Week 3

Data Science Laboratory 1

Dr John Evans
`j.evans8@herts.ac.uk`

University of Hertfordshire UH

Plan for today

Good Practice

Basic Techniques with matplotlib

- Basic plotting
- Histograms
- Subplots

Visualisation

- Visualisation is at the heart of data science – it allows us to communicate efficiently with others, be it a simple description of a single dataset, showing the relationship between different datasets, or revealing trends and correlations.
- It can enable us to identify outliers and generate ideas for models.

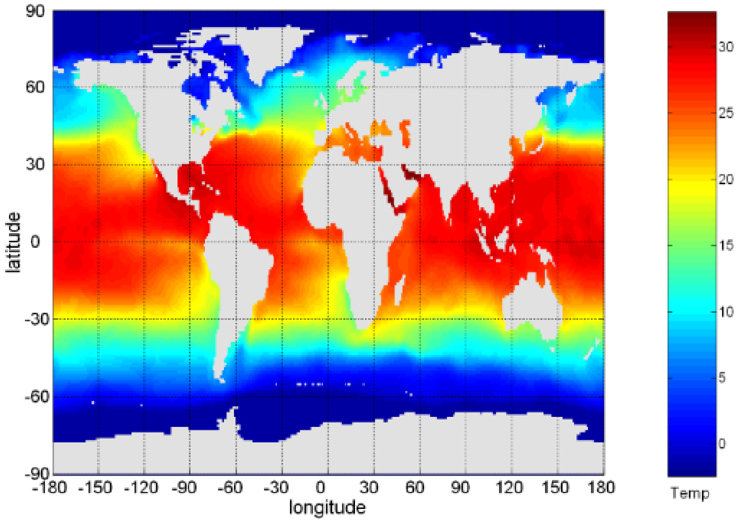
Visualisation

- Visualisation is at the heart of data science – it allows us to communicate efficiently with others, be it a simple description of a single dataset, showing the relationship between different datasets, or revealing trends and correlations.
- It can enable us to identify outliers and generate ideas for models.
- At its best, data visualisation is beautiful (`https://flowingdata.com`).
- At its worst, it is simply appalling (`https://viz.wtf`).

Why visualisation?

- The main motivation for using visualisation is so that people can quickly absorb large amounts of visual information and find patterns in it. For example, in the Figure on the next slide we have the Sea Surface Temperature (SST) in degrees Celsius for July, 1982.
- This figure quickly summarises the information from approximately 250,000 numbers and can be easily interpreted in a matter of seconds.
- For example, it is easy to see that the ocean temperature is highest at the equator and lowest at the poles.
- This demonstrates a key guiding principle in visualisations: **simplicity is king**.

SST Figure



90
60
30
0
-30
-60
-90
latitude
-180
-150
-120
-90
-60
-30
0
30
60
90
120
150
180
longitude
30
25
20
15
10
5
0
Temp

Good practice

No matter what tool we use, there are some general principles of good practice when it comes to data vizualisation:

Good practice

No matter what tool we use, there are some general principles of good practice when it comes to data vizualisation:

1. Less is more: a ‘busy’ or crowded visualisation is hard to read. Think of the main message we want our graphics to convey, and then use as few components as possible to achieve that.

Good practice

No matter what tool we use, there are some general principles of good practice when it comes to data vizualisation:

1. Less is more: a ‘busy’ or crowded visualisation is hard to read. Think of the main message we want our graphics to convey, and then use as few components as possible to achieve that.
2. Be consistent: our plots and figures should conform to a single style. This can be our own design, or following the convention or style-guide of a particular project or organisation. A good example can be seen in the look-and-feel of graphical data presented in *The Economist*.

Good practice

No matter what tool we use, there are some general principles of good practice when it comes to data vizualisation:

1. Less is more: a ‘busy’ or crowded visualisation is hard to read. Think of the main message we want our graphics to convey, and then use as few components as possible to achieve that.
2. Be consistent: our plots and figures should conform to a single style. This can be our own design, or following the convention or style-guide of a particular project or organisation. A good example can be seen in the look-and-feel of graphical data presented in *The Economist*.
3. Attention to detail: ensure we take the time to make our visualisations professional. Do not mix font styles, employ adequate spacing between labels, make sure tick marks are visible, margins are sensible, etc.

Good practice

4. Think of our audience: make our visualisations accessible to all. Carefully consider our colour schemes (make sure our palette is colour-blind friendly for example), line styles, marker sizes, and so-on.[1]

[1] `https://thenode.biologists.com/data-visualization-with-flying-colors/research`

Good practice

4. Think of our audience: make our visualisations accessible to all. Carefully consider our colour schemes (make sure our palette is colour-blind friendly for example), line styles, marker sizes, and so-on.[1]
5. Be self-contained: often our visualisation will be accompanied by some explanatory text or other description. But imagine if someone *only* had the graphical part – say a single graph. We should make sure that the visualisation includes everything necessary for someone to understand what is being shown. This includes descriptive axis labels, a title, a legend and possibly annotation (as long as this does not clash with Rule 1 above).

[1] `https://thenode.biologists.com/data-visualization-with-flying-colors/research`

Basic techniques with matplotlib

- The most common visualisation package in Python is matplotlib.
- It is a desktop plotting package designed for creating (mostly two-dimensional) publication-quality plots.

[2] See `http://seaborn.pydata.org`

Basic techniques with matplotlib

- The most common visualisation package in Python is matplotlib.
- It is a desktop plotting package designed for creating (mostly two-dimensional) publication-quality plots.
- The project itself was started by John Hunter in 2002 with the goal of enabling a MATLAB-like plotting interface in Python.
- Over time, it has spawned a number of add-on toolkits for data visualisation that use matplotlib for their underlying plotting.
- One of these is seaborn[2] which is explored in the lab.

[2]See `http://seaborn.pydata.org`

Basic techniques with matplotlib

- The most common visualisation package in Python is matplotlib.
- It is a desktop plotting package designed for creating (mostly two-dimensional) publication-quality plots.
- The project itself was started by John Hunter in 2002 with the goal of enabling a MATLAB-like plotting interface in Python.
- Over time, it has spawned a number of add-on toolkits for data visualisation that use matplotlib for their underlying plotting.
- One of these is seaborn[2] which is explored in the lab.
- The matplotlib and IPython communities have collaborated to simplify interactive plotting from the IPython shell (and now Jupyter Notebook).
- It now supports various GUI backends on all operating systems and can export visualisations to all of the common vector and raster graphics formats (pdf, svg, jpg, png, bmp, gif etc.).

[2]See `http://seaborn.pydata.org`

Using matplotlib (Spyder)

- If we wish to use matplotlib from within a script, then the function `plt.show` is our friend.
- This starts an event loop that looks for all currently active `Figure` objects and opens one or more interactive windows that displays our figure(s). We can then run this script from the command-line prompt if we wanted (e.g. `$ python myplot.py`).
- This will result in a window opening with our figure.

Using matplotlib (Spyder)

- If we wish to use matplotlib from within a script, then the function `plt.show` is our friend.
- This starts an event loop that looks for all currently active `Figure` objects and opens one or more interactive windows that displays our figure(s). We can then run this script from the command-line prompt if we wanted (e.g. `$ python myplot.py`).
- This will result in a window opening with our figure.
- Alternatively, if we run the script in Spyder then it will appear in our window.
- Note however that the `plt.show` command should only be used once per Python session, and is most often seen at the very end of the script. Multiple `show` commands can lead to unpredictable backend-dependent behaviour, and should mostly be avoided.

Using matplotlib (Jupyter)

- If instead we want to plot from Jupyter notebook, then we can do this just as easily.
- IPython is built to work well with matplotlib, so long as we specify matplotlib mode. To enable this mode, we use the following magic command after starting notebook:

  ```
  %matplotlib notebook
  ```
- This will lead to interactive plots embedded within the notebook.

Using matplotlib (Jupyter)

- If instead we want to plot from Jupyter notebook, then we can do this just as easily.
- IPython is built to work well with matplotlib, so long as we specify matplotlib mode. To enable this mode, we use the following magic command after starting notebook:

  ```
  %matplotlib notebook
  ```
- This will lead to interactive plots embedded within the notebook.
- If we instead used the magic command `%matplotlib inline` then this will lead to static images of our plot embedded in the notebook.
- Note that `plt.show` is not required in IPython's matplotlib mode.

Basic plotting with matplotlib

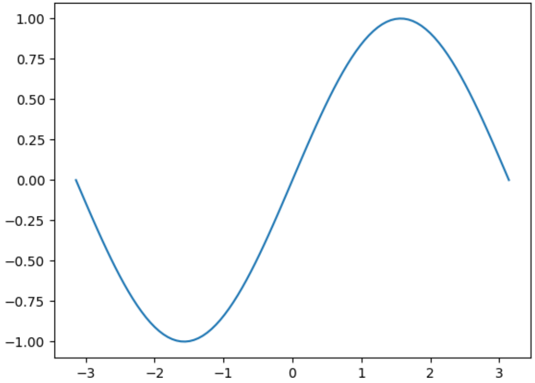
```
import numpy as np
import matplotlib.pyplot as plt

# A flag to determine if we show the plot on the screen
show = False
# Set up a set of 100 values over +/- pi
x = np.linspace(-np.pi,np.pi,100)

#plot sin(x) versus x
plt.plot(x, np.sin(x),label='sin(x)')

#save the plot as a .png file
plt.savefig('fig1.png')

#show it on the screen if show==True
if show: plt.show()
```

Note the way we are importing Matplotlib. The ‘pyplot’ bit refers to a set of functions within the Matplotlib package that allow users to make visualisations in a style similar to MATLAB.

Example continued

- We can call the `plot` function, providing the two arrays that represent the variables we want to appear on the x and y axes, as well as a string label for the data we are plotting using the keyword argument `label`.
- We save the plot as an image (PNG format) on disk, and also display to the screen if we want. The result is as follows:

Example concluded



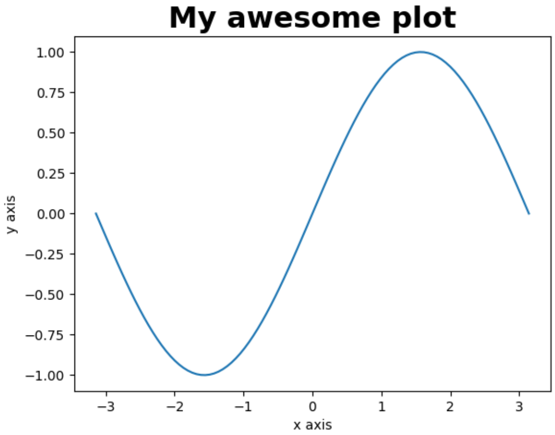
Adding titles

- We can now add some titles for the plot, and the axes

```
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title('My awesome plot', size=22, weight='bold')
plt.savefig('fig2.png')
```

- Note that we can control the font style using keyword arguments when we use text in matplotlib. The result is as follows:

Titles added



Adding another line

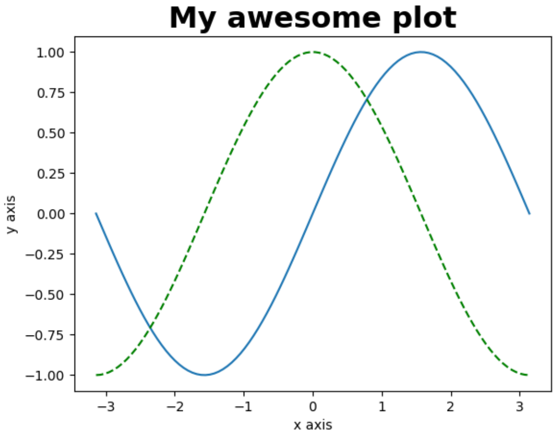
- Next we add another line representing `cos(theta)`. This time, we want to control the look of the line:

```
plt.plot(x, np.cos(x), linestyle='dashed',
color='green',label='cos(x)')
plt.savefig('fig3.png')
```

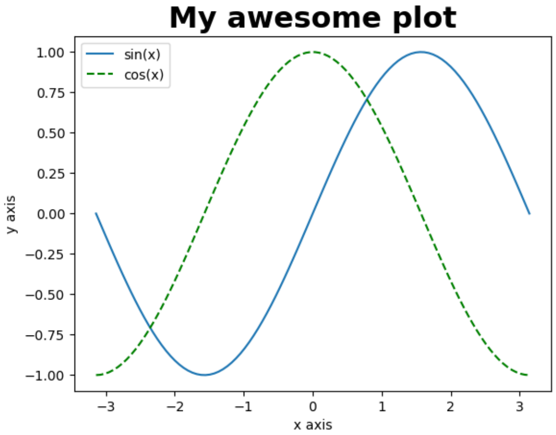
- Here we have used keyword arguments to control the colour and style of the line.
- Now we should add a legend, explaining what the lines mean:

```
plt.legend(loc='best')
plt.savefig('fig4.png')
```

New line added



...with a legend



How the legend works

- The labels supplied when we call each `plot` function is used to associate with each of the lines – note that matplotlib can automatically detect what the different components are (in this case a solid blue and dashed green line), but we can also explicitly specify what elements appear in the legend.
- The `loc` keyword argument sets the position of the legend, which in this case is automatically determined as the 'best' position that does not interfere with the content of the plot.
- Other options are, for example 'bottom right' or 'top left'. We can also supply a tuple of two numbers representing the coordinates of the lower left corner of the legend.

Scatter plots

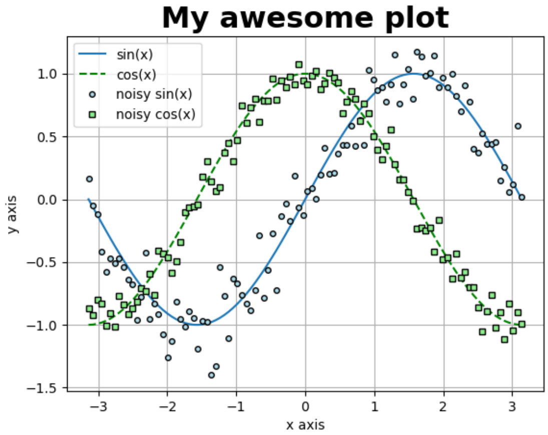
We can simulate some ‘noisy’ data where we simply perturb the pure sinusoidal signals using Gaussian noise – i.e. using the NumPy `normal` generator:

Scatter plots

We can simulate some ‘noisy’ data where we simply perturb the pure sinusoidal signals using Gaussian noise – i.e. using the NumPy `normal` generator:

```
#make some noisy data by adding Gaussian noise to the analytic functions
some_random_sin_points = np.sin(x) + np.random.normal(0,0.2,size=100)
some_random_cos_points = np.cos(x) + np.random.normal(0,0.1,size=100)

#plot noisy sin(x)
plt.plot(x,some_random_sin_points,marker=’o’,linestyle=’None’,markerfacecolor=’lightblue’,
markeredgecolor=’black’,markersize=4,label=’noisy sin(x)’)

#plot noise cos(x)
plt.plot(x,some_random_cos_points,marker=’s’,linestyle=’None’,markerfacecolor=’lightgreen’,
markeredgecolor=’black’,markersize=4,label=’noisy cos(x)’)

#let’s add a grid for the hell of it
plt.grid()

#update the legend
plt.legend(loc=’best’)
plt.savefig(’fig5.png’)
```

The result


My awesome plot
sin(x)
cos(x)
noisy sin(x)
noisy cos(x)
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3
x axis
y axis

What have we done?

- To display the data as points only, we set the `linestyle='none'` and specify a marker ('o'=circle, 's'=square)[3].
- We can control the edge and face colour of the markers using keyword arguments, as well as their size.
- Again, we supply a label for the data to be used in the legend, that we replot.
- Finally, just to show off, we overlay a grid to make reading off the values easier.

[3] `https://matplotlib.org/api/markers_api.html#module-matplotlib.markers`

More resources

- It should be obvious at this point that we can control the appearance of markers, lines and text through the use of keyword arguments.
- We have not covered all the possible options here, so it is recommended to look at the relevant documentation to explore the possibilities: `https://matplotlib.org/contents.html`.
- It may also be worth looking at Part IV of Vander Plas or Chapter 9 of McKinney.

Histograms

- In data science, and statistics in general, we often refer to distributions. This could be some probability distribution, or the distribution of values in some sample.
- We can display distributions easily using the `hist` function.

[4] Also called a Gaussian distribution or probability bell curve.

Histograms

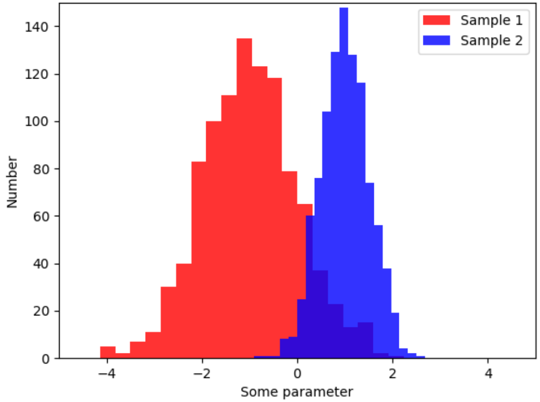
- In data science, and statistics in general, we often refer to distributions. This could be some probability distribution, or the distribution of values in some sample.
- We can display distributions easily using the `hist` function.
- In the following we use the NumPy random module to generate two samples of 1000 elements.
- Sample 1 is selected from a Normal distribution[4] with mean -1 and standard deviation 1,
- Sample 2 is sampled from a Normal distribution with mean 1 and standard deviation 0.5.
- We want to plot and compare these distributions as histograms:

[4] Also called a Gaussian distribution or probability bell curve.

Comparisons as histograms

```
#define the samples
sample1 = np.random.normal(-1,1,size=1000)
sample2 = np.random.normal(1,0.5,size=1000)

#manually set the plot range [xmin, xmax, ymin, ymax]
plt.axis([-5, 5, 0, 150])

#plot the histograms
plt.hist(sample1,bins=20,color='red',alpha=0.8,label='Sample 1')
plt.hist(sample2,bins=20,color='blue',alpha=0.8,label='Sample 2')

#set out labels
plt.xlabel('Some parameter')
plt.ylabel('Number')

#add a legend
plt.legend(loc='upper right')

#save the plot
plt.savefig('fig6.png')
```

The histograms



Normal distribution

- Recall, a Normal distribution is symmetric about the mean μ with width determined by the standard deviation σ.
- More accurately, the range covered by $\mu \pm \sigma$ includes about 68% of the distribution,
- $\mu \pm 2\sigma$ includes about 95% of the distribition,
- while $\mu \pm 3\sigma$ includes about 99.7% of the distribution.

Normal distribution

- Recall, a Normal distribution is symmetric about the mean μ with width determined by the standard deviation σ.
- More accurately, the range covered by $\mu \pm \sigma$ includes about 68% of the distribution,
- $\mu \pm 2\sigma$ includes about 95% of the distribition,
- while $\mu \pm 3\sigma$ includes about 99.7% of the distribution.
- This is telling us that data near the mean are more frequent, while those further away from the mean are less frequent.
- One reason for its importance is the Central Limit Theorem which states that if we have a data set/population (again with mean μ and standard deviation σ) and take sufficiently large random samples from the population with replacement, then the distribution of the sample means will be approximately normally distributed.

Histograms and bins

- Here we have demonstrated how we can manually control the axis ranges using `axis()`.
- In the `hist` calls, we supply the arrays representing our samples, and define how many evenly spaced bins we want to break them up into.
- We can explicitly set the bins ourselves or, like we do here, set an integer number that means that the bin edges are automatically calculated.

Binning

- Consider the following example of data which has been sorted into increasing order.

$$4, 8, 15, 21, 21, 24, 25, 28, 34$$

- We can partition this data into *equal-frequency* bins of size 3:

Bin 1	4, 8, 15
Bin 2	21, 21, 24
Bin 3	25, 28, 34.

Binning

- Alternatively, we could partition data into *equal-width* bins.
- Here, we choose a width (typically, we set $\mathit{width} = \frac{\mathit{max} - \mathit{min}}{\mathit{no.\ of\ bins}}$) and then each bin has range $\mathit{min} + \mathit{width}$.

Binning

- Alternatively, we could partition data into *equal-width* bins.
- Here, we choose a width (typically, we set $\textit{width} = \frac{\textit{max} - \textit{min}}{\textit{no. of bins}}$) and then each bin has range $\textit{min} + \textit{width}$.
- Take another data set:

$$[5, 10, 11, 13, 15, 35, 50, 55, 72, 92, 204, 215].$$

- Suppose we once again want 3 bins, then $\textit{width} = 70$. In this case, we have the following:

Bin 1	5, 10, 11, 13, 15, 35, 50, 55, 72
Bin 2	92
Bin 3	204, 215.

A final note

- Note that the `alpha` keyword argument in the `hist` call sets a transparency for the colour, so we can see where the distributions overlap (on a 0–1 scale where `alpha=0` is totally transparent).
- Again, there are a number of other keyword arguments to `hist` that control its behaviour, and to understand what they do the reader is directed to `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.hist.html`.

Controlling figure size

- In the examples above we let matplotlib decide what size the plot should be, but we can control this easily using the `figure()` function.
- For example, the following will set the size of the plot to 6 inches by 6 inches.

```
plt.figure(figsize=(6,6))
```

- This is useful if we are preparing a figure to appear in a document or presentation.

Subplots

Often, it is useful to compare several plots side-by-side, or arranged in a grid, all in one figure. We do this using sub-plots.

Subplots

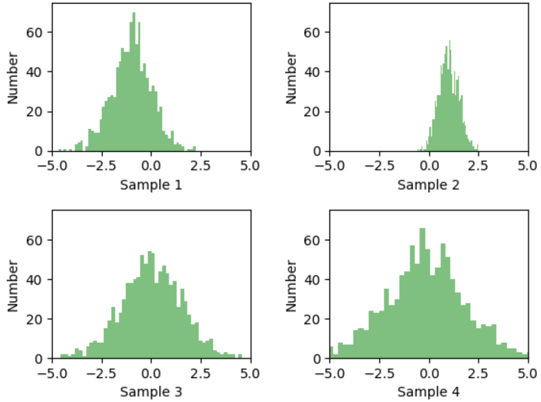
Often, it is useful to compare several plots side-by-side, or arranged in a grid, all in one figure. We do this using sub-plots.

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

#four random samples
sample1 = np.random.normal(-1,1,size=1000)
sample2 = np.random.normal(1,0.5,size=1000)
sample3 = np.random.normal(0,1.5,size=1000)
sample4 = np.random.normal(-0.2,2,size=1000)

#put these in an iterable list
data = [sample1,sample2,sample3,sample4]

#use the subplots_adjust function to make a bit more space between the plots
plt.subplots_adjust(hspace=0.4, wspace=0.4)
```

Subplots

```
#now loop over the data list
for i,d in enumerate(data):
    plt.subplot(2,2,i+1)
    plt.axis([-5, 5, 0, 75])
    plt.hist(d,bins=50,color='green',alpha=0.5)
    plt.xlabel('Sample '+str((i+1)))
    plt.ylabel('Number')

plt.savefig('fig7.png')
```

Subplots


Number
0
20
40
60
−5.0
−2.5
0.0
2.5
5.0
Sample 1
Sample 2
Sample 3
Sample 4

What have we done?

- We use the `subplots_adjust` function to increase the spacing between the individual plots – this is just one of the ways to improve presentation.
- We use the built in `enumerate` function. The argument is a list (in this case holding our samples), and what is returned is a pair of values representing a counter (called `i` here) and the corresponding item in the list (which we call `d` here).

What have we done?

- We use the `subplots_adjust` function to increase the spacing between the individual plots – this is just one of the ways to improve presentation.
- We use the built in `enumerate` function. The argument is a list (in this case holding our samples), and what is returned is a pair of values representing a counter (called `i` here) and the corresponding item in the list (which we call `d` here).
- The arguments to `subplot` are the total number of sub-plots in the horizontal and vertical direction (in this case 2 and 2), and the final argument is the counter of the sub-plot we are working on.
- Irritatingly, this is indexed from 1, unlike the usual convention. As a result, we have to do `i+1` here, as the first counter returned by `enumerate` is zero.
- After that, we define our axes ranges, then plot the corresponding histogram of samples represented by `d`.
- We add a label to tell us what each plot represents, and then come out of the loop and save the figure.

Summary

- We have introduced matplotlib.
- We have seen how it can be used to produce basic plots, scatter plots and histograms.
- We have also seen how to change the size of plots and how to produce subplots.

Exercise: Before next time, read pp. 63–70 of the notes.