# Lab Two: Describing and Visualizing Data

*Alex Davis*

In this lab we will go over the basics of exploring, describing, and visualizing your data. First let's make sure you've set your working directory. For this lab you will need to have the following packages installed:

1. car
2. dplyr
3. reshape2
4. modeest
5. descr
6. psych

Remember, the syntax to install a package is install.packages("packagename"). **Note**: these packages should already be installed from the class initialization script.

Once you have all of these installed, you need to load them into your library. Then you load your data set. We should include all of this at the beginning of the file.

Once you've run your setup, you are ready to start the lab.

## Part One: Factoring

As we went over in the last lab, there are different object types in R. To keep it simple we break it down into two broad groups: 1) factors and 2) numeric/integer.

A factor is nominal data that you want to aply a label to. Think race, gender, party identification, etc.

An integer is data that consists of numbers, where it is ranked (ordinal and interval).

When we read data into R (importing our .csv file), R attempts to classify the data by type. When you have data that is text, R will likely classify it as a factor.

Let's start by looking at an example of a factor variable. In our class dataset, the variable f.party.2 is a factor variable that tells us the party affilation of the individuals who answered our survey. Let's take a look. The table() function allows us to look at the variable by category

```
table(ds$f.party.2)
```

```
##
##  Dem  Ind  Rep
##  869  356 1185
```

Now try to take the mean of the variable:

```
mean(ds$f.party.2)
```

```
## Warning in mean.default(ds$f.party.2): argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

Notice that it doesn't work. This is because you cannot take the mean of a factor variable. That would be like trying to take the mean of words.

Lets use the str() function to look at the structure of the variable:

```r
str(ds$f.party.2)
```

```
##  Factor w/ 3 levels "Dem","Ind","Rep": 3 1 3 NA 3 2 1 3 3 NA ...
```

This should tell us that this variable is a factor with three levels, "Dem", "Ind", and "Rep".

With many data sets, data is initially coded in numbers, and factored afterwards. For example, the basic political party variable in our data set is numeric:

```r
table(ds$party)
```

```
## 
##    1    2    3    4 
##  869 1185  356   91
```

Unless someone had the codebook, they would not know what 1,2,3, or 4 stood for. This is one reason we factor data. Factoring data in R serves two broad purposes:

1. Applies labels to data
2. Tells R to treat the data as categorical/nominal

At a very basic level, you can factor a variable without applying labels to it. At the least, this tells R to treat it as categorical. This methods works when you need to factor a variable quickly. The basic syntax is to use the factor() command. It is important to note that when you are factoring a variable, you are in essence creating a new variable that is a factored version of an older variable. Therefore you also need to use the <- command and pick a name for the new factored variable.

Typically we put "f." before the variable name to remind us it is factored. Let's try it:

```r
ds$f.gender <- factor(ds$gender)
```

Remember that we put the $ sign to tell R what dataset to draw from, or to tell R what dataset to assign the new factored variable to. Let's look at a table of the new variable:

```r
table(ds$f.gender)
```

```
## 
##    0    1 
## 1520 1026
```

while this factored variable is split into two categories, we don't intuitively know which number is male and which is female. We should now assign labels to the variable. When factoring a variable, you need to tell R how many levels are in the variable, and which labels to assign to those levels. In this variable there are two levels, 0 and 1, and we need to assign the labels Women and Men to those variables. Lets factor the variable completely, then create a table to look at it.

```r
ds$f.gender <- factor(ds$gender, levels = c(0,1), labels = c("Women", "Men"))
table(ds$f.gender)
```

```
## 
## Women   Men 
##  1520  1026
```

Notice how, in the syntax, you use the vector command, c(), to tell R the levels and labels of the variable.

Now let's factor the political party variable, where 1 = Dem, 2 = Rep, 3 = Ind, and 4 = Other, then create a table of the variable.

```r
ds$f.party <- factor(ds$party, levels = c(1,2,3,4), labels = c("Dem", "Rep", "Ind", "Other"))
table(ds$f.party)
```

```
##
##   Dem   Rep   Ind Other
##   869  1185   356    91
```

Now take a look at the structure of the variable. It should tell us it is a factor with 4 levels.

```
str(ds$f.party)
```

```
##  Factor w/ 4 levels "Dem","Rep","Ind",..: 2 1 2 NA 2 3 1 2 2 4 ...
```

There are other types of data conversions as well. In most cases, you will follow the basic syntax used to factor, except you could use "numeric" or "integer". For most purposes, numeric and integer are the same.

### Coerce Factoring

Sometimes the typical commands do not work, and you are forced to coerce the variable. When you coerce the variable, you are telling R to treat the variable as if it were a different object type. The coercive arguments are:

1. as.factor()
2. as.numeric()
3. as.integer()

Let's try to convert a factor variable into a numeric variable using the numeric() argument.

This doesn't work, so we should try using the coerce method:

```
ds$n.party <- as.numeric(ds$f.party)
```

Now let's examine the new variable:

```
table(ds$n.party)
```

```
##
##    1    2    3    4
##  869 1185  356   91
```

```
str(ds$n.party)
```

```
##  num [1:2547] 2 1 2 NA 2 3 1 2 2 4 ...
```

Good work! Now let's move on to recoding.

## Part Two: Recoding

In R, we use the recode() function to reassign values to a variable. There are several broad purposes to recode a variable, including

1. To correct or change incorrect data
2. To restructure the data, making it easier for calculations
3. To emphasize some intended effect.

One example situation in which you would want to recode would be if you wanted to look at age groups instead of exact ages. If everyone in your survey reported their exact age (e.g. 54, 23, etc.) but you wanted to break down your data by age groups, like 18-25, 26-35, etc. the recode function would be the one to use.

To perform a recode, follow this basic syntax:

recoded.variable <- recode(old.variable, "recode commands")

Let's make it more clear by perfoming a recode. Similar to how we put "f." before a factored variable to remind us that it is factored, let's put "r." in front of our new recoded variable to remind us it is a recoded version of an already existing variable.

Let's recode the ideology variable in our class data set. Currently the ideology variable goes from 1 to 7, with 1 being very liberal and 7 being very conservative. Let's simplify the variable by recoding it into three levels, instead of the current 7. Broadly we will have a liberal level, a moderate level, and a conservative level. Let's do a recode that sorts everyone who answered 1 or 2 into 1 level, everyone from 3 to 5 into another, and 6 to 7 into a third. We primarily use the recode function from the car package.

```
ds$r.ideol <- car::recode(ds$ideol, "1:2=1; 3:5=2; 6:7=3; else=NA; NA=NA")
```

Let's use the table() function to look at the new recoded variable:

```
table(ds$r.ideol, useNA = "always")
```

```
##
##    1    2    3 <NA>
##  401 1084 1039   23
```

Notice how we include "else=NA; NA=NA" at the end of the recode function. This tells R to regard any other responses, whether missing data or data that for some reason is outside the original range, as NA, and to treat all existing NAs as NAs. Sometimes you also need to include "-99=-99" or "-99=NA" in the function as well.

A couple other notes about the syntax: in the recode function, you put all the recode arguments **inside one set of quotation marks**. Because of this, you need to separate each argument with a semicolon. If you try to use a comma, R will give you an error message.

You can also save quite a bit of time by using a colon to tell R a range of values. In our recode function we put "1:2=1" meaning all values from 1 to 2 will be assigned a one. This is a preferred method over typing "1=1;2=1;etc."

There is not a set in stone way you should break down or categorize your data in a recode. Let your research question, your model design, and ultimately your data itself determine how you should do recodes. For example, if you asked a question on a survey about how much individuals support something one a scale of 1 to 4, which 1 being very supportive and 4 being not supportive, you might want to consider recoding that variable so that higher values equal more support.

Now let's look at our race variable:

```
table(ds$race)
```

```
##
##    1    2    3    4    5    6    7
## 2227   79  119   10    2   76   23
```

In this survey, 1 indicates white, 2 indicates African-American, and 3 through 7 indiciate a variety of other options (Native American, Asian, Pacific Islander, 2+ races, and Other).

Try recoding this variable to go from 7 levels to 3, where 1 is still white, 2 is African American, and 3 includes all others.

```
ds$r.race <- car::recode(ds$race, "1=1; 2=2; 3:7=3; else=NA; NA=NA")
table(ds$r.race)
```

```
##
##    1    2    3
## 2227   79  230
```

### Factoring and Recoding

Now let's combine our knowledge of factoring and recoding. Let's factor our newly recoded race variable to apply the labels to the three levels that currently exist. Remember, 1 indicates white, 2 indicates African American, and 3 indicates another race.

```
ds$f.race.2 <- factor(ds$r.race, levels = c(1,2,3), labels = c("White", "African-American", "Other"))
table(ds$f.race.2)
```

```
##
##            White African-American            Other
##             2227               79              230
```

Now let's do the same with our recoded ideology variable, where 1 indicates liberal, 2 is moderate, and 3 is conservative.

```
ds$f.ideol <- factor(ds$r.ideol, levels = c(1,2,3), labels = c("Liberal", "Moderate", "Conservative"))
table(ds$f.ideol)
```

```
##
##      Liberal     Moderate Conservative
##          401         1084         1039
```

### Creating a Dummy Variable

One reason that you might want to use your knowledge of both factoring and recoding would be to create a dummy variable. A dummy variable is a binary indicator (0 or 1) of some category, so that we can see if there is an effect from that particular category versus the rest. Dummy variables are used a lot in political science, so it is important that we understand how to create them.

Let's create a dummy variable from our newly recoded race variable for being African American. Recall that in our recoded race variable, 2 indicates African American. For this dummy variable, we want to make African American be 1, with everyone else being 0.

```
ds$r.AfAm <- car::recode(ds$r.race, "2=1; else=0; NA=NA")
table(ds$r.AfAm)
```

```
##
##    0    1
## 2468   79
```

Now let's factor the variable to apply labels to the already existing levels of 0 and 1.

```
ds$f.AfAm <- factor(ds$r.AfAm, levels=c(0,1), labels = c("Non African-American", "African-American"))
table(ds$f.AfAm)
```

```
##
## Non African-American     African-American
##                 2468                   79
```

## Part Three: Building and Sorting Your Data

In R, you have the option of building your data within the program itself. There are a variety of ways to do this, and we will go over the basics here.

The first function you need to know is the rnorm() function. This function allows us to generate normally-distributed random values within a given range, with specified means and standard deviations (more on that later in the class!)

Let's build some random data here:

```
one <- rnorm(100, mean=3)
two <- rnorm(100, mean=7)
three <- rnorm(100, mean=1)
```

We just created three elements (one, two, and three) that contain 100 random values with different means. Now let's combine the three elements into columns, To do this we use the cbind() function. Let's put the three elements into a column with the label "four":

```
four <- cbind(one, two, three)
```

Now let's put the three elements into *rows*. We do this with the rbind() function. Let's create the rows in a new object labelled "five":

```
five <- rbind(one, two, three)
```

### Apply functions

In R there is a family of apply() functions. These functions give us a way of sorting through and examining our data. Let's use the apply() function to look through the data we just created. Here is the basic syntax:

apply(X, margin, fun. . . )

Basically this is telling R to use the apply function, on a particular object, by a particular margin (1 for rows and 2 for colums), and to use a particular function (mean, sum, etc.)

Let's use the apply function on our object "four". The object "four" is a combination of columns, so the margin equals 2. Let's look at the mean of each column in "four":

```
apply(four, 2, mean)
```

```
##      one      two    three
## 2.894955 6.996884 1.001694
```

We just used the apply() function to tell R to report the mean value of each column.

Now let's use the apply() function on a combination of rows. Our object "five" is a combination of rows, so let's use that. Within the apply() function, we specify that we want to look at rows by indicating that the margin is 1. Let's find the mean of all the rows:

```
apply(five, 1, mean)
```

```
##      one      two    three
## 2.894955 6.996884 1.001694
```

Notice that these are the same values as the ones found in the earlier apply() function. This is because we assigned the same elements to "four" and "five".

There are other functions in the apply() family. For a great tutorial on the whole group visit:

https://www.datacamp.com/community/tutorials/r-tutorial-apply-family

However, let's take a look at only two more of the apply functions, sapply() and tapply().

Let's create an object, "six", that contains two lists, "a" and "b", with specified values:

```
six <- list(a=1:10, b=11:20)
```

Now let's use the sapply() function to tell R to give us the mean of each list:

```
sapply(six, mean)
```

```
##    a    b
##  5.5 15.5
```

The sapply() function is useful because it can be used on vectors and matrices.

Now let's look at the tapply() function. The tapply() function allows us explore data by specific parameters. Let's switch over to using our class data set. Here we will use tapply() to explore the breakdown of age and ideology within our class data set.

```
tapply(ds$age, ds$ideol, summary)
```

```
## $`1`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   26.00   51.25   61.00   60.08   72.00   87.00 
## 
## $`2`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    24.0    51.5    63.0    60.4    70.0    92.0 
## 
## $`3`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   23.00   51.00   60.00   58.02   69.00   88.00 
## 
## $`4`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   18.00   47.00   60.00   57.61   69.00   93.00 
## 
## $`5`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   22.00   51.00   61.00   59.70   69.25   90.00 
## 
## $`6`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   23.00   54.00   63.00   62.59   72.00   99.00 
## 
## $`7`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   25.00   55.00   64.00   62.48   71.00   97.00
```

Notice that we inlcuded the "summary" command at the end. This returned a lot of information about ideology and age. We can be more specific, like asking R to tell us only the mean age of each ideology score:

```
tapply(ds$age, ds$ideol, mean)
```

```
##        1        2        3        4        5        6        7 
## 60.08197 60.40143 58.01622 57.60771 59.69512 62.58721 62.47578
```

### Other Methods of Exploring Your Data

Something important to know about R is that there are often many different ways to do a single obejctive. For example, we just used the tapply() function to look at the breakdown of ideolgoy by age. However, we can also use the by() function. The syntax is for by() is very similar to tapply(), so let's try it:

```
by(ds$age, ds$ideol, summary)
```

```
## ds$ideol: 1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   26.00   51.25   61.00   60.08   72.00   87.00
## --------------------------------------------------------
## ds$ideol: 2
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    24.0    51.5    63.0    60.4    70.0    92.0
## --------------------------------------------------------
## ds$ideol: 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   23.00   51.00   60.00   58.02   69.00   88.00
## --------------------------------------------------------
## ds$ideol: 4
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   18.00   47.00   60.00   57.61   69.00   93.00
## --------------------------------------------------------
## ds$ideol: 5
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   22.00   51.00   61.00   59.70   69.25   90.00
## --------------------------------------------------------
## ds$ideol: 6
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   23.00   54.00   63.00   62.59   72.00   99.00
## --------------------------------------------------------
## ds$ideol: 7
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   25.00   55.00   64.00   62.48   71.00   97.00
```

Notice how the results are essentially the same. Now let's use by() to examine a different relationship, age and gender. We wil now tell R to give us the mean age of each gender:

```r
by(ds$age, ds$gend, mean)
```

```
## ds$gend: 0
## [1] 60.2
## --------------------------------------------------------
## ds$gend: 1
## [1] 60.60039
```

Did you forget which value corresponds with each gender? Let's remind ourselves by using the factored gender variable:

```r
by(ds$age, ds$f.gend, mean)
```

```
## ds$f.gend: Women
## [1] 60.2
## --------------------------------------------------------
## ds$f.gend: Men
## [1] 60.60039
```

Let's now turn to other methods of exploring your data. The filter function allows you to filter out all the data except what meets the specific parameters you specify. Let's imagine that we want to only look at the data for men. We can use the filter() function to do so. The basic syntax follows this format:

filter(data, parameter1, parameter2, etc.)

We'll want to be sure to create specific names for the objects we assign the filters to. Let's now filter so that we will only look at the men in our class data set. Let's name the new data "ds.men"

```r
ds.men <- filter(ds, ds$gender==1)
```

Now let's imagine we want to only look at men over 30 years old. The filter function allows us to specify multiple parameters. All we need to do is tell R is filter us the data for men and, and more specifically men over 30.

```r
ds.men30 <- filter(ds, ds$gender==1 & ds$age > 30)
```

What if you wanted to look at a breakdown of this new dataset by ideology? You'd use the summary() function we've been using:

```r
summary(ds.men30$ideol)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   1.000   4.000   5.000   4.812   6.000   7.000       7
```

Another way we can filter our data is by using the slice() function. This allows us to filter the data by rows. For example, let's say you wanted to look at only the first 100 rows of our class data set:

```r
ds100 <- slice(ds, 1:100)
```

From here you would treat ds100 like you would any dataset!

### Subsetting Your Data

Another way you can breakdown your data is by subsetting it. This is similar to the way we use the filter() function. One difference is that the subset() argument comes in base R, whereas you have to make sure you have dplyr loaded to use filter(). Let's start by subsetting the data to only men:

```r
men.ds <- subset(ds, gender==1)
```

From here you would use men.ds just like any other data set.

We can also subset the data by selecting specific columns, by both names and numbers:

```r
ds.sub1 <- subset(ds, select = c(party:glbcc_risk))
ds.sub2 <- subset(ds, select = c(1:15))
```

You can also subset by selecting specific variables.

You can largely set whatever parameters you need by following the operator functions in R:

1. < less than
2. <= less than or equal to
3. greater than
4. = greater than or equal to
5. == exactly equal to
6. != not equal to
7. ! not (example: !x - pronounced "not x" )
8. | Or (example: x | y - pronounced "x OR y")
9. & And (example: x & y - pronounced "x AND y")

## Part Four: Working with Nominal Data

Often times we have to work with nominal data. This is data that does not necessarily have a numeric value, but rather is categorized by a word or label. Think political party. Let's take a look at our factored party variable:

```
table(ds$f.party)
```

```
##
##   Dem   Rep   Ind Other
##   869  1185   356    91
```

If we were analyzing this data, the "Other" category might not tell us much. Let's create a new variable that excludes the "Other" responses. We'll do this by recoding the factored party variable to count the "Other" responses as NAs

```
ds$f.party.2 <- car::recode(ds$f.party, "'Dem'='Dem'; 'Rep'='Rep' ;'Ind'='Ind' ;'Other'=NA;
                            else=NA;NA=NA")
table(ds$f.party.2)
```

```
##
##  Dem  Ind  Rep
##  869  356 1185
```

Notice that when we recode a factored variable, we need to put the label names in apostrophe? marks *within the quotation marks.*

### Finding the Mode

When working with nominal data, there are some statistical methods that don't make any intuitive sense. For example, if we were looking at our new factored political party variable, we couldn't find the mean, because that wouldn't make any sense. However, we could find the mode. Recall from mathematics that the mode is the value, or in this case the label, that occurs the most often in a group of values or labels.

In R, we can find the mode using the "modeest" package, which we should have installed and loaded at the beginning of the lab. Within the package we use the mlv() function. It is important to note that the mlv() package does not work for factored variables. This might seem counterintuitive, but there are easy ways to get around it. What we have to do is tell R to think of the factored variable as if it were numeric, or an integer. In this case, we will use the as.integer command to do so:

```
mlv(as.integer(ds$f.party.2), na.rm = TRUE)
```

```
## Mode (most frequent value): 3
## Bickel's modal skewness: -0.5082988
## Call: mlv.integer(x = as.integer(ds$f.party.2), na.rm = TRUE)
```

After running this test we see that the mode is 3. In order to check which nominal label corresponds with 3, we can simply print out a table of the factored variable:

```
table(ds$f.party.2)
```

```
##
##  Dem  Ind  Rep
##  869  356 1185
```

Here we see that 3 corresponds with Republican, so Republican is the most frequent value.

Another method we can use to find the mode is to make a table, and sort the table by how often the labels occur.

```
sort(table(ds$f.party.2), decreasing = TRUE)
```

```
##
##  Rep  Dem  Ind
## 1185  869  356
```

Here we see the same result, but found it using a different method.

We can also use the freq() function to look at the frequency:

```
freq(ds$f.party.2, plot = FALSE)
```

```
## ds$f.party.2
##       Frequency Percent Valid Percent
## Dem         869  34.119         36.06
## Ind         356  13.977         14.77
## Rep        1185  46.525         49.17
## NA's        137   5.379
## Total      2547 100.000        100.00
```

This gives us how many are in each group, but also the percentages. If we wanted to construct a table that shows the percentage breakdown only, we would first create a table of the variable, then use prop.table(), then multiply by 100.

```
party.table <- table(ds$f.party.2)
party.table.per <- prop.table(party.table)*100
party.table.per
```

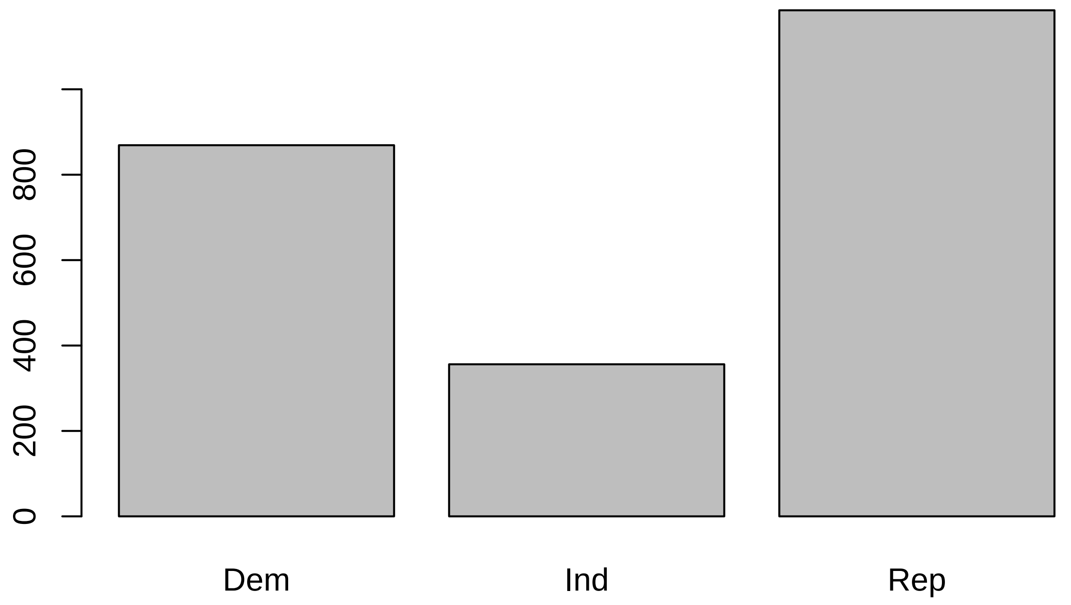
```
##
##      Dem      Ind      Rep
## 36.05809 14.77178 49.17012
```

### Visualizing Nominal Data

Now let's venture into some early and simple visualizations. R has nearly countless ways to visualize data, but let's start with a really simple method: a barplot.

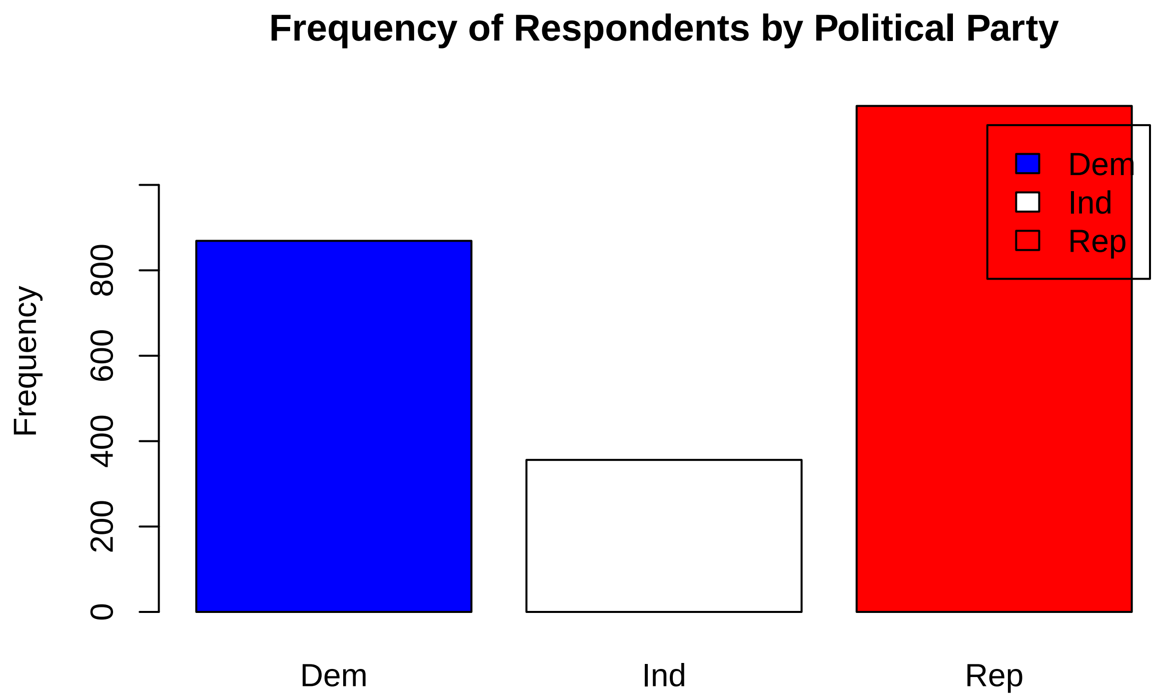
Barplots are great for visualizing nominal data. When creating a barplot, you first use the barplot() function, and then specify the object that you want visualized. Let's visualize the table of our political party breakdown:

```
barplot(table(ds$f.party.2))
```



Instead of having to type "table(ds$f.party.2)" inside the barplot function, we could use the party.table object we created earlier. Let's do that, but this time add a legend and some color:

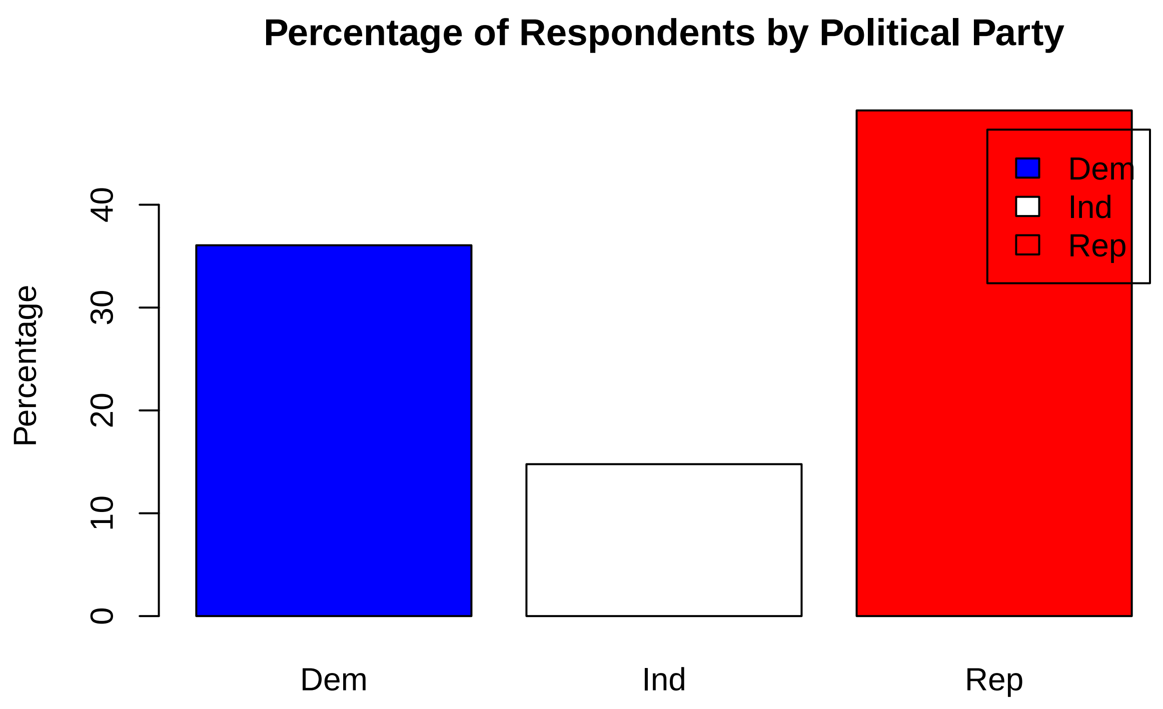
```r
barplot(party.table, beside=TRUE, legend=rownames(party.table), col = c("Blue", "White", "Red"),
        ylab ="Frequency", main="Frequency of Respondents by Political Party")
```



The specific syntax needed to customize visualiztions will get easier with time.

Remember the table we created that shows the percentage breakdown of the parties? Let's visualize that this time:

```r
barplot(party.table.per, beside=TRUE, legend=rownames(party.table.per), col = c("Blue", "White", "Red")
        ylab ="Percentage", main="Percentage of Respondents by Political Party")
```



## Part Five: Working with Ordinal Data

Recall that ordinal data is data that is assigned numeric values, but on an ordered scale. One very intuitive type of ordinal data is educaton level. Edcation level is on an ordered scale: some high school is higher than no high school, a high school diploma is higher than some high school, some college is higher than high school, etc.

Let's use the education variable in our class data set to explore how to work with ordinal data. First let's look at a table of the education variable:

```
table(ds$education)
```

```
##
##   1   2   3   4   5   6   7   8
##  41 341 132 524 205 713 438 149
```

We can see that there are 8 categories in the ordered scale, but we aren't displaying what levels of education they are. Let's factor the variable to do so:

```
ds$f.education <- factor(ds$education, levels=c(1,2,3,4,5,6,7,8),
                   labels=c("< HS","HS/GED","Vocational/Technical", "Some College",
                        "2 year/Associates", "Bachelor's Degree", "Master's degree", "PhD/JD/MD"))
table(ds$f.education, useNA = "always")
```

```
##
##                 < HS               HS/GED Vocational/Technical
##                   41                  341                  132
##         Some College    2 year/Associates    Bachelor's Degree
##                  524                  205                  713
##      Master's degree            PhD/JD/MD                 <NA>
##                  438                  149                    4
```

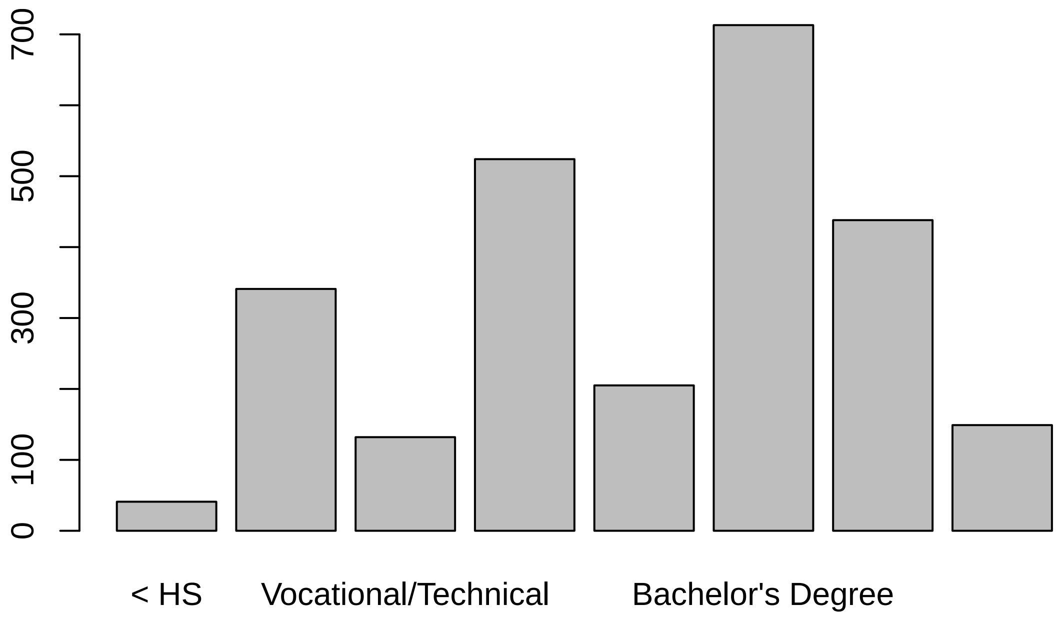
Let's practice finding the mode of education:

```
sort(table(ds$f.education), decreasing = TRUE)
```

```
##
##    Bachelor's Degree         Some College      Master's degree
##                  713                  524                  438
##               HS/GED     2 year/Associates            PhD/JD/MD
##                  341                  205                  149
## Vocational/Technical                 < HS
##                  132                   41
```

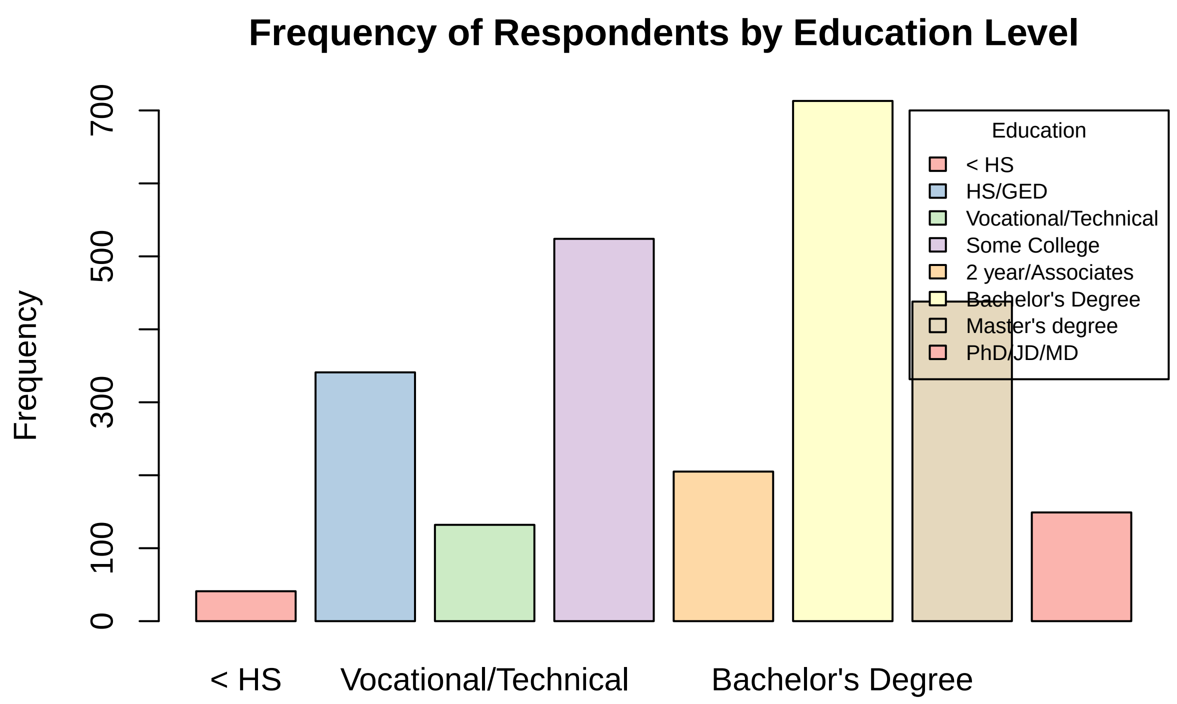
Now let's visualize our education data:

```
barplot(table(ds$f.education))
```



Now let's create a table object for the education data and create a new visualization with some more detail:

```
educ.table <- table(ds$f.education)
barplot(educ.table, beside=TRUE, legend=rownames(educ.table),
        col = c("#fbb4ae", "#b3cde3", "#ccebc5", "#decbe4", "#fed9a6", "#ffffcc", "#e5d8bd"),
        ylab ="Frequency", main="Frequency of Respondents by Education Level",
        args.legend = list(title = "Education", x = "topright", cex = .7), ylim = c(0, 700))
```



Those colors listed are hexadecimal colors, which R recognizes. You can check out color schemes here: http://colorbrewer2.org/

## Part Six: Working with Interval Data

Interval data is similar to ordinal data, but with interval data the difference between levels does have some intuitive value to it. For this section we will use an ordinal measure, but treat it as if it is interval. In our class data set we have a variable that measures an individual's perception of the risk posed by global climate chage, glbcc_risk, on a scale of 0 being no risk to 10 being extreme risk.

Within the "psych" package, there is a function called "describe". Let's use the describe() function to examine the variable:

```
describe(ds$glbcc_risk)
```

```
##    vars    n mean   sd median trimmed  mad min max range  skew kurtosis
## X1    1 2536 5.95 3.07      6    6.14 2.97   0  10    10 -0.32    -0.93
##      se
## X1 0.06
```

Now let's find the mode:

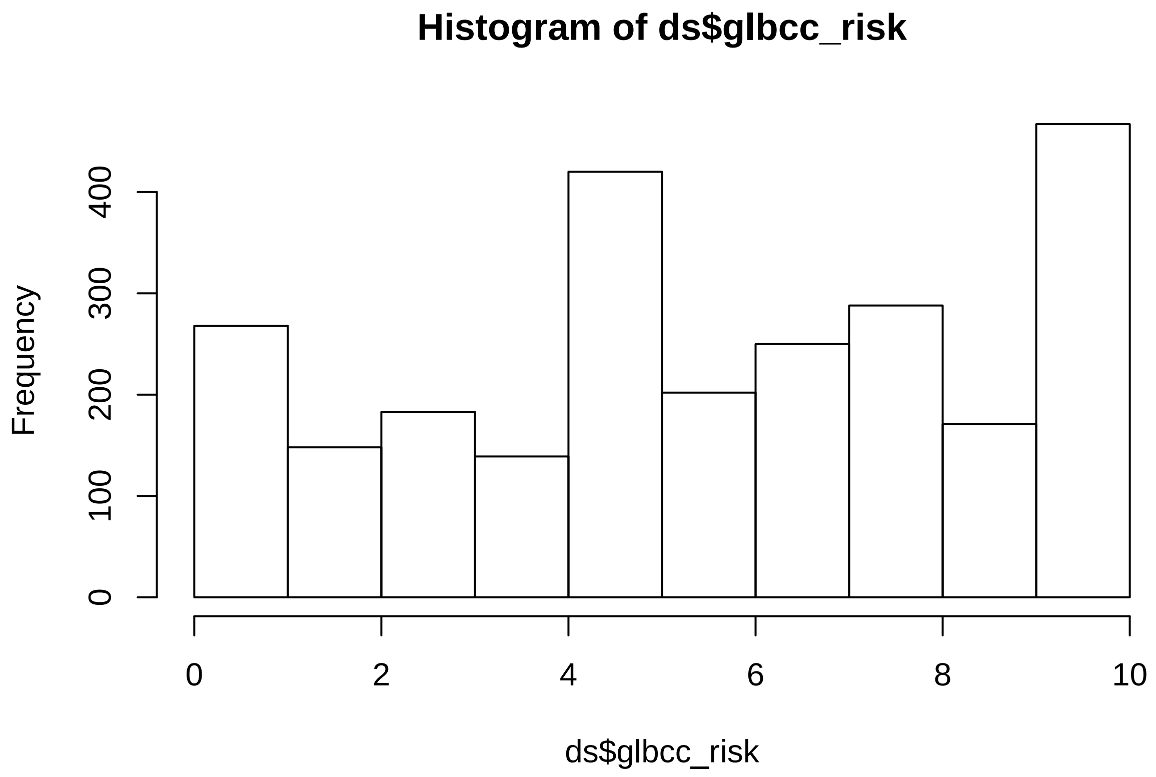
```
mlv(ds$glbcc_risk, na.rm = TRUE)
```

```
## Mode (most frequent value): 10
## Bickel's modal skewness: -0.8158517
## Call: mlv.integer(x = ds$glbcc_risk, na.rm = TRUE)
```

As we can see, the most frequent value is 10.

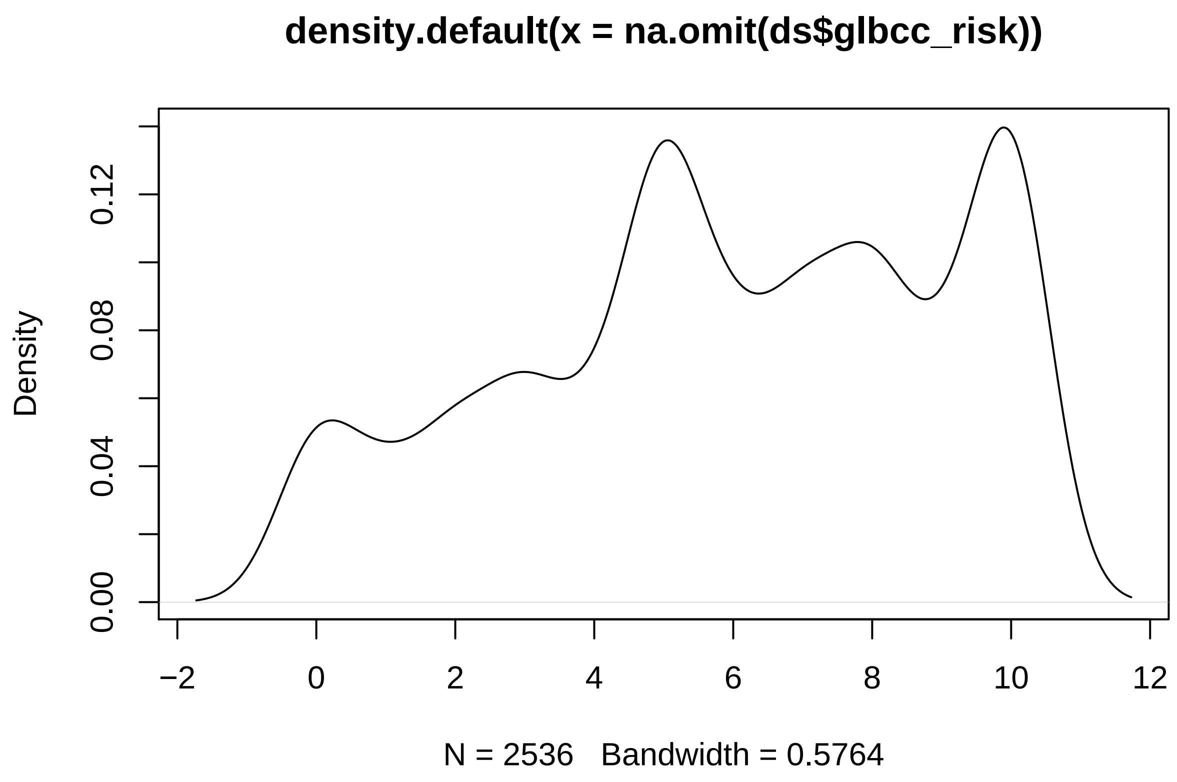
When visualizing interval data, it is more appropriate to use a histogram instead of a barplot. This is because a histogram displays values on a continuous scale, not individual values separated from others. Let's make a histogram using the hist() function.

```r
hist(ds$glbcc_risk)
```



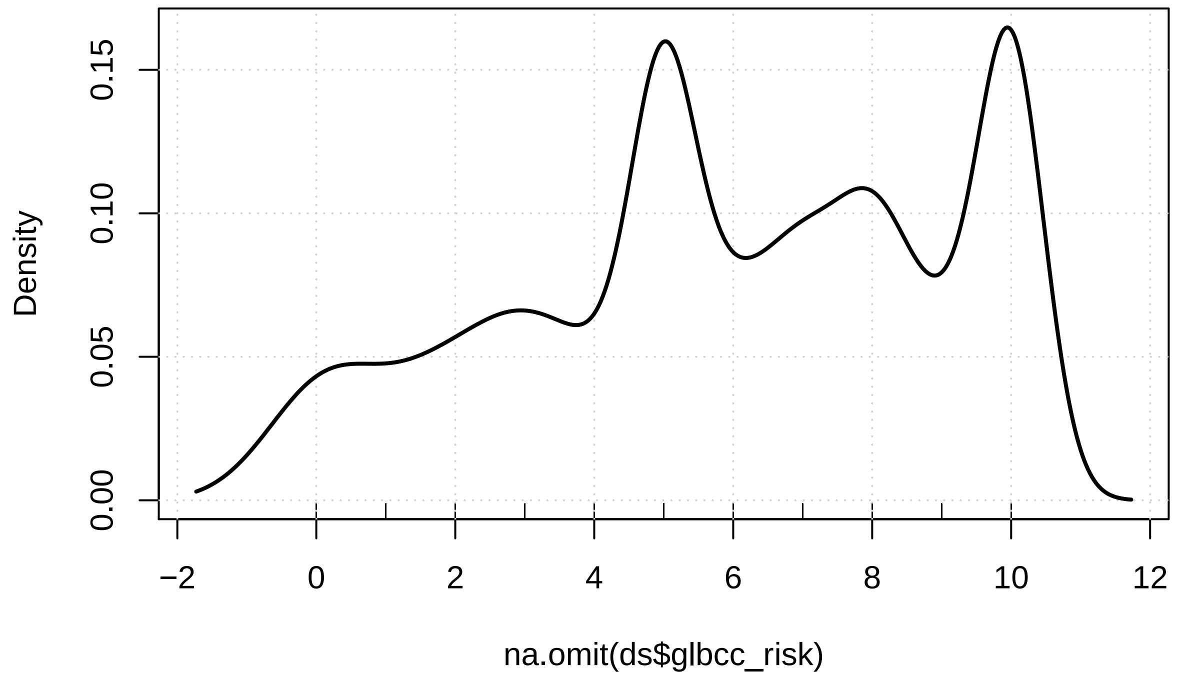
Now let's look at the density of the variable. To do this, we need to remove all missing values from the variable. The na.omit() fuction does this. Then we tell R to plot the density.

```r
density.gcc <- density(na.omit(ds$glbcc_risk))
plot(density.gcc)
```

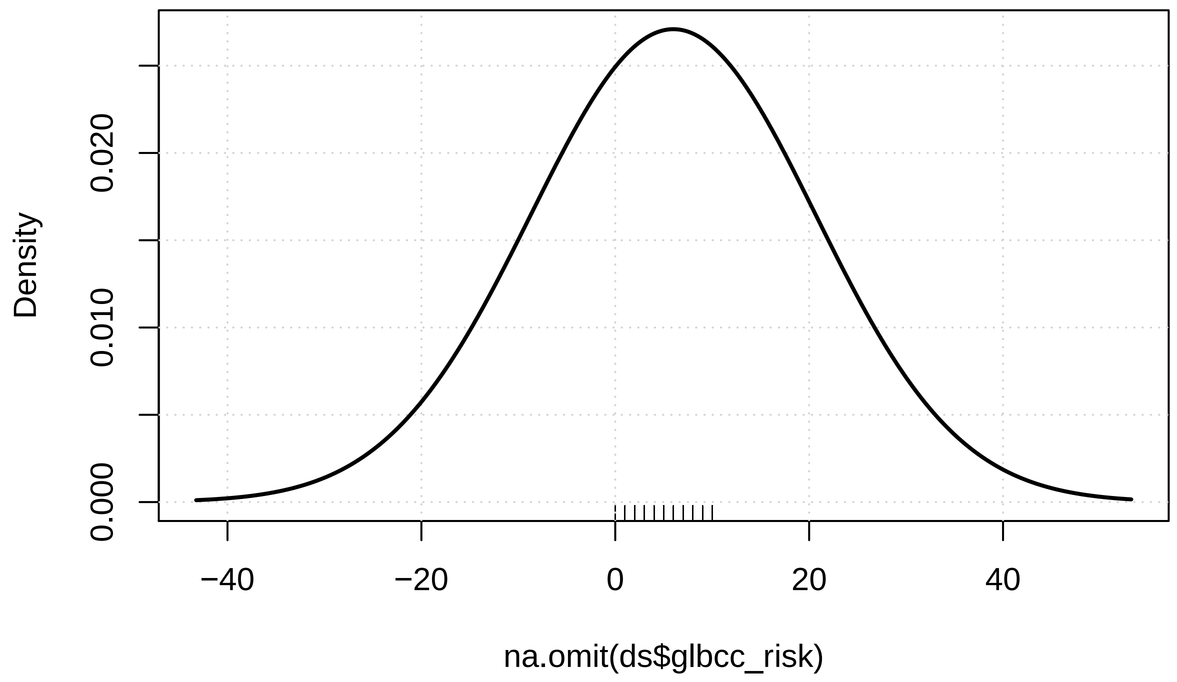


We can also do this by using the densityPlot function in the "car" package. Again, we need to make sure the NAs are ommitted. First we will do the basic plot, and then we'll adjust the bandwitdth.

```
densityPlot(na.omit(ds$glbcc_risk))
```

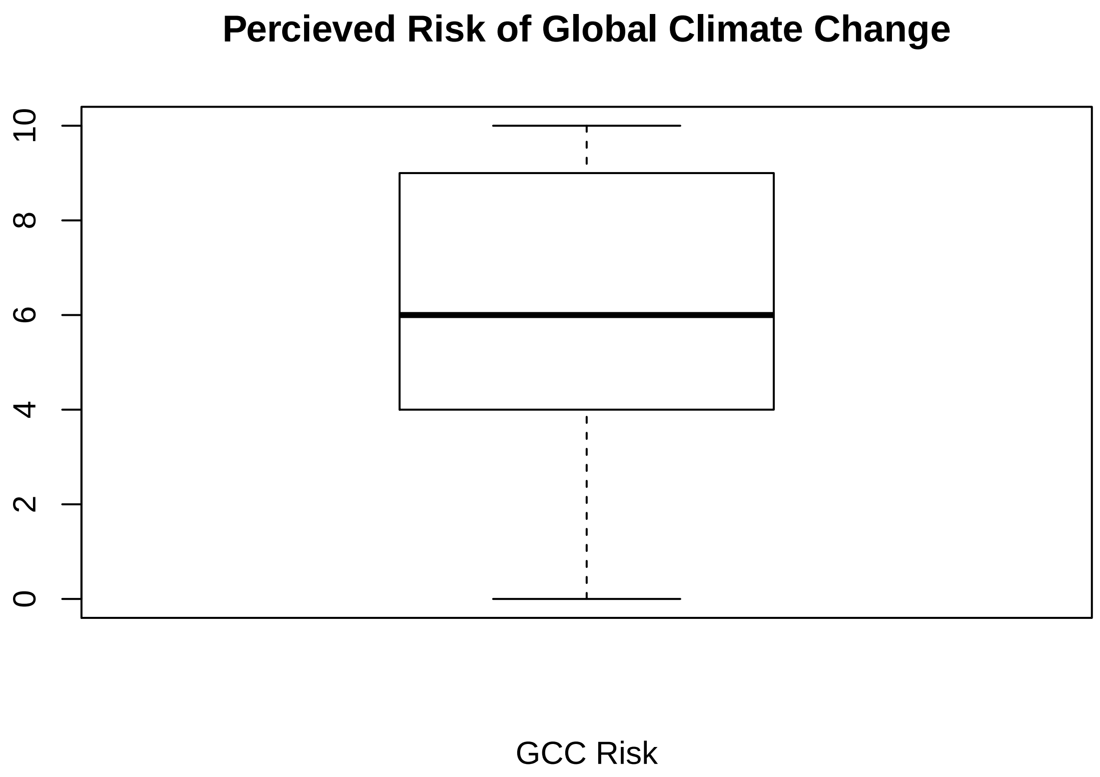


```
densityPlot(na.omit(ds$glbcc_risk), adjust = 25)
```



Now let's create a boxplot of the variable, which is a different way to look at it:

```
boxplot(ds$glbcc_risk, xlab = "GCC Risk", main = "Percieved Risk of Global Climate Change")
```


Percieved Risk of Global Climate Change
GCC Risk