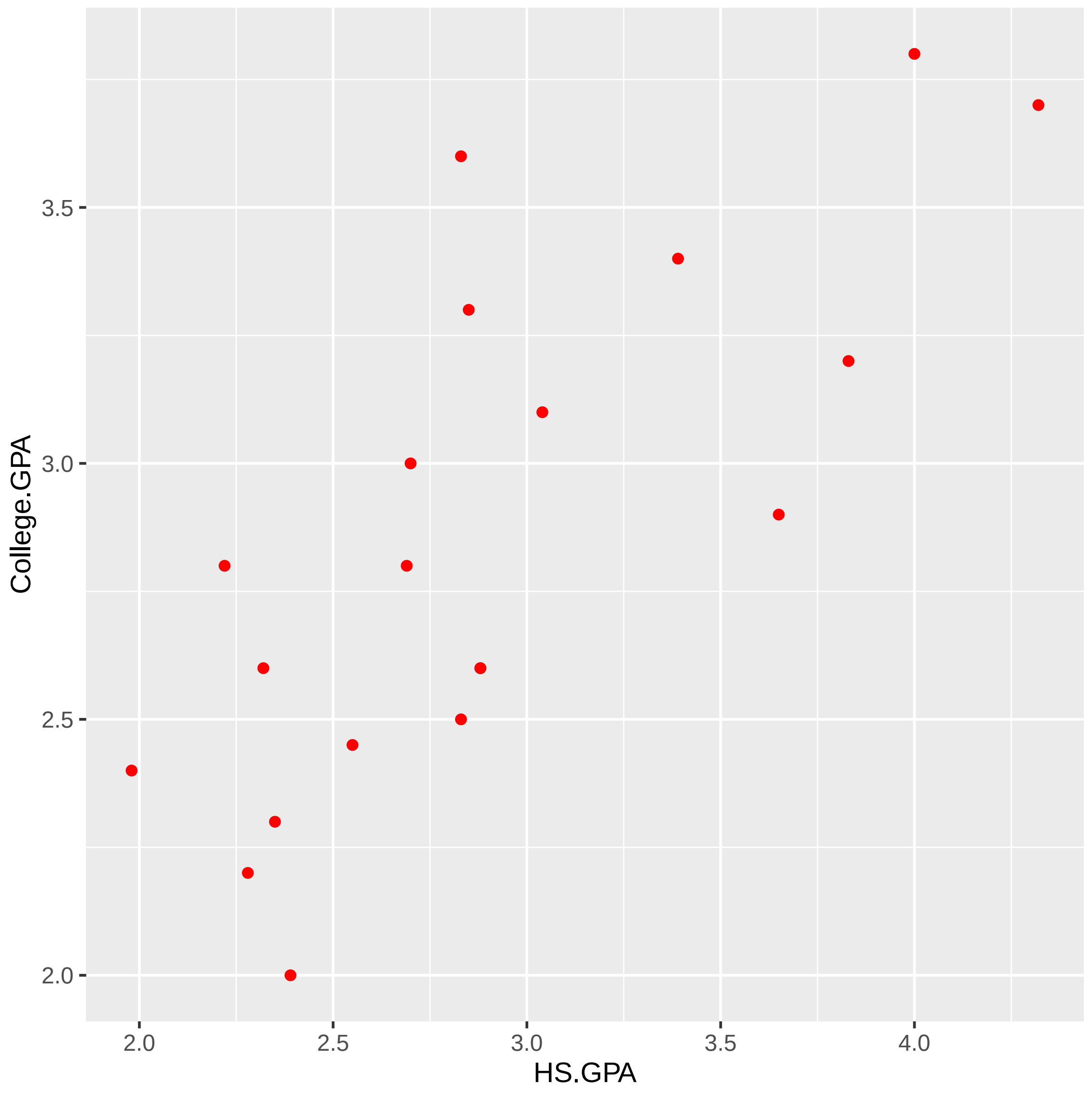
The `ggplot2` package (not in the default installation of R) is likely the second most popular way to produce plots in R and its user base continues to grow. The "gg" portion of `ggplot2` corresponds to the *Grammar of Graphics* book written in 2005

(2nd edition; 1st edition in 1999) by Leland Wilkinson. This book presented some ideas and concepts for plots but not the tools for "how to" create them. Hadley Wickham subsequently created the `ggplot` package and now the `ggplot2` package that allowed for the "how to". Wickham also wrote *ggplot2: Elegant Graphics for Data Analysis* in 2009 based on the package (PDF available from the library) and has a website at `http://ggplot2.org` that provides details about the package as well. Another online resource available from our library is *ggplot2 Essentials* by Donato Teutonico.

There are two main plotting functions: `ggplot()` and `qplot()`, where the "q" stands for "quick". The `ggplot()` function is the more flexible of the two. The `qplot()` can be more easy to work with for simple plots. After invoking one of these functions, one usually calls a number of other functions that add *layers* to a plot. This is somewhat similar to what we had with `plot()` and then adding items with functions like `curve()`, `text()`, `legend()`, ...

We will first focus on using `ggplot()`. The main two arguments in `ggplot()` correspond to the data frame (`data`) and what in the data frame (`mapping`) will be plotted, referred to as the *aesthetics*. Note that all aesthetics need to be in a data frame, which is different than with the `graphics` package! After using `ggplot()`, we then need to specify what to do with these aesthetics. Below is a simple illustration of this process for a scatter plot.

```
> library(package = "ggplot2")
> save.plot <- ggplot(data = gpa, mapping = aes(x = HS.GPA, y = College.GPA))
> save.plot + geom_point(color = "red")
```



In the code, the results from `ggplot()` were saved into an object that I decided to call `save.plot`. Next, I "added" to this object a *geometric object*, also known as a "geom", which plotted the (x, y) coordinates as points. Every geom has a `data` and `mapping` argument like `ggplot()` so that you can change the data set and aesthetics as needed. Their default values are `data = NULL` and `mapping = NULL`, which means use the same values as in `ggplot()`. The `color` argument is known as a parameter because it is constant for all values for `x` and `y`.

There are a number of alternative ways to obtain the previous plot. One way is to just use the same line for all of the code:

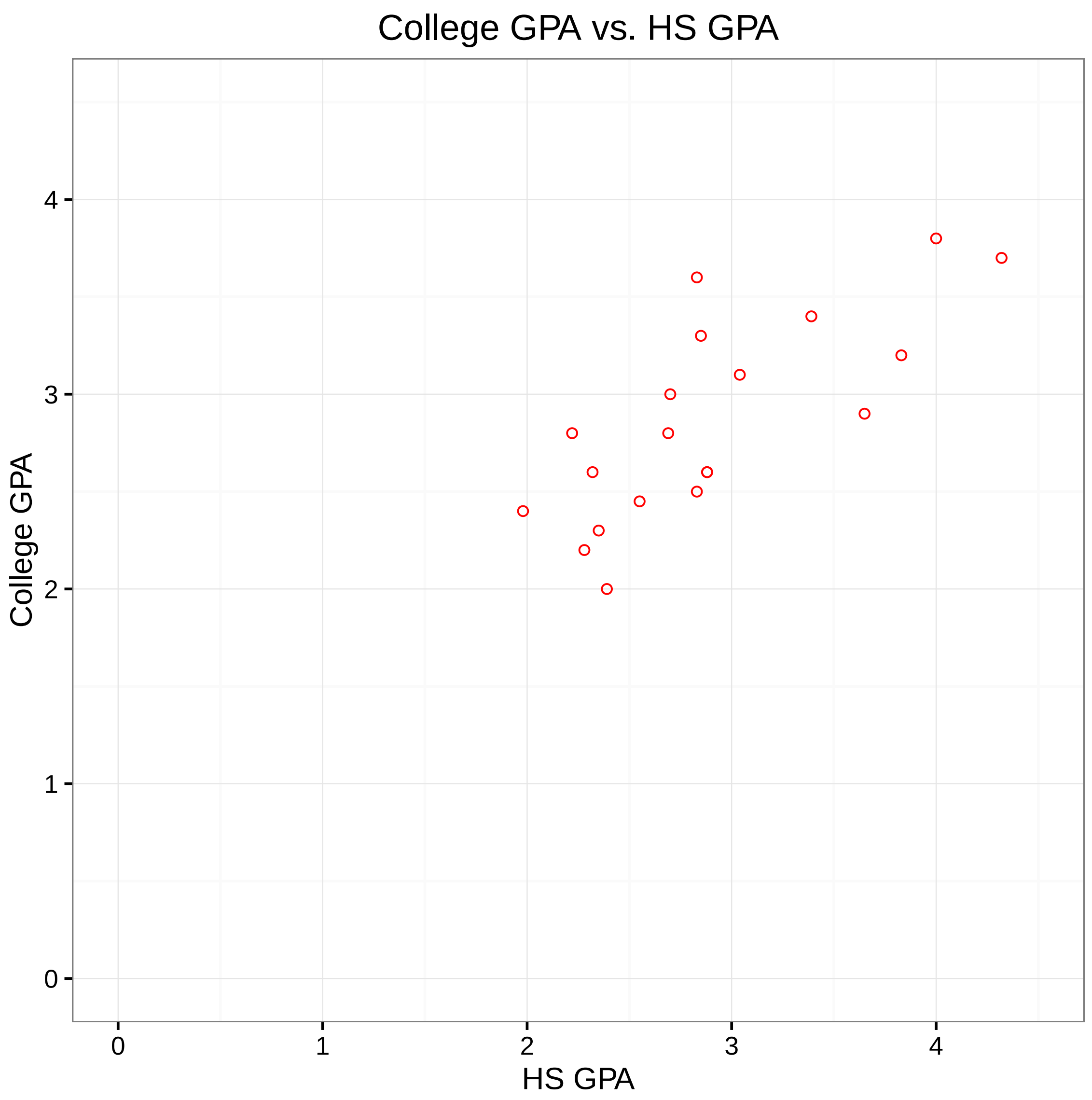
```
> # Code not executed
> ggplot(data = gpa, mapping = aes(x = HS.GPA, y = College.GPA)) +
    geom_point(color = "red")
```

Other ways involve the `qplot()` function. This function actually tries to "guess" what type of plot to construct based on the variables given. Below are a few ways that one can obtain the same plot as the previous one.

```
> #' Code not executed
> # Produces a legend - color is mapped as an aesthetic here
> # (like a variable in a data frame). This is the default
> # behavior of qplot(). See p. 47 of Wickham (2009).
> qplot(x = HS.GPA, y = College.GPA, data = gpa, geom = "auto",
    color = "red")
> # No legend - color is set as a constant with I()
> qplot(x = HS.GPA, y = College.GPA, data = gpa, geom = "auto",
    color = I("red"))
> qplot(x = HS.GPA, y = College.GPA, data = gpa, geom = "point",
    color = I("red"))
```

Through including other geoms, labels, axis controls, and stat functions, one can obtain the following much better looking plot.

```
> # Removes awful gray background!
> theme_set(new = theme_bw())
> save.plot + geom_point(color = "red", shape = 1) + xlim(0, 4.5) +
    ylim(0, 4.5) + ggtitle(label = "College GPA vs. HS GPA") +
    xlab(label = "HS GPA") + ylab(label = "College GPA")
```



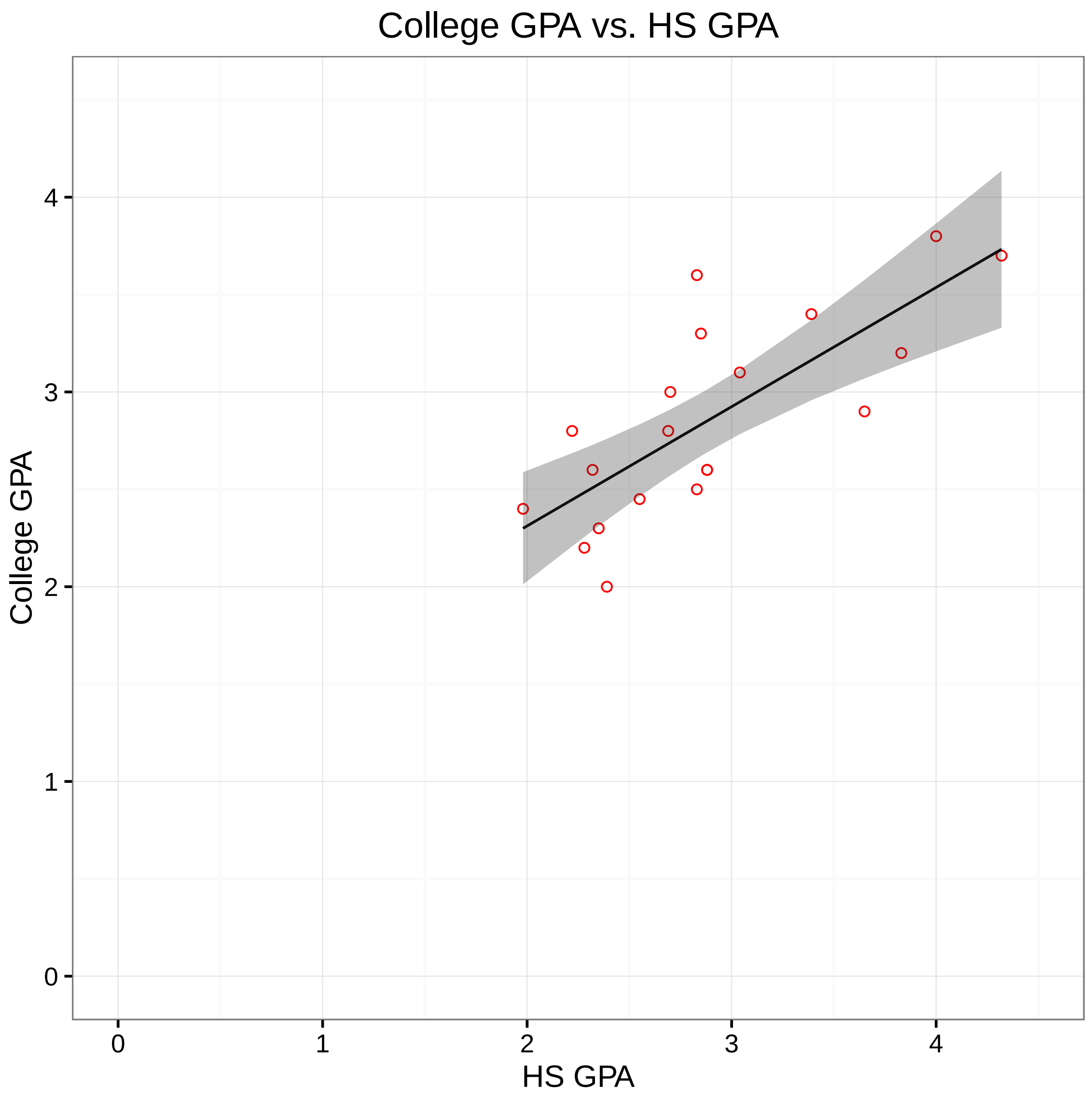
Comments regarding the code:

- The `theme_set()` function sets a new color and style theme for all subsequent plots (default is `theme_gray()`). You can see settings for a theme by running the corresponding theme function at a prompt. A completely new theme can be created as well. See my `new.theme()` function in the program.
- Line types, plotting symbols, and colors are the same as in the `graphics` package.

Customizing plots

Next are additional plots to demonstrate how we can get something closer to the plots constructed with the graphics package.

```
> save.pred <- as.data.frame(predict(object = mod.fit, interval = "confidence",
    level = 0.95))
> save.pred$HS.GPA <- gpa$HS.GPA
> save.pred$College.GPA <- gpa$College.GPA
> save.plot2 <- save.plot + geom_point(color = "red", shape = 1) +
    xlim(0, 4.5) + ylim(0, 4.5) + ggtitle(label = "College GPA vs. HS GPA") +
    xlab(label = "HS GPA") + ylab(label = "College GPA")
> save.plot2 + geom_line(data = save.pred, mapping = aes(y = fit)) +
    geom_ribbon(data = save.pred, mapping = aes(ymin = lwr, ymax = upr),
        alpha = 0.3)
```



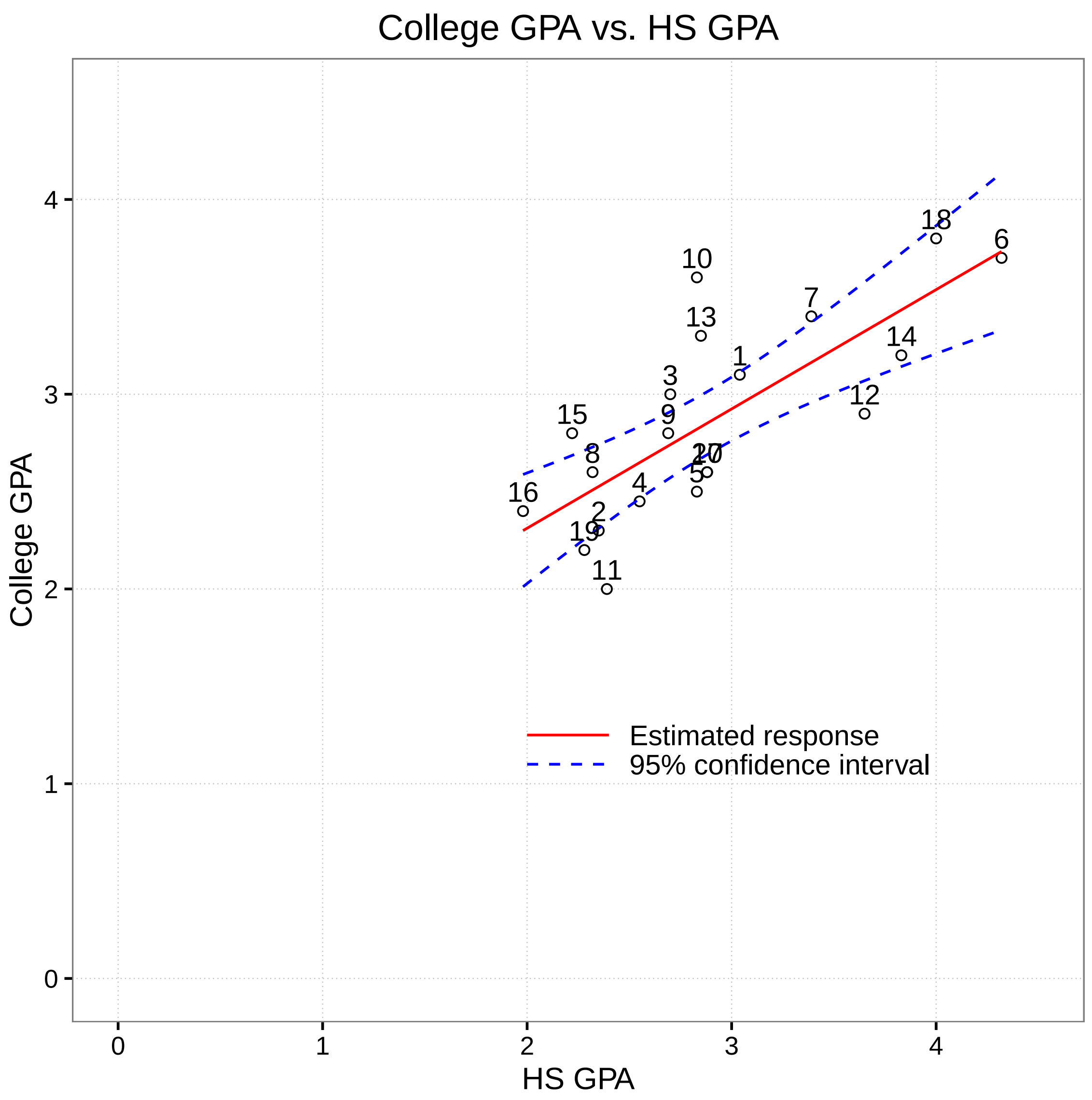
The code below would produce essentially the same plot. This is included here to demonstrate that many common tasks are available in geom functions.

```
> save.plot2 + geom_smooth(formula = y ~ x, level = 0.95, method = "lm",
     color = "black", alpha = 0.3)
```

Remember that the `y` and `x` values in `geom_smooth()` come from the original aesthetic defined in `save.plot`! While the shading in the plot is nice, I cannot represent the confidence interval band without it using geoms. Instead, I had to use more sophisticated coding:

```
> save.plot3 <- save.plot + geom_point(color = "black",
     shape = 1) + ggtitle(label = "College GPA vs. HS GPA") +
     xlab(label = "HS GPA") + ylab(label = "College GPA")
> # Functions used in drawing Y^ and confidence
> # interval bands
> yhat <- function(x, mod.fit) {
     # Could use predict() too
     mod.fit$coefficients[1] + mod.fit$coefficients[2] *
         x
 }
> lower.bound <- function(x, mod.fit) {
     predict(object = mod.fit, newdata = data.frame(HS.GPA = x),
         interval = "confidence", level = 0.95)[, 2]
 }
> upper.bound <- function(x, mod.fit) {
     predict(object = mod.fit, newdata = data.frame(HS.GPA = x),
         interval = "confidence", level = 0.95)[, 3]
 }
> # Data frame containing legend location and labels
> legend.df <- data.frame(x = c(2.5, 2.5), y = c(1.25,
     1.1), name = c("Estimated response", "95% confidence interval"),
     color.line = c("red", "blue"), linetype = c("solid",
         "dashed"))
> legend.df
    x    y                    name color.line linetype
1 2.5 1.25      Estimated response        red    solid
```

```
2 2.5 1.10 95% confidence interval       blue   dashed
> # Theme changes
> theme_bw()$panel.grid.major  # Current
List of 4
 $ colour  : chr "grey90"
 $ size    : num 0.2
 $ linetype: NULL
 $ lineend : NULL
 - attr(*, "class")= chr [1:2] "element_line" "element"
> chris.theme.changes <- theme(panel.grid.major = element_line(color = "gray",
     linetype = "dotted")) + theme(panel.grid.minor = element_blank())
> chris.theme.changes$panel.grid.major  # New
List of 4
 $ colour  : chr "gray"
 $ size    : NULL
 $ linetype: chr "dotted"
 $ lineend : NULL
 - attr(*, "class")= chr [1:2] "element_line" "element"
> # Create plot
> save.plot3 + stat_function(fun = yhat, args = list(mod.fit = mod.fit),
     color = "red") + stat_function(fun = lower.bound,
     args = list(mod.fit = mod.fit), color = "blue",
     linetype = "dashed") + stat_function(fun = upper.bound,
     args = list(mod.fit = mod.fit), color = "blue",
     linetype = "dashed") + coord_cartesian(ylim = c(0,
     4.5), xlim = c(0, 4.5)) + chris.theme.changes +
     geom_text(mapping = aes(y = College.GPA + 0.1,
         label = 1:20), size = 4) + geom_text(data = legend.df,
     mapping = aes(x = x, y = y, label = name), size = 4,
     hjust = 0) + geom_segment(data = legend.df, mapping = aes(x = x -
     0.5, y = y, xend = x - 0.1, yend = y), color = c("red",
     "blue"), linetype = c("solid", "dashed"))
```



Comments:

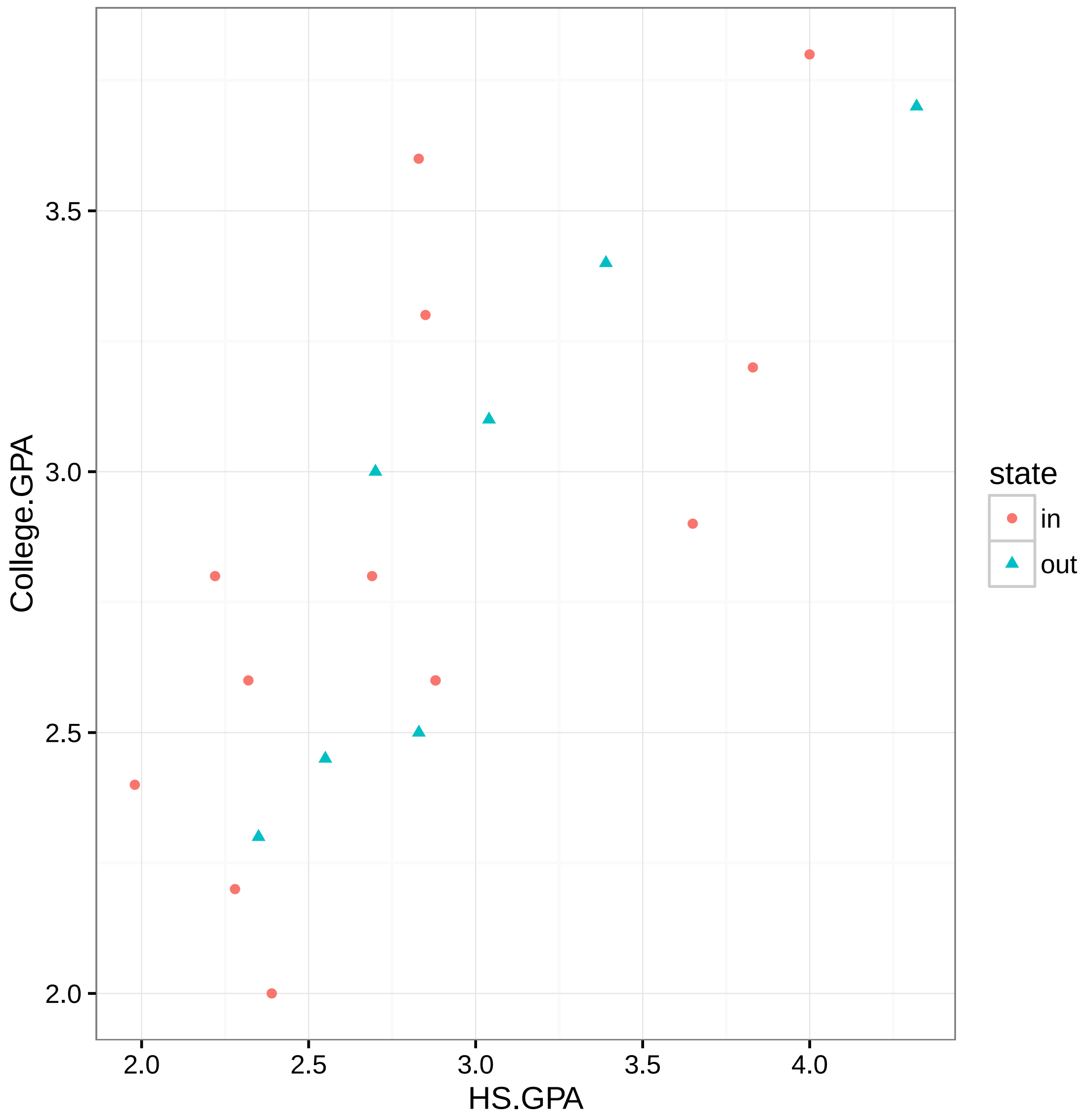
- Functions of the naming convention `stat_<name>()` provide a statistical transformation of the data. The `stat_function()` function here works like `curve()`. Simply, the minimum and maximum values of the x aesthetic are taken and the mathematical function specified is evaluated with 101 equally spaced points (`n = 101` is default). The corresponding y values are found and connected by a line
- `coord_cartesian()` was used to limit the x and y-axis scales. In previous code, I used `xlim()` and `ylim()` for a similar purpose. Unfortunately, these two functions cause the minimum and maximum values for the `x` and `y` aesthetics to change to

the limits given, which subsequently causes `stat_function()` to be evaluated over these `x` and `y` ranges too (we need to keep the x range to those from the high school GPAs to avoid extrapolation beyond the range of the data).

- `geom_text()` provides one way to add text annotations to a plot. The `annotate()` function could be used for this purpose too when wants more freedom to put text anywhere on the plot without restriction to the current aesthetics. The `size` value is in millimeters.
- The `ggplot2` package is known for "automatic" legend construction. Unfortunately, this does not work for the legend that I need here. For this case, I constructed my own data frame with the necessary items in it so that I could construct the legend manually with `geom_text()` and `geom_segment()`.

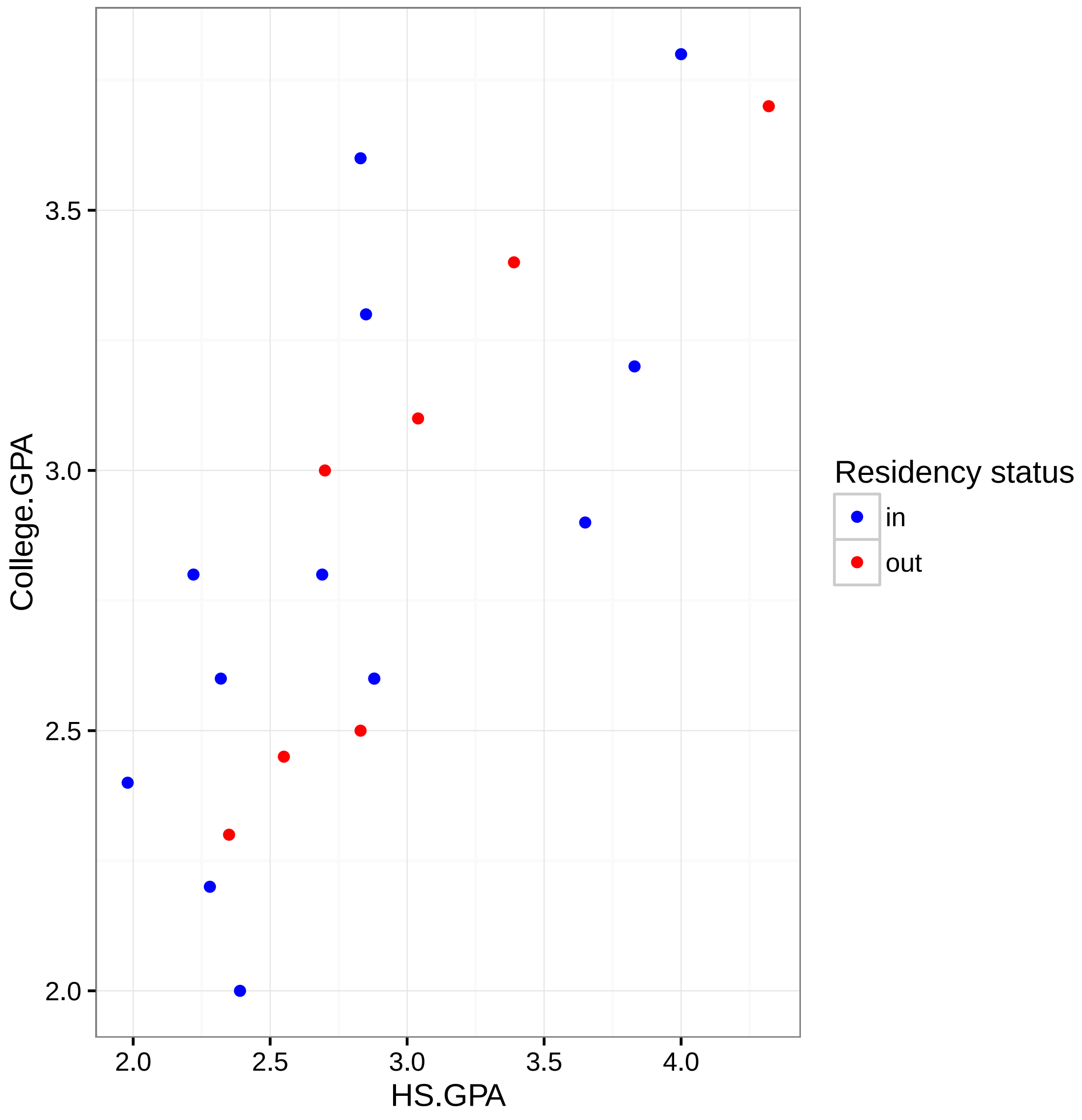
For simple plots, `ggplot2` can automatically provide a legend. For example, consider the situation where the residency status of a student is added to the plot:

```
> state <- c(rep(x = "out", times = 7), rep(x = "in", times = 13))
> # Create a new data frame that adds the new variable to it.
> # Alternatively, one could have kept the old data frame and
> # used gpa$state <- state
> gpa2 <- data.frame(gpa, state)
> save.plot4 <- ggplot(data = gpa2, mapping = aes(x = HS.GPA, y = College.GPA,
     color = state, shape = state))
> save.plot4 + geom_point()
```

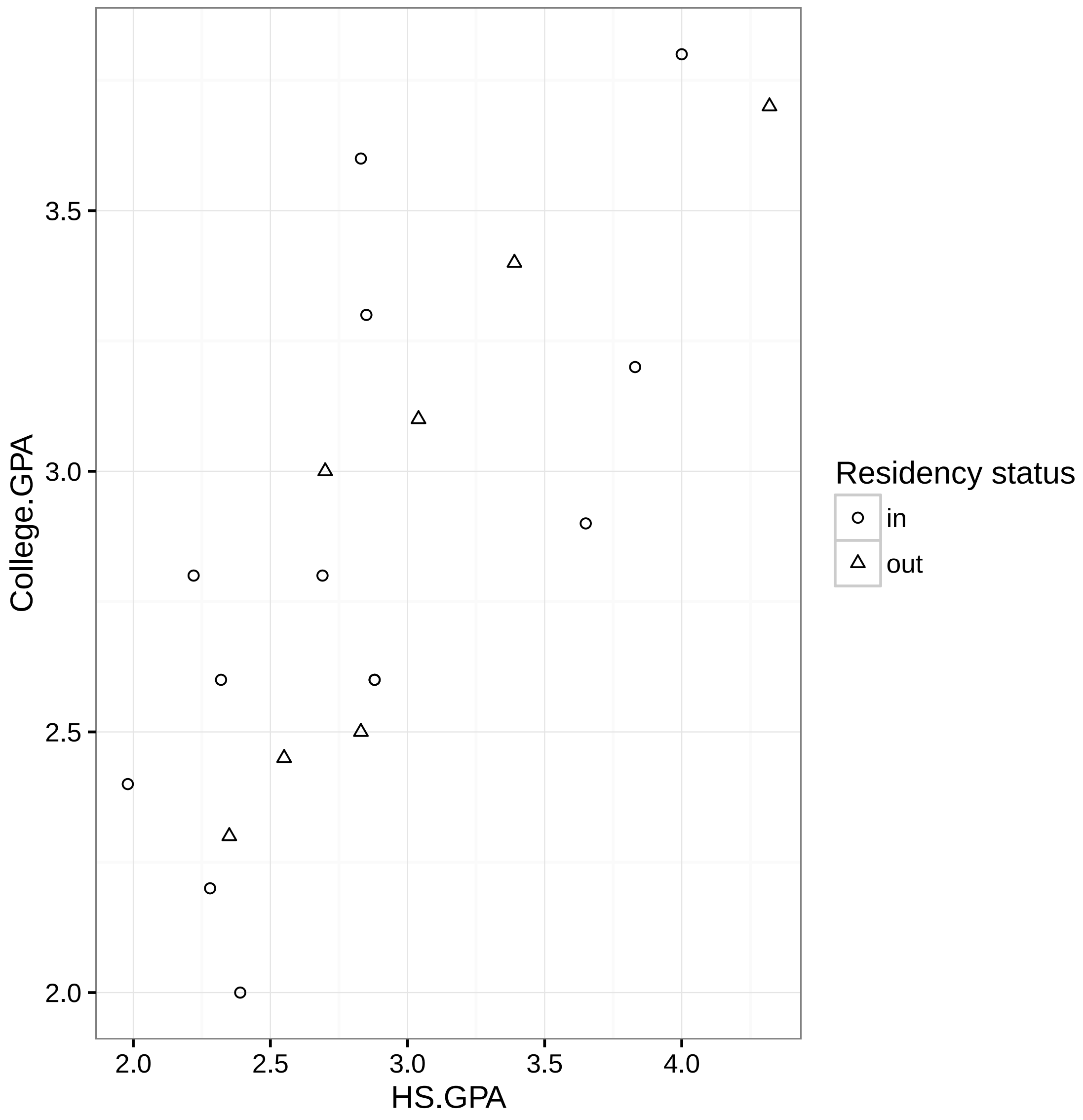


Customizing the plot further is possible:

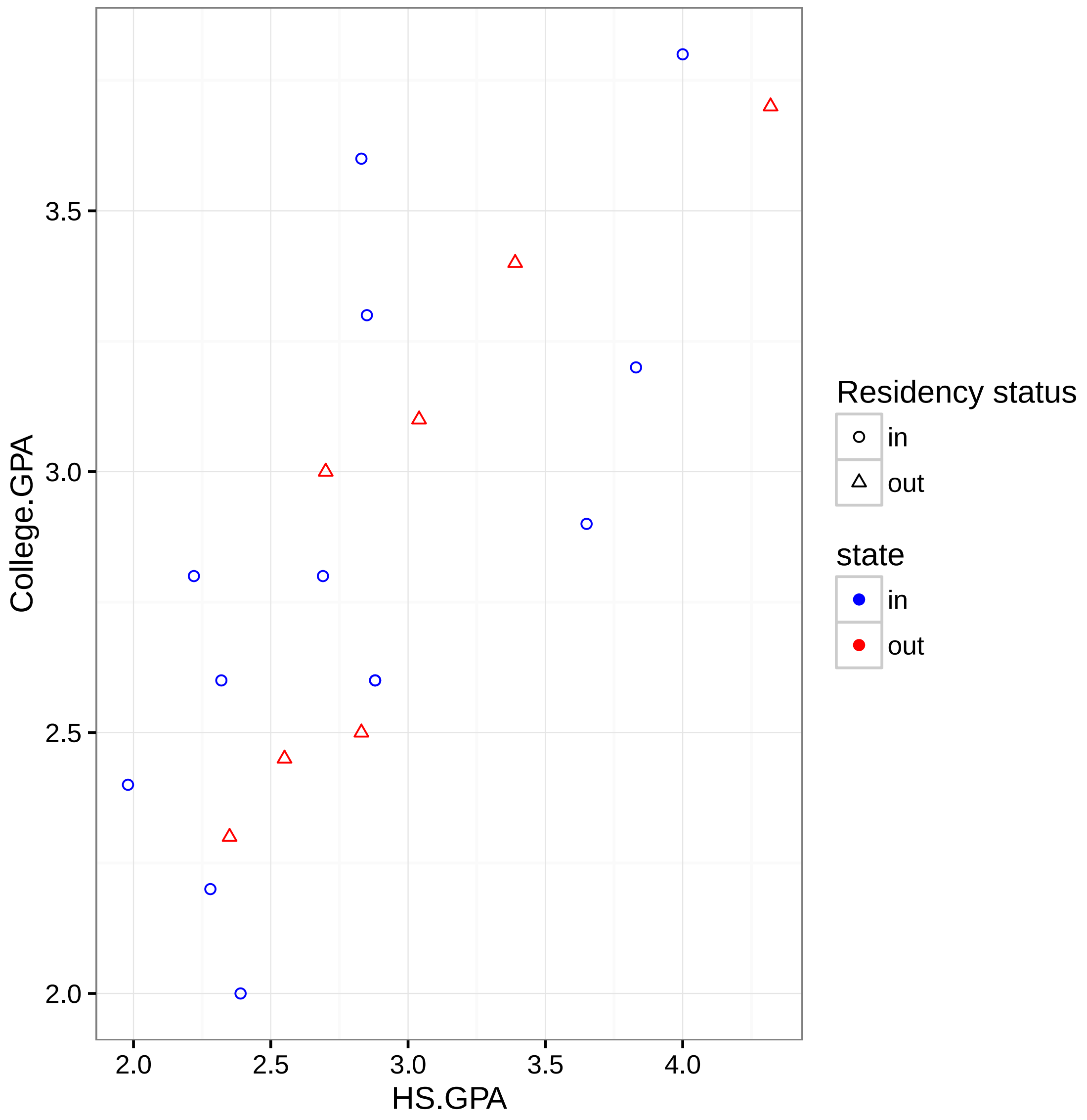
```
> # Control color of plotting point
> save.plot5 <- ggplot(data = gpa2, mapping = aes(x = HS.GPA,
    y = College.GPA, color = state))
> save.plot5 + geom_point() + scale_color_manual(values = c(out = "red",
    `in` = "blue")) + guides(color = guide_legend(title = "Residency status"))
```



```
> # Control style of plotting point
> save.plot6 <- ggplot(data = gpa2, mapping = aes(x = HS.GPA,
     y = College.GPA, shape = state))
> save.plot6 + geom_point() + scale_shape(solid = FALSE) +
     guides(shape = guide_legend(title = "Residency status"))
```



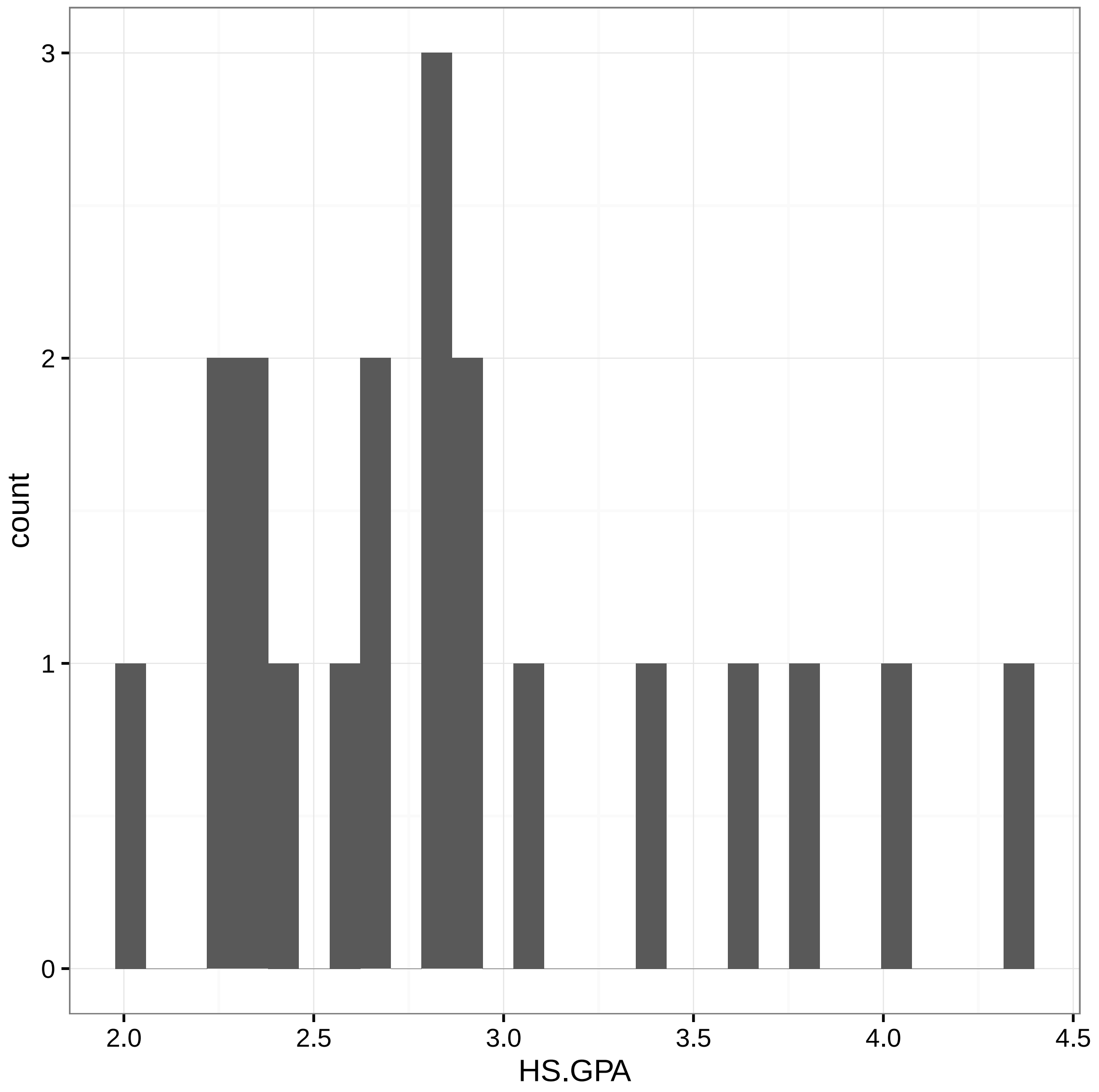
```
> # Control color and style does not work well
> save.plot7 <- ggplot(data = gpa2, mapping = aes(x = HS.GPA,
     y = College.GPA, shape = state, color = state))
> save.plot7 + geom_point() + scale_shape(solid = FALSE) +
     scale_color_manual(values = c(out = "red", `in` = "blue")) +
     guides(shape = guide_legend(title = "Residency status"))
```



Useful single variable summary plots

Histograms are produced with the help of `geom_histogram()`. The default implementation of this function uses 30 classes (bins) and produces quite a poor plot!

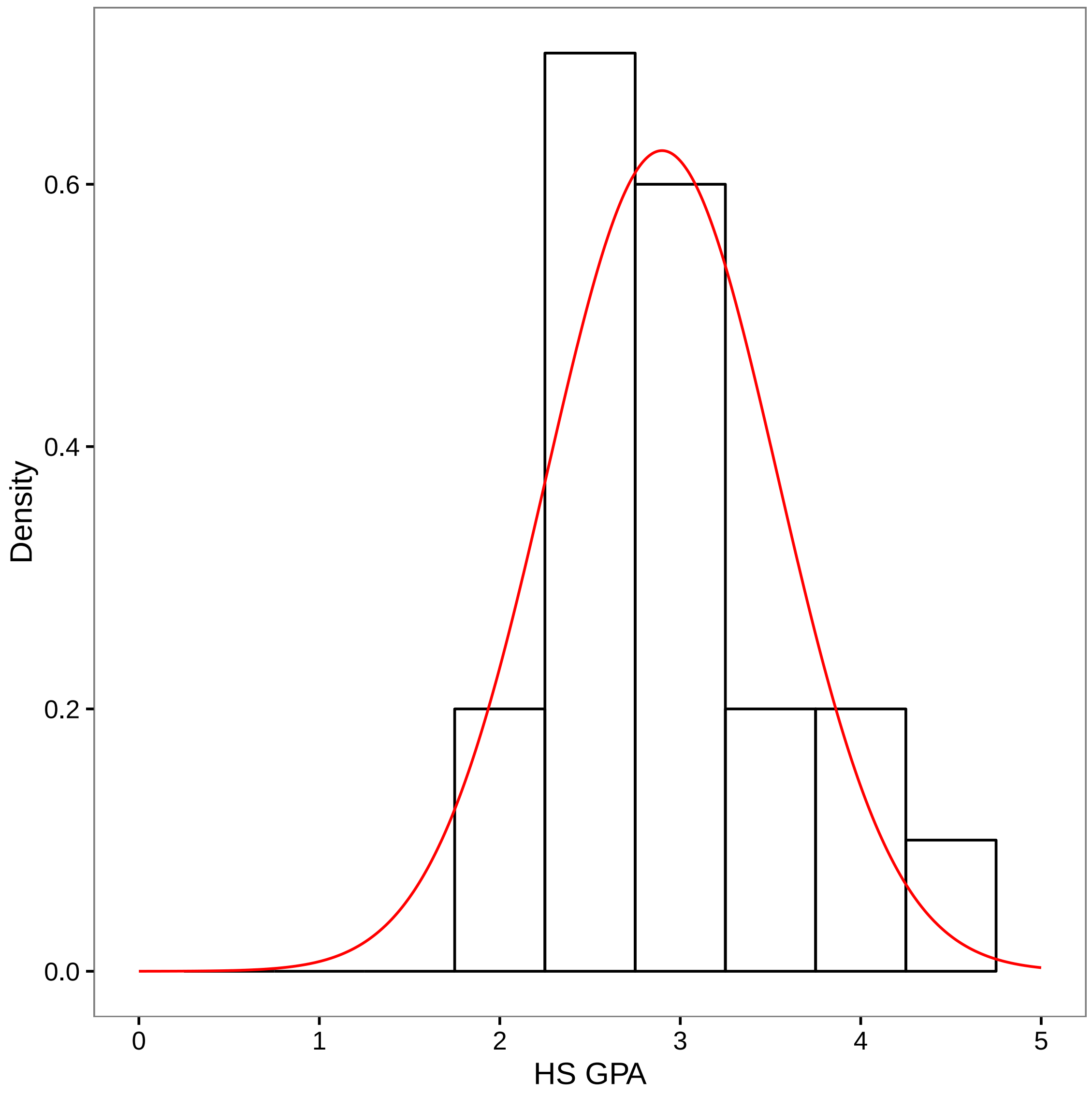
```
> ggplot(data = gpa, mapping = aes(x = HS.GPA)) + geom_histogram()
`stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```



Setting the y aesthetic is not needed because a histogram summarizes only one variable.

Below is a histogram with a normal distribution approximation. I tried to use a style similar to what was done earlier for the histogram with the `graphics` package.

```
> chris.theme.changes2 <- theme(panel.grid.major = element_blank()) +
     theme(panel.grid.minor = element_blank())
> ggplot(data = gpa, mapping = aes(x = HS.GPA)) + xlab("HS GPA") +
     ylab("Density") + chris.theme.changes2 + geom_histogram(aes(y = ..density.
     fill = NA, color = "black", binwidth = 0.5) + xlim(0,
     5) + stat_function(fun = dnorm, args = list(mean = mean(gpa$HS.GPA),
     sd = sd(gpa$HS.GPA)), color = "red", n = 1000)
```



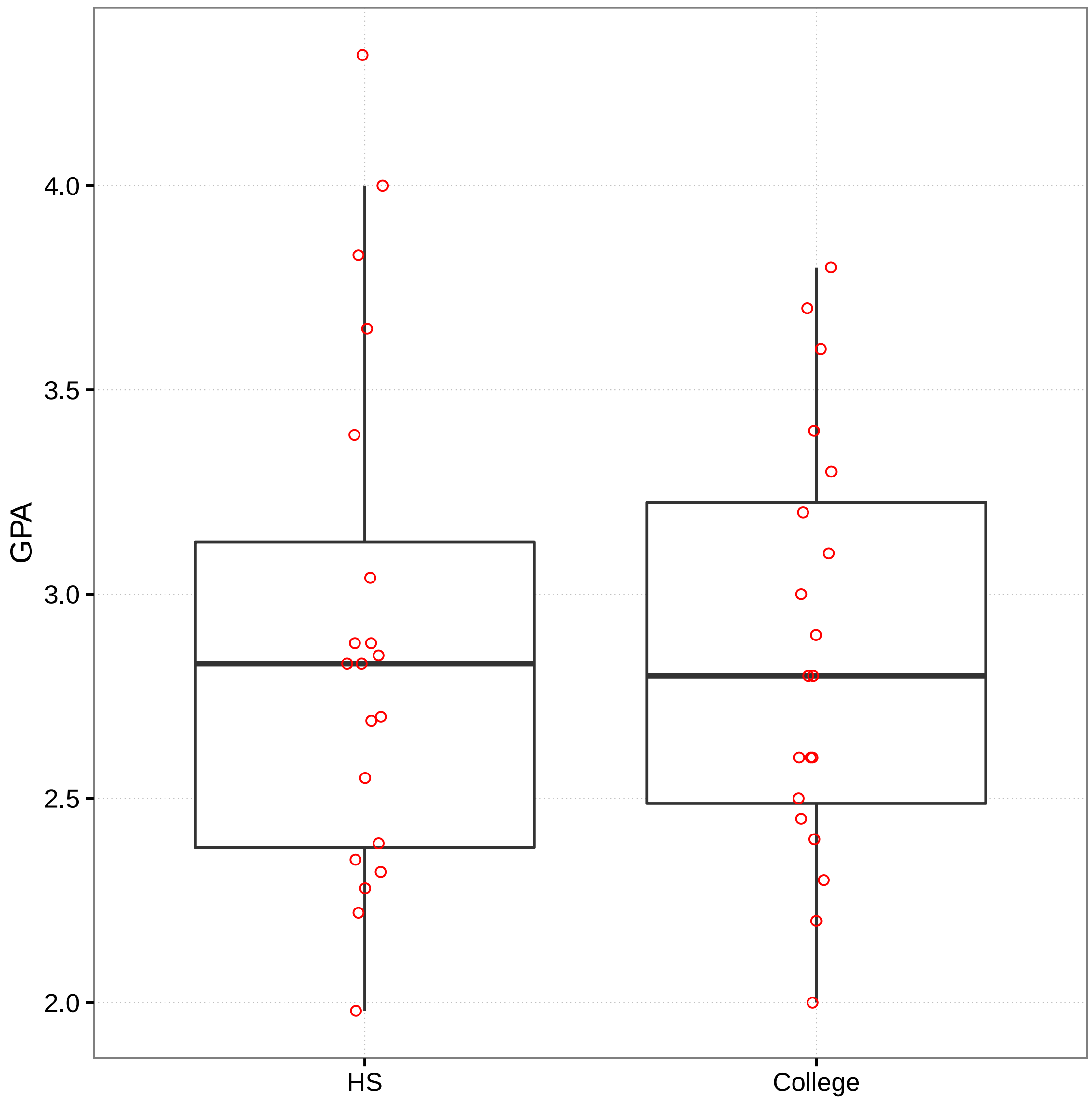
Comments:

- The height on the bars on the right side is a little different than what was obtained with `hist()`. The reason is due to the discreteness of the data and the classes chosen. The `geom_histogram()` function uses $3.5 \leq x < 4$ for the classes and `hist()` uses $3.5 < x \leq 4$.
- Rather than plotting the frequencies per class (i.e., the counts), the y-axis needs to be rescaled so that a normal density function can be plotted. This is specified by setting a `y` aesthetic named `..density..`. The double dot around "density" is used to differentiate it from an actual variable with the same name.

- The `binwidth` argument specifies the width of the classes. To specify the number of classes instead, use the `bins` argument.
- The `x` aesthetic cannot be passed into `args` for `mean` and `sd`. Therefore, I could not use a general way to find the mean and standard deviation needed for the normal distribution.
- The `n` = 1000 argument value was used with `stat_function()` to make the normal distribution more smooth.

There are times when a data frame is not in the correct format for a particular plot of interest. As described earlier for box and dot plots, one could use the `HS.college` alternative format to `gpa` to produce these plots. For `ggplot2`, this alternative format must be used:

```
> head(HS.college, n = 2)
  school  gpa
1     HS 3.04
2     HS 2.35
> tail(HS.college, n = 2)
    school gpa
39 College 2.2
40 College 2.6
> chris.theme.changes3 <- theme(panel.grid.major = element_line(color = "gray",
     linetype = "dotted")) + theme(panel.grid.minor = element_blank(),
     axis.title.x = element_blank())
> set.seed(8912)
> ggplot(data = HS.college, mapping = aes(x = school, y = gpa)) +
     ylab(label = "GPA") + chris.theme.changes3 + geom_boxplot(outlier.shape = 
     geom_point(position = position_jitter(height = 0, width = 0.1),
         shape = 1, color = "red")
```



Unfortunately, the limits for the whiskers are obtained in an unconventional manner. The upper limit is drawn to the largest observation which is smaller than $Q_3 + 1.5(Q_3 - Q_1)$, and the lower limit is drawn to the smallest observation which is larger than $Q_1 - 1.5(Q_3 - Q_1)$.

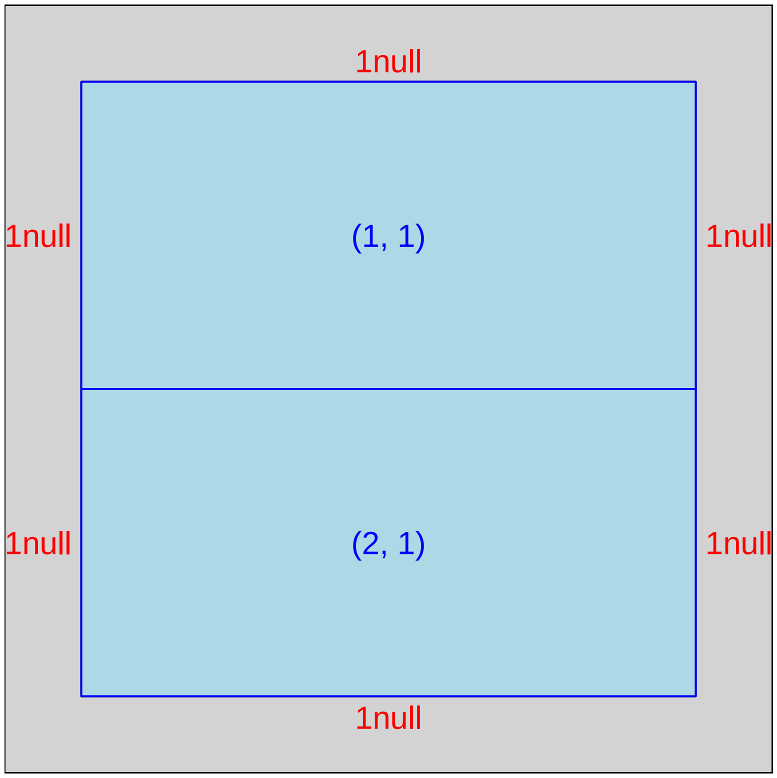
Layout of plots

The `mfrow` and `mfcol` arguments of `par()` and the `layout()` function can not be used to place multiple plots into one graphics window. Instead, we need to use the `grid` package for this purpose. This package provides a very general basis for creating graphics. Because it is so general, both the `ggplot2` and `lattice` packages are based on it. Essentially, the gglot2 and lattice packages provide an easier way to use functions and syntax from the grid package.

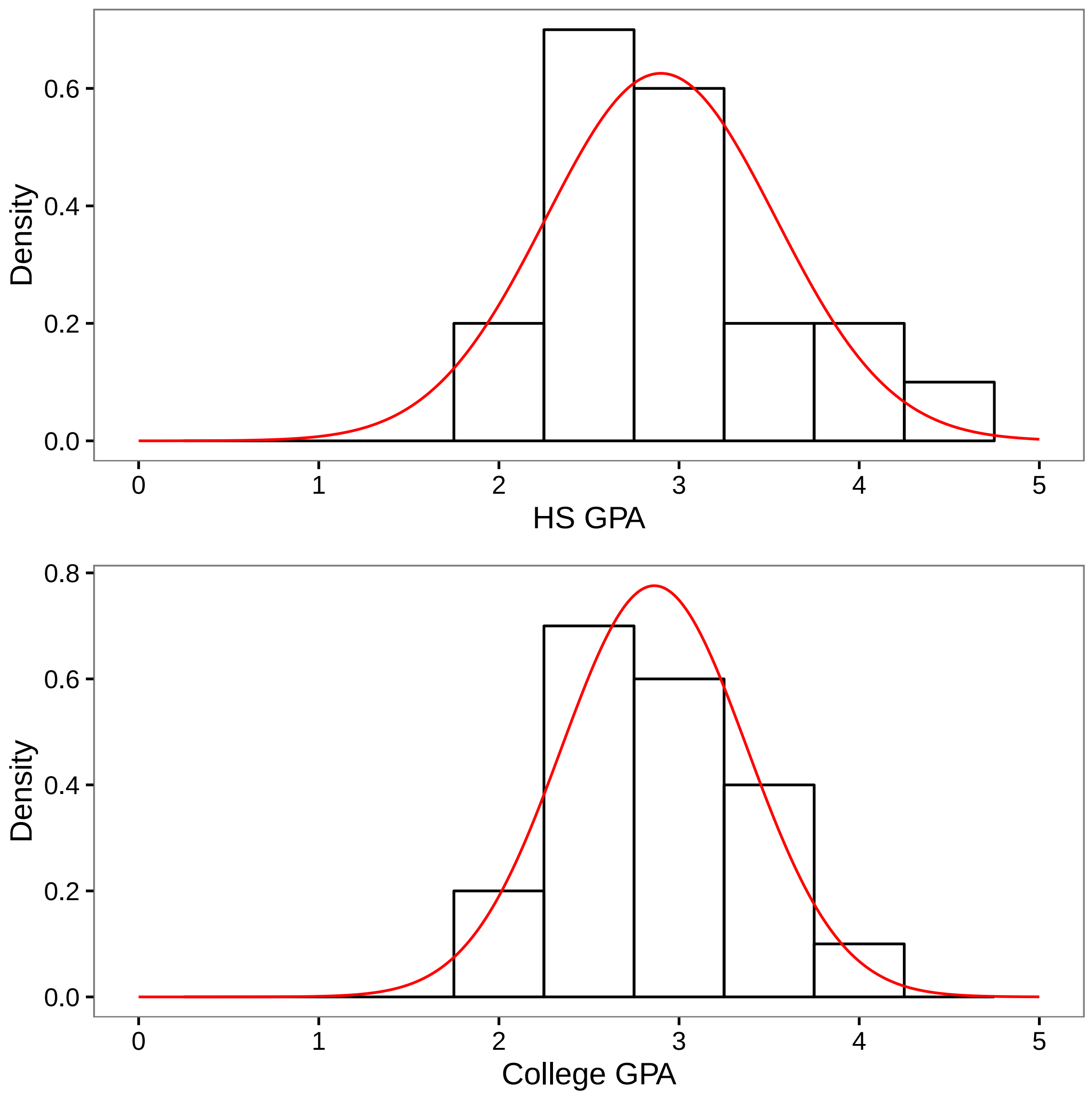
Below is how we can create a 1 row and 2 column layout of plots for a graphics window.

```
> library(package = "grid")
> # Removes any previous graphics window settings
> grid.newpage()
> layout2x1 <- grid.layout(nrow = 2, ncol = 1)
> # Push layout onto graphics window - PDF file here has an
> # extra space
> pushViewport(viewport(layout = layout2x1))
```

```
> # Show layout - null units because did not define using exact
> # measurements
> grid.show.layout(l = layout2x1)
```

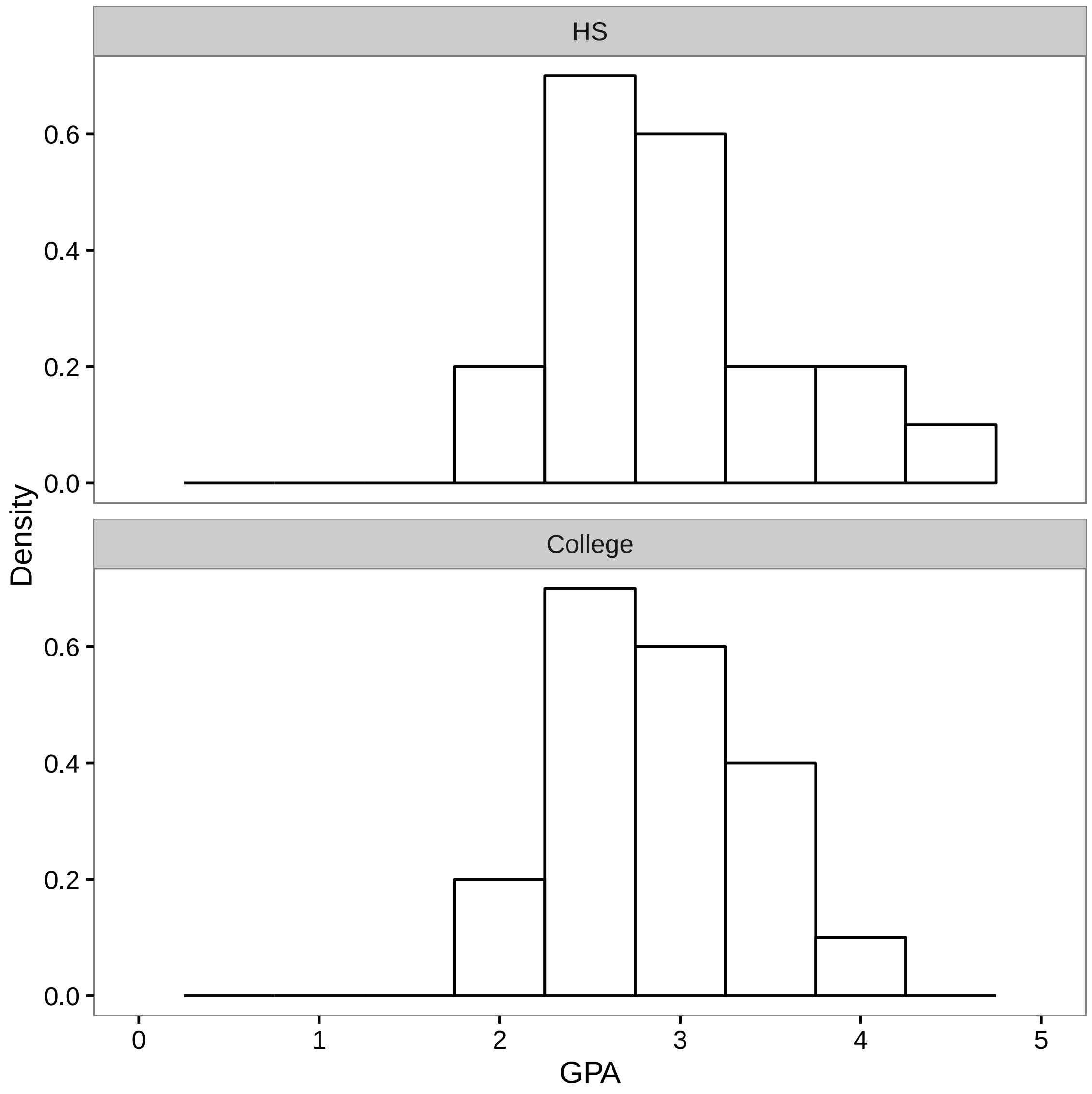


```
> plot1 <- ggplot(data = gpa, mapping = aes(x = HS.GPA)) +
     xlab("HS GPA") + ylab("Density") + chris.theme.changes2 +
     geom_histogram(aes(y = ..density..), fill = NA,
         color = "black", binwidth = 0.5) + xlim(0,
     5) + stat_function(fun = dnorm, args = list(mean = mean(gpa$HS.GPA),
     sd = sd(gpa$HS.GPA)), color = "red", n = 1000)
> plot2 <- ggplot(data = gpa, mapping = aes(x = College.GPA)) +
     xlab("College GPA") + ylab("Density") + chris.theme.changes2 +
     geom_histogram(aes(y = ..density..), fill = NA,
         color = "black", binwidth = 0.5) + xlim(0,
     5) + stat_function(fun = dnorm, args = list(mean = mean(gpa$College.GPA),
     sd = sd(gpa$College.GPA)), color = "red", n = 1000)
> pushViewport(viewport(layout = layout2x1))
> print(plot1, vp = viewport(layout.pos.row = 1, layout.pos.col = 1))
> print(plot2, vp = viewport(layout.pos.row = 2, layout.pos.col = 1))
```



Another way to place multiple plots in the same graphics window is to actually combine them into one plot based on a variable(s) in a data set. These types of plots are referred to as *Trellis plots*, and they were developed by the statistics group at AT&T Bell Labs in the early 1990s for S. They have since been incorporated into a number of packages, including `ggplot2` through its use of *faceting*.

```
> ggplot(data = HS.college, mapping = aes(x = gpa)) + chris.theme.changes2 +
    geom_histogram(aes(y = ..density..), fill = NA, color = "black",
        binwidth = 0.5) + xlim(0, 5) + ylab("Density") + xlab("GPA") +
    facet_wrap(~school, nrow = 2)
```



Comments:

- The normal distribution overlay is not included on the histograms because `stat_func()` is not meant to be used individually for each *panel* (individual plot). If this function is included, the exact same result will be given in each panel.[1]
- A smoothed density estimate could be included uniquely on each plot by using `geom_density(color = "red")`. Oddly, this has a side effect that results in a vertical red line at y = 0.
- `facet_grid()` can be used to have more control over a matrix

[1]Potential work arounds are suggested at `http://stackoverflow.com/questions/1376967/using-stat-function-and-facet-wrap-together-in-ggplot2-in-r` and `https://statbandit.wordpress.com/2011/07/29/a-ggplot-trick-to-plot-different-plot-types-in-facets` .

of panels. For example, `facet_grid(a~b)` would put individual plots into a matrix with dimension (# of levels of a)×(# of levels of b). Also, `facet_grid(a+b~.)` creates a matrix with dimension (# of levels of a + # of levels of b)×1.

- The `scales = "free"` argument can be included in a facet function to allow the x and y-axis scales to vary among the panels.

One of the most beneficial aspects of using Trellis plots is to plot multivariate data. For example, suppose a data set has 3 variables measured on a continuous scale and 2 variables that are categorical. Scatter plots for two of the continuous variables can be constructed by conditioning on the levels of the other variables (including the remaining continuous variable by using a *shingle* through creating a new categorical variable with `cut_interval()`).

Additional notes

- The `ggplot()` function creates an object that contains the basic set-up for a particular plot. Parts of the plot are created by using geom's and other functions. To help see this, suppose the code for the first plot created in this section is saved into an object and then we examine components of it:

```
> temp1 <- save.plot + geom_point(color = "red")
> class(temp1)

[1] "gg"     "ggplot"

> methods(class = "gg")

[1] +
see '?methods' for accessing help and source code

> methods(class = "ggplot")
```

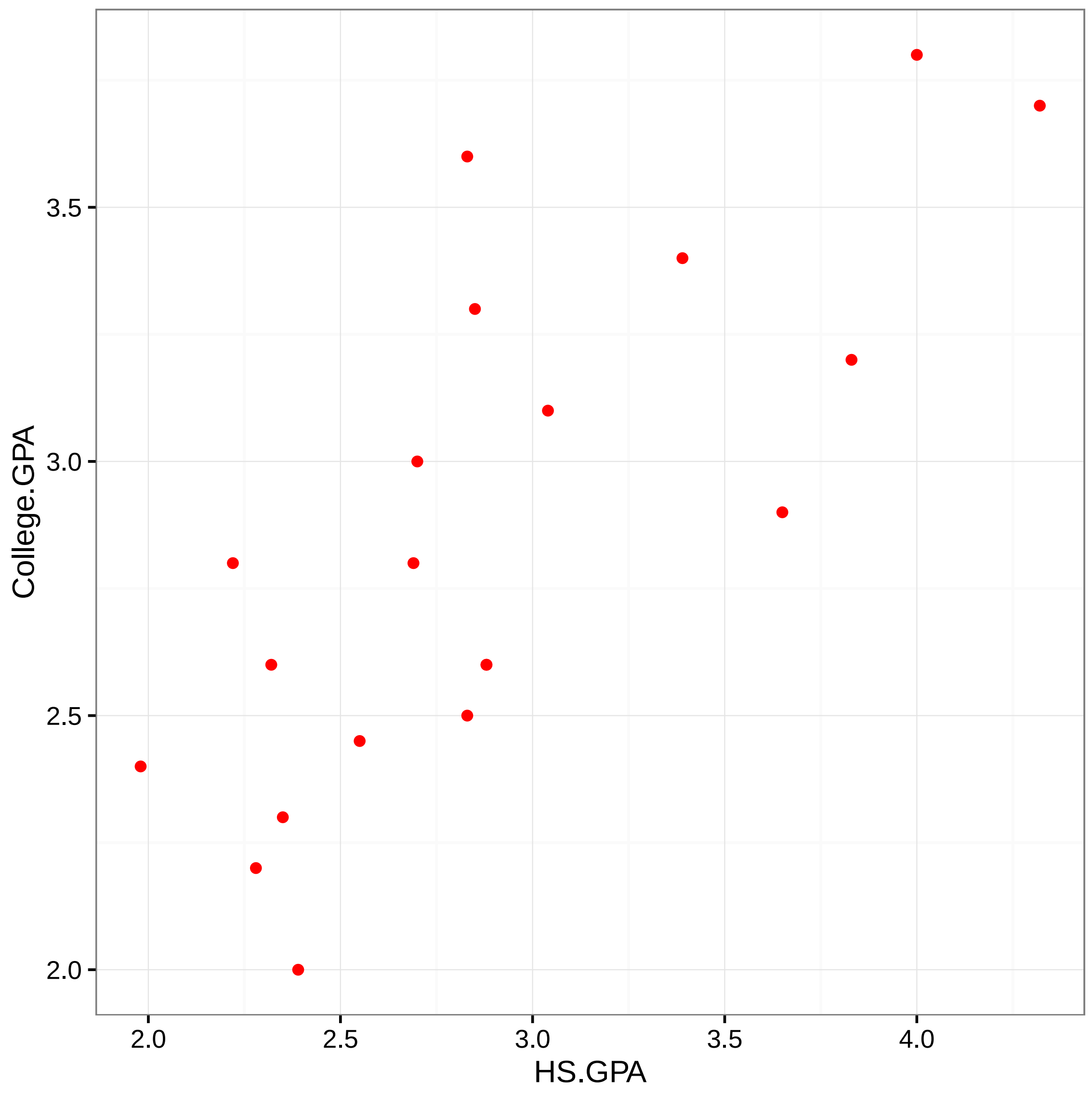
```
[1] grid.draw plot      print     summary
see '?methods' for accessing help and source code

> getAnywhere(print.ggplot)  # Method function

A single object matching 'print.ggplot' was found
It was found in the following places
  registered S3 method for print from namespace ggplot2
  namespace:ggplot2
with value

function (x, newpage = is.null(vp), vp = NULL, ...)
{
    set_last_plot(x)
    if (newpage)
        grid.newpage()
    grDevices::recordGraphics(requireNamespace("ggplot2", quietly = TRUE),
        list(), getNamespace("ggplot2"))
    data <- ggplot_build(x)
    gtable <- ggplot_gtable(data)
    if (is.null(vp)) {
        grid.draw(gtable)
    }
    else {
        if (is.character(vp))
            seekViewport(vp)
        else pushViewport(vp)
        grid.draw(gtable)
        upViewport()
    }
    invisible(data)
}
<environment: namespace:ggplot2>

> # getAnywhere(summary.ggplot) # Method function
> print(temp1)  # Show plot
```



```
> summary(temp1)

data: HS.GPA, College.GPA [20x2]
mapping:  x = HS.GPA, y = College.GPA
faceting: facet_null()
-----------------------------------
geom_point: na.rm = FALSE
stat_identity: na.rm = FALSE
position_identity

> names(temp1)

[1] "data"          "layers"        "scales"        "mapping"       "theme"
[6] "coordinates" "facet"         "plot_env"      "labels"

> temp1$mapping
```

```
* x -> HS.GPA
* y -> College.GPA

> temp1$layers

[[1]]
geom_point: na.rm = FALSE
stat_identity: na.rm = FALSE
position_identity

> temp1$labels

$x
[1] "HS.GPA"

$y
[1] "College.GPA"
```

- Note that `print()` is a generic function used with all packages that is run whenever an object is given alone at a command prompt. Thus, if `x <- 1`, then `print(x)` and `x` issued at a command prompt will run `print.default()`.
- The `print.ggplot()` function uses functions from the `grid` package to take what is in `temp1` to create a plot.
- My program contains another example of showing what is inside a more complicated object created with `ggplot()`. This example corresponds to the scatter plot with the regression model and confidence interval bands.

- I have found the documentation at `http://docs.ggplot2.org/current` to be better than the documentation via `help()`.
- Chapter 3 of my STAT 873 lecture notes discusses how to use the `lattice` package. This package is very good at constructing Trellis plots.

- The `last_plot()` function provides a way to add layers to the current plot in the graphics window. For example, `last_plot() + geom_vline(xintercept = 5)` adds a vertical line at x = 5.

- Many individuals think `ggplot2` is the best package for graphics. I think the `graphics` package is better except when Trellis plots are needed. I especially dislike some of the defaults in `ggplot2`, its flexibility, and `qplot()` for often drawing the wrong type of plot. For example, consider a situation where one needs to be construct a complex plot for a paper to be published. Roughly speaking, the amount of code needed to get the plot 80% done, say, is less for `ggplot2` than for `graphics`. However, the overall code for a 100% done plot is often less for `graphics` than for `ggplot2`.

- The current edition of the *R Graphics* book has chapters on `grid` and `ggplot2`. Example programs are available at `http://www.stat.auckland.ac.nz/~paul/RG2e`.

- There are also a number of packages that specialize in specific types of plots. For example, the `rgl` package produces 3D, rotatable plots. Chapter 3 of my STAT 873 lecture notes and Chapters 8 and 11 of my STAT 870 lecture notes provide examples of how to use this package.