

W4995 Applied Machine Learning

Visualization and Matplotlib

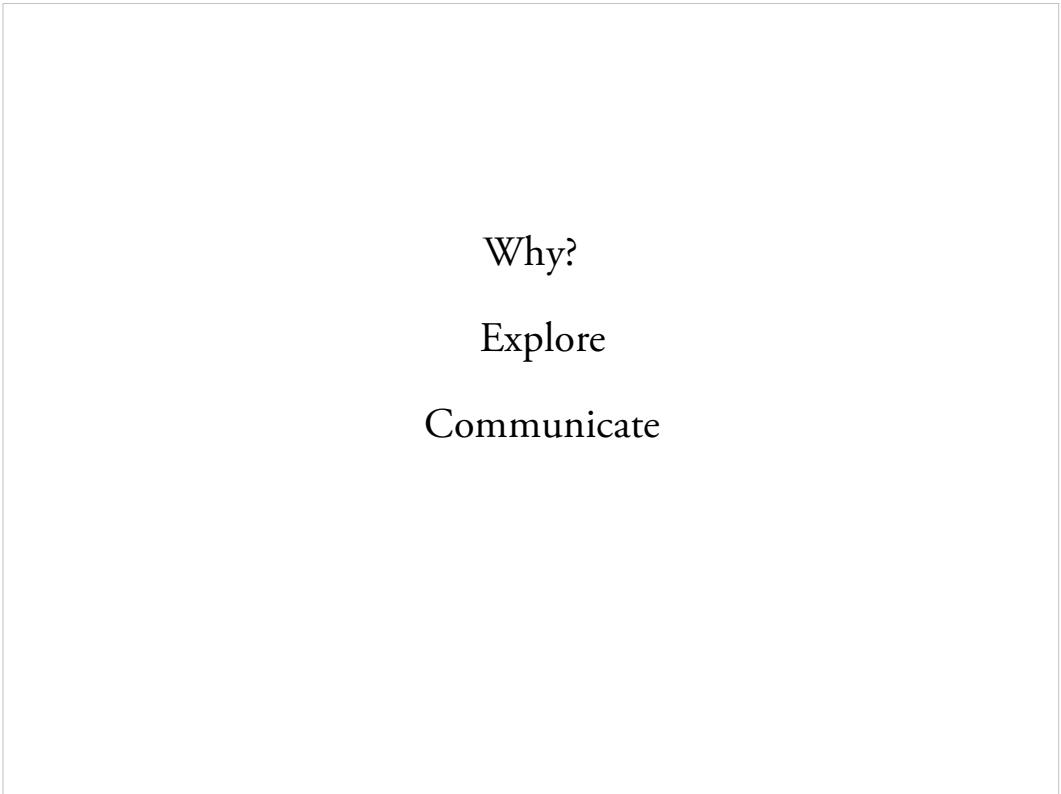
01/25/17

Andreas Müller

For the rest of today I want to talk about visualization and matplotlib. I thought a bit about it and I think numpy and pandas are better learned by doing, and there's some interesting bits about matplotlib and visualization that I think are worth going through. I was hoping to go into visualization techniques and practices a bit more, but unfortunately I don't think we have the time and I really want to get going with machine learning next week.

Principles of data visualization

I want to briefly talk about some very basics of data visualization. I'll post to great books as references if you're curious.



Why?

Explore

Communicate

So first, I want to ask why we might want to visualize data. And I think there are two main reasons.

The first is for ourselves, to explore the data, make hypothesis and find interesting trends.

The other is to communicate either the whole data, or particular aspects or findings about the data. I think both are important, but for the matter of this class, I think the exploration is more important.

In general, you should think about what the point of any visualization is, and whether the method you chose is the best for this purpose.

Above else, show the data.
Maximize the data-ink ratio.
- E. Tufte

Tools matter.
- W. S. Cleveland

So here are more quotes, from the two books I mentioned. The first two are from Edward Tufte, the godfather of statistical visualization. He says “above else, show the data”. So the data should be in focus, and not any fancy method you use to show it.

He also says “Maximize the data-ink ratio”. What he means here is that all the ink you’re using (even if it’s virtual ink) should show the data, and should vary with the data. You want fewer static elements in your graphic, and more elements that are actually important to show the data.

The other quote, from William Cleveland is: Tools matter. I think for him the tools are more the visualization methods, but I think also the software tools matter, which is why I want to talk about them.

Visual Channels

length (1D size)	- — —	colour hue	
angle	/ —	texture density	
curvature)))	texture pattern	
shape	+ • ■ ▲	position (2D)	
area (2D size)	• ■ ■ ■	depth (3D position)	
volume (3D size)	• ■ ■ ■	motion	
lightness black/white	■ ■ ■ ■	blur/sharpness	
colour saturation	■ ■ ■ ■	containment	
transparency	■ ■ ■ ■	connection	

Before we start with plotting, here's a quick summary of visual channels that I took from the thesis "Systematising glyph design for visualization" that I posted on courseworks.

These are ways in which you can show information.

There is obviously length, as in bar-charts, angle, curvature, shape as in graphs and scatter plots.

There's area, which is often useful, but it's important to say whether you're using area or distance to show information. For circles that's often unclear.

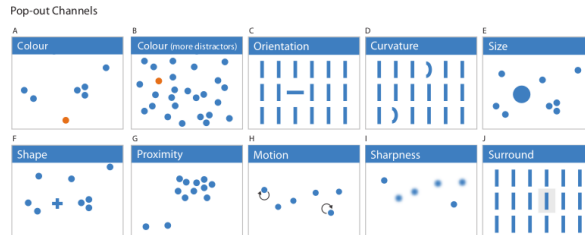
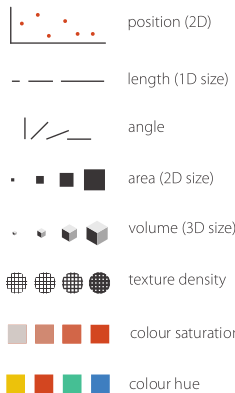
There is 3d volume, which I really try to avoid. Then there's color. There's lightness, saturation, transparency and hue, and they all work very differently. It's generally accepted that using hues for quantitative changes is not a great idea, and lightness or saturation works much better.

Textures are something I also try to avoid, position is clearly one of the most important channels, while the rest are not something I use very often - though containment and connection can be helpful.

Picking Channels

Quantitative validated

Cleveland and McGill, 1983
Heer and Bostock, 2010
MacKinley, 1986



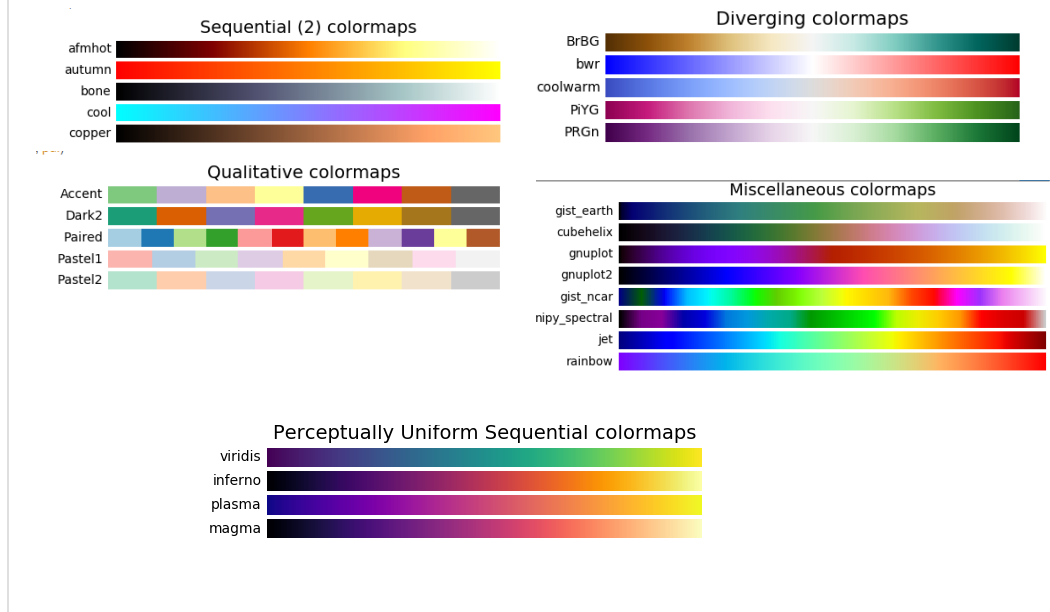
There have been studies of which of these are good ways to relate quantities. The winner is position, closely followed by length. These are clearly the most accurate ways to depict a value. Then angle, area, volume, texture, saturation and finally hue.

So hue is basically the worst way to encode any quantitative information. Unfortunately these studies didn't include brightness.

It's also good to be aware of which of these channels have pop-out effect, which allows you to find particular items very quickly. Here you can see color, orientation, curvature, size, symbol, movement, blur and contrast all working well - actually you can't see the movement, but I think you would agree.

It's important to know that the number of different values are important to how much something pops out, and if you use too many orientations or sizes or colors, this effect is lost.

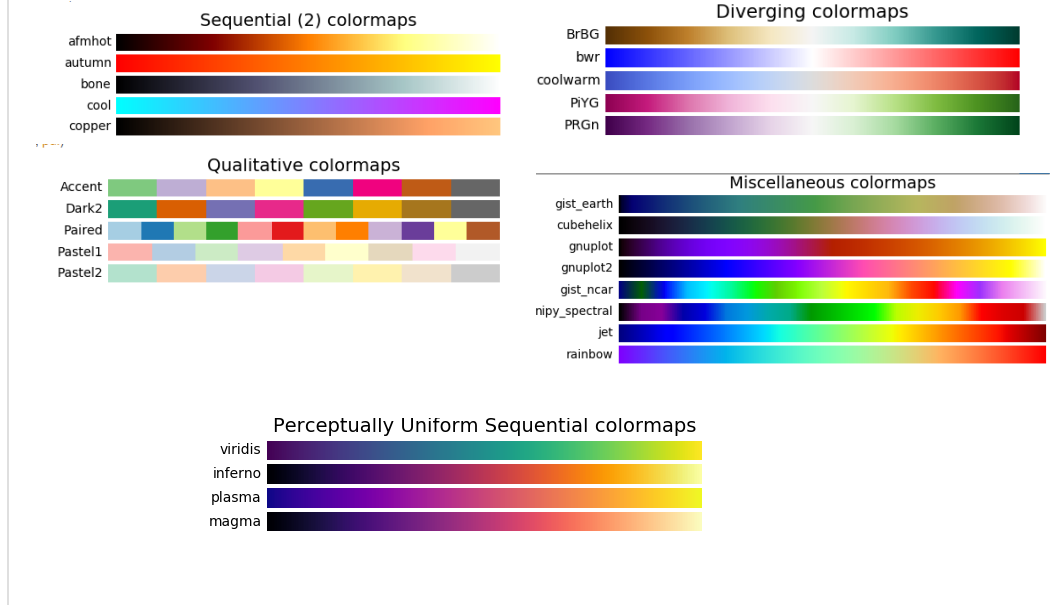
Colormaps



But let's talk more about color, in particular color maps. Color maps are ways to map quantities from a continuous number to a color, and they are used whenever you plot something with color - or even with gray scale.

There are several different types that are important to distinguish. Sequential colormaps usually go from one hue or saturation to another, while also changing lightness. They have two extremes and interpolate between them. That works well if you're particularly interested in the extreme values, but they might not offer a lot of contrast in the middle. There's also diverging colormaps, which have a focus point in the middle, these here have either grey or white, and then have different hues going in either direction. This is great if you have a neutral point and then deviations for that, like profits that can be zero or positive and negative, and you can easily see which side of the zero any point is on.

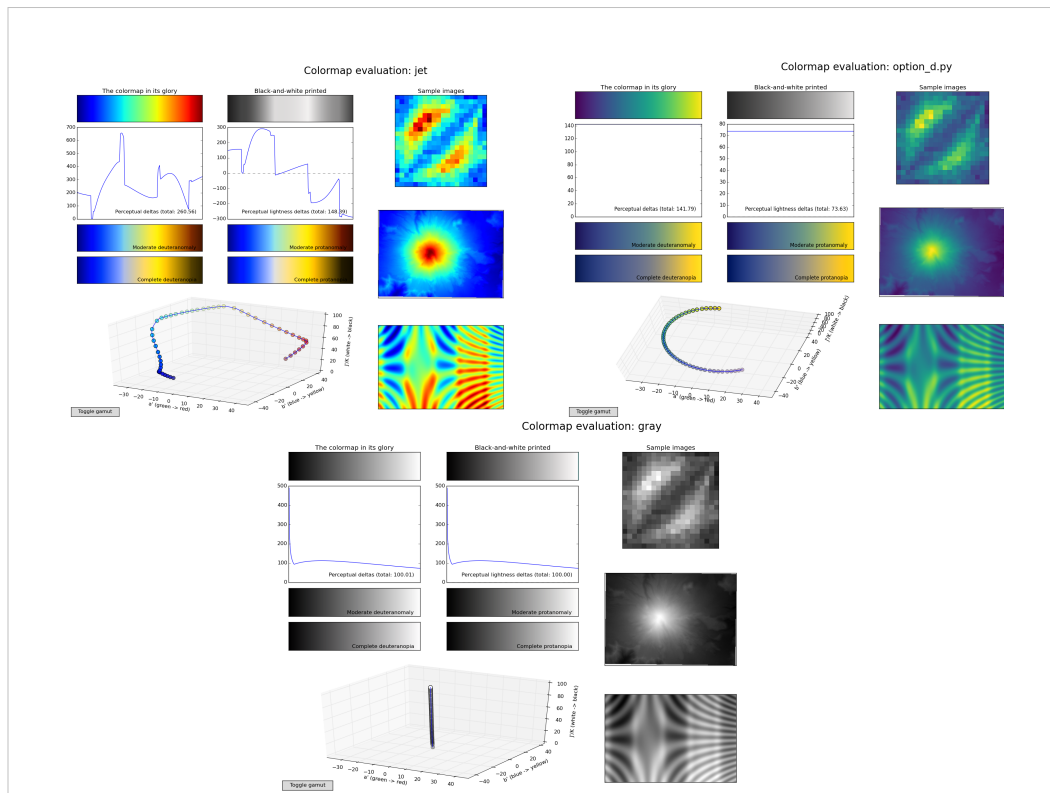
Colormaps



Quantitative colormaps are meant to not show a continuum, but just some discrete values, and they are designed in such a way to have optimum contrast for a particular number of discrete values.

Finally, there are the miscellaneous colormaps shown here, and you can see them really everywhere, in particular rainbow colormaps like jet and rainbow. Don't use them. Really, never ever use them. I'll explain why in a second, but if I see it in this course, not only will I take points of your homework, I'll also look at you really disappointedly.

And there are other colormaps that have more than two hues, but that don't have the problems that these very colorful ones have.

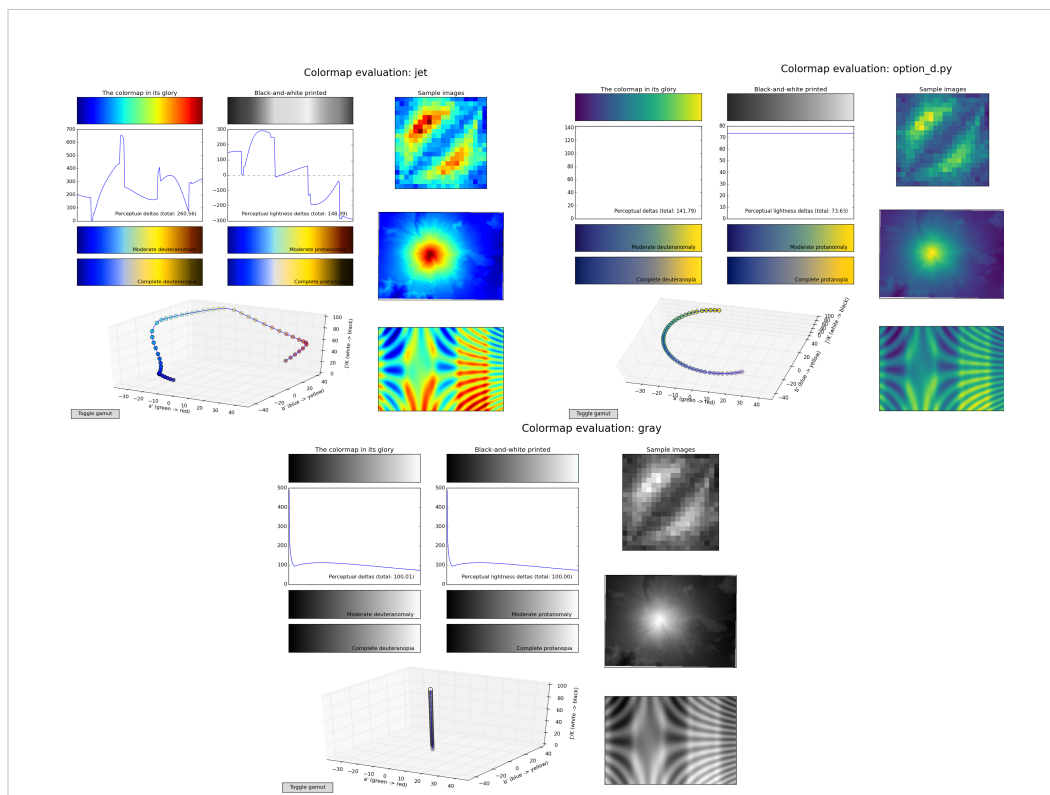


So what's the problem with these other colormaps.

This is an evaluation of the grey color map. Please focus on the parts to the right. There are three heat maps here. So each pixel here represents a floating point value on some scale.

Now let's look at this in the jet color map. you see that green ring that looks like it has some boundary? Or the red core? Where are these in the greyscale?

They are not there, because they are not in the data. So why does it look like there are edges? here is the color map, and this shows the differences in perceived color. You see these spikes? that's where we perceive edges, because the color map has edges! Also here is the colormap converted to greyscale, say if someone printed it. and below are the brightness deltas. Do you see something? It's not monotonous. It doesn't go from dark to bright, it backs up on itself....



And here's one of the perceptual uniform colormaps In comparison.

You can see that there's a bit more detail then in the gray one, but no artificial contours appear. That's because there are no perceptual edges. Also, the change in brightness is constant, so if you convert it to greyscale, you just got the greyscale colormap back.

The other plots here show how it looks for various forms of color-blindness, and the path of the colormap in a perceptual 3d color space.

I posted a cool video explaining way more details on this on courseworks.



matplotlib

Which brings us to matplotlib.

So matplotlib is the python plotting library on which basically all python plotting libraries build, and because it's so important, we're gonna go through it in some detail.

Matplotlib was designed as a general plotting library and not really with data analysis in mind, which is why it is a bit cumbersome in some cases, and not as elegant for data science as you might be used to coming from R.

But I think it's still good to know how to effectively create graphics in python if you're gonna work with python, which we will.

Matplotlib is very powerful, and any figure you can imagine, you can create with matplotlib.

Unfortunately sometimes it takes a bit of code, though.

matplotlib v2

update now!
(you can enable classic style if your really want)

Other libs

- pandas plotting - convenience
- seaborn - ready-made stats plots
- bokeh - alternative to matplotlib for in-browser
- several ggplot translations / interfaces

There are some other interface you should look into which I won't discuss in class.

Pandas has some built-in plotting on top of matplotlib. It's basically shortcuts to do some operations more easily on dataframes, and I highly encourage you to use it. You'll get the most out of it if you actually understand matplotlib, though.

Similarly seaborn has many tools for statistical analysis and visualization, also using pandas dataframes, also based on matplotlib.

There is a completely separate project, bokeh, which uses javascript (but not d3) to do visualizations in the browser. I'm not entirely convinced yet, though.

There's also several reimplementations and interfaces to ggplot, if that's what you're into.

Imports

```
from matplotlib.pyplot import *  
from pylab import *  
from numpy import *
```

```
import matplotlib.pyplot as plt  
import numpy as np
```

So before we start, a word on imports.

What do you think about these imports?

Why are they bad?

They hide where functionality came from. Is this squareroot from numpy or math or your own function? Is this plot from seaborn or pandas or matplotlib or something else?

Never use import *. It makes your code less readable. And we don't want that, right?

Always use explicit imports and standard naming conventions.

I haven't decided if I'll subtract points for that, but if I see that I will know that I failed.

matplotlib & Jupyter

`% matplotlib inline`

- sends png to browser
- no panning or zooming
- new figure for each cell
- no changes to previous figures

`% matplotlib notebook`

- interactive widget
- all figure features
- need to create figures explicitly
- ability to update figures

I highly recommend that you use jupyter to play around with matplotlib figures and visualizations.

To show matplotlib figures embedded in jupyter, you have to call one of two magics. Jupyter magics are those that start with a percentage.

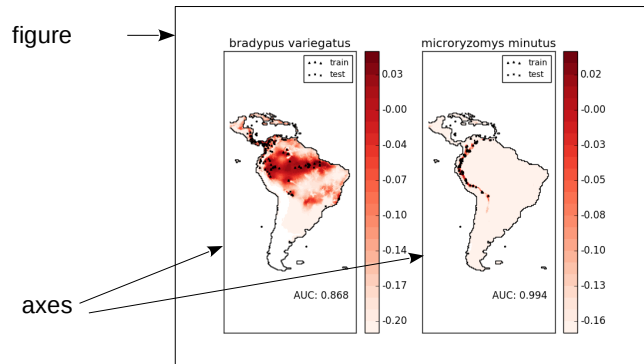
There's matplotlib inline and matplotlib notebook, and you should run one of them, and only one of them, at the very start of the notebook.

Both will allow you to show the figures inside the browser, but they are quite different. [read/explain table]

Figures and axes

figure = one window or one image file

axes = one drawing area with coordinate system



by default: each figure has one axis

Now let's start with the actual matplotlib.

The most basic concepts in matplotlib are Figures and axes. A figure is a single window or a single output file. A figure can contain multiple axes, which are basically subplots. Axes are the things you actually draw on, and usually each one has a single coordinate system.

By default, each figure has one axes.

So here is a figure with two axes from the scikit-learn examples.

Creating Figures and axes

1st way: don't.

Creates figure with axes on plot command

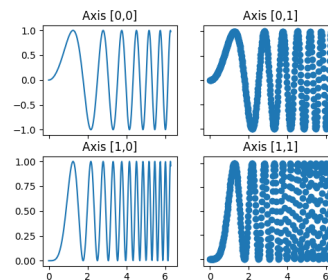
2nd way: **fig = plt.figure()**

Creates a figure with axes, sets current figure.

Can add more / different axes later.

3rd way: **fig, ax = plt.subplots(n, m)**

Creates a figure with a regular grid of $n \times m$ axes.



There's several ways to create figures and axes.

The first one is that you don't. With your first plotting command, matplotlib will automatically create a figure, and that figure will contain axes and that's what you'll draw on. Once you created a figure, each subsequent plot command will apply to the current axes, which is usually the last axes you created.

So if you want a second figure, you'll have to create it explicitly, for example using the `plt.figure` command. That creates the single default axes, but you can also add more later.

You can also create a figure with a grid of axes by using the `subplots` command (note the s in the end).

You can see how that looks on the right for a 2 by 2 grid. I use that a lot. `fig` is an object representing the figure and `ax` is a numpy array of size n time m , with each entry being one axes object.

More axes via subplots

ax = plt.subplot(n, m, i) # or `plt.subplot(nmi)`

places ax at position “i” in n x m grid (1-based index)

```
ax11 = plt.subplot(2, 2, 1)
```

```
ax21 = plt.subplot(2, 2, 2)
```

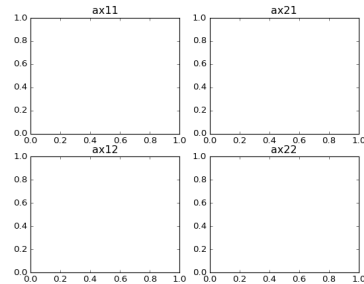
```
ax12 = plt.subplot(2, 2, 3)
```

```
ax22 = plt.subplot(2, 2, 4)
```

equivalent:

```
fig, axes = plt.subplots(2, 2)
```

```
ax11, ax21, ax12, ax22 = axes.ravel()
```



If you just created a figure but you want more axes, you can also add them one subplot at a time with the subplot command, this time without the s.

The subplot command takes three numbers - you can leave out the comma between them if they are single digits, but please don't.

The first two numbers specify a imaginary grid for the whole figure, let's say I want a 2 by 2 grid.

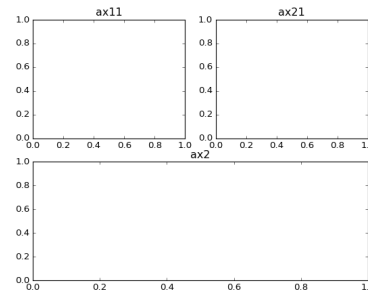
The third number is at which position in that grid I want my figure created. The numbers start with one and go column by column. I created the 2 by 2 grid here and labeled the axes according to the variable name.

you could create all of them at the same time with the subplot commands, but there are two reason why you might choose not to.

More axes via subplots

ax = plt.subplot(n, m, i) # or **plt.subplot(nmi)**
places ax at position “i” in n x m grid (1-based index)

```
ax11 = plt.subplot(2, 2, 1)
ax21 = plt.subplot(2, 2, 2)
ax2 = plt.subplot(2, 1, 2)
```



The first one is that the grid you specify doesn't need to be the actual grid. It just tells the single subplot about where it should be. So you can create all kinds of different layouts. For example, I can create a 2 by 2 plot where the second row is a single axes, by telling it it should be at the second position for a 2 by 1 plot.

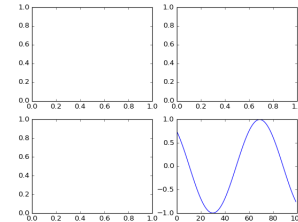
This allows me to create irregular grids, which is often quite useful.

Two interfaces

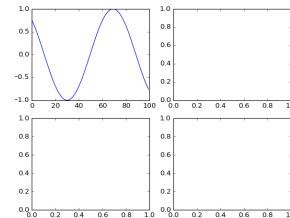
Stateful interface - applies to current figure and axes

object oriented interface - explicitly use object

```
sin = np.sin(np.linspace(-4, 4, 100))  
fig, axes = plt.subplots(2, 2)  
plt.plot(sin)
```



```
fig, axes = plt.subplots(2, 2)  
axes[0, 0].plot(sin)
```



So now is the part where things become a bit tricky. There's basically two different ways to use matplotlib, two interfaces if you want, one is the stateful one, and one is the object oriented interface.

If you're using the stateful interface, you just call the module-level plot functions like `plt.plot` here, and it will plot in the current axes. What do you think will happen here?

In the object based interface, you are explicit about which axes you want to draw into, so you hold onto your axes objects, and call the plotting commands on those! What do you think will happen here?

This is another reason why some people like to add subplots one by one: they use the stateful interface. They add a subplot, plot into it, then add the next one and so on. So the plot commands are exactly the same for both interfaces, but there are some differences.

Differences between interfaces

Stateful

```
plt.title  
plt.xlim, plt.ylim  
plt.xlabel, plt.ylabel  
plt.xticks, plt.yticks
```

Object oriented

```
ax.set_title  
ax.set_xlim, ax.set_ylim  
ax.set_xlabel, ax.set_ylabel  
ax.set_xticks, ax.set_yticks  
(& ax.set_xtick_labels)
```

```
ax = plt.gca()    # get current axes  
fig = plt.gcf()   # get current figure
```

The formatting options in the object oriented interface all have a “set_” in front of them, while they don’t in the stateful interface.

Also, for setting the tick marks, there are separate commands for the locations and the labels in the object oriented interface, but not in the stateful interface.

In general, the object oriented interface provides more functionality and is more explicit and powerful, but the stateful interface is a bit terser.

If you started plotting but then you want to use something that’s only part of the object oriented interface, you can always get the current axes or current figure with the gca or gcf commands.

I use the stateful interface if I have a single axes and I don’t need any of the advanced functionality of the object oriented interface, so basically for simple plots.

Whenever I have more than one axes objects, so for any grid, I always use the object oriented interface - but that’s just my personal preference.

Plotting commands

- Gallery:
<http://matplotlib.org/gallery.html>
- Plotting commands summary
http://matplotlib.org/api/pyplot_summary.html

I hope that sufficiently obfuscated everything.

So lets finally start with the plotting.

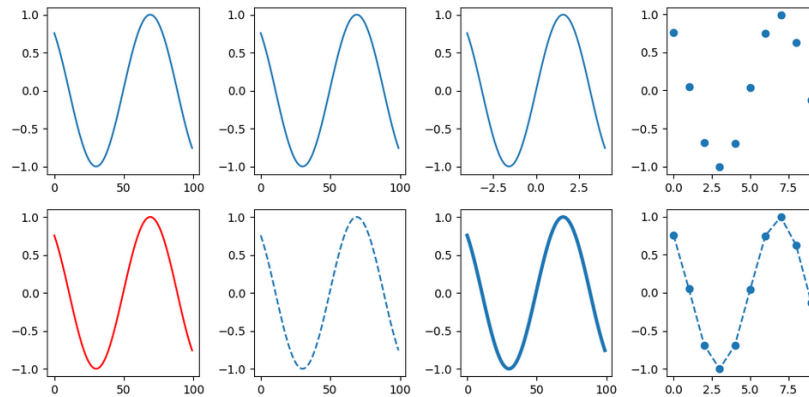
matplotlib has two pages that are helpful for figuring out how to plot something: the gallery and the plotting commands summary.

I will go through only some of the commands from the summary that I think are particular important and their most important aspects.

plot

```
fig, ax = plt.subplots(2, 4, figsize=(10, 5))  
ax[0, 0].plot(sin)  
ax[0, 1].plot(range(100), sin) # same as above  
ax[0, 2].plot(np.linspace(-4, 4, 100), sin)  
ax[0, 3].plot(sin[::10], 'o')  
ax[1, 0].plot(sin, c='r')  
ax[1, 1].plot(sin, '--')  
ax[1, 2].plot(sin, lw=3)  
ax[1, 3].plot(sin[::10], '--o')  
plt.tight_layout() # makes stuff fit - usually works
```

why?



Clearly plot is the most important one.

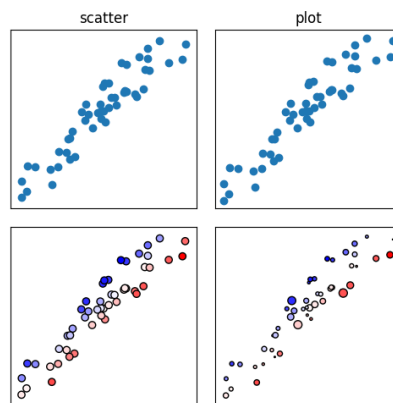
Plot allows you to do line plots and scatter plots.

If you give a single variable, it will plot it against its index, if you provide two, it will plot them against each other. By default, plot creates a line-plot, but you can also use “o” to create a scatterplot. You can change the appearance of the line in many ways, including width, color, dasheding and markers.

scatter

```
x = np.random.uniform(size=50)
y = x + np.random.normal(0, .1, size=50)

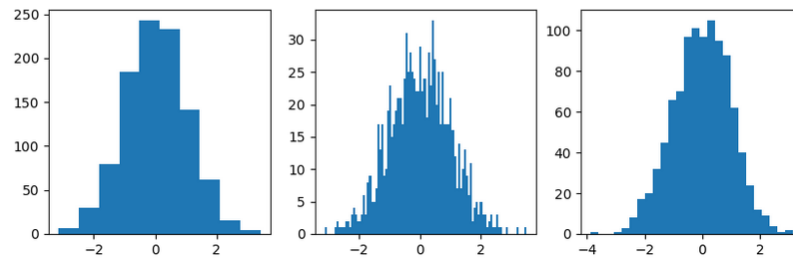
fig, ax = plt.subplots(2, 2, figsize=(5, 5),
                        subplot_kw={'xticks': (), 'yticks': ()})
ax[0, 0].scatter(x, y)
ax[0, 0].set title("scatter")
ax[0, 1].plot(x, y, 'o')
ax[0, 1].set title("plot")
ax[1, 0].scatter(x, y, c=x-y, cmap='bwr', edgecolor='k')
ax[1, 1].scatter(x, y, c=x-y, s=np.abs(np.random.normal(scale=20, size=50)), cmap='bwr', edgecolor='k')
plt.tight_layout()
```



While plot can create scatter plots, those are quite limited. the scatter function allows you to do scatter plots that not only encode the position of the points, but visualizes additional variables via color or size. In the bottom left, I colored the points by their distance to the diagonal, in the bottom right, I also added random size variations. Here I also used “subplot_kw” which allows you to specify some arguments for all subplots in a figure. I use it here to remove the ticks.

histogram

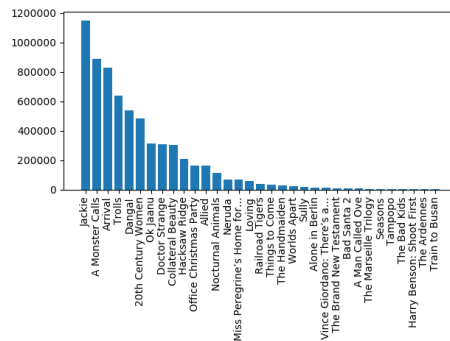
```
fig, ax = plt.subplots(1, 3, figsize=(10, 3))
ax[0].hist(np.random.normal(size=1000))
ax[1].hist(np.random.normal(size=1000), bins=100)
ax[2].hist(np.random.normal(size=1000), bins="auto")
```



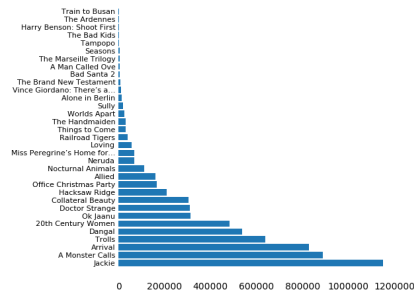
Histograms are clearly also important. By default, histograms have ten bins, which is never right. You can adjust the number of bins, and I usually try to find the point when it will be too fine. There's also a heuristic for finding the binsize which you can use with `bins="auto"`

bars

```
plt.figure()
plt.bar(range(len(gross)), gross)
plt.xticks(range(len(gross)), movie, rotation=90)
plt.tight_layout()
```



```
plt.figure()
plt.barh(range(len(gross)), gross)
plt.yticks(range(len(gross)), movie, fontsize=8)
ax = plt.gca()
ax.set_frame_on(False)
ax.tick_params(length=0)
plt.tight_layout()
```



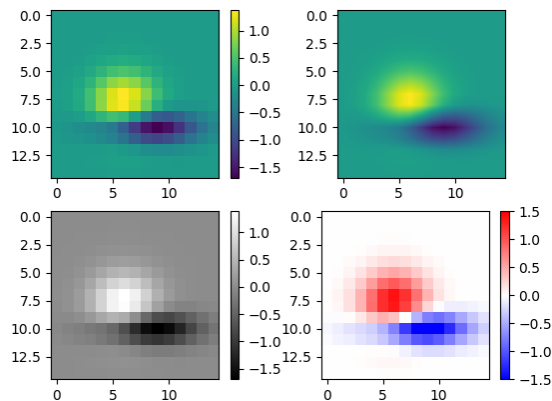
For barcharts, you always need to provide the position of the bar as well as the length. Usually that's done via a range.

If you use ticklabels, it's usually a good idea to rotate them so you can actually read them. But I don't really like tilting my head, so I often prefer horizontal bar-charts, which work the same way.

The way I specified the positions here, the first bar is at the bottom. We could flip the axes or change the position if we wanted it at the top.

heatmaps

```
fig, ax = plt.subplots(2, 2)
im1 = ax[0, 0].imshow(arr)
ax[0, 1].imshow(arr, interpolation='bilinear')
im3 = ax[1, 0].imshow(arr, cmap='gray')
im4 = ax[1, 1].imshow(arr, cmap='bwr', vmin=-1.5, vmax=1.5)
plt.colorbar(im1, ax=ax[0, 0])
plt.colorbar(im3, ax=ax[1, 0])
plt.colorbar(im4, ax=ax[1, 1])
```



You can plot heatmaps with the `imshow` command.

Previous to matplotlib v2 this automatically enabled interpolation, which you can see at the top right.

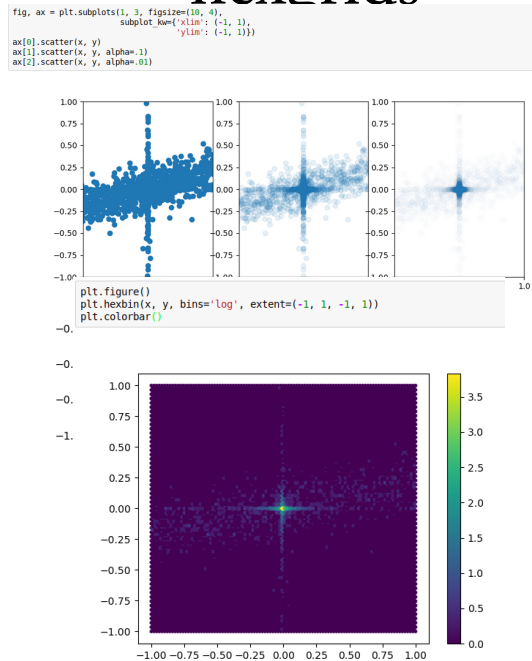
Interpolation might hide data or might give the impression of more data than there is, by showing a smooth transition.

You should always disable interpolation for heatmaps.

At the bottom you can see some results with a gray colormap and with a diverging color map. Here, the background is zero and it makes sense to represent the neutral differently. So I ensured that white is mapped to zero, and we can clearly distinguish positive from negative entries, which is much harder with the other color maps.

Doing colorbars on multiple axes can be tricky. You need to store the matplotlib image that is returned by `imshow` in an object, and then call the `colorbar` command with the image and the axes to which you want to attach the colorbar.

hexgrids



A command that I discovered way too late is the hexgrid. Hexgrids basically allow two-dimensional density maps.

If you have a lot of points, a scatterplot can become too crowded to understand what's going on.

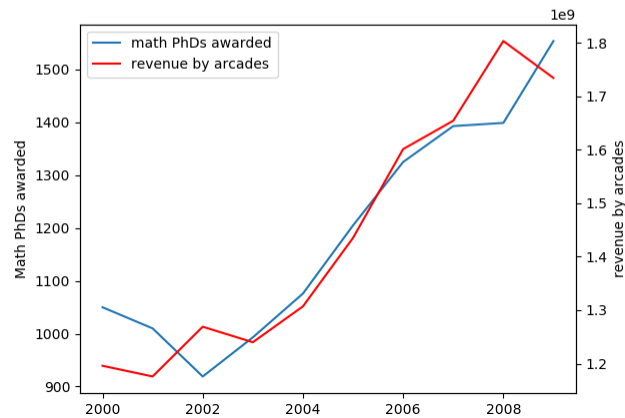
You can work around that a bit by using the alpha value, but that often throws away a lot of information.

A better way is to use a hexgrid and plot the density in each grid cell directly. That also allows the use of arbitrary colormaps.

Using hexgrids also allows you to map the density, for example using a logarithm, if the differences in density are very large between different regions.

Twin x / twiny

```
plt.figure()
ax1 = plt.gca()
line1, = ax1.plot(years, phds)
ax2 = ax1.twinx()
line2, = ax2.plot(years, revenue, c='r')
plt.legend((line1, line2), ("math PhDs awarded", "revenue by arcades"))
ax1.set_ylabel("Math PhDs awarded")
ax2.set_ylabel("revenue by arcades")
```



The last thing I want to mention is twin axes.

Here I show two time series, the number of math PhDs awarded in the us and the revenue made by arcades in the US.

If I plot them both in the same coordinate system, the number of PhDs will look just flat, because it lives on a completely different scale.

Using the object oriented interface, I can create a twin x axis for the revenue to show both time series with their own scales.