Working with a single variable

Teaching note:

When facing a new set of data, a data scientist often begins his analysis by examining individual variables in a variety of ways to see what it looks like. A great deal can be learned from analyzing individual variables one at a time: you are familiar with means, medians, and standard deviations, but with visual tools such as KDE, CDF, and QQ plots, we can learn even more about the way a variable is distributed amongst a population. This tutorial samples some of the ways we can explore individual variables in R.

Getting the data:

For this assignment, we'll be using survey data that captures customers' perceptions of their supplier. The data in HBAT.csv rates several attributes of the supplier "HBAT". (See the document HBAT-description for full details.)

To load the data into your R session, you'll use the `read.csv` function, which is really just a special case of the `read.table` function that loads the data from a CSV file as an R data frame. Change your working directory to the folder that contains our tutorial datasets, and enter the following line of R code:

```
hbat <- read.csv("HBAT.csv")
```

If you would prefer to use code that works regardless of R's working directory (for example, if you are writing a script you will save and run again in the future), you could use the full address on disk of the data file, like so:

```
hbat <- read.csv("C:\\datasets\\HBAT.csv")
```

Examine the data set:

A few commands let you quickly look at the whole data set:

1. The `names` function is a quick way to get the list of variables:

   ```
   names(hbat)
   ```

2. Look at the entire data frame by typing its name:

   ```
   hbat
   ```

3. There was far too much data to fit on-screen at once, so try instead using the `head` function to look at only the first few rows and get a feel for the type of data in each variable:

```
head(hbat)
```

4. The `str` function gives you an even more compact overview of the variables, showing the name, data type, and first few observations of each variable on a single line:

   ```
   str(hbat)
   ```

5. The `summary` function gives you some simple statistics about each variable: the minimum, maximum, median, 1st and 3rd quartiles, and the mean:

   ```
   summary(hbat)
   ```

Examine a single variable:

Let's focus on the variable Website_User_Friendliness for a while. We can start by shortening its name to something a little easier to type:

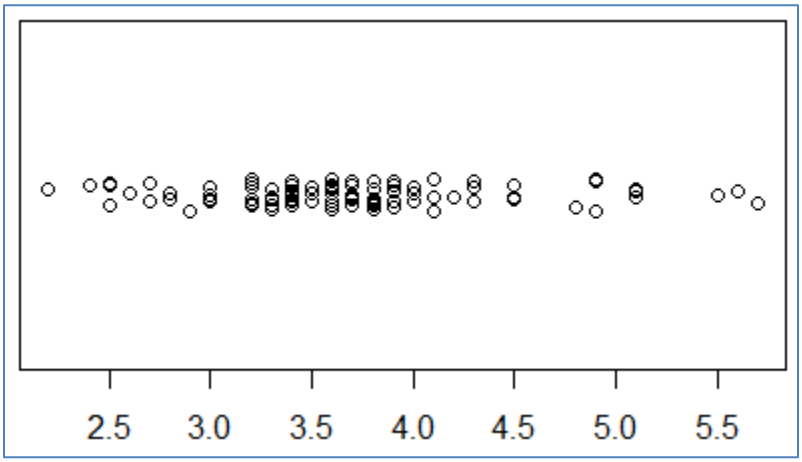
```
wuf <- hbat$Website_User_Friendliness
```

Let's learn as much as possible about this variable:

6. Compute basic summary statistics

   ```
   length(wuf)  # the number of observations
   summary(wuf)
   mean(wuf)
   median(wuf)
   mode(wuf)
   quantile(wuf)  # by default, gives quartiles
   quantile(wuf,probs=seq(0,1,0.1))  # gives deciles
   ```

7. Visualizing the data will help us to gain a better understanding of it than we can glean from statistics alone. Is the variable evenly distributed? Does it follow a "bell curve" or normal distribution? Is it lopsided or discontinuous? A "dot plot" is a simple visualization of the distribution, placing one dot or tick mark on a horizontal line for each observation of the variable.



From the plot we can plainly see that many of the surveys ranked the website with scores between 3.0 and 4.0, suggesting something of a curved distribution.

To generate this plot in R, we need to know a little about R's `plot` function. Every data point that is plotted must have Cartesian x and y coordinates. In the above example, the x coordinate is the observed variable. The y coordinates are (almost) the same, so that the dots will line up horizontally. The following line of code generates y values by repeating the value 1 one hundred times (the same as the number of x observations):

```
yvalues <- rep(1,100)
```

Since we don't want the dots to be precisely on top of one another and obscure their density, we use the R function `jitter` to make them just slightly different. Our dot plot is actually a "jitter plot" now:

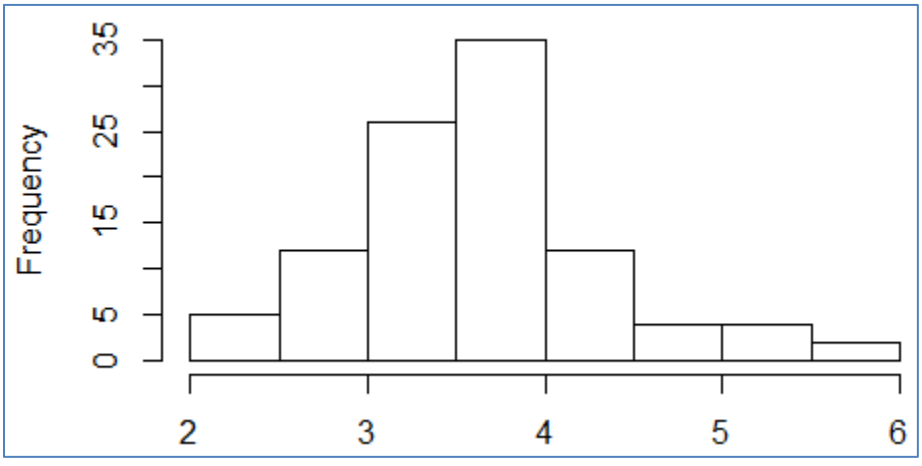
```
yvalues <- jitter(rep(1,100))
```

We could plot the values with the simple code plot(wuf,yvalues) but it will not look very good. Because R automatically determines the range of the x and y axes to display, it will cause the jittered values to spread out visually over the whole plot. To constrain our perspective, we want to specify the limits of the u axis with a `ylim` parameter. We also remove the y axis with `yaxt="n"` and its label with `ylab=""`:

```
plot(wuf,yvalues,ylim=c(0.8,1.2),yaxt="n",ylab="")
```

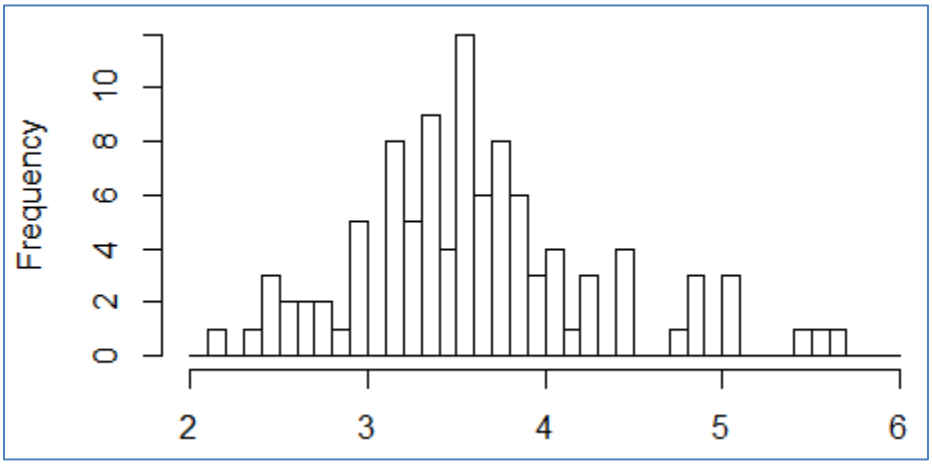
8. A histogram is another way, perhaps a more intuitive way than a dot plot, to visualize the distribution of the variable. Histograms are very easy to generate in R:

```
hist(wuf)
```



The bin widths by default do not give a great deal of detail. We may specify with the `breaks` parameter where we want the dividing lines to go. For example, to assign a bin to every tenth of a point between 2.0 and 6.0:

```
hist(wuf,breaks=seq(2,6,0.1))
```



By the way, we can also generate a quick text-based histogram called a "stem and leaf" plot with this simple line of R code:

```
stem(wuf)
```

Which gives us:

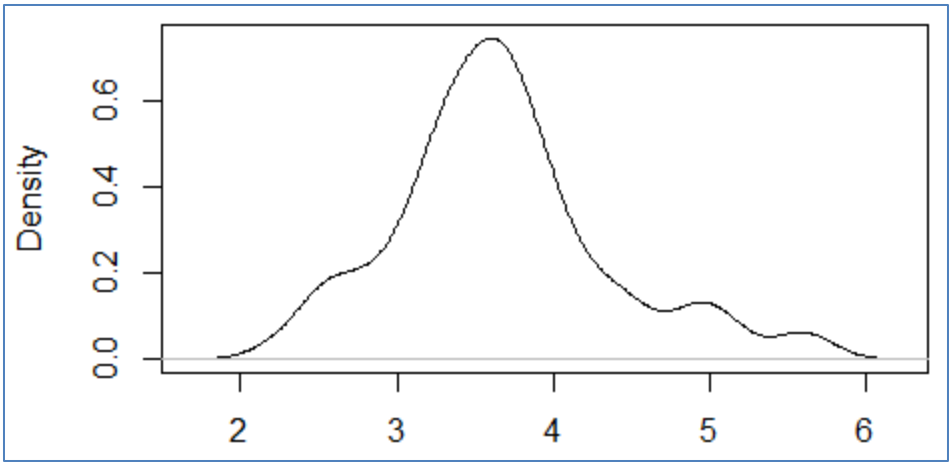
```
  The decimal point is at the |

  2 | 24
  2 | 5556677889
  3 | 000002222222233333444444444
  3 | 55556666666666667777778888888899999
  4 | 00011112333
  4 | 55558999
  5 | 111
  5 | 567
```

One of the drawbacks to a histogram is that there is no formal rule dictating the number or width of bins. Narrower bins give more detail, but are sensitive to gaps when the number of observations is relatively small. Wider bins may give a more reliable shape of the distribution, but with less detail. Another drawback is that histograms of the same data may look different if the dividing lines are moved left or right.

9. An alternative visualization that mitigates at least the latter of these problems is a kernel density estimate (KDE) plot. Unlike a histogram, this is one that can't be easily done by hand, so it is a new possibility that software like R has made available to data scientists:

```
plot(density(wuf))
```

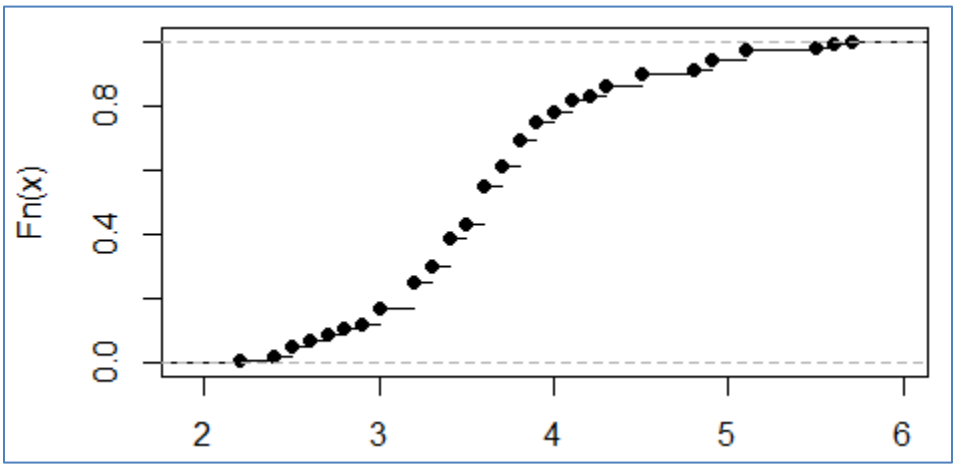


Like a histogram, a KDE plot can be made smoother (by using a larger "bandwidth") or more detailed (a smaller bandwidth). Unlike a histogram, a KDE plot does not depend on any arbitrary choice about the location of bins. To see a more detailed version of the KDE plot above, try it with a bandwidth of 0.1 or even 0.05:

```
plot(density(wuf,bw=0.1))
plot(density(wuf,bw=0.05))
```

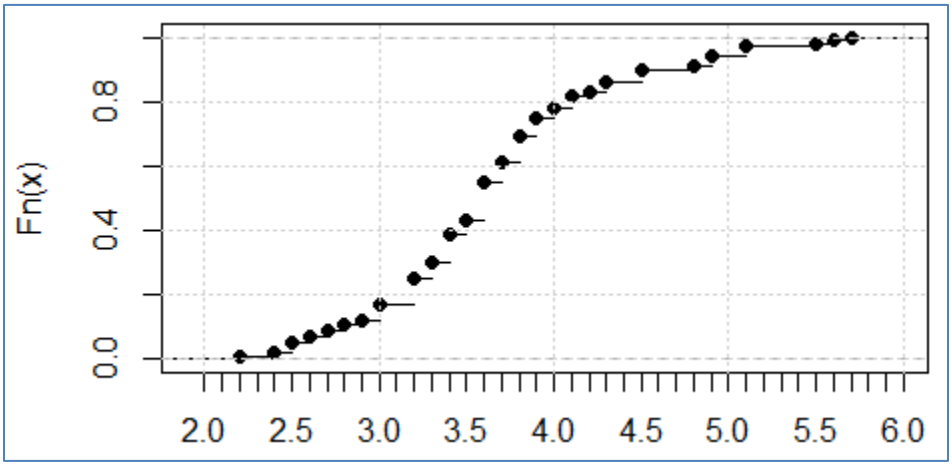
10. The histograms and KDE plots make it clear that the most frequent survey responses gave the website a score somewhere between 3 and 4, but just how many surveys does that reflect? The human eye is not very good at estimating the area under a curve, so we turn to a visualization of the variable's cumulative distribution function (CDF). The CDF for each x represents the fraction of the data points that are less than ("to the left of") x. This allows us to see, for example, what fraction of the survey responses were greater than, less than, or between any given values:

```
plot(ecdf(wuf))  # ecdf stands for 'empirical CDF'
```



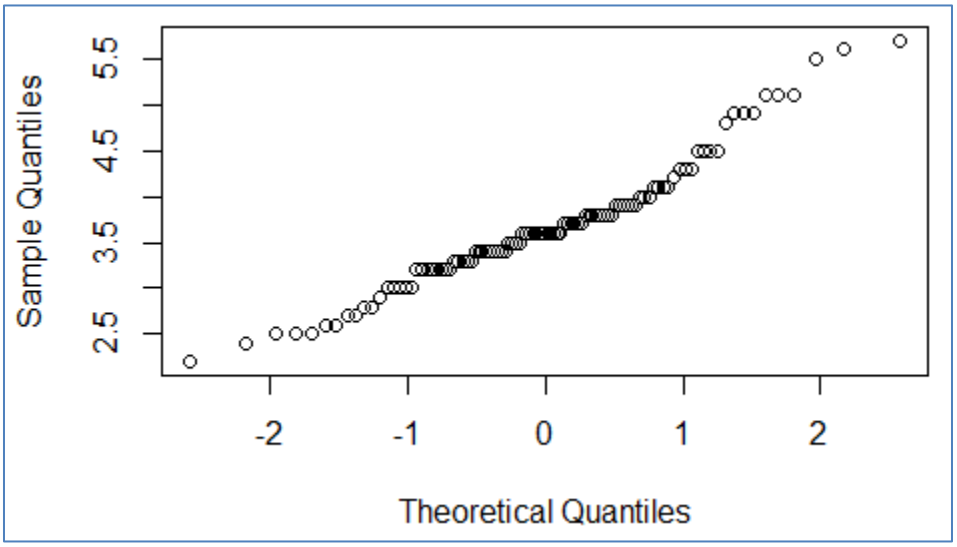
From the CDF plot we can see that fewer than 20% (or 0.2) of the survey responses scored the website below 3 points, but about 80% scored it less than 4 points. Fully 40% appear to have scored it between 3.5 and 4.0. It would be easier to read the graph if it had a few more guidelines and tick marks, so let's improve it like so:

```
plot(ecdf(wuf),xaxp=c(2,6,40))  # put 40 tick marks between 2.0 and 6.0
grid()  # add grid lines to the plot
```



11. A variation on the idea of CDF plot will allow us to ask one more question about the distribution of the data: is it "Normal" (i.e. a Gaussian distribution)? One tool for doing this type of test is a quantile-quantile or "QQ" plot.

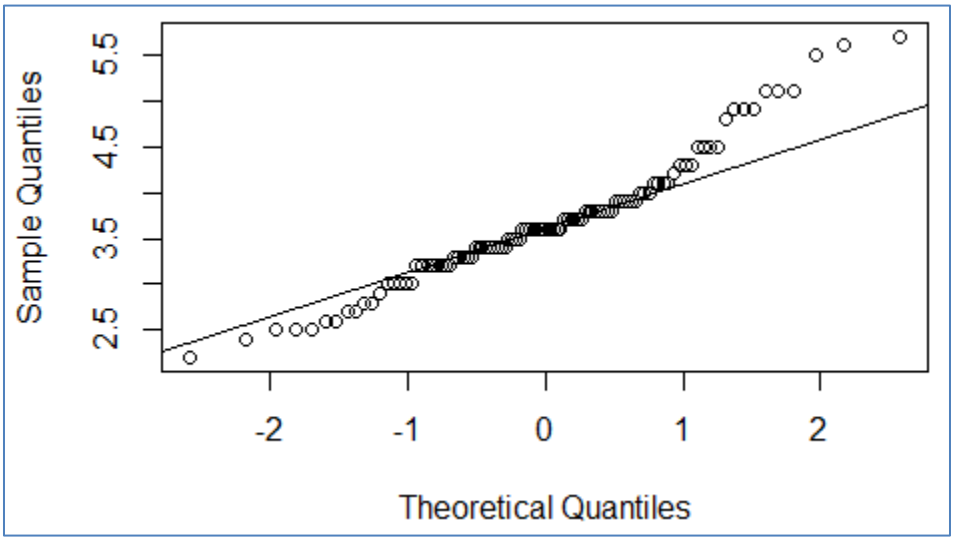
```
qqnorm(wuf)
```



In this plot, observations are plotted in order from the least to greatest with a y coordinate

equal to the observed value and x equal to the theoretically-predicted observation at that quantile. A straight diagonal line would indicate a perfect fit to the theoretical distribution. Add a guide line to the plot to see how far this variable actually deviates from the Normal:

```
qqline(wuf)
```



We see that the deviation is most pronounced at the tails. The 98th observation for example was much greater than the expected 97th percentile value if the data were normally distributed.

The more general function `qqplot()` can be used to compare a variable to other theoretical distributions or other empirical datasets. To get more information about how to use a function in R, type a "`?`" followed by the function name, without parentheses:

```
?qqplot
```