Working with three or more variables

Teaching note:

We can analyze an individual variable in a number of ways to understand its distribution. We can analyze pairs of variables to understand their relationships. A special case of two-variable analysis is the time series. Building on previous tutorials, we will begin in this lesson to look at some of the analyses we can do with three or more variables to show how they interact or fit together into a bigger picture. The visualizations we will use include several interesting tools: interaction plots, small multiples, 3D surface plots and heat maps.

Getting the data:

For this tutorial we will use a 1985 dataset giving the specifications of a large number of import cars, available online in the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml/datasets/Automobile. This website is a great source of free data sets to experiment and learn with. We have added the variable names to the CSV file. To read the data into R, and briefly examine it, use:

```
imports <- read.csv("imports-85.csv")
str(imports)
head(imports)
```

The data requires a bit of clean-up. In particular, many of the numeric variables are seen by R as "factor" variables. To transform them into numeric data, we first transform them into character data (text) and then into numbers. Clean up the horsepower, MPG, and price variables like so:

```
imports$horsepower <- as.numeric(as.character(imports$horsepower))
imports$highway.mpg <- as.numeric(as.character(imports$highway.mpg))
imports$price <- as.numeric(as.character(imports$price))
```

Even worse, the number of cylinders is written out long-form (e.g. "four", "six") instead of with numerals. R doesn't understand these are numbers, so it puts them in an arbitrary order:

```
Levels: eight five four six three twelve two
```

We fix this by first re-naming the "levels" of the variable with numerals, and then converting the variable as we did the others:

```
levels(imports$num.of.cylinders) <- c(8,5,4,6,3,12,2)
imports$num.of.cylinders <-
as.numeric(as.character(imports$num.of.cylinders))
```

Transforming the variables to numbers incidentally changed missing values from the placeholder "?" to the value NA which is understood by R to mean missing data. To save us some headaches down the line, we're going to simply discard the incomplete rows with `na.omit()`.

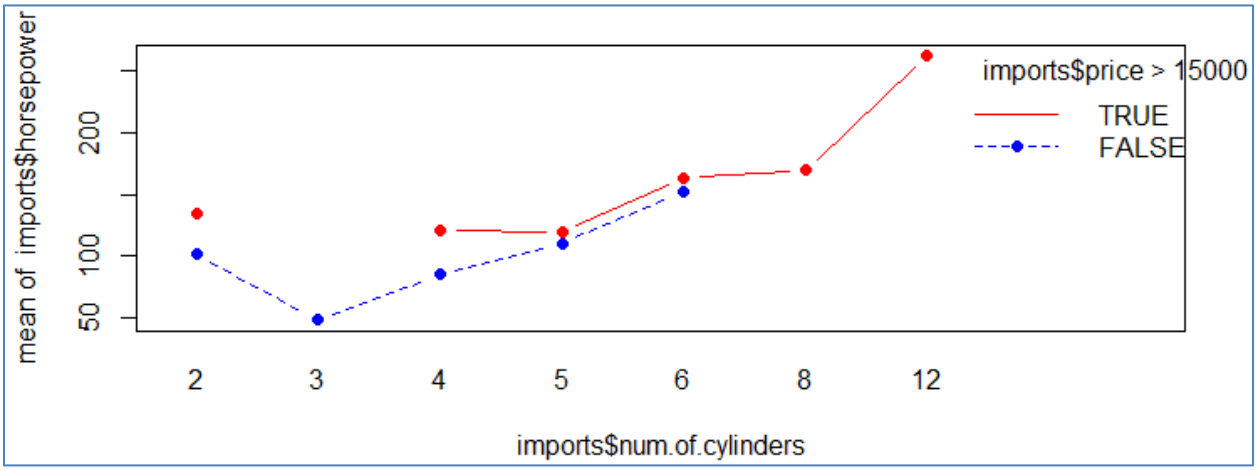
```
imports <- na.omit(imports)
```

Visualizing an interaction:

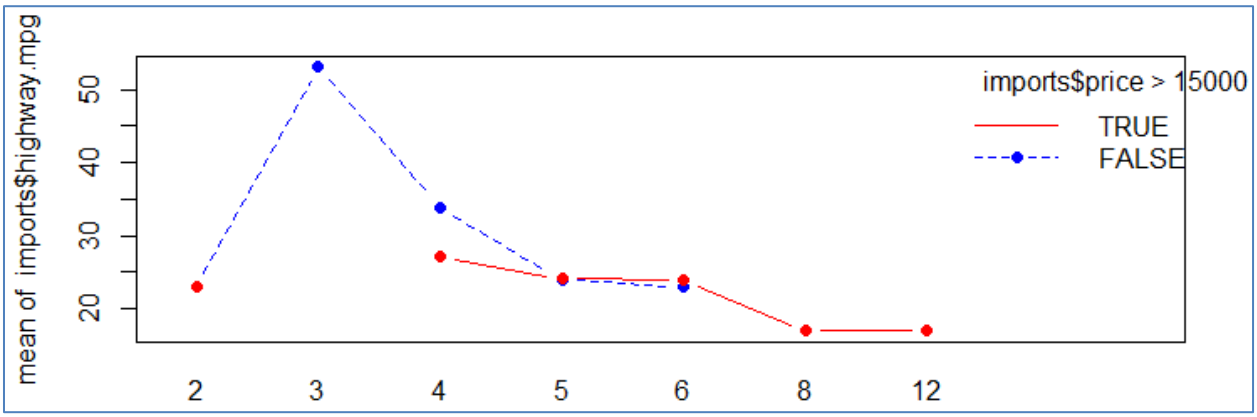
An "interaction effect" is present when the level of a third variable alters the relationship between two variables of interest. For example, we'd like to know if the relationship between number of cylinders, and horsepower, is the same for expensive cars as it is for inexpensive ones.

1. Essentially what we can do is separate the data into two groups and plot the relationship between cylinders and horsepower for both groups on the same plot. (We did something similar at the end of the "working with two variables" tutorial.) However, R has a convenient function just for producing interaction plots. It automates a lot of the intermediate work, such as summarizing the data (e.g. taking the mean).

```
interaction.plot(imports$num.of.cylinders,  # variable for the x axis
                 imports$price>15000,       # the interaction variable
                 imports$horsepower,        # variable for the y axis
                 mean,                      # how to summarize the data
                 pch=19,type="b",col=c("blue","red"),legend=TRUE)
```



In this case the relationship seems similar for both groups, but it's interesting to note the gaps in the data. No cars priced under $15,000 had more than six cylinders, for example. We see a somewhat different interaction for the relationship between cylinders and highway miles-per-gallon:



2. What if we wanted to view this relationship in more than two price categories? We can create a "multiplot", also known as a "small multiples" visualization, in which essentially the same plot is repeated several times in parallel for a different subset of the data.

   The following code gives us a simple scatter plot for num.of.cylinders and horsepower variables. The parameters remove the x and y axis tick marks and labels. The `par()` statement reduces the margins to a minimum:

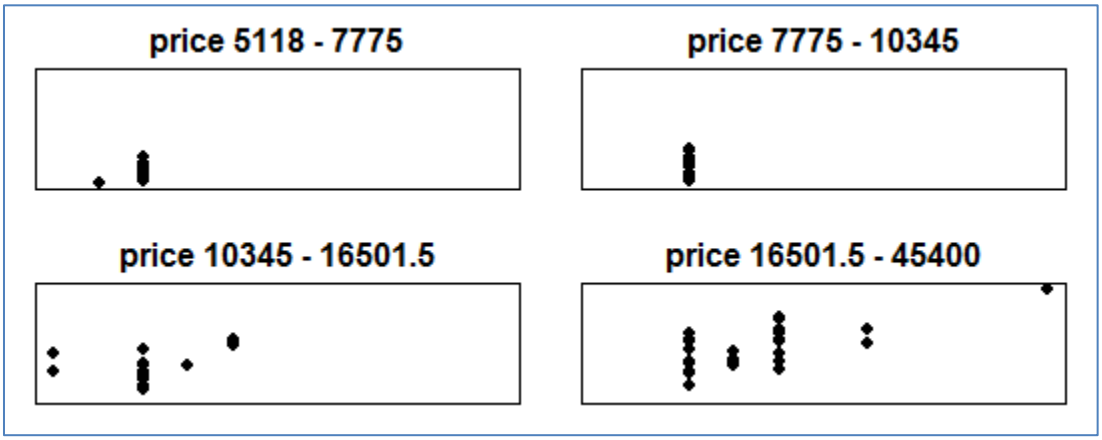
```
par(mar=c(1,1,2,1))
plot(imports$num.of.cylinders,imports$horsepower,
  main="cylinders x HP plot", xaxt="n",yaxt="n",xlab="",ylab="",pch=19)
```

   Now we'll loop through the data by quartiles and reproduce the same graph four times. The line `par(mfrow=c(2,2))` sets up a 2×2 multi-plot in which our plots will be placed, in order, from left to right and then top to bottom. (Use the `mfcol` parameter instead if you wish to place plots from top to bottom first, then left to right.) The next line finds the minimum, 1st quartile, median, 3rd quartile, and maximum value of price. Then we use a simple `for` loop to produce four plots.

   The `thisquartile` variable is a logical vector – a vector of `TRUE` and `FALSE` values that tells us which data are in the quartile we're focusing on now. The `plot()` command in the loop is just like the one above except that it limits the data by the `thisquartile` index (so it only plots one quartile's data in each of the four plots). It also gives each of the four plots a unique title, and forces their x and y axes to have the same range.

```
par(mfrow=c(2,2))
quarts <- quantile(imports$price)

for(q in 1:4) {
  thisquartile <- (imports$price >= quarts[q])
                  & (imports$price <= quarts[q+1])
  plot(imports[thisquartile,]$num.of.cylinders,
       imports[thisquartile,]$horsepower,
       main=paste("price",quarts[q],"-",quarts[q+1]),
       xaxt="n",yaxt="n",xlab="",ylab="",pch=19,
       xlim=c(2,12), ylim=c(48,262))
}
```



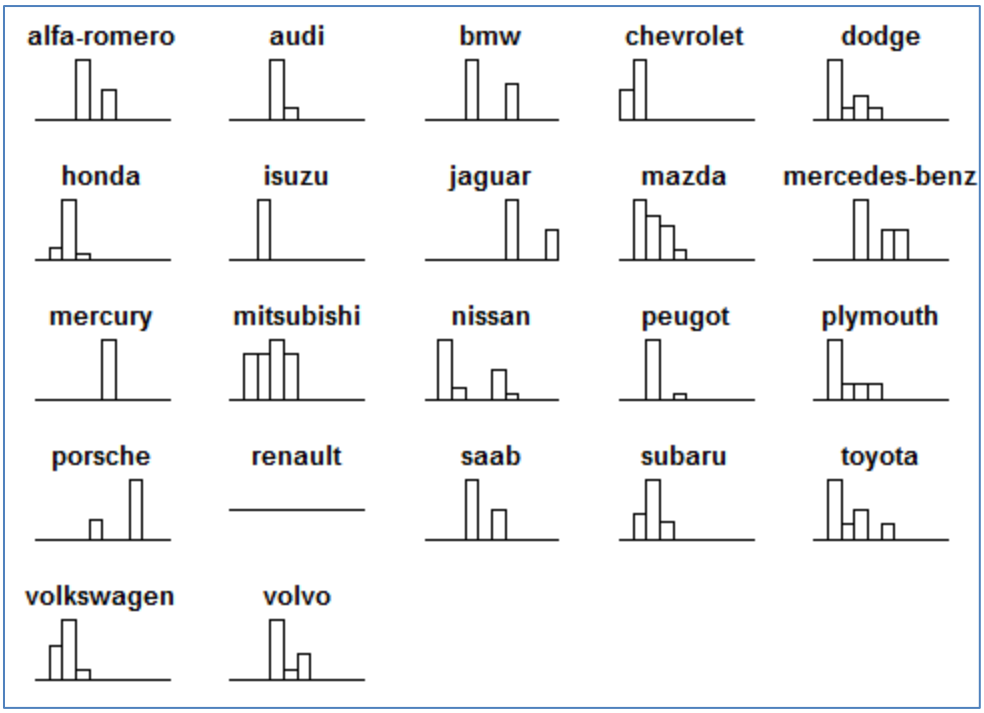
More small multiples:

The "small multiples" idea is often useful when comparing data by a non-quantitative variable. In our case, we might want to compare some feature of the data for different makes of cars.

1. This plot repeats a histogram of horsepower for each import make:

```
par(mfrow=c(5,5))
par(mar=c(1,1,2,1))
for(m in levels(imports$make)) {
  hist(imports[imports$make==m,]$horsepower,
       breaks=seq(25,275,25),
       xaxt="n",yaxt="n",xlab="",ylab="",
       main=m)
}
```

   The resulting plot:



Using three dimensions:

A typical plot uses x and y dimensions to represent two variables. We can extend this by adding a z dimension to represent a third variable.

1. We'll plot the auto makes on one axis, a number of variables on another axis, and the mean value of each variable on the third (z) axis. First, let's generate the data:

   ```
   makes <- aggregate(imports,by=list(imports$make),FUN=mean)
   ```

   The `aggregate()` function splits the `imports` data frame into subsets by `make`, applies the `mean()` function to each subset, and returns a data frame of the summarized data. It is a very versatile function. Next, since there are a lot of variables measured in this data set, we'll take a subset of five interesting ones to focus on:

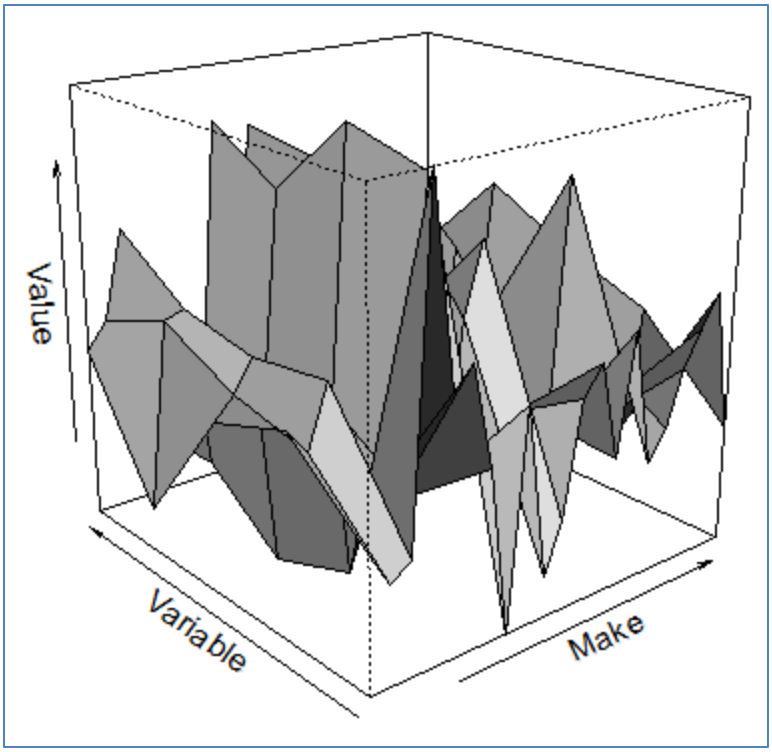
   ```
   variables <-
     c("highway.mpg","horsepower","curb.weight","wheel.base","price")
   zdata <- makes[variables]
   ```

   We named the subset `zdata` because it contains the mean values themselves but does not include the names of the makes or the names of the variables. Those will be our x and y axes. Because these variables are in very different scales (e.g. MPG ranges between 18 and 47, while price ranges from 6000-35000), we will normalize each variable by its mean and standard deviation using the `scale()` function. Finally we change zdata from a data frame to a simpler matrix because the plotting function we will use requires it.

   ```
   zdata <- scale(zdata)
   zdata <- as.matrix(zdata)
   ```

   A "surface plot" is a 3D version of a line graph in which the z values are represented by the apparent height of the points. A surface plot needs numeric values for the variables, so we use the sequence 1:21 instead of the make names, and 1:5 instead of the variable names, and make a surface plot using the `persp()` function:

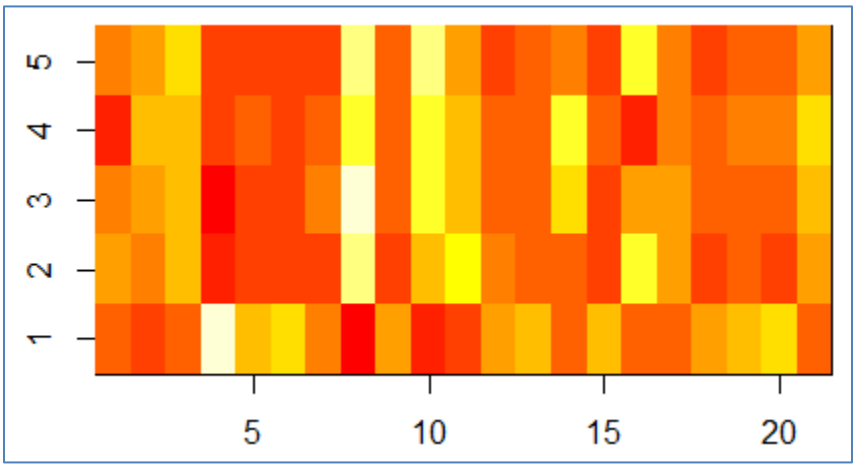
   ```
   persp(1:21,1:5,zdata,
         theta=-40,phi=20,r=5,d=1,  # 3D positioning parameters
         ltheta=-10,lphi=10,shade=1,  # 3D light/shade parameters
         xlab="Make",ylab="Variable",zlab="Value"
         )
   ```



As flashy as a surface plot looks, though, it is almost indecipherable. Part of the problem is the difficulty of placing the make and variable names on the x and y axes, but also it is simply very hard for the eye to correlate z values with their x and y positions.

2. A less flashy but more useful variation is a false-color plot, often called a "heat map". Instead of using a third dimension, this data visualization uses color or a shade of grey to indicate the value of the third variable. R has a `heatmap()` function but it does rather more than we need, so we will use the simpler `image()` function:

```
image(1:21,1:5,zdata)
```



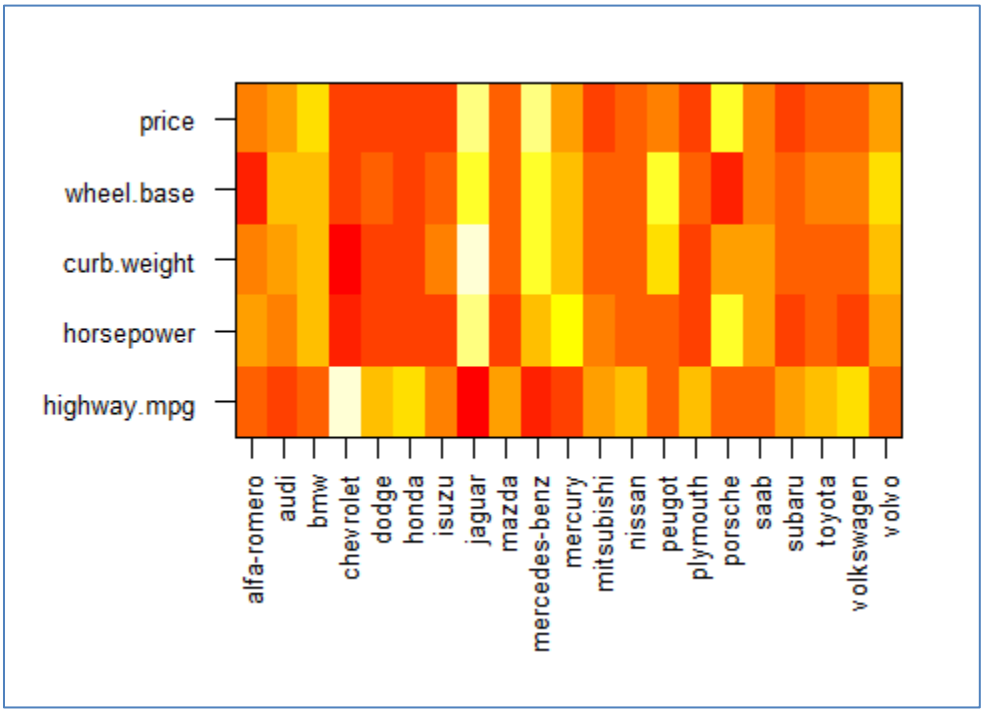
You can see the basic idea already in this quick and dirty heat map, but we need to make

a few improvements. We will suppress the numerical x and y axis tick marks with `xaxt="n",yaxt="n"`, and add new axes with text labels by using the `axis()` function. This will necessitate setting wider margins on the left and bottom sides of the plot, using the `par()` function to set the `mar` parameter. Finally, we'll draw a subtle box around the plot area with `box()`.

```
par(mar=c(7,6,2,2))  # make space in margins for axis labels

image(1:21,1:5,zdata,
      xlab="",ylab="",
      xaxt="n",yaxt="n")

axis(1,at=1:21,labels=makes[,1],las=2,cex.axis=0.8)
axis(2,at=1:5,labels=variables,las=2,cex.axis=0.8)
box()
```



The resulting plot is much easier to read than the 3D surface plot we tried earlier. However, it's still a bit unclear: which colors represent higher values and which represent lower values? (Answer: red is lowest, white is highest) We can specify alternate color schemes for the plot by specifying a `col` parameter. Try `col=rainbow(256)`, `col=terrain.colors(256)`, or `col=cm.colors(256)` and see if you like the results.

Personally, we don't. Fortunately, there exist third-party packages for R that can help you generate your own color schemes. One is `gplots`, with the `colorpanel()` function.

```
install.packages("gplots",dependencies=TRUE) # if you haven't already
library(gplots)
```

This code creates a new palette consisting of 256 shades between white (for the lowest values) and red (for the highest):

```
myshades <- colorpanel(256,"white","red")
```

Now plug it in to the `col` parameter of your `image()` function:

```
image(1:21,1:5,zdata,
      col=myshades,
      xlab="",ylab="",
      xaxt="n",yaxt="n")
```

And the result is easier to read, even if you are printing it on a black-and-white printer: