Visualizing Data in R

ANTH 3720-001 Archaeological and Forensic Science Lab Methods: Data Analysis with R

Elic Weitzel

Jan. 29-31, 2021

If you would like the original R Markdown file, find it on my GitHub page at https://github.com/weitzele/Basic-R-Tutorial

1 Visualizing Data

Visualizing data is one of the most important aspects of data science. Graphic representations of your data are often the key way in which you communicate information to your audience.

There are an incredible variety of ways to visualize data. Some approaches are better suited towards certain types of data than others, and here we will discuss some of the most common visualization methods used for archaeological and forensic data.

2 Scatterplots

Scatterplots are an incredibly useful and versatile visualization technique for presenting data. They are a simple and useful way to throw data onto a plot with two dimensions and see each data point individually.

Let's load some data to visualize from the `archdata` package. This is a package of real archaeological datasets compiled by David Carlson as a companion to his book *Quantitative Methods in Archaeology Using R*. Install the package either by clicking on the Packages tab in the bottom right of RStudio, or using the `install.packages()` function. Then, once the package is installed, make sure to run `library(archdata)` to load this package into our R session.

```
library(archdata)
```

```
## Warning: package 'archdata' was built under R version 4.0.3
```

To create a scatterplot in R, we use the `plot()` function. If you inspect the documentation for this function, you'll find some very useful information on plotting and you'll see that the basic syntax for using this function is `plot(x, y)`:

```
?plot
```

Therefore, to create a plot with this function, simply supply it with data to plot on the x-axis of the graph and corresponding data to plot on the y-axis.

Let's look at one particular dataset from the `archdata` package called ESASites, which the documentation describes as a dataset on 43 Early Stone Age archaeological assemblages from Norway. You can inspect this dataset using `?`, and then load it into your environment using the `data()` function. Once you run `data(ESASites)`, an object named `ESASites` will appear in the environment in the top right quadrant of your screen. You'll see it has 43 rows representing different assemblages and 16 columns representing different types of artifacts.

```
?archdata

?ESASites
```

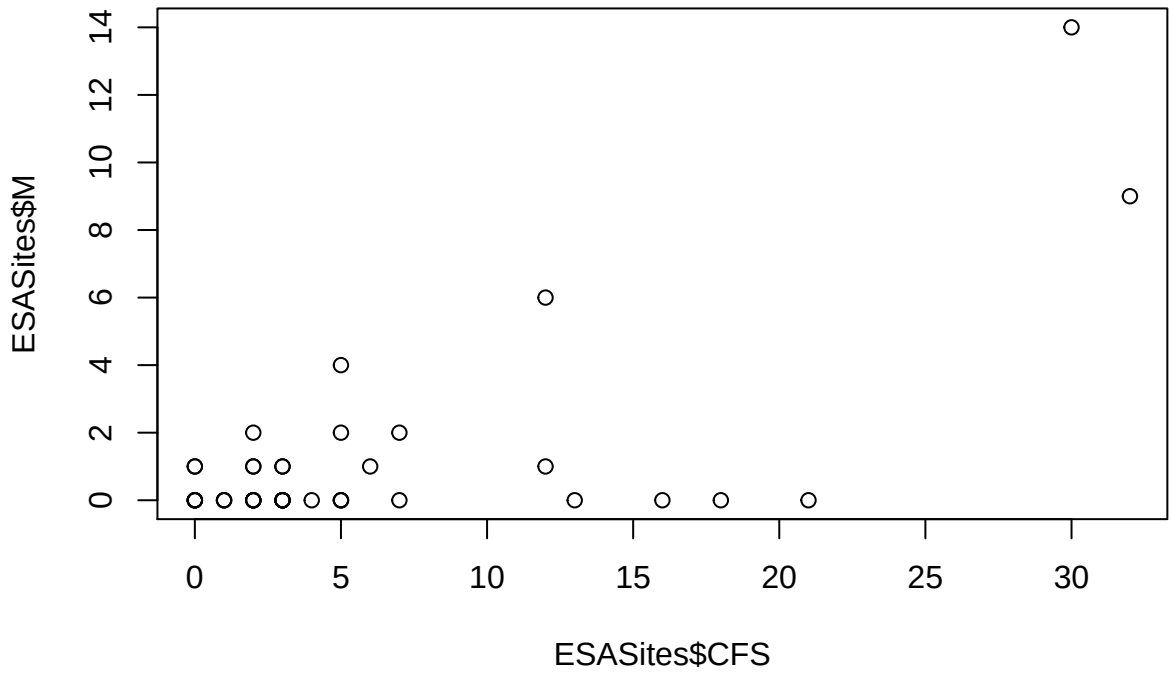
```
data("ESASites")
```

With the data loaded, let's see if there's a relationship between the number of microliths (M) and the number of core and flake scrapers (CFS) for each of the 43 assemblages. To do so, we will plot microliths on the y-axis and core and flake scrapers on the x-axis, specifying this in the `plot()` function as follows:

```
colnames(ESASites) #view the abbreviations for each artifact type
```

```
##  [1] "TA"  "BA"  "TOA" "AA"  "M"   "FK"  "BK"  "NK"  "CFS" "BS"  "DS"  "Bu"
## [13] "Ax"  "Ch"  "SAx" "Pf"
```

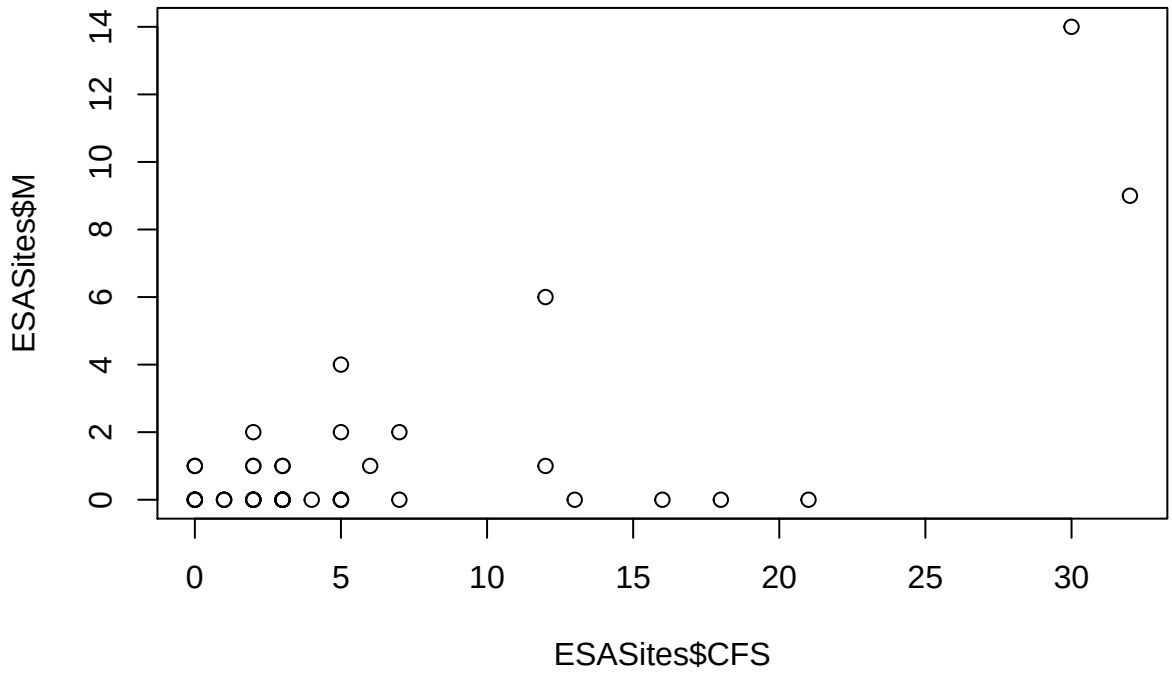
```
plot(x = ESASites$CFS, y = ESASites$M) #plot the data
```



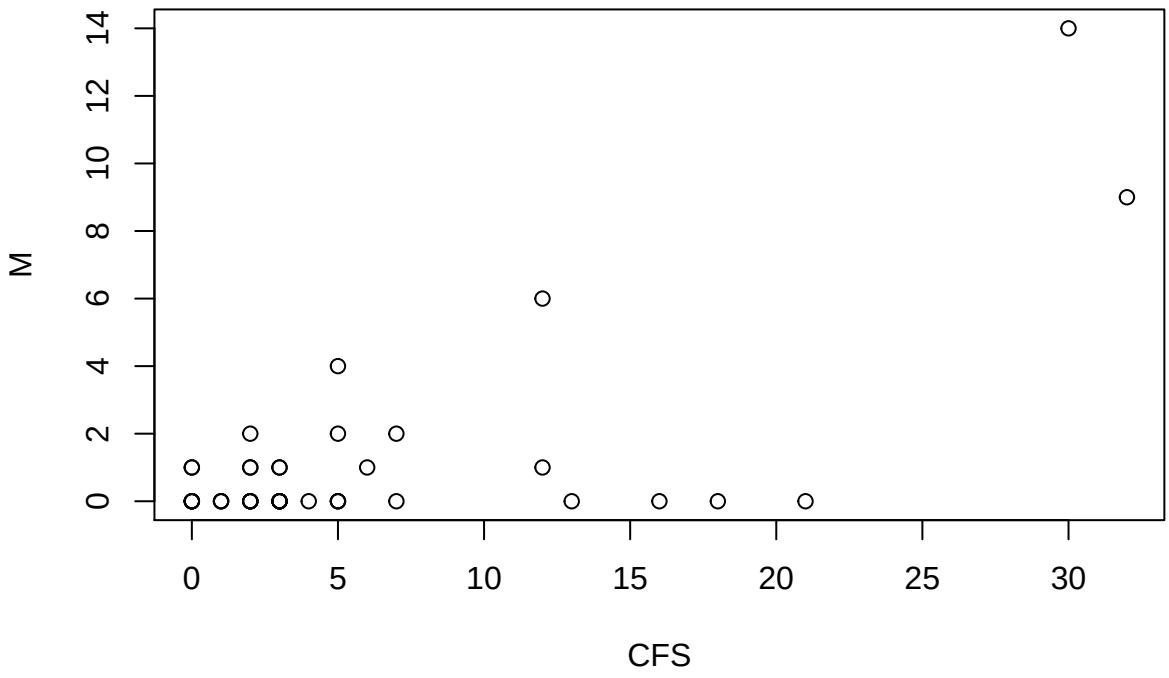
And voila, we have a plot! It looks like there is a positive, but perhaps weak, relationship between the count of scrapers and microliths.

The `plot()` function in R is incredibly versatile, so now we can explore some more ways to use it. For starters, I don't often use the syntax that we just used to plot data. I use the following syntax:

```
plot(ESASites$M ~ ESASites$CFS) #option 1
```




```
plot(M ~ CFS, data = ESASites) #option 2
```



The `plot()` function permits you to use a tilde (`~`) when plotting. It can roughly be translated to *"as a function of"* in this context, thus `y ~ x` can be verbalized as "y as a function of x". In the plot function above, which plots `ESASites$M ~ ESASites$CFS`, this represents microliths as a function of core and flake scrapers. Note that when using this syntax, the y-axis variable is specified first and the x-axis variable comes second, after the `~`.

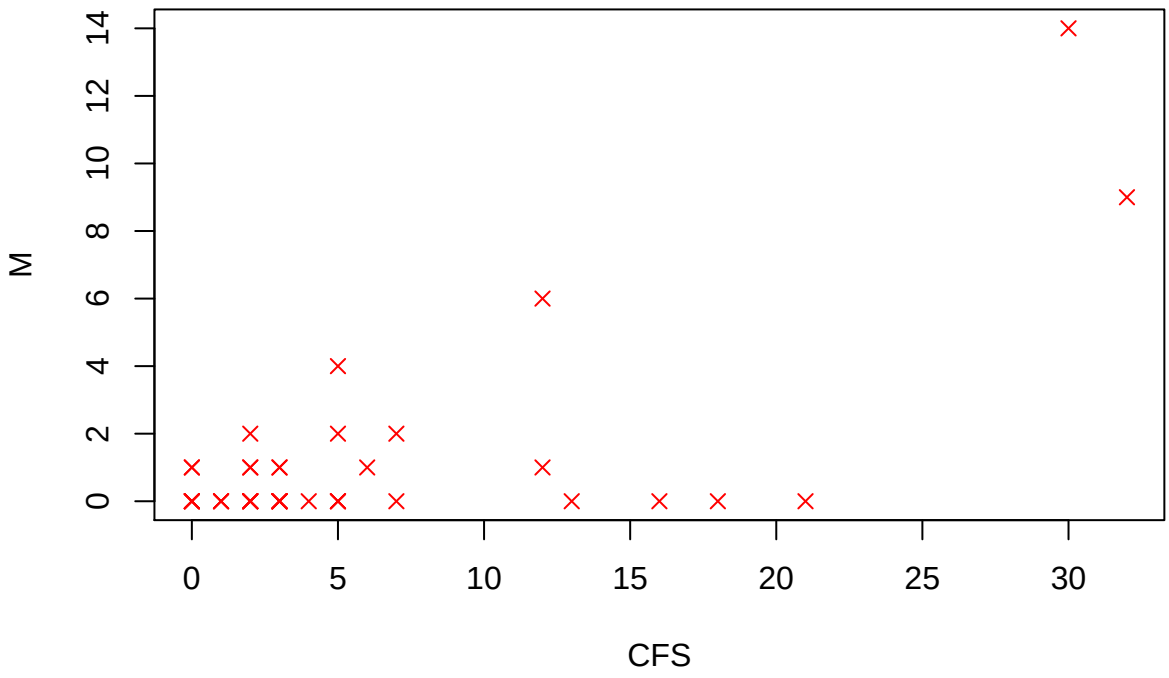
In the code above, option 1 and 2 illustrate two different ways to say the same thing using `plot()`. In option 1, `M` and `CFS` are specified by using the `$` operator to tell R to look in the `ESASites` data frame for columns of those names.

In option 2, I employ an additional argument of the `plot()` function: `data`. By telling R that the `data` for this plot come from the object `ESASites`, you can avoid having to using the `$` operator which results in less code as you only have to type `ESASites` once instead of twice. I normally use this syntax when plotting, though there are circumstances in which other options may be better so you should be aware of them all.

2.1 Point Styles and Colors

Now, this plot is rather dull looking... It is of course possible to change the colors and styles and everything else about this plot. Let's start by changing the style of the points using the `pch` argument in the `plot()` function, and then by changing the color of the points using the `col` argument.

```
plot(M ~ CFS, data = ESASites, pch = 4, col = "red")
```



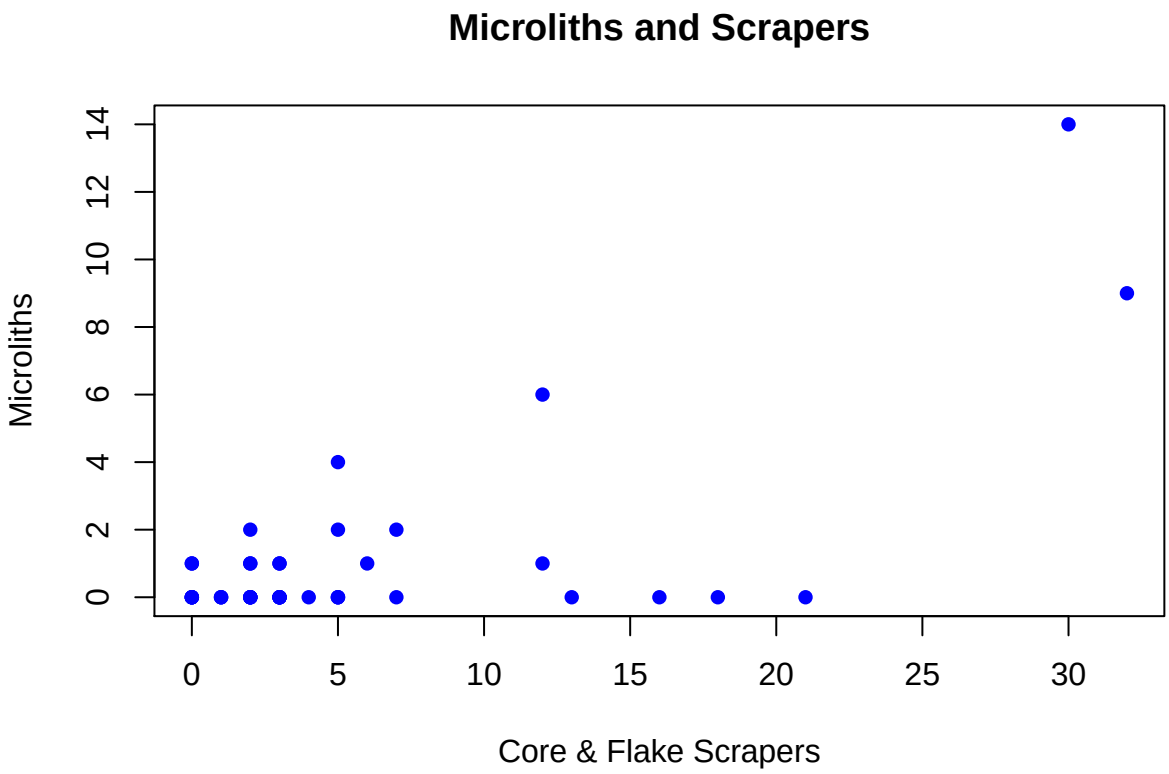
Now, the data are shown as red x's in our plot. The `pch` argument contains 25 different point styles ranging from filled and open circles to triangles to squares to even more complex shapes. Similarly, the `col` argument permits you to change the color of these points according to a wide variety of colors that R knows.

Experiment a bit by changing `pch` to different values between 1 and 25 and by changing `col` to other common colors. Note, however, that the value provided to `pch` is an integer while that provided to `col` is a character. In R, character values in this context must be given in quotation marks. Thus, `col = blue` will not work, but `col = "blue"` will.

2.2 Axis Labels and Titles

Now, let's add more arguments to our plot to make it even better looking. We can specify the `xlab` and `ylab` arguments to change the x-axis and y-axis labels, respectively. Remember, as character strings, axis labels must be in quotation marks. Let's also add a main title to the graph using the `main` argument.

```
plot(M ~ CFS, data = ESASites, pch = 16, col = "blue", xlab = "Core & Flake Scrapers",
     ylab = "Microliths", main = "Microliths and Scrapers")
```



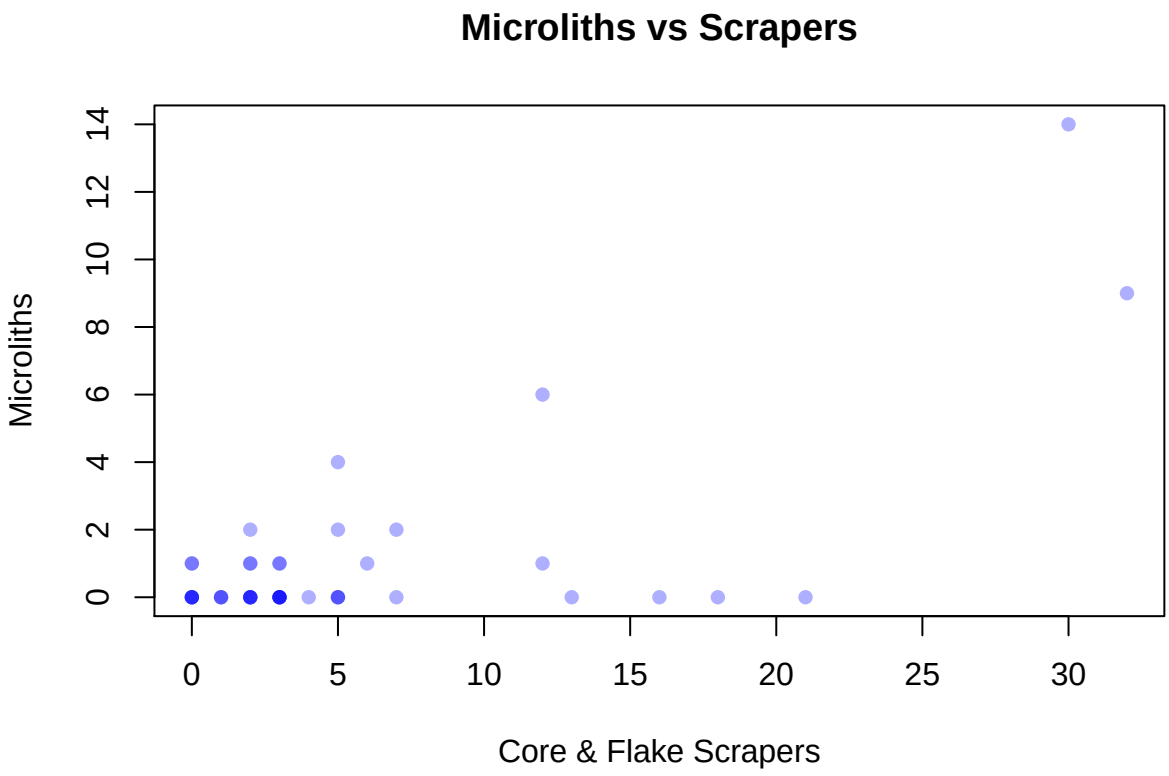
2.3 Color Transparency with Hex Codes

This plot is looking pretty good. However, if you count the points on the plot, there are only 23. There are supposed to be 43 based on the number of assemblages in this dataset. Where are the other 20 points? They're there, but we can't see them because they're overlapping with other points because multiple rows in our data frame share the same values of CFS and M.

Let's make all of these data more easily seen by changing the transparency of our points. The easiest way to do this is to switch from using a character string to specify the colors as `blue` or `black`, and instead use hexadecimal values, or hex values. Every color has a six digit hex code associated with it which is based on color theory. For example, the hex code for black is `#000000` while the code for red is `#FF0000`. Note that a pound sign (#) must be inserted prior to the hex code. When using hex color codes in plotting, you still have to put the code in quotation marks as you would if you were specifying a character string for the `col` argument. You can Google search for any color's hex code.

Then, to make a color transparent, simply add a two-digit value to the end of the hex code that represents *percent transparency*. Thus, as `#FF0000` is red, a hex code of `#FF000050` is red, but 50% transparent. So let's make our blue dots on our plot transparent.

```
plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF50", xlab = "Core & Flake Scrapers",
     ylab = "Microliths", main = "Microliths vs Scrapers")
```



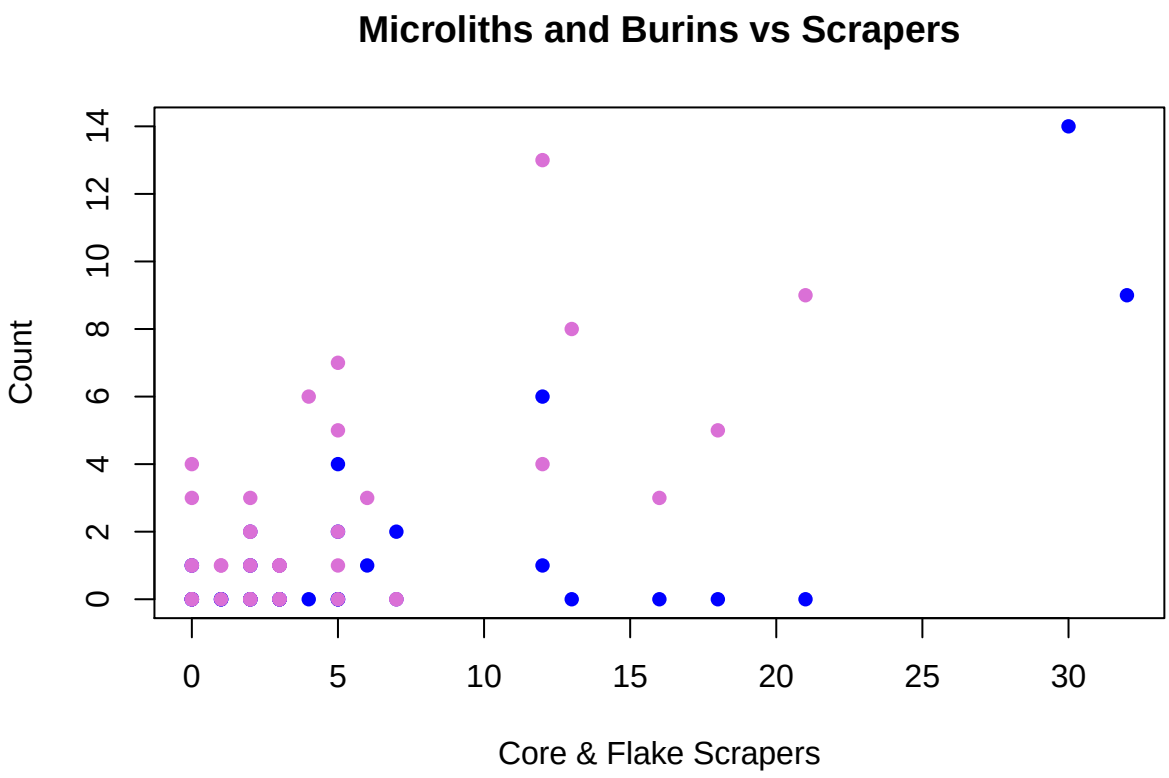
Note that with the blue points on the plot, some points appear to still be very dark blue while others are very light blue. This is because, in some cases, the plot is placing several transparent blue points on top of each other. Where there is only one transparent blue point, the color appears lighter, whereas in spots where multiple transparent blue points are superimposed, the color appears darker. This is a useful plotting trick when you have overlapping points.

2.4 Adding Additional Data with the `points()` Function

Let's add a bit more complexity to our plot, now that we have a handle on some basic plotting arguments.

What if we want to visualize both microliths as a function of scrapers, but also burins? We could add these new data points to our plot using a second line of code and the `points()` function, which operates similarly to the `plot()` function but is used to add points to an already-existing plot, not create a new one. Burins are coded as `Bu` in the `ESASites` dataset.

```
plot(M ~ CFS, data = ESASites, pch = 16, col = "blue", xlab = "Core & Flake Scrapers",
     ylab = "Count", main = "Microliths and Burins vs Scrapers")

points(Bu ~ CFS, data = ESASites, pch = 16, col = "orchid")
```

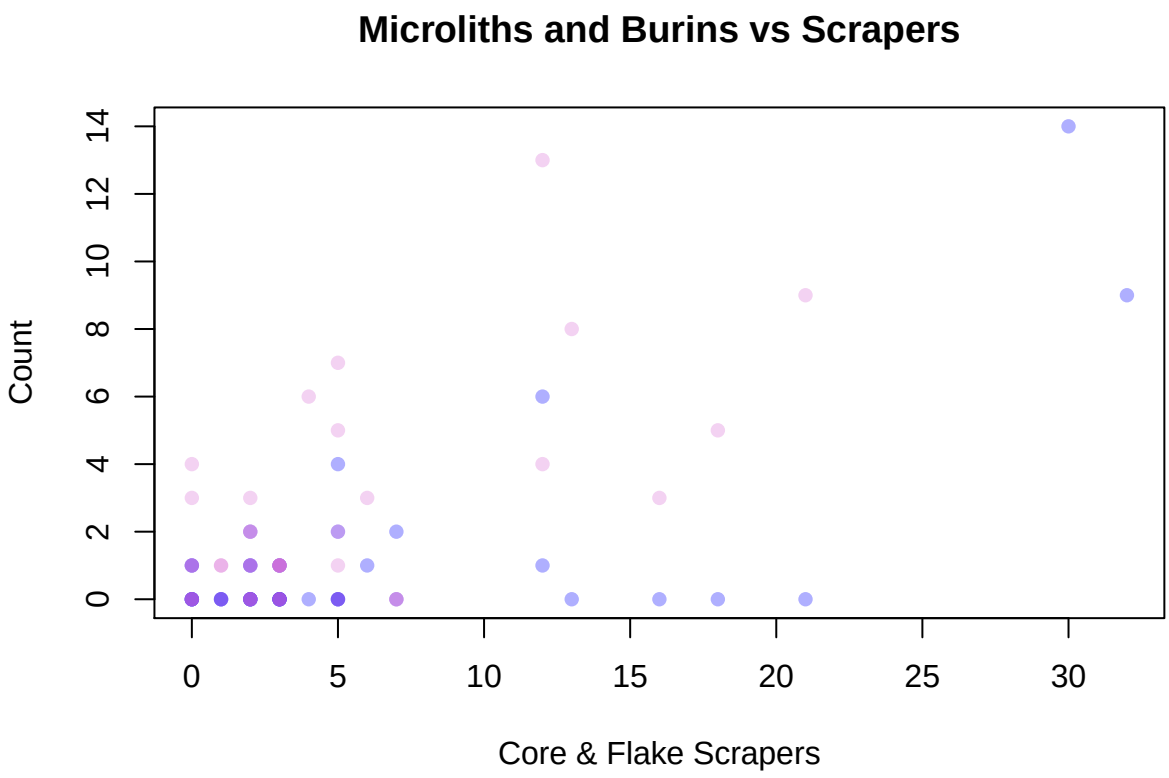


Now, we have a plot which shows the counts of microliths (in blue) and burins (in purple) as a function of the count of core and flake scrapers. I changed the labels a bit to reflect this new plot.

However, because we ran the `points` function after the `plot` function, some of the purple points are plotting on top of the blue points, covering them up and making it seem like there are only a few data points for microliths. There are a few options for making our plot better in light of this superimposition.

We could try and make the purple and blue points transparent, as we did above.

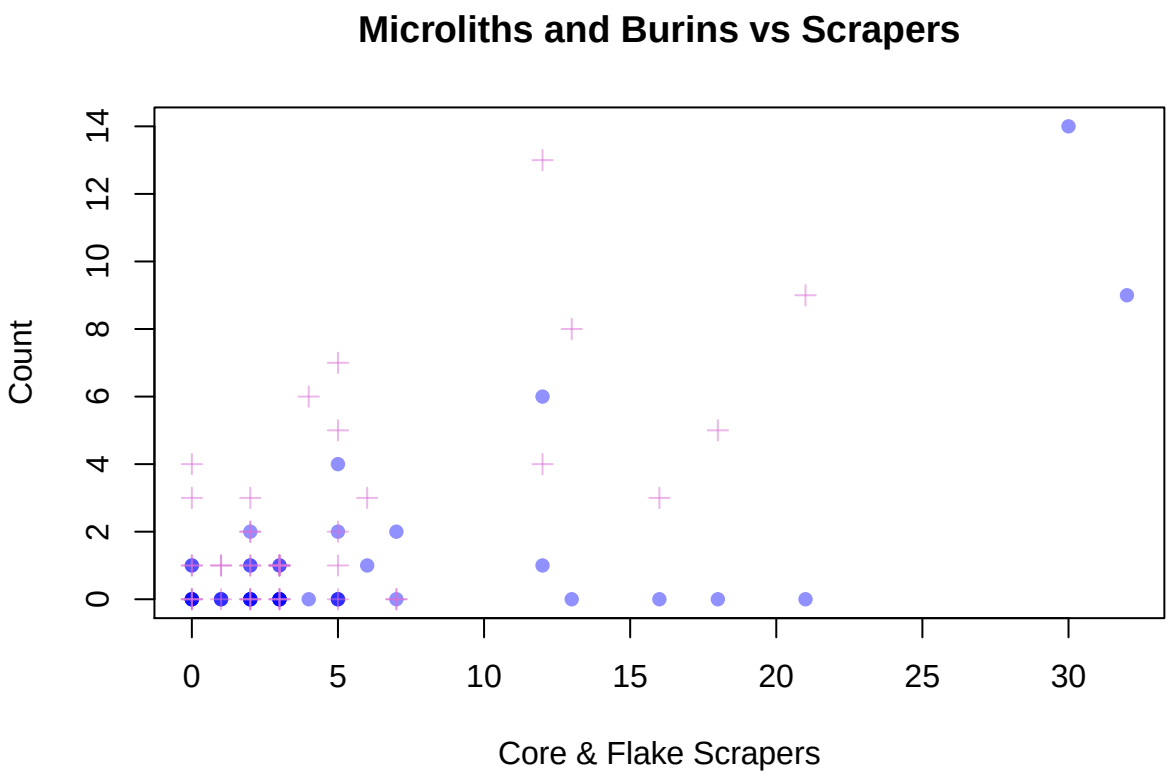
```
plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF50", xlab = "Core & Flake Scrapers",
     ylab = "Count", main = "Microliths and Burins vs Scrapers")

points(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d650")
```



This looks better! It's possible that things could still get a bit confusing where there are multiple blue points and multiple purples points all superimposed, though... If you really feel like you need to see each individual point, there are some additional options.

We could change the `pch` value of the burin points to a shape which will allow the blue circles to still appear:

```
plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "Core & Flake Scrapers",
     ylab = "Count", main = "Microliths and Burins vs Scrapers")

points(Bu ~ CFS, data = ESASites, pch = 3, col = "#da70d680")
```

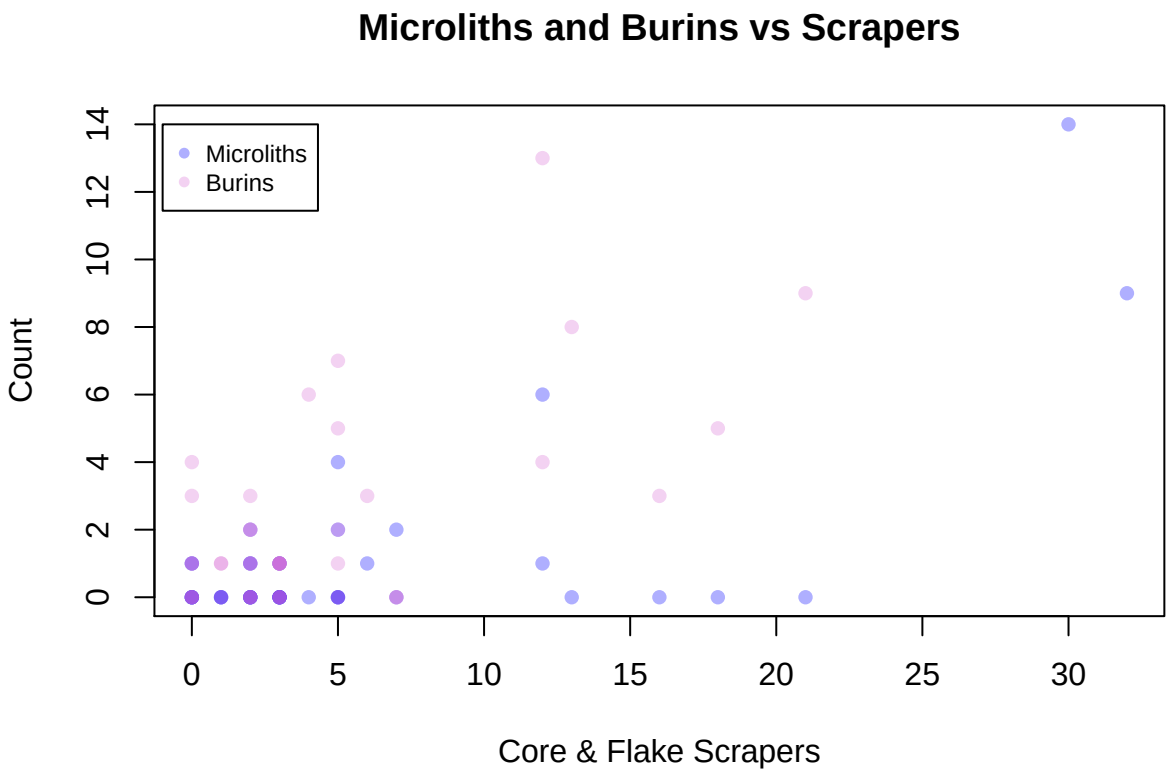


Now we can more easily see where blue and purple points are overlapping. Though this still may not be ideal for your purposes, so there are additional options.

2.5 Legends

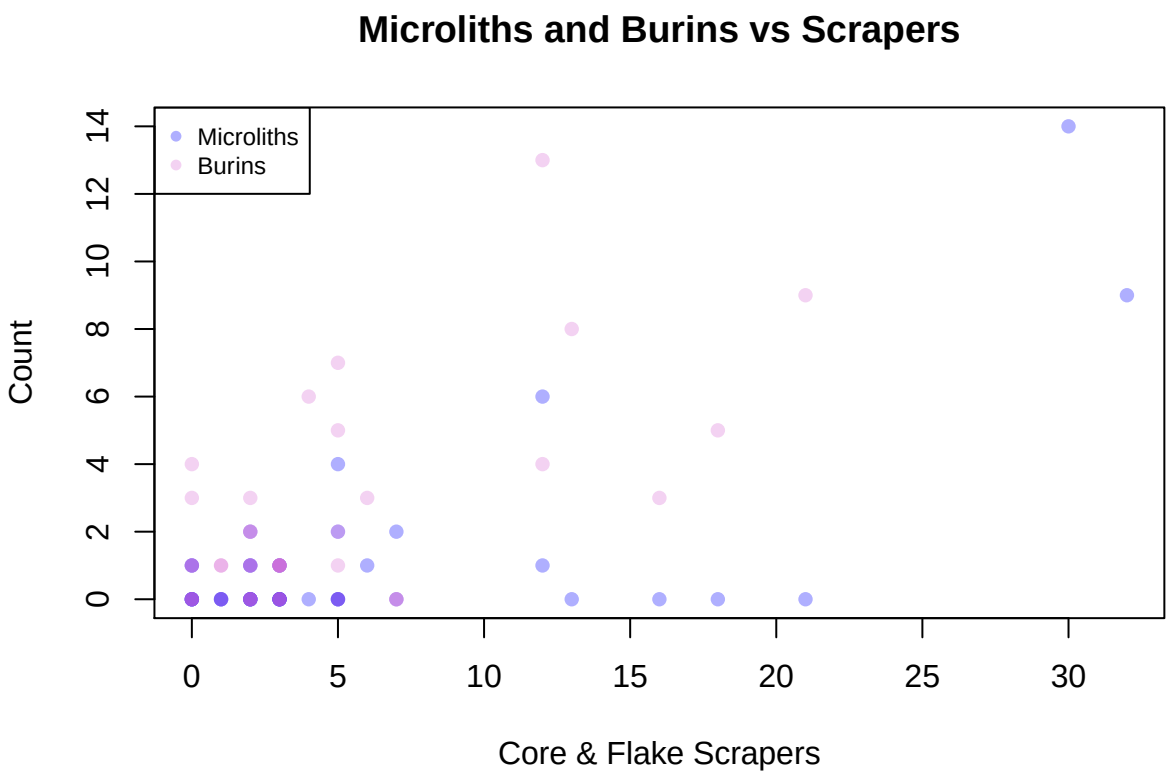
With plotting multiple sets of data on the same plot, we'll often want to add a legend to explain which points belong to which variable. We can use the `legend()` function to do this.

```
plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF50", xlab = "Core & Flake Scrapers",
     ylab = "Count", main = "Microliths and Burins vs Scrapers")

points(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d650")

legend(x = -1, y = 14, legend = c("Microliths", "Burins"), pch = 16, col =
         c("#0000FF50", "#da70d650"), cex = 0.75)
```



The `legend()` function can get a bit complicated to work with, but it can also make your life much easier when plotting. The `legend()` function above is set to draw at coordinates of -1 and 14, but you can manipulate this as you see fit using trial and error. You can also very conveniently specify "topleft" or "bottomright" or something like that, and the `legend()` function will understand you.

```
plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF50", xlab = "Core & Flake Scrapers",
     ylab = "Count", main = "Microliths and Burins vs Scrapers")

points(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d650")

legend("topleft", legend = c("Microliths", "Burins"), pch = 16, col =
         c("#0000FF50", "#da70d650"), cex = 0.75)
```



To make this legend work, I'm telling it to plot itself in the "topleft" and that the two pieces of information it needs are related to "Microliths" and "Burins". I do this in the `legend` argument. I then tell the legend that the point types are both `pch = 16`, though I could also do this by typing `pch = c(16, 16)`. I then tell it the two colors of the two legend elements using the `col` argument. Finally, because I don't want the legend to be huge, I make it a bit smaller using the `cex` argument. The default is `cex = 1` so I shrink our legend a bit by specifying `cex = 0.75`.

2.6 Multi-Panel Plots

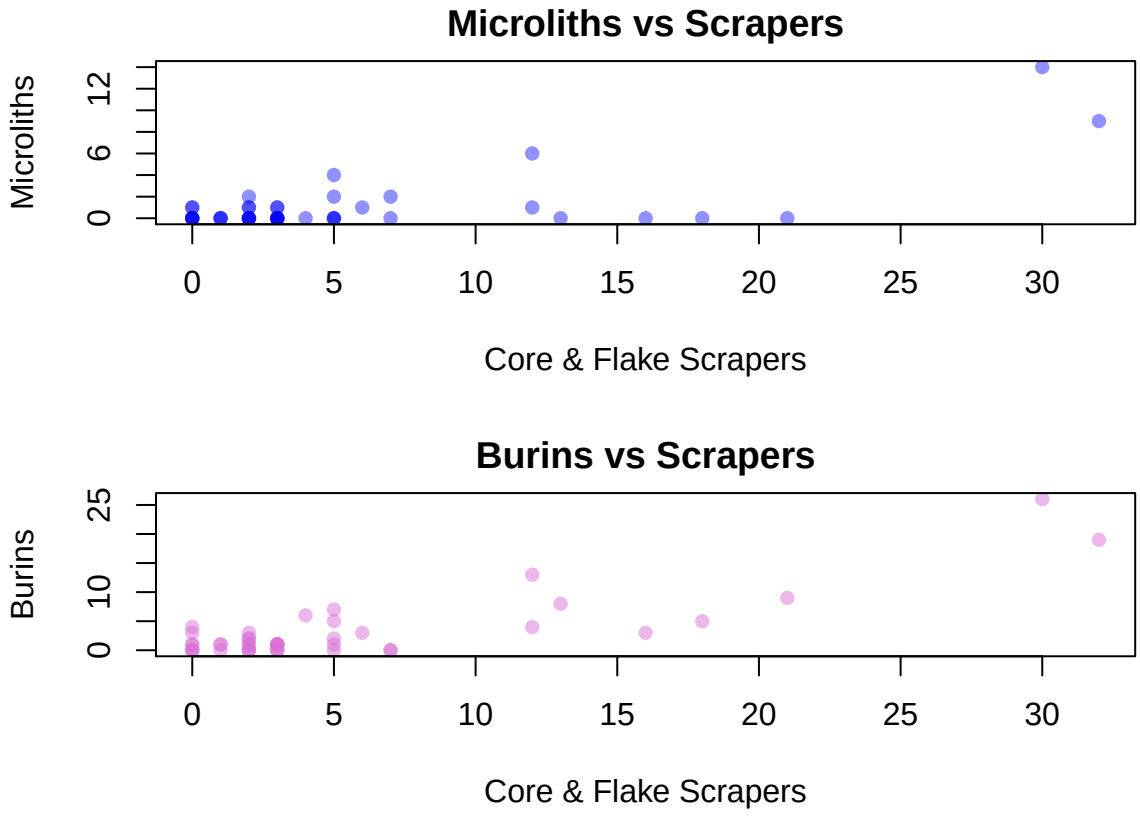
Instead of trying to fit both microliths and burins onto the same plot, we could split them up between two plots. R allows us to make both of these plots part of the same image, though, in effect creating a two-panel plot.

To create such a figure, we need to adjust the plotting parameters. We do this using the `par()` function and the arguments it takes. To start with, let's adjust the `mfrow` argument, which controls how many different plots we can add to our figure. Let's specify `mfrow = c(2, 1)` which means we want to create two rows of plots and one column, as R always specifies `c(rows, columns)`.

We next want to add another argument to our `par()` function - the `mar` argument - which controls the margins of our figure. The default margins setting for R are `mar = c(5.1, 4.1, 4.1, 2.1)`. Often, when trying to create multi-panel plots, you run into errors because the default margin settings are not compatible with the size of the figure you're trying to create. Therefore, let's adjust the `mar` argument to avoid this. I arrived at the values specified below through a bit of trial and error.

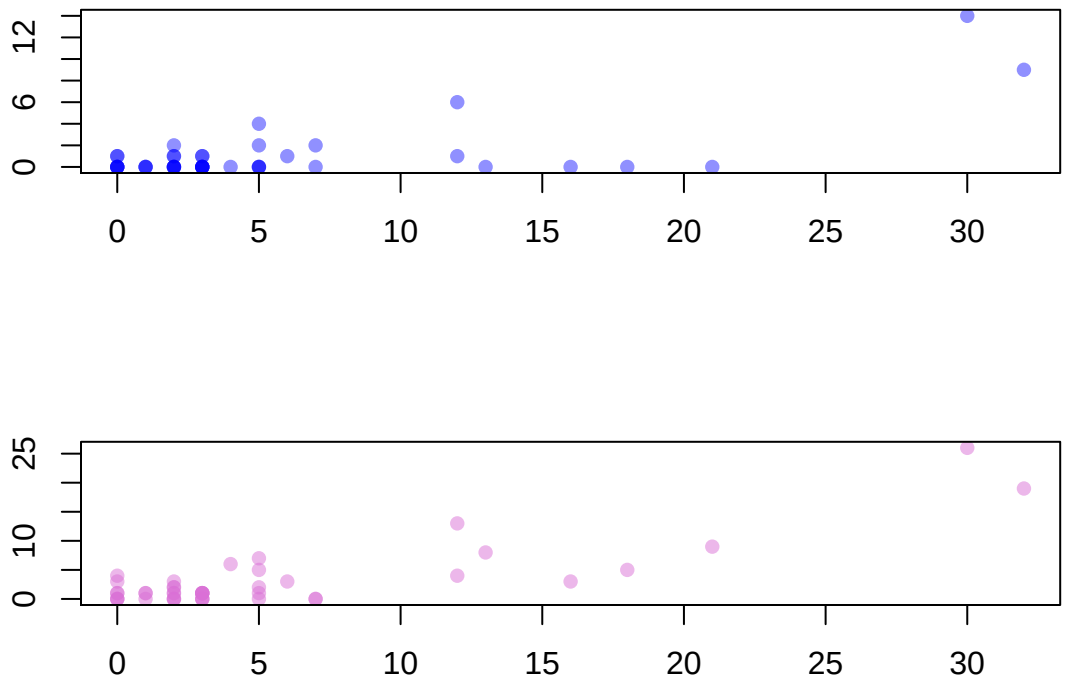
```
par(mfrow = c(2, 1), mar = c(5, 5, 2, 2))
```

```r
plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "Core & Flake Scrapers",
     ylab = "Microliths", main = "Microliths vs Scrapers")

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "Core & Flake Scrapers",
     ylab = "Burins", main = "Burins vs Scrapers")
```



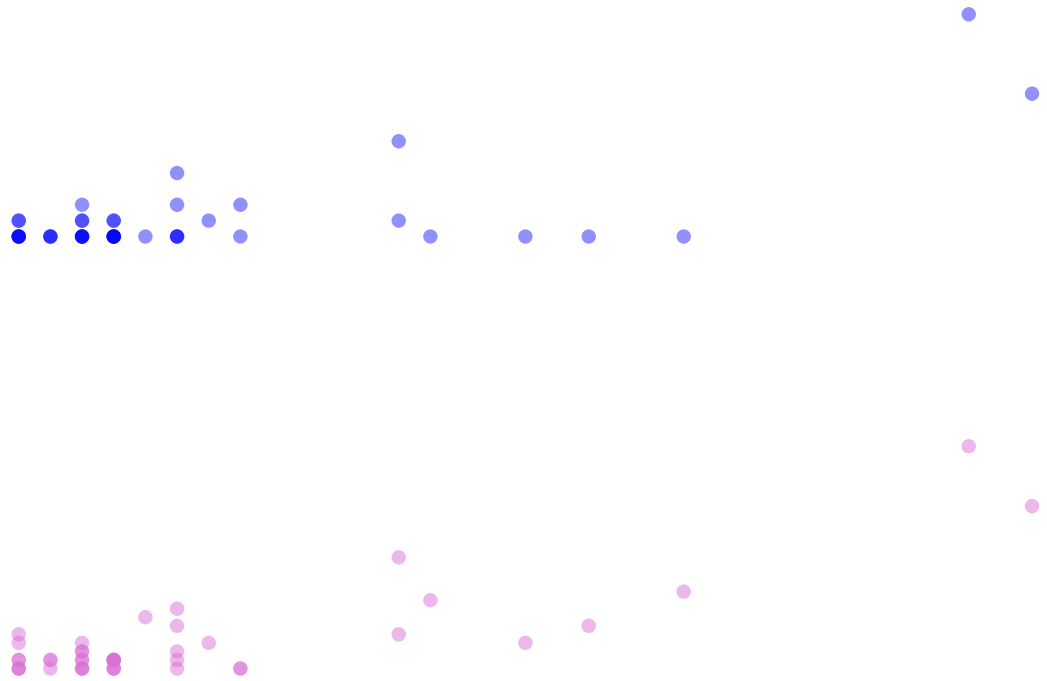
This looks awful... The plots themselves are miniscule, you can't even read the y-axis, and the labels are huge. Let's get rid of the axis labels and main titles to clear up some space. To do this, set `xlab` and `ylab` equal to `""`. If you don't specify any `xlab` or `ylab` argument, R will default to the name of the vector being plotted (e.g. "M" and "Bu"), which we don't want.

```r
par(mfrow = c(2, 1), mar = c(5, 5, 2, 2))

plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "", ylab = "")

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "", ylab = "")
```



Okay, now that we have some negative space to work with let's make some additional changes. Since the x-axis in both plots is identical, let's get rid of it for the top plot and only have it on the bottom plot. To do this, let's turn off the axes for each plot using the `axes = F` argument in each plot function. Then we can add new, custom ones afterwards in locations that we specify. Let's also adjust our `mar` argument in the `par()` function a bit to make these plots larger.

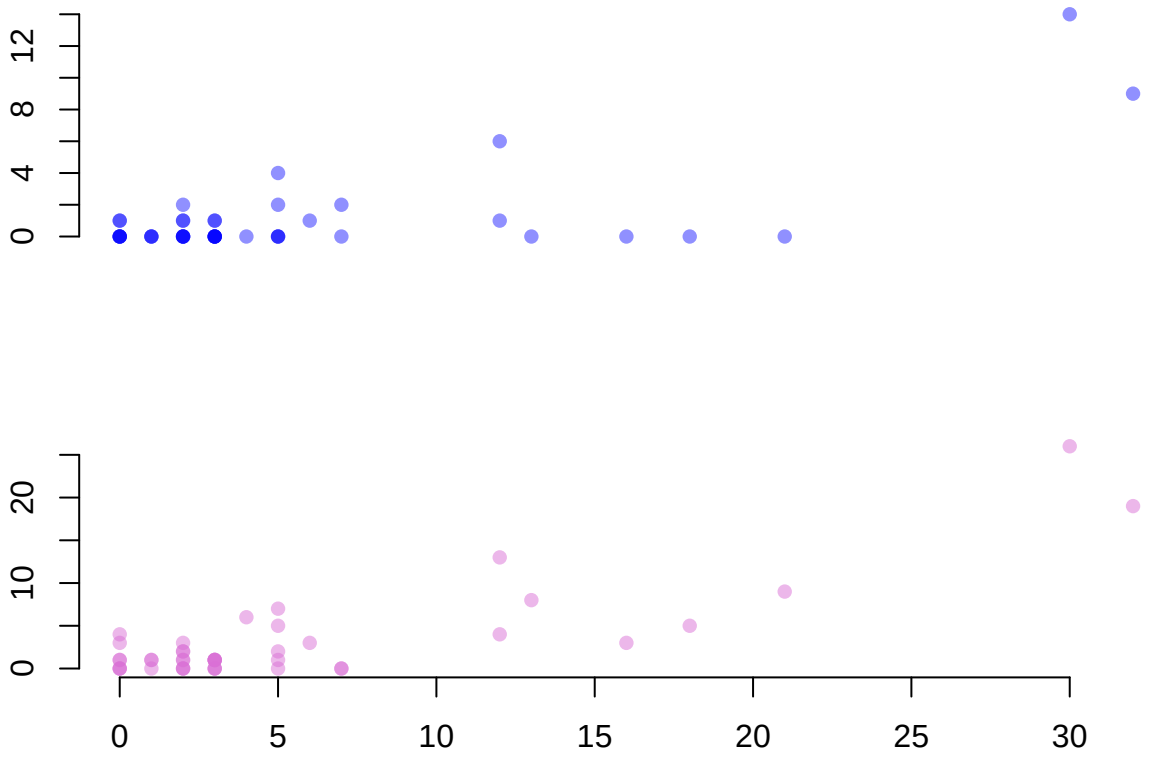
```
par(mfrow = c(2, 1), mar = c(4, 3, 1, 1))

plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "", ylab = "",
     axes = FALSE)

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "", ylab = "",
     axes = FALSE)
```



Okay, now we have only the data points plotted and nothing else. So let's add an x-axis at the bottom and a y-axis at the side. We can do this using the `axis()` function and specifying some arguments inside. The first argument this function takes, as you can see by running `?axis` in your R Console, is `side`. Just as in the `mar` argument, R orders the sides clockwise starting from the bottom: 1, 2, 3, 4 is equivalent to bottom, left, top, right. We want to plot our first axis, the x-axis, on the bottom, so we specify `side = 1`. For our next axis, the y, we specify `side = 2`, and since R knows that `side` is the first argument in this function, we don't even need to specify `side =` and can simply write `2` as long as the `2` is the first argument we include in the function.

We must note, however, that `axis()` functions, like many other extra plotting functions, are added on to the `plot()` you have most recently created. Since we plot the blue points first and then the purple ones, by adding `axis()` afterwards, we would only be modifying the most recent `plot()`. Therefore, we will need to add `axis()` functions after each plot.

```
par(mfrow = c(2, 1), mar = c(4, 3, 1, 1))

plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "", ylab = "",
     axes = FALSE)

axis(2)

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "", ylab = "",
     axes = FALSE)

axis(2)

axis(1)
```



Okay, things are starting to look better, but we're not quite there yet... In my opinion, that x-axis is a bit too close to the purple points. We can move it down a bit by using the `line` argument to specify that instead of plotting the axis at `line = 0`, which is the default, we want to plot at `line = 0.5`.

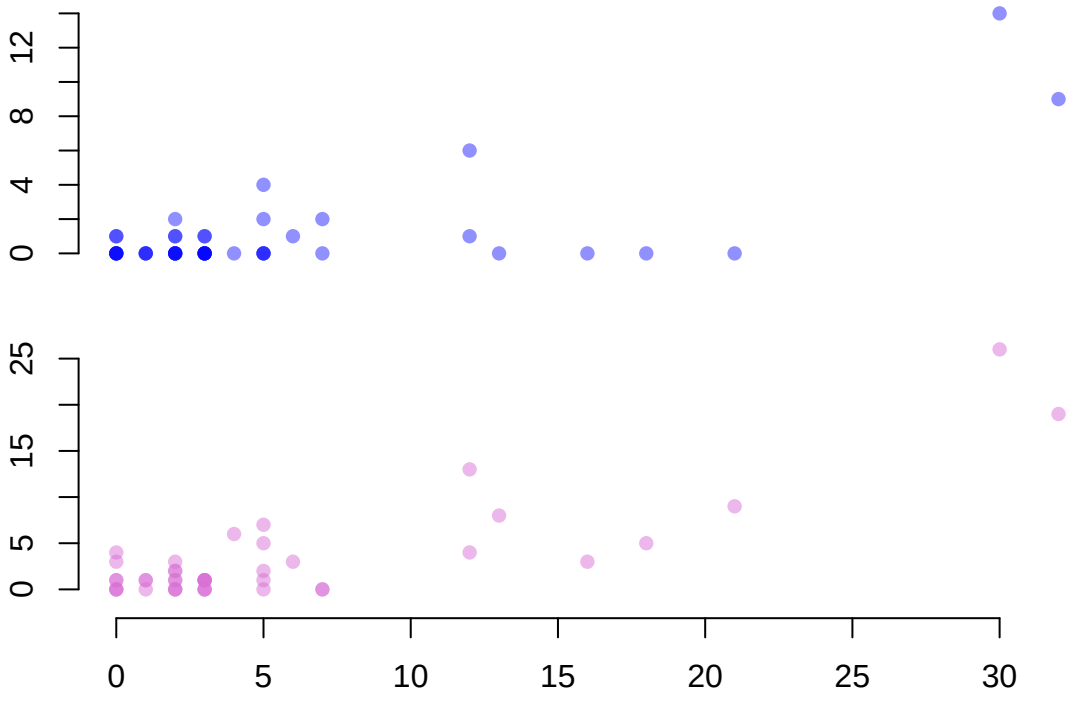
```
axis(1, line = 0.5)
```

This second axis is more appropriately spaced to my eye. But see how adding a new `axis()` function to our pre-existing plot just superimposed a new x-axis onto our old one? We would need to run the `plot()` functions all again to get a new figure that looks good.

But before doing that, I also want to cut down on all that white space in between the top and bottom plots. Right now, we're specifying `mar = c(4, 3, 1, 1)`. This means we're creating 4 units of space at the bottom of *each* plot. In reality, we don't want this much space below the first plot, only the second one. So let's make use of a new `par()` argument for this: `oma`. This stands for *outer margins*.

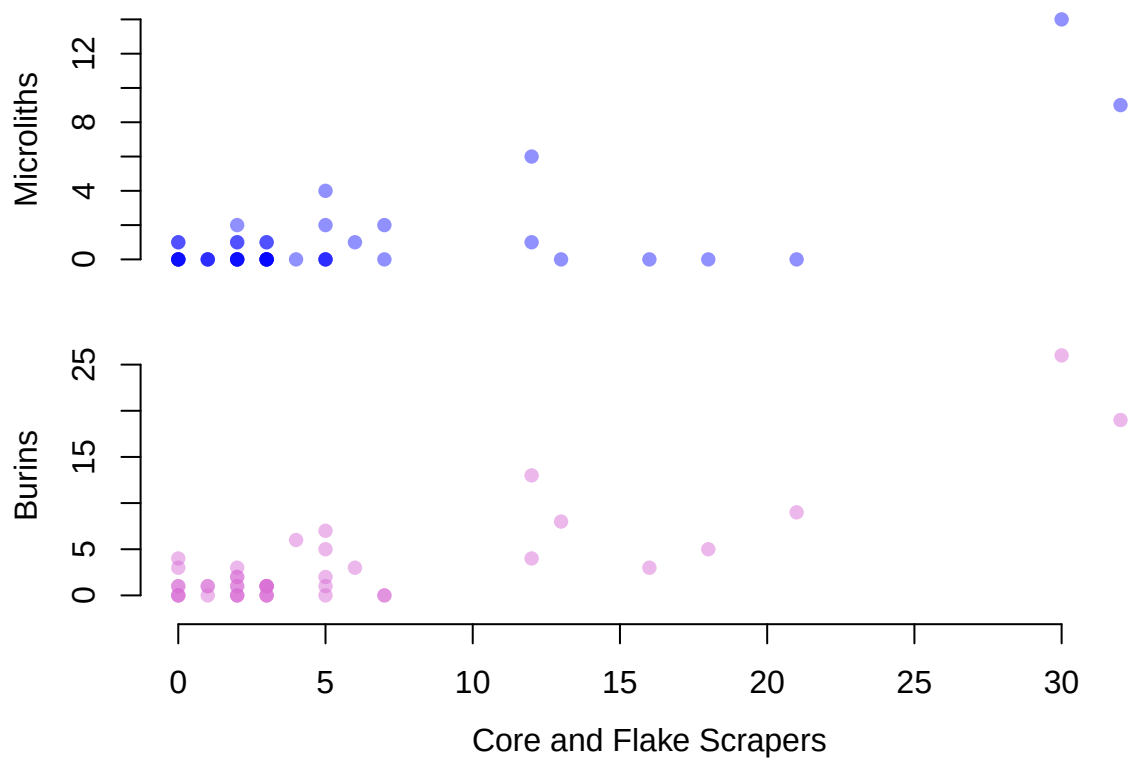
```
par(mfrow = c(2, 1), mar = c(1, 1, 1, 1), oma = c(4, 3, 1, 1))

plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "", ylab = "",
     axes = FALSE)

axis(2)

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "", ylab = "",
     axes = FALSE)

axis(2)
```

```
axis(1, line = 0.5)
```



Now things are looking even better! We changed the `mar` argument to equal `c(1, 1, 1, 1)` so that there is only 1 unit of space around each of the plots. But by setting `oma = c(4, 3, 1, 1)`, we added 4 units of space below the *overall* figure and 3 units of space the left of the *overall* figure, not just next to each individual plot. This now leaves us space to label our axes.

```
par(mfrow = c(2, 1), mar = c(1, 1, 1, 1), oma = c(4, 3, 1, 1))

plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "", ylab = "",
     axes = FALSE)

axis(2)
mtext("Microliths", side = 2, line = 2.5)

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "", ylab = "",
     axes = FALSE)

axis(2)
mtext("Burins", side = 2, line = 2.5)

axis(1, line = 0.5)
mtext("Core and Flake Scrapers", side = 1, line = 3)
```

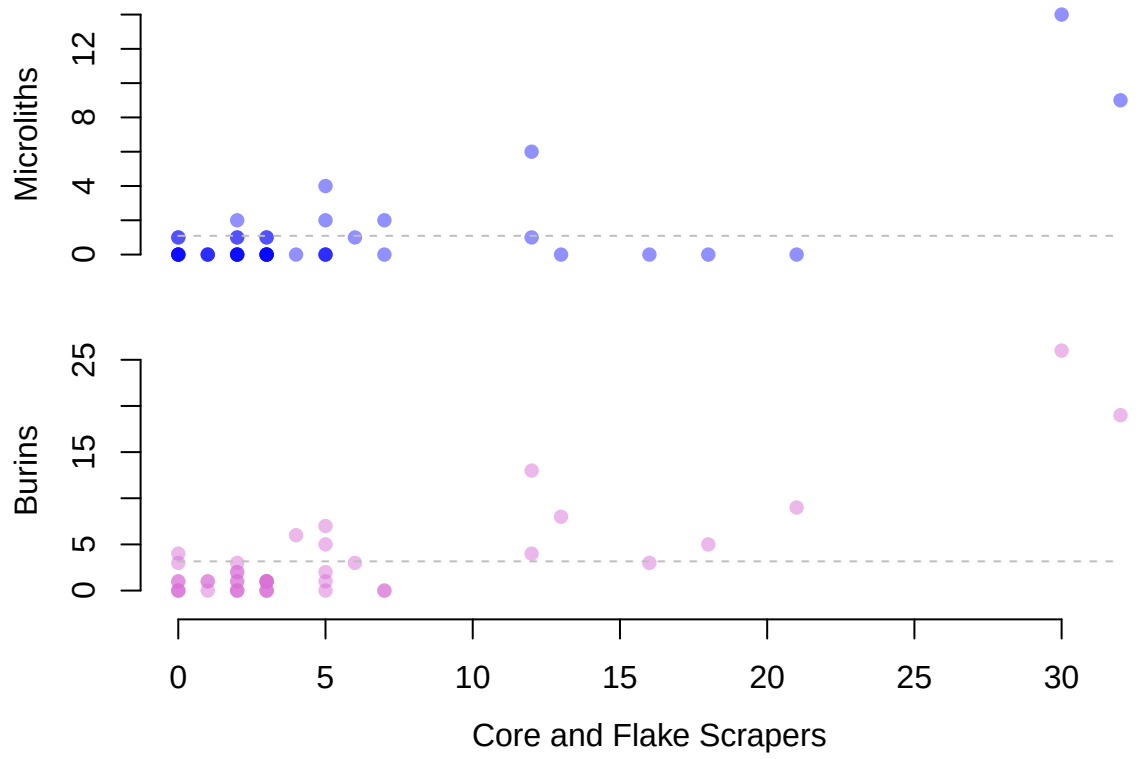


Now we're getting there! It may take a bit of trial and error to figure out what `line` you want to place your `mtext` on, but experiment as you need to.

As a final addition, let's add lines on each plot that show the mean counts of microliths and burins. We can do this using the `segments()` function and specifying the x0, y0, x1, anc y1 coordinates of our line segment. This means specifying where you want the line to start on the x-axis, then on the y-axis, and then specifying where you want the line segment to end on the x-axis then y-axis. For our plots, let's start the line at 0 and extend it to 32 on the x-axis, and have it start and end at the average count for each artifact type. We can apply the `mean()` function to `ESASites$M` and `ESASites$Bu` to do this. Let's also change this line segment's type, specified with the `lty` argument in the `segments()` function, to make it a dashed line (`lty = 2`).

```
par(mfrow = c(2, 1), mar = c(1, 1, 1, 1), oma = c(4, 3, 1, 1))

plot(M ~ CFS, data = ESASites, pch = 16, col = "#0000FF70", xlab = "", ylab = "",
     axes = FALSE)
segments(0, mean(ESASites$M), 32, mean(ESASites$M), lty = 2, col = "gray")

axis(2)
mtext("Microliths", side = 2, line = 2.5)

plot(Bu ~ CFS, data = ESASites, pch = 16, col = "#da70d680", xlab = "", ylab = "",
     axes = FALSE)
segments(0, mean(ESASites$Bu), 32, mean(ESASites$Bu), lty = 2, col = "gray")

axis(2)
mtext("Burins", side = 2, line = 2.5)
```

```
axis(1, line = 0.5)
mtext("Core and Flake Scrapers", side = 1, line = 3)
```



Note that you can also add more arguments to `mtext`, `axis`, and other functions to change additional aspects of your plot. It is possible to change the colors of your axes and labels, to adjust the range of values over which these axes are drawing, or to add more tick marks or fewer to an axis. All of these can be investigated by using the `?` function followed by the specific function you're curious about.

And remember that you can always add additional things to your plot, such as a legend or shapes (lines, circles, squares, etc.) to draw attention to certain things.

3 Bar plots

Now that we have a handle on scatterplots, let's turn to other types of plotting. Bar plots, or bar graphs, are a common technique in archaeology and forensics for visualizing data. In a bar plot, each bar represents some numerical value corresponding to some discrete category. As an example, let's use a different dataset from Texas on a skeletal popluations of individuals from a Late Archaic period cemetery.

First, load the dataset.

```
data("EWBurials")
```

Now, let's say we were interested in how many individuals of each age class were buried at this site. We could create a bar graph of the count of individuals for each of the age categories in this dataset. So first, let's explore these age categories.

3.1 Categorical Data, i.e. Factors

First, we can simply ask R to show us this `Age` vector within the `EWBurials` data frame:

```
EWBurials$Age #the age class values for each of the 49 individuals
```

This returns a list of 49 age class observations, one for each of the individuals in this burial population.

At the bottom of this output, we see a line that says `Levels: Child Adolescent Young Adult Adult Middle Adult Old Adult`. This is in reference to the fact that this is a character vector that contains a type of data known in R as **factors**. A factor is the R term for a category. If we use the `str()` function to investigate the `EWBurials` data frame we can see this:

```
str(EWBurials$Age) #the structure of the Age vector of EWBurials
```

```
##  Factor w/ 6 levels "Child","Adolescent",..: 3 3 6 3 6 3 6 3 3 5 ...
```

When we run this function, we see that the `EWBurials$Age` vector is a factor object with 6 levels. This means that R understands this vector to contain a series of categories: 6 of them, to be precise.

To see what exactly these six levels are, we can use the `levels()` function:

```
levels(EWBurials$Age) #the factor levels for this vector
```

```
## [1] "Child"        "Adolescent"   "Young Adult"  "Adult"        "Middle Adult"
## [6] "Old Adult"
```

Now we know that these six levels correspond to six different age classes found in this burial population.

3.2 Plotting Factors with Bar Plots

Now, since we're interested in plotting the number of skeletons belonging to each age class, we need to manipulate our data a bit. If we tried to simply create a bar plot of this `Age` vector right now, R would get confused and spit out an error message because our vector isn't the right form for the `barplot()` function.

```
barplot(EWBurials$Age)
```

```
## Error in barplot.default(EWBurials$Age): 'height' must be a vector or a matrix
```

What we really need is a data frame with a column for the age category and a column for the count of individuals within that age category. This data frame would therefore have 6 rows and 2 columns. There's a useful function we can use to manipulate our `EWBurials$Age` vector and get it into this format: the `table()` function.

```
table(EWBurials$Age)
```

```
##
##        Child   Adolescent  Young Adult        Adult Middle Adult    Old Adult
##            2            3           19            3           10           12
```

This `table()` function returns a list of the counts of individuals for each of the six age categories, exactly as we wanted. If we check out the structure of the output of this function we see something a bit more complex than what we're used to:

```
str(table(EWBurials$Age))
```

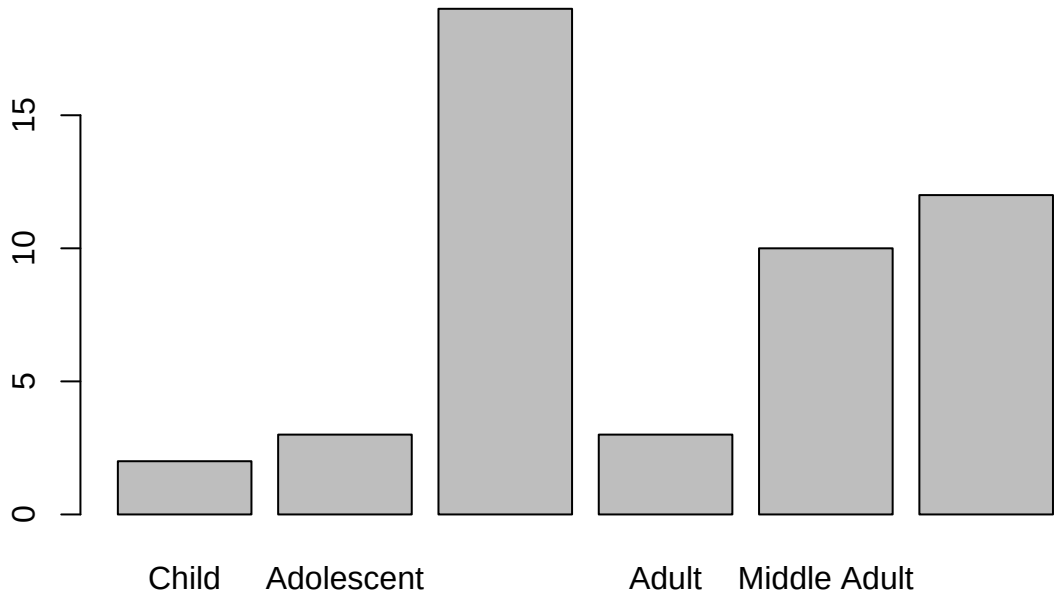
```
##  'table' int [1:6(1d)] 2 3 19 3 10 12
##  - attr(*, "dimnames")=List of 1
##   ..$ : chr [1:6] "Child" "Adolescent" "Young Adult" "Adult" ...
```

This `table()` function is outputting an object which is a list of integer values, hence the `int [1:6]` in the first row of this output. These integers are provided, and they match the output of `table(EWBurials$Age)` from above: 2, 3, 19, 3, 10, 12.

The next line of this `str()` function output tells us that this series of integers have associated attributes, hence the `attr`. These attributes are in the form of a character vector of length six, hence `$ : chr [1:6]` followed by a list of the age classes. This is why the output from `table()` was not just the list of six numbers, but they were labeled with their corresponding age classes. The table function outputs a labeled set of values, which is very useful for our purposes here.

As such, we can assign the output of this `table()` function to a new object, and then plot that object using the `barplot()` function.

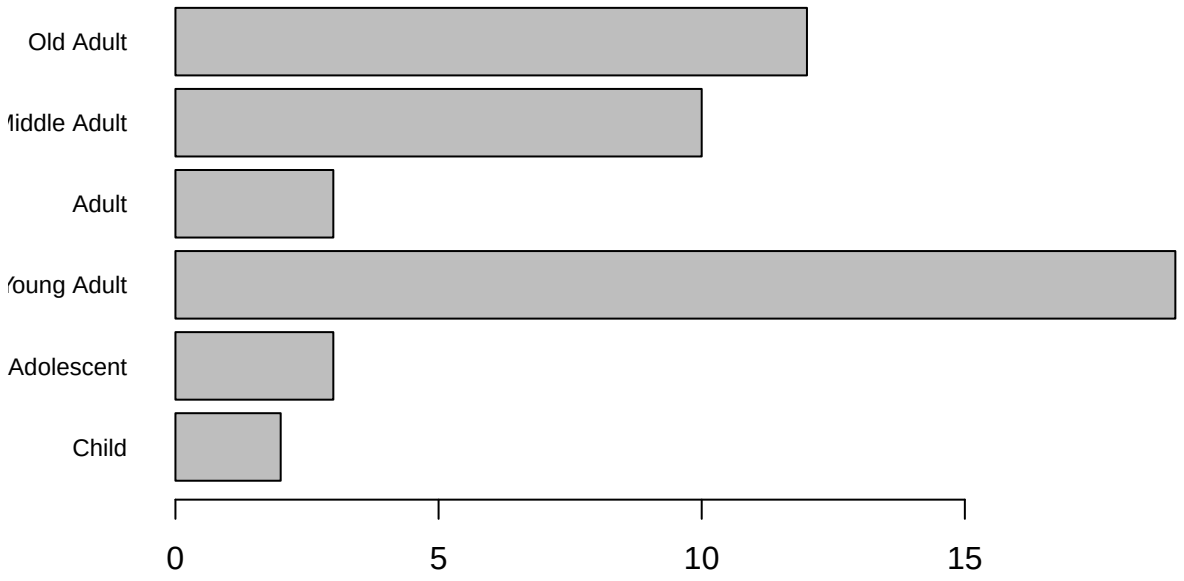
```
burial.data <- table(EWBurials$Age) #assign the output from table to an object

barplot(burial.data) #plot the data from this new object
```



3.3 More About Bar Plots

Now that we have a sense for manipulating factors (i.e. categorical data) in R, we can work to spruce up our bar plot a bit. Depending on the width of your plotting window in the bottom right of RStudio, you might not be able to see all of the age category labels on the x-axis. We can work on this to improve the plot.

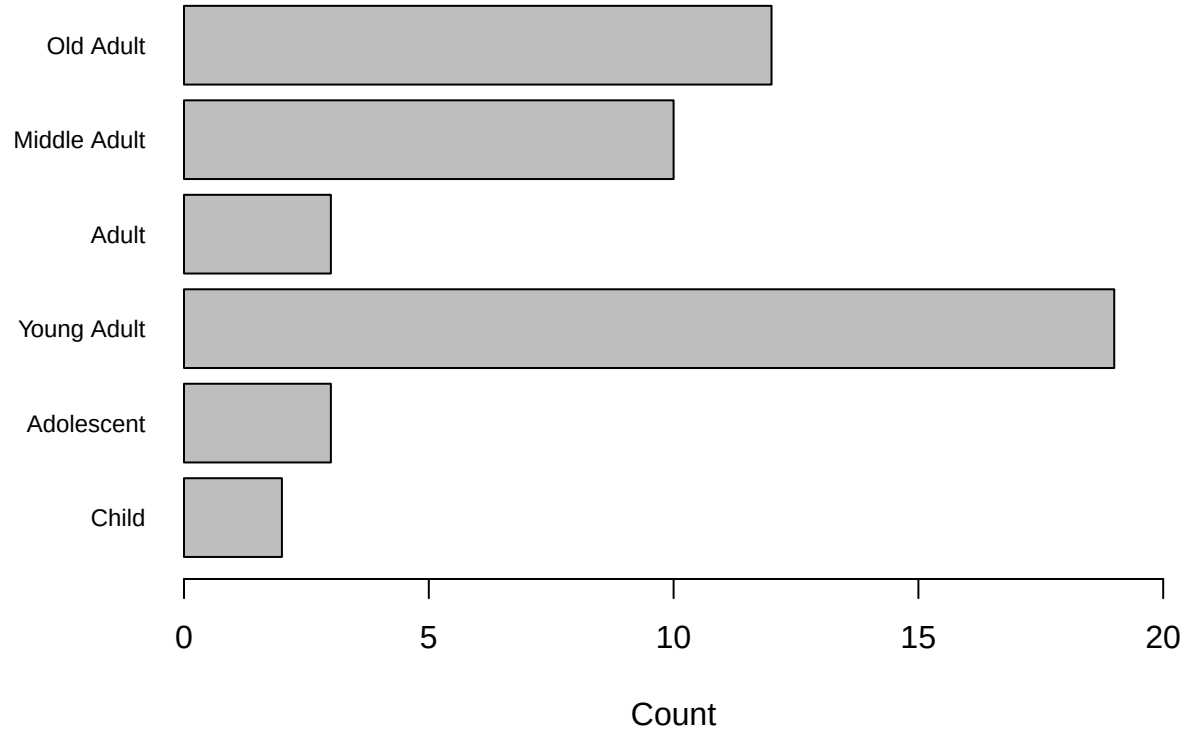
```
barplot(burial.data, horiz = TRUE, cex.names = 0.75, las = 1)
```



Here, I've made use of various arguments within the `barplot()` function to alter our plot. I've specified `horiz = T` to flip the plot so that the bars extend to the right instead of up. I've made the age category labels smaller using `cex.names` and I've plotted these age category labels horizontally instead of vertically by specifying `las = 1`. These arguments are listed in the help documentation for the function, and you can also simply Google search for how to do certain things. For example, I don't believe the `las` argument is listed in the documentation for `barplot()` but if you Google something like "change bar plot axis labels orientation R" you will find out how to use this `las` argument to do so.

Now in this case, our y-axis labels are extending past the plot margins, so we could use the `par()` function and the `mar` argument to fix this as follows.

```
par(mar = c(5, 5, 2, 2))
barplot(burial.data, horiz = TRUE, cex.names = 0.75, las = 1, xlab = "Count", xlim = c(0, 20))
```



I also added an x-axis label using `xlab` and changed the limits of the x-axis using `xlim`. Note that when altering the x- or y-axis limits, you need to specify these limits within a `c()` function as it is a concatenated pair of values.

If you want to create stacked or grouped bar plots, there are numerous online tutorials and resources on how to do so, just as there are for any question you might have about R.

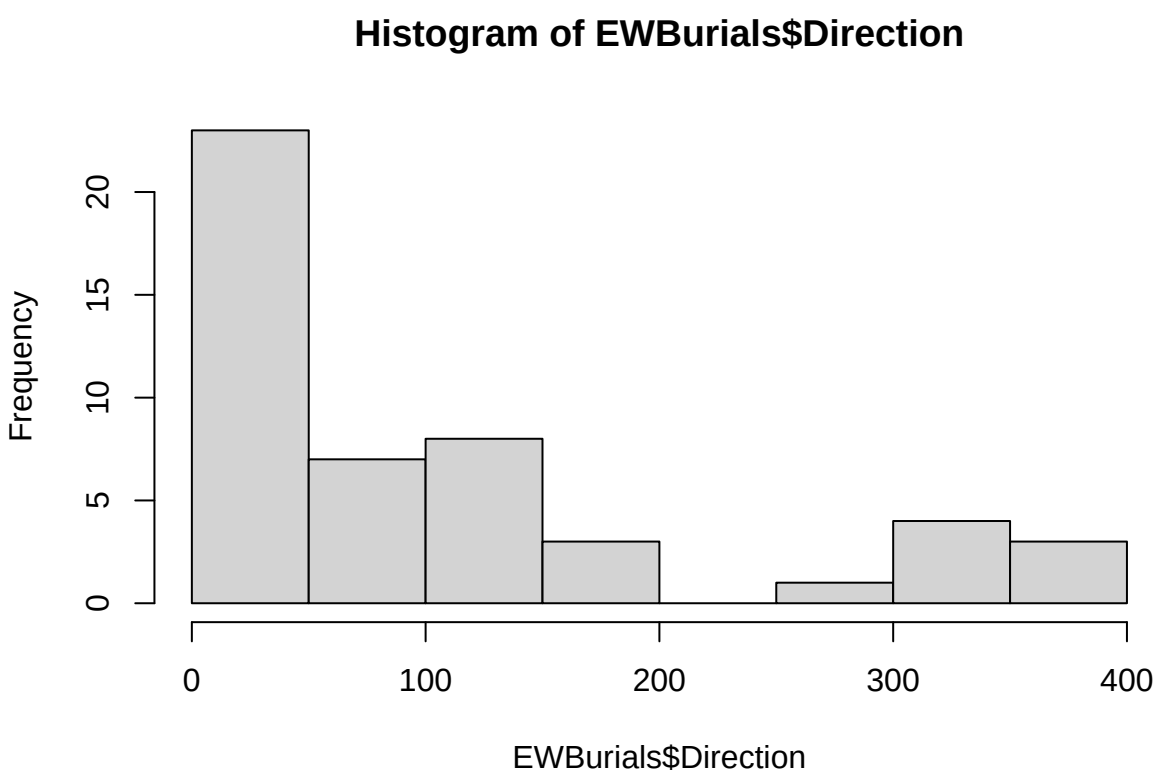
4 Histograms

Histograms are superficially similar to bar plots, however they serve a different purpose. You will still end up with a plot of bars that show some value, but histograms are meant to divide up some continuous variable into what are called *bins* and plot bars for each bin. Therefore, while bar plots are used to show some frequency values for categories of data on the x-axis, histograms show some frequency values for continuous data on the x-axis.

Let's use the `Direction` vector from the `EWBurials` data frame for this. This vector contains information on the orientation of the skeletons in this cemetery measured in degrees east of north. In this format, North is 0 degrees, East is 90 degrees, South is 180 degrees, and West is 270 degrees east of north.

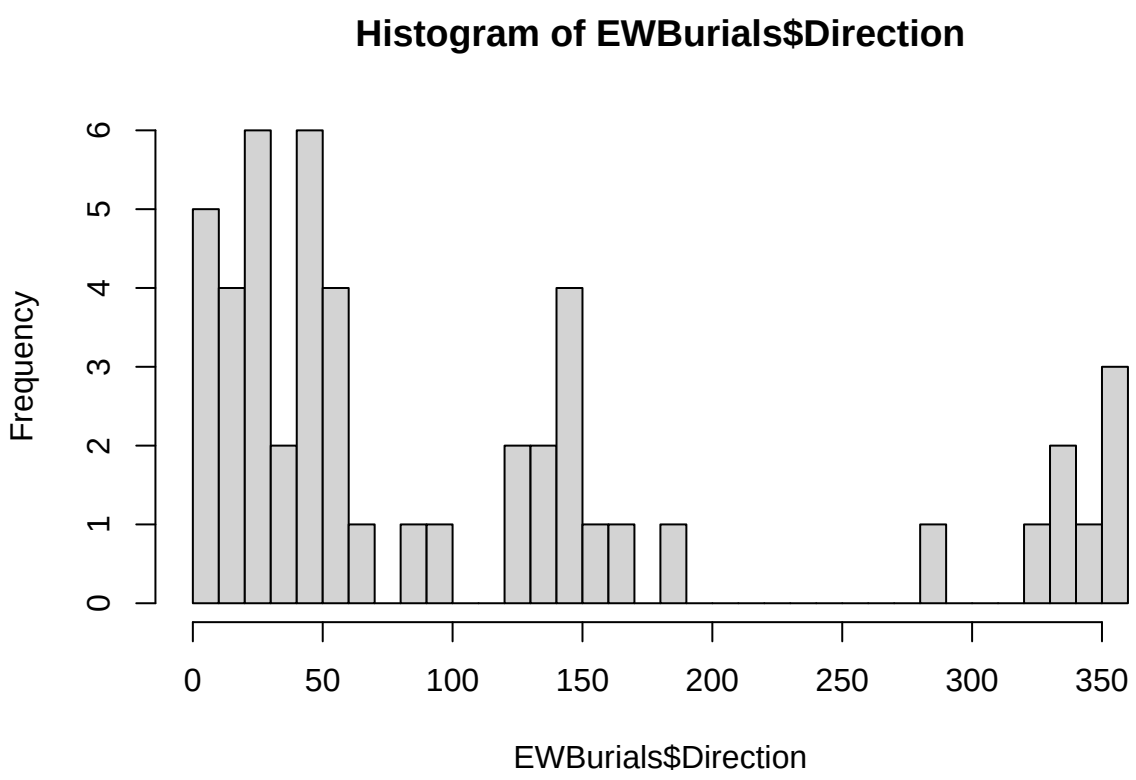
Let's use the `hist()` function to plot a histogram showing the breakdown of these various orientations and their frequencies in this burial population.

```
hist(EWBurials$Direction)
```



This histogram reveals that the most common orientation for skeletons in this cemetery is between North and East. But the bins on this histogram are rather wide... With these bin widths, all orientations between 0 and 50 degrees are lumped together. Furthermore, because the bin width is 50 here, the x-axis goes up to 400 degrees east of north, which is impossible. So let's fix this using the `breaks` argument.

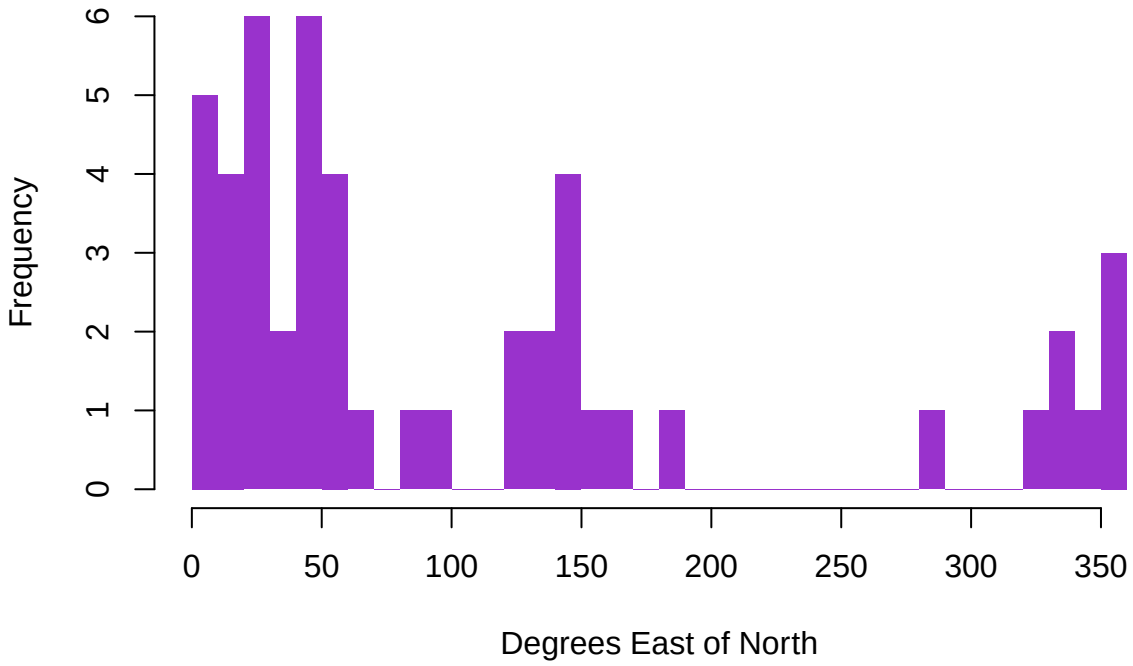
```
hist(EWBurials$Direction, breaks = seq(0, 360, 10))
```



This looks better. The `breaks` argument specifies bin widths for the histogram. It's a bit of a tricky argument to use, but the most effective way to change bin widths is to simply specify exactly what you want. This is what I did by setting `breaks` equal to `seq(0, 360, 10)`. This `seq()` function tells the `histogram()` function to bin the data according to a sequence of values ranging from 0 to 360, but to count by 10s. Therefore, I'm telling it that I want a bins to start at values of 0, 10, 20, 30... and 360.

We can also make this histogram a bit prettier by manipulating the axis labels, main title, bin colors, and whether we want bins to have borders or not:

```
hist(EWBurials$Direction, breaks = seq(0, 360, 10), xlab = "Degrees East of North",
     main = "", col = "darkorchid", border = F)
```



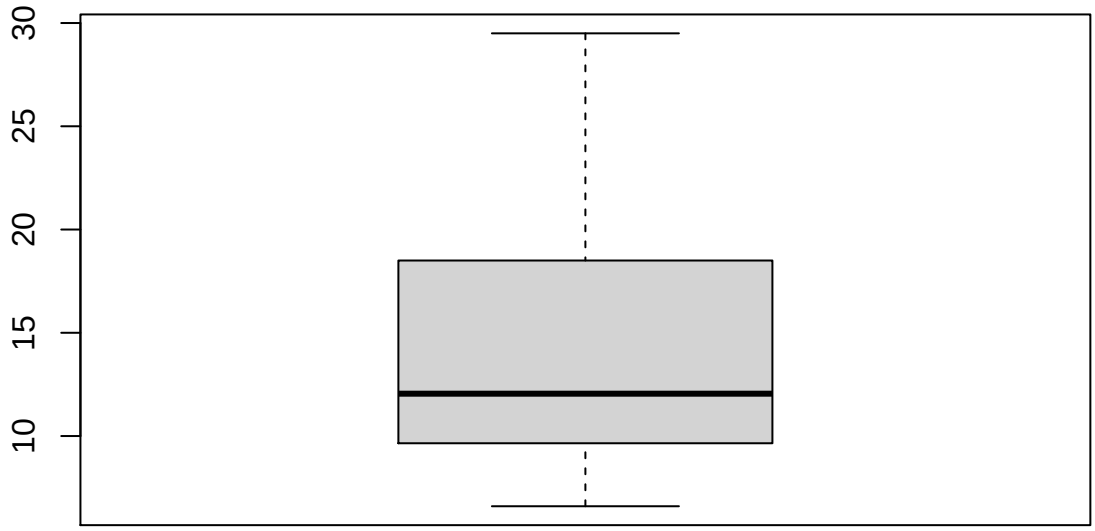
Now we can quite clearly see that most individuals in this cemetery are oriented between North and North-East, though there is a good bit of variation.

5 Boxplots

Another very common type of plot for archaeologists, forensic anthropologists, and other scientists to use is a boxplot. Boxplots show the viewer how the data are distributed according to *quartiles*, which represent the data dividied into four equally sized groups.

Let's use the `BACups` dataset for our boxplots: a series of measurements on sixty Early and Late Bronze Age ceramic cups from Italy.

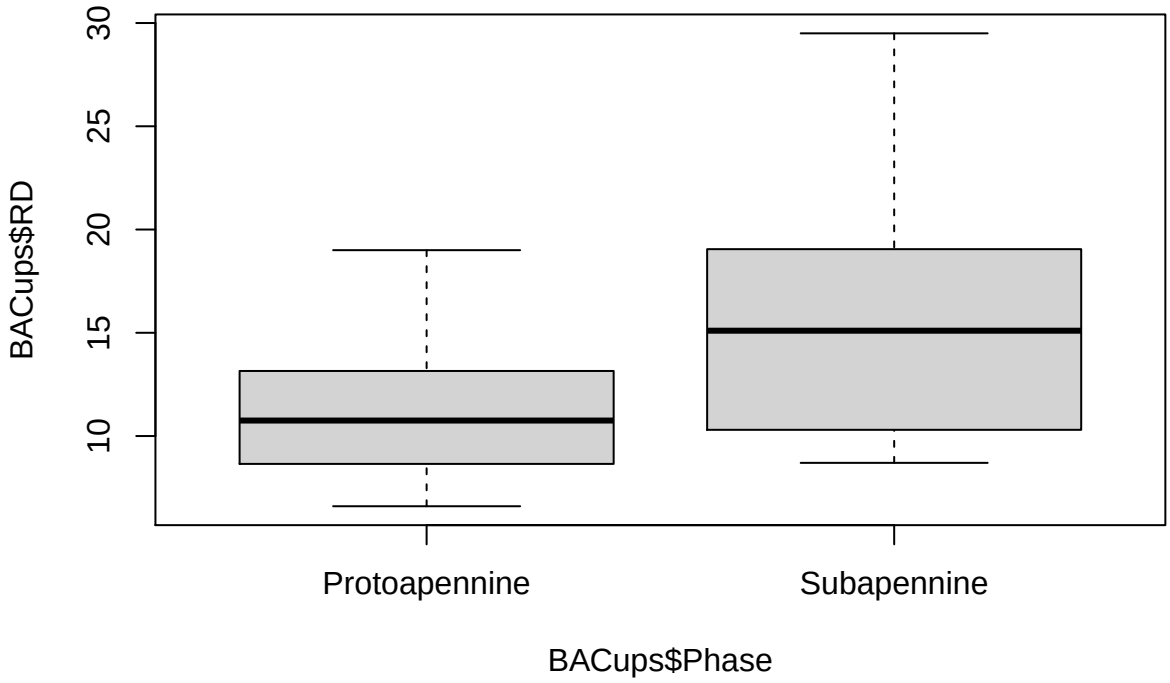
```
data("BACups") #load the dataset

boxplot(BACups$RD) #plot the rim diameter of the cups
```



Here, we have created a boxplot of the rim diameter of the cups in this dataset. We can see that the mean diameter is somewhere around 12 cm, shown by the dark black line on the plot. The first quartile extends from around 6 cm to 10 cm, the second quartile from ~10 cm to ~12 cm, the third from ~12 cm to ~19 cm, and the fourth from ~19 cm to ~30 cm. The first and fourth quartiles are shown as "whiskers" on either side of the "box" which comprises the middle two (second and third) quartiles.

Now, let's compare rim measurements between the two different time periods included in this dataset: the Early and Late Bronze Age. We can do this using the syntax $y \sim x$ in the `boxplot()` function.

```
boxplot(BACups$RD ~ BACups$Phase)
```



Now we have a plot of rim diameters for Protoapennine (Early Bronze Age) cups compared to Subapennine (Late Bronze Age) cups. It looks like cup diameter increases substantially in the Late Bronze Age. Let's check the summary statistics of these two groups to see:

```
summary(BACups[BACups$Phase == "Protoapennine", ]$RD)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   6.600   8.725  10.750  11.445  13.075  19.000
```

```
summary(BACups[BACups$Phase == "Subapennine", ]$RD)
```

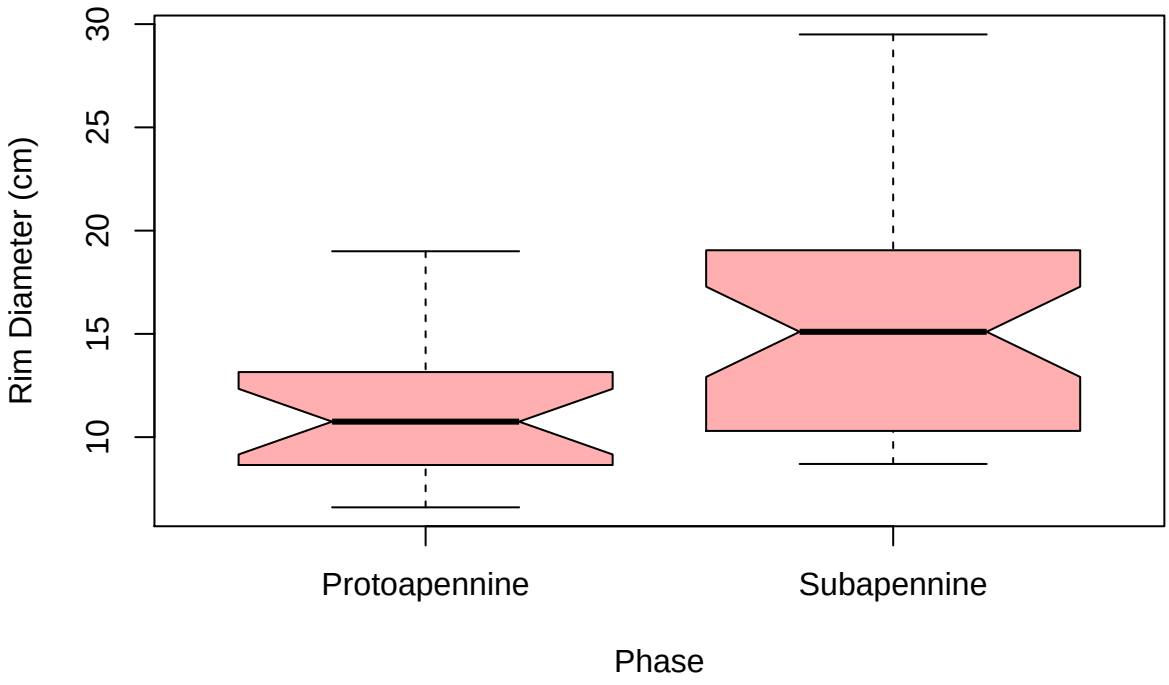
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    8.70   10.40   15.10   15.31   19.02   29.50
```

The above code might look at bit complex, but I'm simply indexing the `BACups` data frame to pull out the rim diameter vectors for each of the two time periods. I do this by telling R to look only at the rows of `BACups` for which `BACups$Phase` is equal to `"Protoapennine"`, and all columns. Remember that indexing uses the syntax [rows, columns] and that leaving a blank space after the comma tells R to return all columns. Therefore, `BACups[BACups$Phase == "Protoapennine", ]` gives me a new data frame of only the Protoapennine data. I then want to tell R to only look at the `RD` column since that's what we plotted above. I do this by adding `$RD` to the end of the indexing brackets. I finally wrap all of this in the `summary()` function, which tells me the range of values as well as the median and mean. Next, I do the same for the Subapennine period.

And with this, we can see that both the mean and median rim diameters of Late Bronze Age cups are larger than those of Early Bronze Age cups.

However, we don't really know if this difference is large enough to be considered *statistically significant*. We'll discuss significance more later in the context of regression models, but as it is a concept that many people care about and are aware of, there is a trick with boxplots to approximate a signficance test. If you add the argument `notch = TRUE` to your code, the resulting boxplots will have notches in them that roughly reflect 95% confidence intervals.

```
boxplot(BACups$RD ~ BACups$Phase, notch = T, ylab = "Rim Diameter (cm)",
        xlab = "Phase", col = "#FF000050")
```



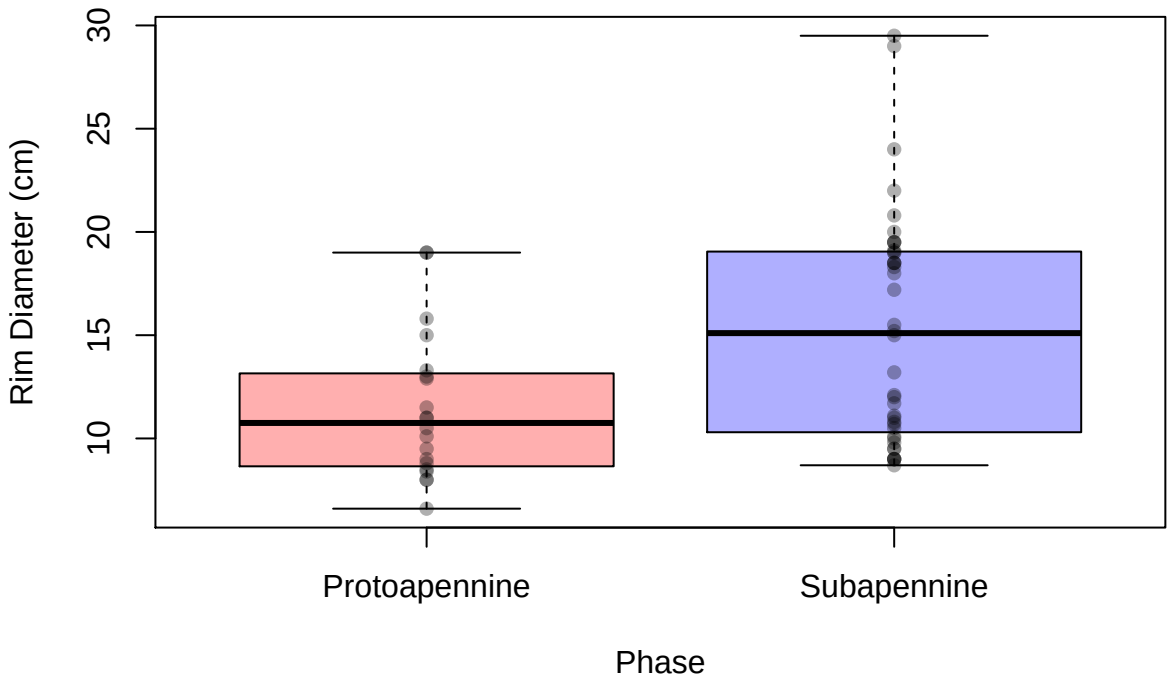
You can see that the notches for these two boxplots do not overlap. This suggests that the rim diameter of Subapennine cups is significantly larger than the rim diameter of Protoapennine cups. I would still recommend running an actual significance test to compute a p-value, but for rough approximations of significance, this trick can be useful.

5.1 A Note on Boxplots

Boxplots are a very common visualization tool, but they are often criticized for hiding data. In the boxplots we just created, you can't actually see the data points that are being used to determine the shapes of those boxplots. The quartiles are based on the data, so you can get a rough sense for the distribution of the data points, but you also have no idea how many observations these boxplots are based on from just looking at the plot. Are there ten Protoapennine cups or 10,000? This plot does not say.

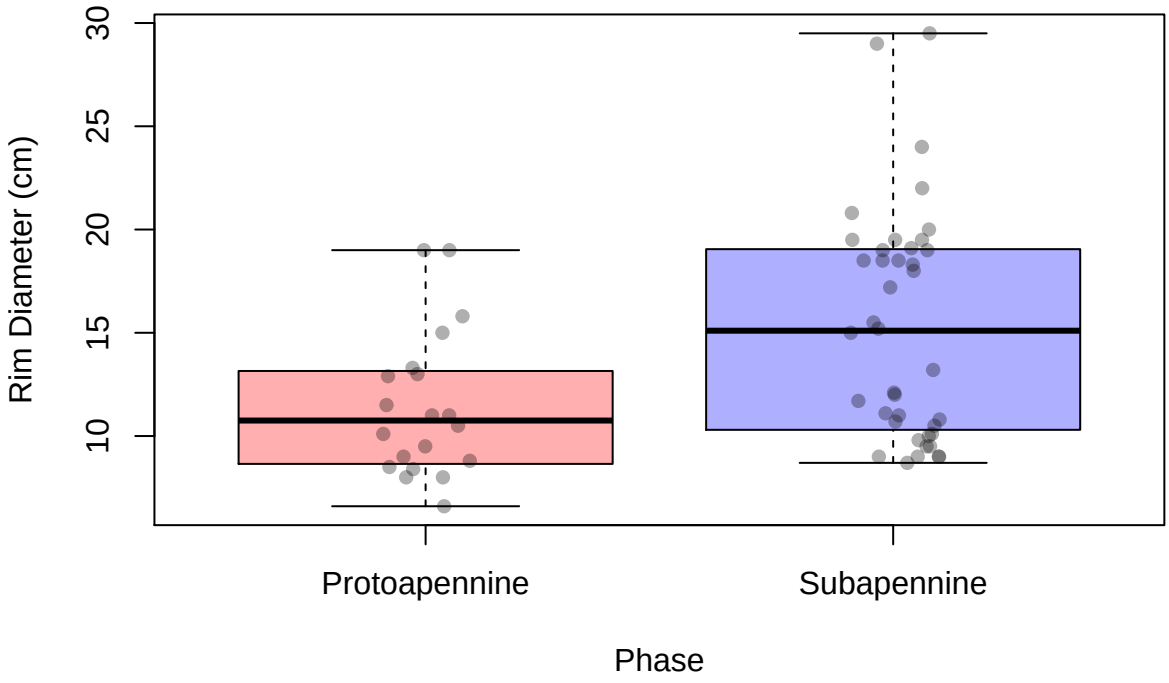
For this reason, many recommend plotting the actual data points on top of, or in place of, boxplots. You can do the former by using the `points()` function to plot your actual data on rim diameter by phase on top of the boxplots:

```
boxplot(BACups$RD ~ BACups$Phase, ylab = "Rim Diameter (cm)",
        xlab = "Phase", col = c("#FF000050", "#0000FF50"))

points(BACups$RD ~ BACups$Phase, pch = 16, col = "#00000050")
```



Because these points are coming from just two categories, there are many being plotted on top of one another. Making the points transparent, as above, helps with this, but we can also do something called *jittering*. Jitttering is a process by which you randomly add and subtract small values from your real values. This has the effect of slightly spreading your data apart in a random way.

```
boxplot(BACups$RD ~ BACups$Phase, ylab = "Rim Diameter (cm)",
        xlab = "Phase", col = c("#FF000050", "#0000FF50"))

points(BACups$RD ~ jitter(as.numeric(BACups$Phase), 0.5), pch = 16, col = "#00000050")
```



To jitter our data a bit, we need to first manipulate `BACups$Phase` and treat it as a numeric vector. This is because jittering adds and subtracts randomly from your data, and you cannot add or subtract a random number from a factor like "Subapennine". Therefore, we wrap `BACups$Phase` in the function `as.numeric()` which gives "Protoapennine" a value of 1 and "Subapennine" a value of 2 (it assigns numbers alphabetically). Now we can jitter it. To do so, we use the `jitter()` function and specify a factor of 0.5 (you may need to experiment with the jittering factor through a bit of trial-and-error until you find what looks best).

6 Saving Plots

Now that you've learned how to create several different types of plots in R, you need to know how to save them! For casually sharing preliminary figures with collaborators and colleagues, you can use the **Export** button in the bottom right quadrant of your RStudio screen. This button exists on the **Plots** tab and is to the right of the **Zoom** button. When you click this **Export** button, you can select to save your plot as an image (e.g. .png), PDF, or to copy the plot to your clipboard so that you can paste it elsewhere.

Using the **Export** button is really only suitable for sharing preliminary figures unofficially because the default export resolution is only 72 dpi (dots per inch). This is really rather poor resolution, and no academic journal that I'm aware of would publish such a low-quality image.

Therefore, to actually export a high-quality (300 dpi +) image from R or RStudio, we will use a set of functions: `jpeg()`, `png()`, and `pdf()`.

To export an image as a .jpg file, we must first make sure that we have specified a working directory for our current R session. You can do this by clicking Session > Set Working Directory > Choose Directory... or Session > Set Working Directory > To Source File Location. Alternatively, you can use the `setwd()` function to specify a file pathway.

Once we have our working directory set, we must run the `jpeg()` function **before** we plot our figure. In this function, we can specify the file name, the export resolution, and the size of the image.

We must also use the function `dev.off()` at the end of our plotting code to tell R to stop exporting this image as a .jpg. Otherwise, additional code you run will be treated as part of this exported image.

```
jpeg("BronzeAgeBoxplot.jpg", height = 4, width = 5, units = "in", res = 600)

boxplot(BACups$RD ~ BACups$Phase, ylab = "Rim Diameter (cm)",
        xlab = "Phase", col = c("#FF000050", "#0000FF50"))

points(BACups$RD ~ jitter(as.numeric(BACups$Phase), 0.5), pch = 16, col = "#00000050")

dev.off()
```

```
## pdf
##   2
```

Here, I have told R to export a .jpg file called "BronzeAgeBoxplot.jpg" to my working directory. This .jpg has a height of 4 inches, a width of 5 inches, and a resolution of 600 dpi. If you run this code (don't forget to run `dev.off()` at the end!) and then open the resulting .jpg in your working directory, you will find a high-quality .jpg ready for publication!