Producing Pretty Plots in Python

Geraint Ian Palmer

Hi. I'm Geraint Palmer, a PhD student in Operational Research at Cardiff University. My work involves looking at, understanding and analysing data: whether that's real world healthcare data, or data generated by mathematical models. An important part of this analysis is visualising the distributions and relationships in the data. Plots are everywhere, in research journals, mainstream media (newspapers, television news), management board meetings. It's because they are a quick and natural way for our brains to comprehend numerical data.

There's a famous set of data, called Anscombe's quartet (pictured here) that emphasises the importance of visualising data. These four sets of data have close to exactly the same descriptive statistics. Mean x and y values, variance in x and y, correlation between x and y, and line of best fit are all the same. But we can see once we visualise that the four data sets, and the four relationships between x and y are very different. Therefore statistics alone might not be enough to tell the full story of data.

Before we talk about visualising data, we need to understand data. There are four main types: Nominal, Ordinal, Quantitative, and Relational.

- Nominal, or categorical data is discrete, and values are mutually exclusive. That is there are not overlaps, and no numerical relevance to the values. For example what is your favourite book?
- In ordinal data, the order of the values are important. However distances between values are unquantifiable. For example, a questionnaire might ask "How happy are you with a product?", "Very happy", "Happy", "Neutral", "Unhappy". There is a clear order, but we cannot measure the distance between Happy and Neutral, and that 'difference' might not be the same difference as between Very Happy and Happy.
- Quantitative data is numeric, and has both order and distances. There are two types, "Interval" data has no absolute zero, and "Ratio" data has an absolute zero. For example dates would be interval data, but height would be ratio data.
- Finally Relational data describes the relationships between discrete objects.

We should understand the types of data we are dealing with before planning a data visualisation. E.g Ratio data should always show the absolute zero on an axis; some chart types are meaningless with nominal data, etc.

As we're planning out data visualisation we need to think about what elements of the chart are showing our data points, and exactly what is differentiating between the values of the data points.

Now this slide is reproduced from a talk given by Professor Kennedy at a data viz workshop, and summarises a lot of research into cognitive and perceptual psychology, and how the brain processes visual information. There are six major ingredients that we use in plots to represent variables: position, length, angles, area, volume, and colour. They are ranked here by how accurately our brain can perceive differences in their values: Humans can pick up subtle differences in position and length much better that differences in volume and colour.

We need to be aware of these things as we plan out a plot, but that doesn't necessarily mean that for every new variable you add to a plot you work your way down this list. Angle and volume for example are very

difficult to incorporate into plots, and sometimes one or more of these elements are inappropriate for the data and design you are working with. But it's something to keep in mind.

Now let's get to actually making a plot. In Python we have the incredibly powerful Matplotlib library for producing plots. Matplotlib is very flexible in allowing you to create the exact image you want, but for this you pay a small price, some might see it as fiddly, and and there are a multitude of features and methods and options. But for simplicity there is something called pyplot: "a collection of command style functions that make matplotlib work like MATLAB". What this means it that you can create full meaningful plots with one command.

Here you can see examples of pyplot's work. A histogram here is as simple as typing 'plt.hist' passing in a list or array of data to plot. These command can also take a wide range of keyword arguments to customise the plot.

Step back a minute, which type of plot should I be using? The top row here shows plots that can be used for one dimensional quantitative data: they illustrate distributions. A histogram shows frequencies of ranges of values; a boxplot summarises descriptive statistics in an image, and a violinplot combines the two. The bottom row shows two dimensional data, although they aren't all equivalent. A scatterplot plots quantitative data againast quantitative data. So does a line plot, however now we assume that x values have no more than one y value. A bar chart plots nominal or ordinal data against quantitative data.

Moving away from the simple interface of pyplot, matplotlib allows far more flexibility by manipulating figure and axis objects.

- The pyplot command subplots can create a figure and axis for us, and attach them together. A figure is your canvas, this is what you view and save. An axis is you plotting area, this is where you add data points and labels.
- To give an axis a ylabel we can use the axis method `set_ylabel`. This takes in the text of the label, and arguments such as fonsize. 'set_xlabel' is the x-axis equivalent.
- Similarly we can set a title with `set_title`
- Ticks are the specific locations of gridlines and the little numbers that help you navigate the grid. `ax.set_xticks` determines their location. `ax.set_xticklabels` determines what is shown on the ticks. Both take a list of numbers/strings, and of course we can also manipulate the y axis.
- If using colour we might require a continuous colorbar. Note that this is a method of the figure itself, not the axis. This takes in a mappable, a plot instance as its argument so that it will adjust its scale appropriately.
- Then there's the data points themselves. We attach these to the axis, but the commands are the same as the pyplot commands we saw before.
- An finally we can annotate the chart with `ax.text`, which takes in relative x and y positions, and the text string itself.

The pyplot command that made our figure and axis is more flexible that we just saw. It's designed for creating subplots, more that one axis per figure. This command makes a 2 by 2 array of plots, and thus returns an array of axes instead. Each individual axis can be customised as before. We access the axes by simply by indexing the array.

So let's walk through an example of create a plot with real data. I have a data set containing the Olympic medal winning times of the Women's 200m sprint from 1948 to 2008.

- I'll plot three line plots, one for the gold medal winning times by year, one for silver and one for bronze.
- Now we might want to choose more appropriate colours, ones that will help understanding of the graph, so we can add colour as a keyword of the 'plot' method. Notice however I didn't just choose gold and silver as the colours. Colour choice is important, I chose more subdued colours so that visibility wasn't compromised. We must also choose colours such that one plot doesn't dominate the others.
- Olympics are every four years, so it would make sense if our evenly spaced ticks reflected the points at which data points are collected. So we can set the ticks, and I've rotated the labels to tidy the plot up a bit.
- We can then add axis labels and a title, so that the plot can be read.
- I'm then going to add another type of plot tot he axis, a scatter plot indicating which medals went to American athletes. Notice some the keyword arguments here: marker shape, size, colour.
- If you notice now, the scatter points lie behind the line plots. This felt messy to me, so I reorder the plots with the 'zorder' keyword argument, giving the line plots a lower order in the z dimension that the scatter points.
- I'll add a legend so we know what each plot component represents.
- All plot components show up in the legend, but we haven't specified what they represent. We can do this by adding the 'label' keyword argument to the plot commands.

I've talked about one and two dimensional plots. What about three dimensional? I'm going to discuss heatmaps, which map numerical values in an x-y coordinate system to z-values represented by a colour. This is done through colormaps. Now a lot of what I'll speak about now is from this brilliant talk from a SciPy conference. It's about choosing appropriate colormaps.

As an example I have two bits of numerical data: a photograph of the Christuskirche in Windhoek, which is a matrix of numbers that we already know how we expect to look, and a 9 by 9 magic square, which we haven't got any preconceived ideas about its structure. We'll apply some popular colormaps onto these and observe the outcomes:

- First we have the most popular colormap around: jet. We see this helps us find some sort of structure in the magic square, but takes some effort to understand. It distorts the image of the church.
- The sequential colormaps 'hot' and 'cold' really distort the church, but perform better on the magic square. However vlues in the middle are difficult to differentiate.
- The less said about 'flag' the better. Although I can see it might have some use picking out very subtle differences of number.
- The monochrome sequential maps work well. Although these might have some problems when printing, or comparing screens with different settings.
- And these are diverging colormaps. You need to be careful with these, does red turn into blue, or blue turn into red? These are designed for datasets with a definite meaningful centre, for example zero.
- So a lot of colour theory and psychology research went into developing 'good' colormaps, and these four were created. They are called perceptually uniform, meaning they are sequencial, and accurately represent data. As a bonus they also work in black and white, and are fine for colourblind people. You should use these!

So let's build up a heatmap using matplotlib's pcolor command.

- Again make figure and axis objects, and plot a heatmap with pcolor. z here is an array or a list of lists containing the z-values.
- Using numpy's meshgrid, we can underlay X and Y values under the heatmap, ensuring accurate scales and ticks.

- And of course we must use a sensible perceptually uniform colormaps.
- Notice that pcolor places the ticks at the left hand extreme of the unit square. We'd like our tick to be in the middle of the unit square, as we won't get a skew when adding gridlines. So I've reset and relabelled the x and y ticks.
- We'll add a colorbar, and we can also give the colorbar a label. The colorbar command takes in the mappable 'hm', wich I've set to be the heatmap itself.
- Add in the x, y labels and the title.
- And finally add gridlines so that the reader doesn't get lost in the graph. Notice I've had the give the heatmap a zorder of 0. Zorder 0 is a special one, that means 'behind the grid', as ordinarily the grid lies behind the plot. But a heatmap saturates the canvas and so the grid would be unseen. And so we have a complete heatmap with matplotlib.

Thanks for listening, and thanks to the Phoenix project for allowing me to travel here, and the PyCon for letting me speak. Here's some libraries I used in this talk. The slides are available here: and there are some useful links and references here.
