Lesson 03 - Basic Plotting

Jun 27, 2016

Background

Staring with the basics: a plot is a **graphical representation** of data. A plot is usually used to convey some sort of message about the data that cannot be easily understood by looking at the data in its native (usually tabular) format.

Plots can be used for **data exploration**, as well as for formal presentation or publication. In this lesson, I will mostly focus on using plots for data exploration. The purpose of the plot, in this scenario, is to give you insight to be able to answer a question you have about data you have collected or curated. This type of plot should show the data with all its messiness and baggage, so to speak.

When you are generating a plot for publication, however, you probably want a final product that is both honest and direct about the message it sends. This can take more time, and is beyond the scope of this lesson. R plots, with customization, make very good plots for publication, so it is worth taking time to learn both basic and advanced R plotting.

There is not just one *right way* to plot a data set. Therefore, keep in mind that your choice of plot can **clarify** or **confuse** the intended viewer, and you might want to rethink how you choose to plot something after you plot it.

To start things off, we need some *clean* data available to plot. Most likely, you will be interested in plotting data that you or your collaborators have generated. However, we will be plotting two other, convenient, types of clean data in this lesson:

1. Open Data (available in R, or online).
2. Random Data (generated in R).[1]

Why random data? Plotting and understanding random distributions can help you to understand your own data (think null hypothesis vs. alternative hypotheses, or null distribution vs. true distribution). We will revisit this concept in more depth during our lesson on data modeling (Lesson 05).

Plot Types

There are many plot types available in base R. We will go through some of the following basic plots, and hopefully get a sense of their strengths and weaknesses for different applications.

1. Histogram
2. Density plot
3. Scatterplot
4. Line graph
5. Boxplot
6. Barplot
7. Correlation matrix
8. Pairs plot

Example Data: airports.csv

```
> download.file("https://dl.dropboxusercontent.com/u/6965753/airport.dat
> dat <- read.csv("airports.csv")
> names(dat)
```

The description of this data set is available here: OpenFlights Airport Database

Before we plot the data, we can ask some basic questions about the data. See if you can understand and implement these types of queries.

How many countries have airports?

```
> length(unique(dat$country))
```

How many airports are in the USA? I will use the `subset()` function, which was introduced last week.

```
> dat.sub <- subset(dat, country=="United States")
> length(unique(dat.sub$name))
```

How many cities in the USA have airports?

```
> length(unique(dat.sub$city))
```

Can you come up with other questions to ask about the data?[2]

Histogram

You encountered the **histogram** plot function `hist()` last week. Let's try it out on this data set.

Let's suppose that we wanted to see the distribution of airports by longitude, binned into 20 bins (breaks).

```
> hist(dat$latitude, breaks=20)
```

Let's try the same thing, but this time look at longitude.

```
> hist(dat$longitude, breaks=20)
```

We can make a finer-graned plot by increasing the number of breaks.

```
> hist(dat$longitude, breaks=100)
> hist(dat$longitude, breaks=10^3)
> hist(dat$longitude, breaks=10^5)
```

- *Which plot did you like best?*
- *Was there a noticeable difference in computation time between the different plots?*

Density Plot

Based on the plots from our histogram, it looks like we can include too few breaks, or too many breaks. One way to get a clear image that conveys a similar message is to plot something known as a **density plot**. This shows a smoothed distribution of the data histogram.[3]

```
> plot(density(dat$longitude))
```

The x-axis is basically the same as in the histogram. Below the x-axis you see the count of the datapoints and the estimate for something called 'bandwidth', which we won't worry about right now (you can learn more about it here).

Instead of frequency (or counts) on the y-axis, which is what we saw in the histogram, we now have a "density."

How might you change the x and y axis labels? If you don't know how, what is a good way to find out?

Scatterplot

Since we know that latitude and longitude have a real, physical/geographical meaning, we can try plotting them against each other in what is known as a **scatterplot**.

We don't have to worry about plotting the continents behind the points right now (though we can try that after the advanced plotting lecture). The data is rich enough that we can get a good sense of geography from the points themselves.

```
> plot(latitude~longitude, data=dat)
```

What is R doing here?

It is basically taking one vector (a column from your data frame) and plotting it against another vector (another column from your data frame). The vectors have to be the same length (which is fine, because all data frames are rectangular), and in the same order (the 10th value of the x vector corresponds to the 10th value of the y vector). If there is a missing (`NA`) value in one of the columns, R will not plot a point for that 'row' of data. So, there should be the same number of points in a scatterplot as there are full, paired x and y coordinates.

You can think of the `~` symbol as indicating regression of one variable on another, or as indicating "y vs. x", in that order.

We can change the format of the plot. Perhaps we want to shrink the points and fill them in so that they are not empty circles. We can use something called `pch` to set the *p* lot *ch* aracter or symbol (I am not sure if that what PCH means, but it is easy to remember).

```
> plot(latitude~longitude, data=dat, pch=16)
```

We can shrink the points using `cex`, which indicates the scaling factor (defaults to 1).

```
> plot(latitude~longitude, data=dat, pch=16, cex=0.5)
```

We still have overlapping points. Let's make a new color, a semi-transparent red. We'll use `rgb` to generate our colors this time.

```
> new.red <- rgb(red=1, blue=0, green=0, maxColorValue=1, alpha=0.2)
> plot(latitude~longitude, data=dat, pch=16, cex=0.5, col=new.red)
```

This gives us a sense of the density of airports in different locations, even when the points are overlapping.

We can highlight a specific set of airports. Perhaps we want to see where the airport in "Bad Gastein" is. We can use a function called `points` to add to the existing plot.

```
> dat.bad <- subset(dat, city=="Bad Gastein")
> points(latitude~longitude, data=dat.bad, pch=8, col="black", cex=2)
```

You can only call `points()` after some sort of plot has been initiated with `plot()`,

`plot.new()` or some other plot function. This is not true for all plot functions– some of them take a parameter called `add`, where you can specify `add=TRUE` or `add=FALSE`.

What country is the "Bad Gastein" airport in? What about the airport in "Chinchilla"?

Let's highlight the airports in the US.

```
> dat.us <- subset(dat, country=="United States")
> plot(latitude~longitude, data=dat, pch=16, cex=0.5, col=new.red)
> points(latitude~longitude, data=dat.us, pch=16, col="black")
```

Does anything look strange to you? Let's explore the data a bit:

```
> dat.us[dat.us$longitude>-50,]
```

Some of these airports are annotated as being in the "United States", but their `timezone.olson` value specifies Africa, Asia, or Europe! So, we see that data exploration through plotting can reveal inconsistencies, errors, or outliers in the data. Good thing we didn't publish this plot! Perhaps OpenFlights will clarify/correct these rows in a future update.

Line graph

We can easily change the plot type in the plot function by specifying `type="l"`, for a **line graph**:

```
> plot(latitude~longitude, data=dat, type="l")
```

Previously when we used `plot()`, R defaulted to `type="p"`, for points. As you might have guessed, there is also a `lines()` function, and like `points()`, it can only be called after a new plot has been made. In this case, a line graph is not a great way to represent our data, since the lines are connecting points that are in no particular order of interest. Longitudinal data (where the y-axis is a phenotype, and x-axis is time) is better viewed as a line graph. *Can you find a data set included with R that would be well suited for visualization using a line graph?*

Boxplot

A **boxplot** gives a useful summary of the distribution of numeric data across discrete or categorical factors. For this type of plot, we will use one of the more exciting open data sets available, "The Number of Breaks in Yarn during Weaving" data set.

```
> data(warpbreaks)
```

Let's try some of our tricks:

```
> head(warpbreaks)
> str(warpbreaks)
> levels(warpbreaks$wool)
> levels(warpbreaks$tension)
> dim(warpbreaks)
> nrow(warpbreaks)
> ncol(warpbreaks)
> mean(warpbreaks$breaks)
> sd(warpbreaks$breaks)
```

Now, let's just ask R to plot one variable vs. the other, using this construction:

```
> plot(breaks~wool, data=warpbreaks)
```

R is defaulting to `type="b"`, for boxplot. *What do the boxes and whiskers indicate?* We can also do the same for tension:

```
> plot(breaks~tension, data=warpbreaks)
```

If we want to see how the **interaction** between wool and tension affect the distribution of the number of breaks, we could try the following:

```
> plot(breaks~wool*tension, data=warpbreaks)
```

However, R just plots the values against each of the categories in subsequent plots. If you want them in the same plot, just try:

```
> boxplot(breaks~wool*tension, data=warpbreaks)
```

There is an R package called `beeswarm` which plots *jittered* points to illustrate the distribution of data in each category in a different way.

Can you install beeswarm and use `add=TRUE` *to overlay a boxplot and a beeswarm plot?*

Barplot

Let's go back to the airport dataset. What if we wanted to know how many airports are in each country? We can use a table, or a **barplot**.

We can ouptut this summary in a table:

```
> table(dat$country)
```

The `summary` function will also do this, and rank order the countries from high to low count. However, it will not show all the results.

```
> summary(dat$country)
```

Let's try to use the function `barplot`.

```
> barplot(dat$country)
```

That didn't work! What's wrong with our command? Well, we have a summary table of the data, which compiles the counts by country. Let's try to plot *that* instead.

```
> barplot(table(dat$country))
```

OK. That worked, but now we only have a few of the country labels. We can use something called `las` to specify the direction of the x and y axis labels.

```
> barplot(table(dat$country), las=2)
```

That looks better. (*What happens when you use* `las=0, las=1`, `las=2`, `las=3`, `las=4` *?*)

We still can't read the names. We can do two things to fix this.

1. We can shrink the text using `cex`, or in this case, `cex.names`:
2. Or we can save the image as a really wide pdf, so that there is space to put the labels in without any overlap.

```
> barplot(table(dat$country), las=2, cex=0.4)
> pdf("barplot-of-airport-data.pdf", width=20, height=5)
> barplot(table(dat$country), las=2)
```

```
> dev.off()
```

The function `pdf()` opens a pdf graphical *dev_ice, and* `dev.off()` *shuts off the _dev_ice. The pdf won't be readable until you complete those two steps, sandwiching the plots that you want to output to pdf in between the two device commands. We can then open the pdf file in the location it is saved, which is your working directory. _Do you know where that is?*

```
> getwd()
```

Did your labels fit? How might you change the argument values in the `pdf()` *function to make sure they fit?* The `mar` argument in the `par()` function specifies default margin values.

```
> par("mar")
```

You can think of these as going clockwise from the bottom (bottom, left, top, right), specified in line height units. Since we want to increase the bottom margin, we can specify `mar=c(9.1, 4.1, 4.1, 2.1)`

```
> pdf("barplot-of-airport-data.pdf", width=30, height=5)
> par(mar=c(15.1, 4.1, 4.1, 2.1))
> barplot(table(dat$country), las=2)
> dev.off()
```

This will overwrite the old pdf. Read more about `par`, `mar` and `las` here.

Correlation matrix

Let's generate some random data! We will eventually use this to plot a **correlation matrix**, since two independent random data sets should not, in general, be correlated.

```
> set.seed(1999)
> random.x <- rnorm(n=1000, mean=5, sd=1)
> random.y <- rnorm(n=1000, mean=5, sd=1)
> plot(random.y~random.x)
```

These points are apparently not correlated with eachother.

What happens if we throw an extra set.seed in, before generating `random.z`, using the same seed?

```
> set.seed(1999)
> random.z <- rnorm(n=1000, mean=5, sd=1)
> plot(random.z~random.x)
```

These points are perfectly correlated (they are not sampled independently – they are 'sampled' identically and both dependent on the same seed). This means that `random.z==random.x`.

Let's bind these up into a data frame, and examine the correlation. In order to plot this, we will briefly foray into the world beyond base R plotting.

```
> random.dat <- cbind(random.x, random.y, random.z)
> install.packages(c("corrplot","corrgram"))
> library(corrplot); library(corrgram)
> cor(random.dat)
> corrplot(cor(random.dat)) # this looks like a domino
> corrgram(random.dat)
```

This is one example of how different packages can do essentially the same thing. This data is not too interesting. Let's find something that has a more natural correlation structure. I chose this data set (called `swiss`) because it has multiple continuous variables:

```
> data(swiss)
> corrgram(swiss)
> corrplot(cor(swiss))
```

Remember, this sort of display looks for correlations, not causation. *Do you have a preference for corrgram or corrplot, and why?* We might want to look a little closer at the actual data points.

Pairs Plot

Thankfully, in base R, there is a great function to look at all pairwise scatter plots in your data set.

```
> pairs(swiss)
```

What do you see? How does the information in this visualization differ from what you saw in the correlation plots?

In Class Exercise

Please assemble yourselves into groups of about 2-3 people.

Step 0. Choose a team name! (This is the most important step. OK, don't waste too much time on this, but make it good – no pressure!)

Step 1. Scan the list of available datasets in R using the following command:

```
> library(help="datasets")
```

Step 2. Choose one of the data sets that interests you (or the one which is the least uninteresting to you), and read it into R using the `data()` function.

Here are some suggested data sets for:

- *Histogram, Density, and/or Scatterplot:* iris, USArrests, mtcars, pressure, quakes, randu, rock, stackloss, trees, women
- *Correlation Plot:* USArrests, mtcars, quakes, randu, rock, trees

Step 3. Find the column names (and row names, if applicable) of the data set you chose.

- In the last class, we learned to use `head()`, `str()`, `levels()`, `dim()`, `nrow()`, `ncol()`, `mean()`, and other functions. Can you apply these to your data set (data frame)?

Step 4. Decide on two columns that you would like to plot against each other (y vs. x) using a `line` or `points` (scatter) plot. If you are using a histogram or density plot, decide on one column to visualize.

- What question are you asking of the data at this point?
- What do you expect to see?

Step 5. Plot the data.

Step 6. Consider the following:

- Was the plot informative?
- Did the plot help to answer the question you had about the data?
- How did the plot differ from what you expected?
- How might you improve the plot (layout, color, text size, format, etc.) so that it is more

publication-worthy?
- What other questions might you ask of the data, and what plots would you use?

Useful Functions

Plot and Graphical Functions

```
* plot()
* par()
* lines()
* points()
* text()
* title()
* mtext()
* axis()
* colorRamp()
* rgb()
* expression()
* layout()
```

Saving / exporting plots

```
* jpeg()
* pdf()
* png()
* dev.copy()
* devSave()
* quartz()
* dev.off()
```

Lattice Plots

For multivariate data, you can use something called the `lattice` package. First, `install.packages("lattice")`, then try out the following plots:

```
* xyplot()
* bwplot()
* histogram()
```

```
* stripplot()
* dotplot()
* splom()
* levelplot()
* contourplot()
```

Homework

Repeat the in class exercise, using the same data set that you chose in class.

This time, visualize the data using two different types of plots, or two different versions of the same plot to answer a single question you have about the data.

What are the benefits of each choice? What would the ideal plot look like?

Extra: What sort of hypothesis might you test, and how would you determine statistical significance of the result (thinking ahead to the lesson on data analysis in R)?

Key Points

- There may be many ways to plot the same data, each method leading to a different insight.
- There may be many different data sets that look the same, if you use a method of data visualization that does not discriminate between them. This can lead to false conclusions.

References

Parts of this lesson were sourced from this site (a quick read!): R Base Graphics: An Idiot's Guide. This adds to what we have covered in class, playing around with coloring and labeling, etc.

Sources of Open Data

- Public Dataset List on GitHub
- OpenFlights Airport Database

Other learning resources for base R plotting

- Clean Graphs in R

- Quick-R: Basic Graphs
- Introduction to R: Base Graphics (ramnathv)
- Graphics in the R Language
- On Boxplots
- BoxPlotR
- Revolution Analytics: graphics
- R Graphics

Books

- Graphical Data Analysis with R (2015), Unwin
- R Graphics, 2nd edition (2011), Murrell
- Statistics at Square One, 9th edition (1997), Swinscow: See Chapter 1 on "Data display and summary"

1. This data is *pseudo* random, because, unlike truly 'random' data, the sequences generated from these functions are replicable, and exhibit properties that are close enough to random, while enabling reproducibility when given a **seed** value. ↩

2. There is the option, strangely enough, to simply call the `plot()` function on a dataset, without specifying any options. In this case, R will choose what kind of plot to use on your behalf. I can't think of any scenario in which I would do this myself, but it can be done! ↩

3. The `density()` function uses a kernel density estimate. It involves a Fourier transform and smoothed linear approximations (not important to know for this lesson). It works well for many types of data, but you will sometimes want to compare it with a histogram. Sometimes you will want to set the bandwidth manually. ↩