An\_Introduction\_Statistics\_Python\_c04

**Chapter 4**

**Display of Statistical Data**

The dominant task of the human cortex is to extract visual information from the activity patters on the retina. Our visual system is therefore exceedingly good at detecting patterns in visualized data sets. As a result, one can almost always *see* what is happening before it can be demonstrated through a quantitative analysis of the data. Visual data displays are also helpful at finding extreme data values, which are often caused by mistakes in the execution of the paradigm or mistakes in the data acquisition.

This chapter shows a number of different ways to visualize statistical data sets.

**4.1 Datatypes**

The choice of appropriate statistical procedure depends on the data type. Data can be *categorical* or *numerical*. If the variables are numerical, we are led to a certain statistical strategy. In contrast, if the variables represent qualitative categorizations, then we follow a different path.

In addition, we distinguish between *univariate*, *bivariate*, and *multivariate* data. *Univariate data* are data of only one variable, e.g., the size of a person. *Bivariate data* have two parameters, for example, the *x*=*y* position in a plane, or the income as a function of age. *Multivariate data* have three or more variables, e.g., the position of a particle in space, etc.

***4.1.1 Categorical***

**a) Boolean**

*Boolean data* are data which can only have two possible values. For example,

• female/male

• smoker/nonsmoker

• True/False

**b) Nominal**

Many classifications require more than two categories. Such data are called *nominal data*. An example is *married/single/divorced*.

**c) Ordinal**

In contrast to nominal data, *ordinal data* are ordered and have a logical sequence, e.g., *very few/few/some/many/very many*.

***4.1.2 Numerical***

**a) Numerical Continuous**

Whenever possible, it is best to record the data in their original continuous format, and only with a meaningful number of decimal places. For example, it does not make sense to record the body size with more than 1 mm accuracy, as there are larger changes in body height between the size in the morning and the size in the evening, due to compression of the intervertebral disks.

**b) Numerical Discrete**

Some numerical data can only take on integer values. These data are called *numerical discrete*. For example *Number of children: 0 1 2 3 4 5* : : :

**4.2 Plotting in Python**

Visualization is most important for numerical data, so I will focus on the following on this data type.

In practice the display of data can be a bit tricky, as there are so many options: graphical output can be displayed as a picture in an HTML-page, or in an interactive graphics window; plots can force the attention of the user, or can automatically close after a few seconds, etc. This section will therefore focus on general aspects of plotting data; the next section will then present different types of plots, e.g., histograms, errorbars, 3D-plots, etc.

The *Python* core does not include any tools to generate plots. This functionality is added by other packages. By far the most common package for plotting is *matplotlib*. If you have installed *Python* with a scientific distribution like *WinPython* or *Anaconda*, it will already be included. *matplotlib* is intended to mimic the style of *Matlab*. As such, users can either generate plots in the *Matlab* style, or in the traditional *Python* style (see below).

*matplotlib* contains different modules and features:

**matplotlib.pyplot** This is the module that is commonly used to generate plots. It provides the interface to the plotting library in *matplotlib*, and is by convention imported in *Python* functions and modules with

import matplotlib.pyplot as plt.

*pyplot* handles lots of little details, such as creating figures and axes for the plot, so that the user can concentrate on the data analysis.

**matplotlib.mlab** Contains a number of functions that are commonly used in

*Matlab*, such as find, griddata, etc.

**“backends”** *matplotlib* can produce output in many different formats, which are referred to as *backends*:

* In a *Jupyter Notebook*, or in a *Jupyter QtConsole*, the command %matplotlib inline directs output into the current browser window. (%pylab inline is a combination of loading pylab, and directing plot-output inline.)
* In the same environment, %matplotlib qt41 directs the output into a separate graphics window (Fig. 2.4). This allows panning and zooming the plot, and interactive selection of points on the plot by the user, with the command plt.ginput.
* With plt.savefig output can be directed to external files, e.g., in PDF, PNG, or JPG format.

***pylab***is a convenience module that bulk imports matplotlib.pyplot (for plotting) and numpy (for mathematics and working with arrays) in a single name space. Although many examples use *pylab*, it is no longer recommended, and should only be used in *IPython*, to facilitate interactive development of code.

***4.2.1 Functional and Object-Oriented Approaches to Plotting***

*Python* plots can be generated in a *Matlab*-like style, or in an object oriented, more pythonic way. These styles are all perfectly valid, and each have their pros and cons. The only caveat is to avoid mixing the coding styles in your own code.

First, consider the frequently used *pyplot* style:

# Import the required packages, # with their conventional names import matplotlib.pyplot as plt import numpy as np

# Generate the data

x = np.arange(0, 10, 0.2) y = np.sin(x)

# Generate the plot plt.plot(x, y)

# Display it on the screen plt.show()

Note that the creation of the required figure and an axes is done automatically by *pyplot*.

Second, a more pythonic, object oriented style, which may be clearer when working with multiple figures and axes. Compared to the example above, only the section entitled *“# Generate the plot”* changes:

# Generate the plot

fig = plt.figure() # Generate the figure

ax = fig.add\_subplot(111) # Add an axis to that figure ax.plot(x,y) # Add a plot to that axis

For interactive data analysis, it is convenient to load the most common commands from *numpy* and *matplotlib.pyplot* into the current workspace. This is achieved with pylab, and leads to a *Matlab*-like coding style:

from pylab import \*

x = arange(0, 10, 0.2) y = sin(x)

plot(x, y)

show()

So, why all the extra typing as one moves away from the pure *Matlab*-style? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and to maintain. For example, the following lines of code produce a figure with two plots above each other, and clearly indicate which plot goes into which axis:

# Import the required packages import matplotlib.pyplot as plt import numpy as np

# Generate the data

x = np.arange(0, 10, 0.2) y = np.sin(x)

z = np.cos(x)

# Generate the figure and the axes

fig, axs = plt.subplots(nrows=2, ncols=1)

# On the first axis, plot the sine and label the ordinate axs[0].plot(x,y)

axs[0].set\_ylabel('Sine')

# On the second axis, plot the cosine axs[1].plot(x,z) axs[1].set\_ylabel('Cosine')

# Display the resulting plot plt.show()

**Code:** “ISP\_gettingStarted.py”2 gives a short demonstration of *Python* for scientific data analysis.

***4.2.2 Interactive Plots***

*matplotlib* provides different ways to interact with the user. Unfortunately, this interaction is less intuitive than in *Matlab*. The examples below may help to bypass most of these problems. They show how to

* Exactly position figures on the screen.
* Pause between two plots, and proceed automatically after a few seconds.
* Proceed on a click or keyboard hit.
* Evaluate keyboard entries.  
     
  **Listing 4.1 L4\_1\_interactivePlots.py**  
   # Source: http://scipy-central.org/item/84/1/simple-  
     
  interactive-matplotlib-plots

'''Interactive graphs with Matplotlib have haunted me. So here I have collected a number of

tricks that should make interactive use of plots simpler. The functions below show how to

- Position figures on the screen (e.g. top left half of display)

- Pause to display the plot, and proceed automatically after a few sec

- Proceed on a click, or a keyboard hit - Evaluate keyboard inputs

author: Thomas Haslwanter date: Nov-2015

ver: 1.1

license: CC BY-SA 4.0

'''

# Import standard packages import numpy as np

import matplotlib.pyplot as plt

# additional packages

try:

import tkinter as

except ImportError:

2.x

import Tkinter as

tk

tk

#different capitalization in Python

t = np.arange(0,10,0.1)

c = np.cos(t)

s = np.sin(t)

def normalPlot():

'''Just show a plot. The progam stops, and only continues

when the plot is closed,

either by hitting the "Window Close" button, or by typing

"ALT+F4". '''

plt.plot(t,s)

plt.title('Normal plot: you have to close it to continue\

nby clicking the "Window Close" button, or by hitting

"ALT+F4"')

plt.show()

def positionOnScreen():

'''Position two plots on your screen. This uses the

Tickle-backend, which I think is the default on all platforms.'''

# Get the screen size

root = tk.Tk()

(screen\_w, screen\_h) = (root.winfo\_screenwidth(), root.

winfo\_screenheight())

root.destroy()

def positionFigure(figure, geometry):

'''Position one figure on a given location on the

screen.

This works for Tk and for Qt5 backends, but may fail

on others.'''

mgr = figure.canvas.manager

(pos\_x, pos\_y, width, height) = geometry

try:

# positioning commands for Tk

position = '{0}x{1}+{2}+{3}'.format(width, height , pos\_x, pos\_y)

mgr.window.geometry(position)

except TypeError:

# positioning commands for Qt5

mgr.window.setGeometry(pos\_x, pos\_y, width, height)

# The program continues after the first plot

fig = plt.figure()

ax = fig.add\_subplot(111)

ax.plot(t,c)

ax.set\_title('Top Left: Close this one last')

# Position the first graph in the top-left half of the screen

topLeft = (0, 0, screen\_w//2, screen\_h//2) positionFigure(fig, topLeft)

# Put another graph in the top right half

fig2 = plt.figure()

ax2 = fig2.add\_subplot(111)

ax2.plot(t,s)

# I don't completely understand why this one has to be

closed first. But otherwise the program gets unstable.

ax2.set\_title('Top Right: Close this one first (e.g. with ALT+F4)')

topRight = (screen\_w//2, 0, screen\_w//2, screen\_h//2) positionFigure(fig2, topRight)

plt.show()

def showAndPause():

'''Show a plot only for 2 seconds, and then proceed

automatically'''

plt.plot(t,s)

plt.title('Don''t touch! I will proceed automatically.')

plt.show(block=False) duration = 2 # [sec] plt.pause(duration)

plt.close()

def waitForInput():

''' This time, proceed with a click or by hitting any key

'''

plt.plot(t,c)

plt.title('Click in that window, or hit any key to

continue') plt.waitforbuttonpress()

plt.close()

def keySelection():

'''Wait for user intput, and proceed depending on the key

entered.

This is a bit complex. But None of the versions I tried

without

key binding were completely stable.'''

fig, ax = plt.subplots() fig.canvas.mpl\_connect('key\_press\_event', on\_key\_event)

# Disable default Matplotlib shortcut keys:

keymaps = [param for param in plt.rcParams if param.find( 'keymap') >= 0]

for key in keymaps:

plt.rcParams[key] = ''

ax.plot(t,c)

ax.set\_title('First, enter a vowel:') plt.show()

def on\_key\_event(event): '''Keyboard interaction'''

#print('you pressed %s'%event.key)

key = event.key

# In Python 2.x, the key gets indicated as "alt+[key]" # Bypass this bug:

if key.find('alt') == 0:

key = key.split('+')[1]

curAxis = plt.gca() if key in 'aeiou':

curAxis.set\_title('Well done!')

plt.pause(1)

plt.close()

else:

curAxis.set\_title(key + ' is not a vowel: try again

to find a vowel ....') plt.draw()

if \_\_name\_\_ == '\_\_main\_\_':

normalPlot() positionOnScreen() showAndPause() waitForInput() keySelection()

**4.3 Displaying Statistical Datasets**

The first step in the data analysis should always be a visual inspection of the raw- data. Between 30 and 50 % of our cortex are involved in the processing of visual information, and as a result our brain is very good at recognizing patterns in visually represented data. The trick is choosing the most informative type of display for your data.

The easiest way to find and implement one of the many image types that *matplotlib* offers is to browse their gallery (http://matplotlib.org/gallery.html), and copy the corresponding *Python* code into your program.

For statistical data analysis, the *Python* package *seaborn* (http://www.stanford. edu/~mwaskom/software/seaborn/) builds on *matplotlib*, and aims to provide a con- cise, high-level interface for drawing statistical graphics that are both informative and attractive. Also *pandas* builds on *matplotlib* and offers many convenient ways to visualize DataFrames.

Other interesting plotting packages are:

* *plot.ly* is a package that is available for *Python*, *Matlab*, and *R*, and makes beautiful graphs (https://plot.ly).
* *bokeh* is a *Python* interactive visualization library that targets modern web browsers for presentation. *bokeh* can help anyone who would like to quickly and easily create interactive plots, dashboards, and data applications (http://bokeh. pydata.org/).
* *ggplot* for *Python*. It emulates the *R*-package *ggplot*, which is loved by many *R*-users (http://ggplot.yhathq.com/).

***4.3.1 Univariate Data***

The following examples all have the same format. Only the “Plot-command” line changes.

# Import standard packages import numpy as np

import matplotlib.pyplot as plt import pandas as pd

import scipy.stats as stats import seaborn as sns

# Generate the data

x = np.random.randn(500)

# Plot-command start --------------------- plt.plot(x, '.')

# Plot-command end -----------------------

# Show plot

plt.show()

1. **Scatter Plots**

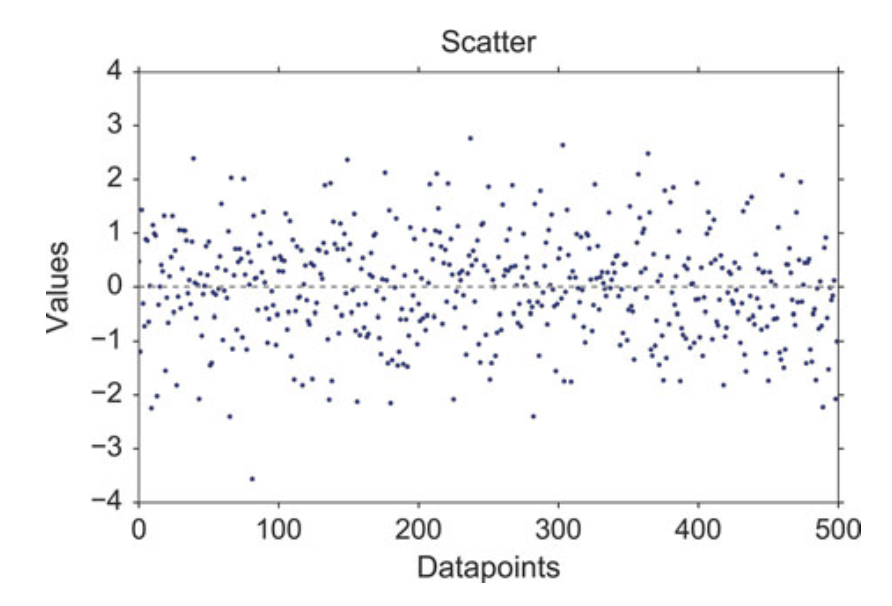
This is the simplest way to represent univariate data: just plot each individual data point. The corresponding plot-command is either

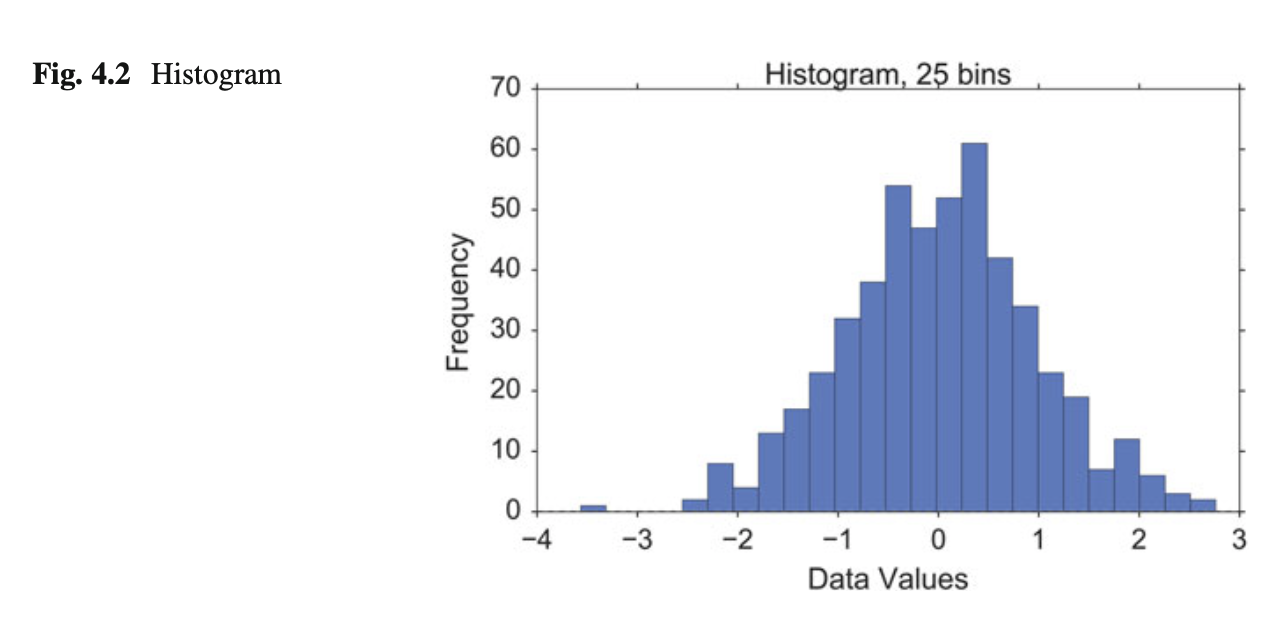
plt.plot(x, '.')

or, equivalently,

plt.scatter(np.arange(len(x), x))

Note: In cases where we only have few discrete values on the *x*-axis (e.g., *Group1, Group2, Group3*), it may be helpful to spread overlapping data points slightly (also referred to as *“adding jitter”)* to show each data point. An exam- ple can be found at http://stanford.edu/~mwaskom/software/seaborn/generated/ seaborn.stripplot.html)





**b) Histograms**

*Histograms* provide a first good overview of the distribution of your data. If you divide by the overall number of data points, you get a *relative frequency histogram*; and if you just connect the top center points of each bin, you obtain a *relative frequency polygon*.

plt.hist(x, bins=25)

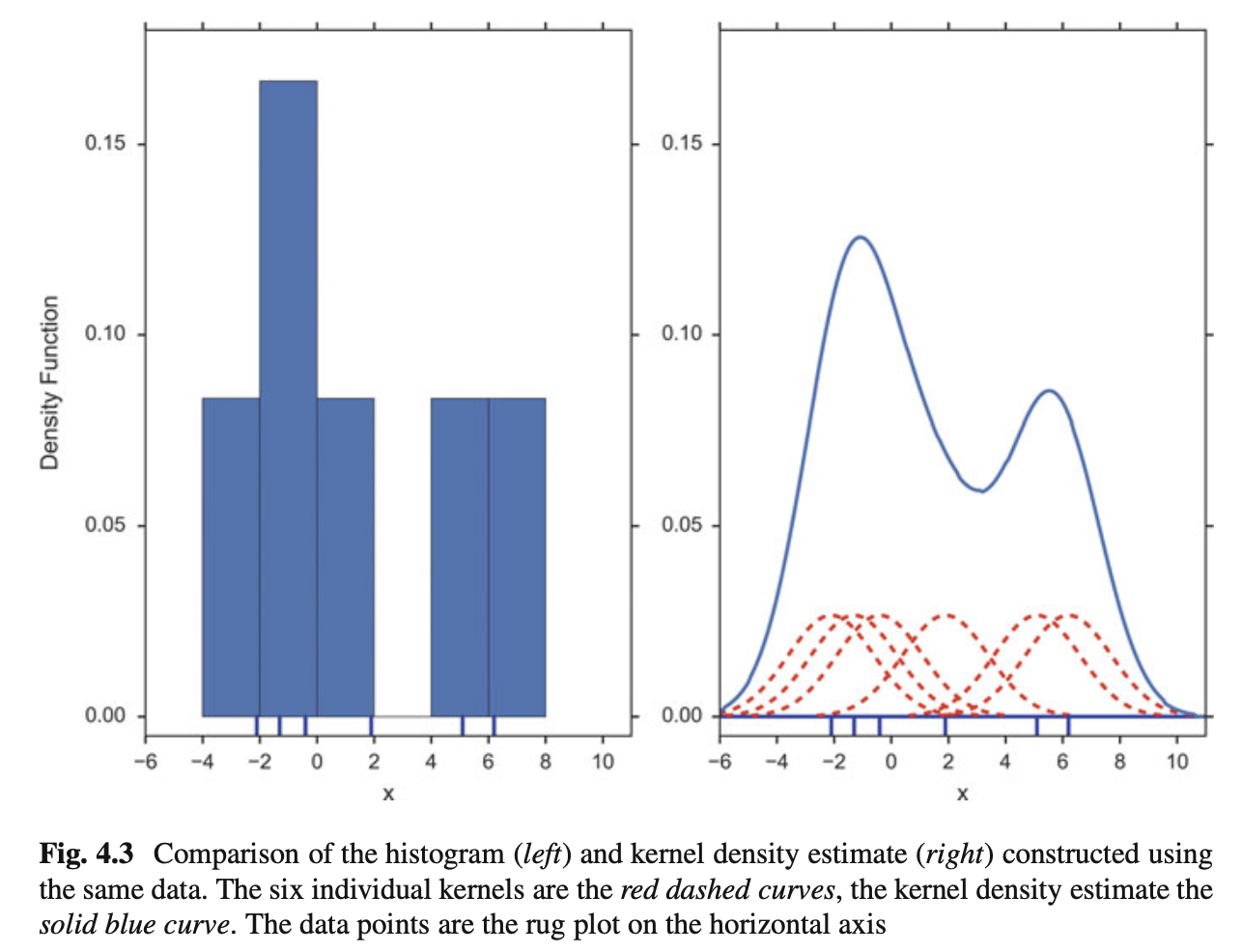
**c) Kernel-Density-Estimation (KDE) Plots**

Histograms have the disadvantage that they are discontinuous, and that their shape critically depends on the chosen bin-width. In order to obtain smooth *probability densities*, i.e., curves describing the likelihood of finding an event in any given interval, the technique of *Kernel Density Estimation* (KDE) can be used. Thereby a normal distribution is typically used for the kernel. The width of this kernel function determines the amount of smoothing. To see how this works, we compare the construction of histogram and kernel density estimators, using the following six data points:

x = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2].

For the histogram, first the horizontal axis is divided into subintervals or bins which cover the range of the data. In Fig. 4.3, left, we have six bins each of width 2. Whenever a data point falls inside this interval, we place a box of height 1/12. If more than one data point falls inside the same bin, we stack the boxes on top of each other.

For the kernel density estimate, we place a normal kernel with variance 2.25 (indicated by the red dashed lines in Fig.4.3, right) on each of the data points *xi*. The kernels are summed to make the kernel density estimate (solid blue curve). The smoothness of the kernel density estimate is evident. Compared to the discreteness

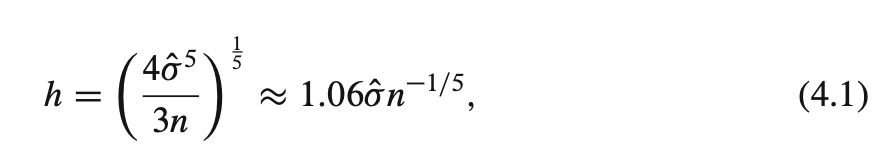


of the histogram, the kernel density estimates converge faster to the true underlying density for continuous random variables.

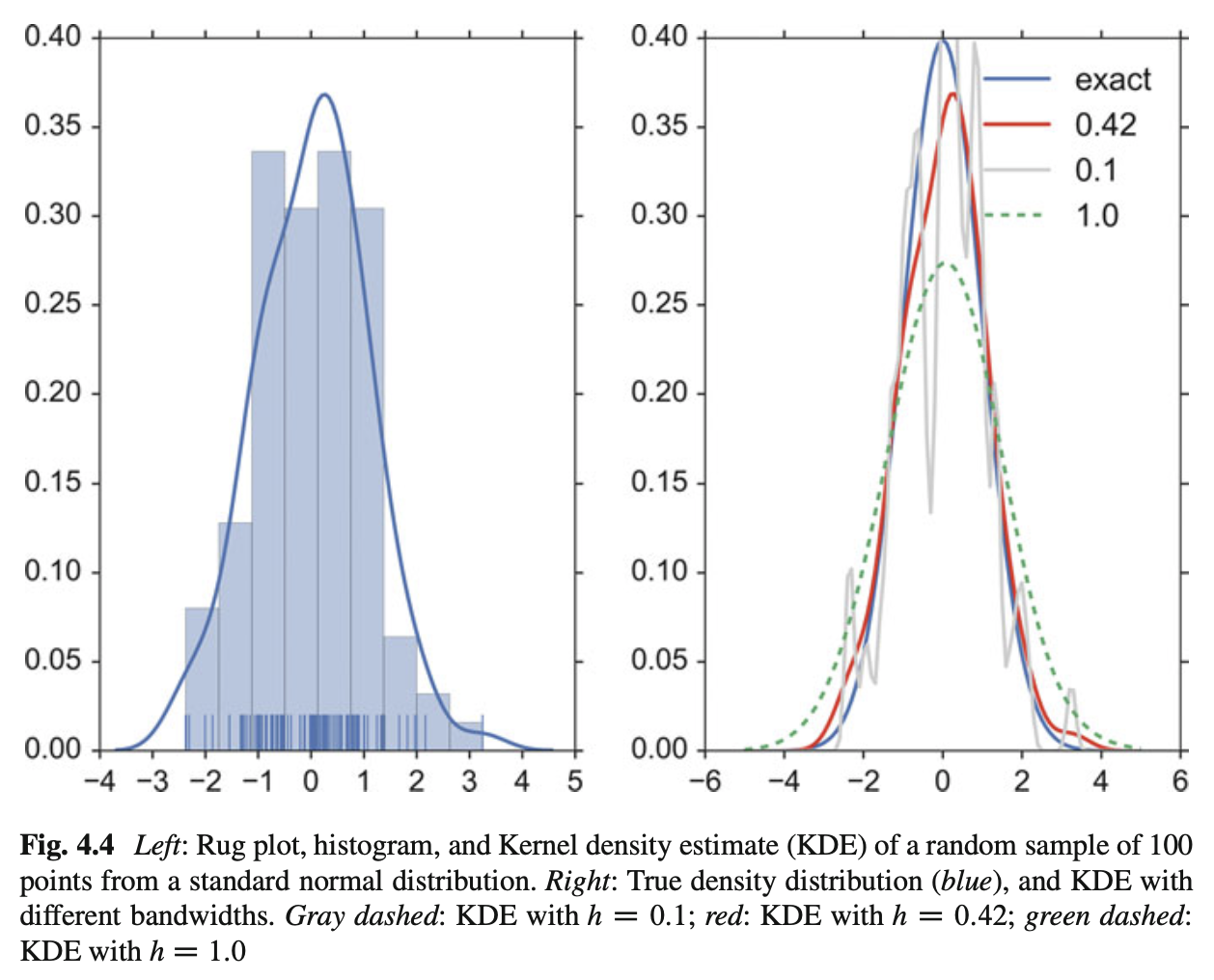
sns.kdeplot(x)

The bandwidth of the kernel is the parameter which determines how much we smooth out the contribution from each event. To illustrate its effect, we take a simulated random sample from the standard normal distribution, plotted as the blue spikes in the *rug plot* on the horizontal axis in Fig. 4.4, left. (A *rug plot* is a plot where every data entry is visualized by a small vertical tick.) The right plot shows the true density in blue. (A normal density with mean 0 and variance 1.) In comparison, the gray curve is undersmoothed since it contains too many spurious data artifacts arising from using a bandwidth *h* D 0:1 which is too small. The green dashed curve is oversmoothed since using the bandwidth *h* D 1 obscures much of the underlying structure. The red curve with a bandwidth of *h* D 0:42 is considered to be optimally smoothed since its density estimate is close to the true density.

It can be shown that under certain conditions the optimal choice for *h* is



where 􏰀O(sigma) is the standard deviation of the samples (“Silverman’s rule of thumb”).

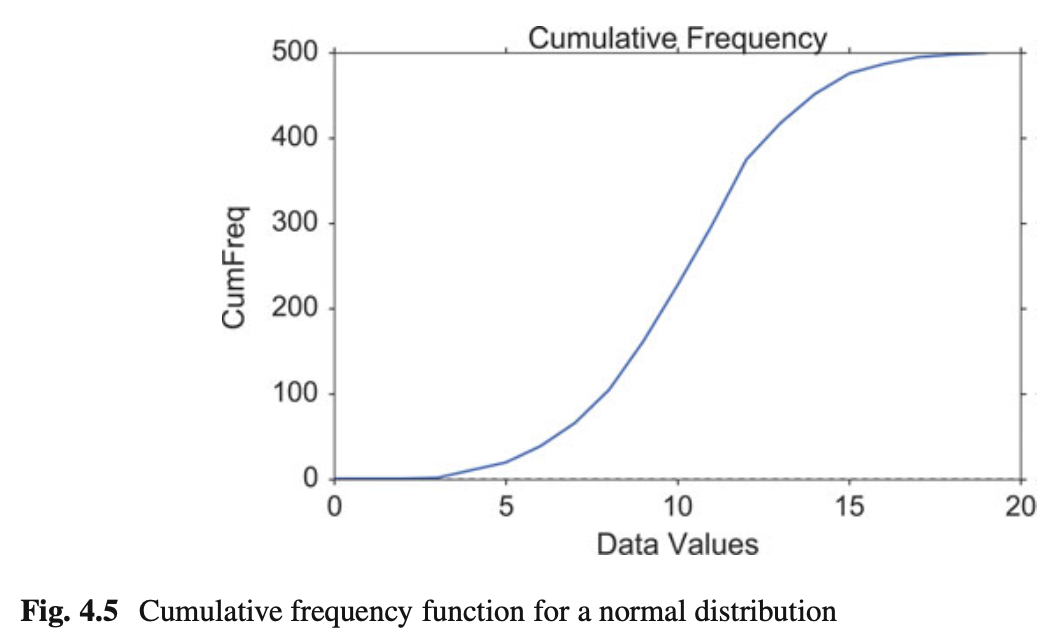


**d) Cumulative Frequencies**

A *cumulative frequency* curve indicates the number (or percent) of data with less than a given value. This curve is very useful for statistical analysis, for example when we want to know the data range containing 95 % of all the values. Cumulative frequencies are also useful for comparing the distribution of values in two or more different groups of individuals.

When you use percentage points, the cumulative frequency presentation has the additional advantage that it is bounded: 0 􏰏 *cumfreq*.*x*/ 􏰏 1

**plt.plot(stats.cumfreq(x,numbins)[0])**



**e) Error-Bars**

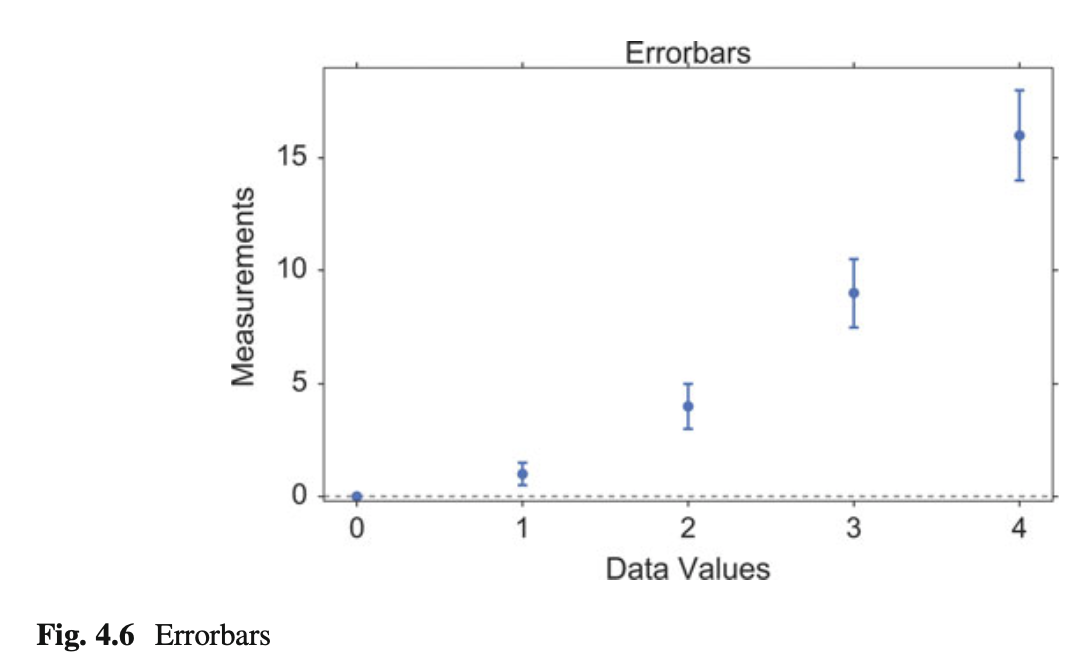
*Error-bars* are a common way to show mean value and variability when comparing measurement values. Note that it always has to be stated explicitly if the error-bars correspond to the *standard deviation* or to the *standard error* of the data. Using *standard errors* has a nice feature: When error bars for the standard errors for two groups overlap, one can be sure the difference between the two means is not statistically significant (*p* > 0:05). However, the opposite is not always true!

index = np.arange(5)

y = index\*\*2

errorBar = index/2 # just for demonstration plt.errorbar(index,y, yerr=errorBar, fmt='o',

capsize=5, capthick=3)



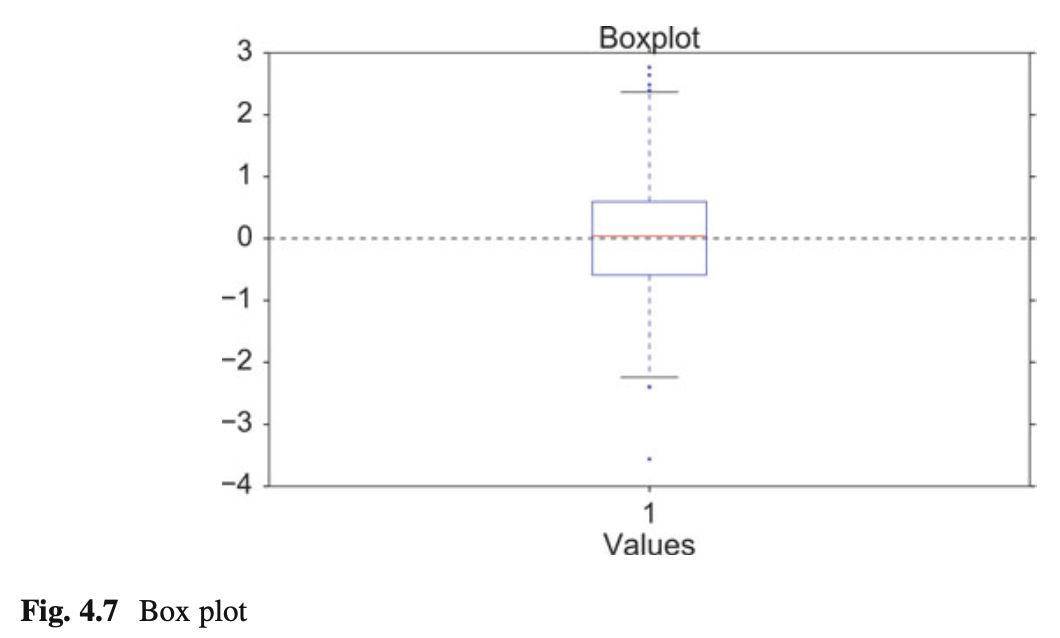
**f) Box Plots**

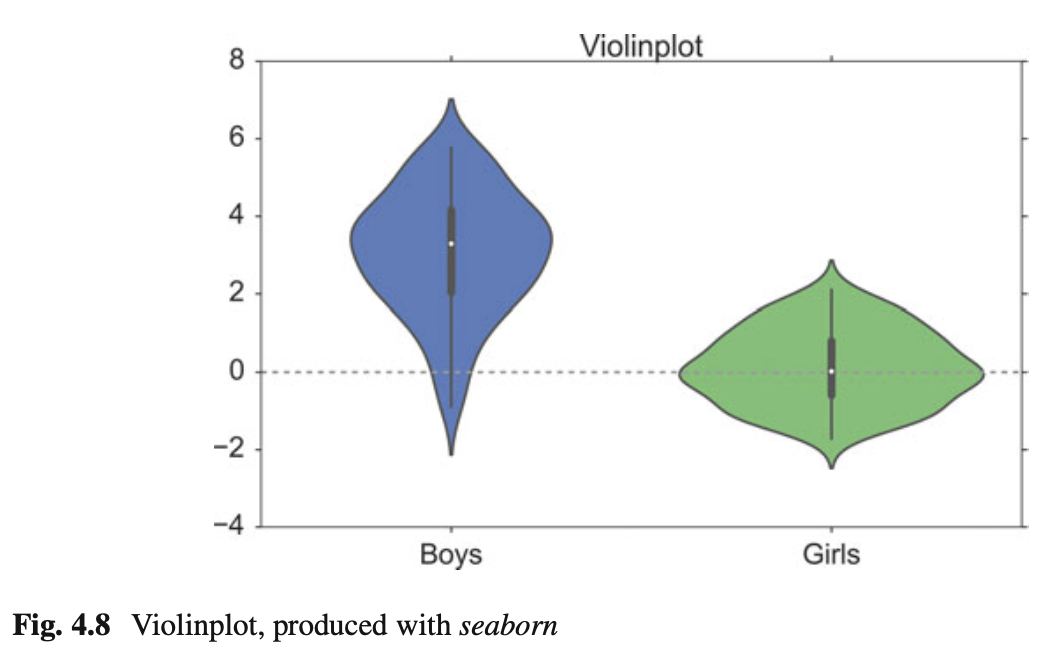
Boxplots are frequently used in scientific publications to indicate values in two or more groups. The bottom and top of the box indicate the first quartile and third quartile, respectively, and the line inside the box shows the median. Care has to be taken with the whiskers, as different conventions exist for them. The most common form is that the lower whisker indicates the lowest value still within 1.5 \* *inter- quartile-range* (IQR) of the lower quartile, and the upper whisker the highest value still within 1.5 \* IQR of the upper quartile. Outliers (outside the whiskers) are plotted separately. Another convention is to have the whiskers indicate the full data range.

There are a number of tests to check for outliers. The method suggested by Tukey, for example, is to check for data which lie more than 1.5 \* IQR above or below the first/third quartile (see Sect. 6.1.2).

plt.boxplot(x, sym='\*')

Boxplots can be combined with KDE-plots to produce the so-called *violin plots*, where the vertical axis is the same as for the box-plot, but in addition a KDE-plot is shown symmetrically along the horizontal direction (Fig. 4.8).





# Generate the data

nd = stats.norm

data = nd.rvs(size=(100))

nd2 = stats.norm(loc = 3, scale = 1.5) data2 = nd2.rvs(size=(100))

# Use pandas and the seaborn package

# for the violin plot

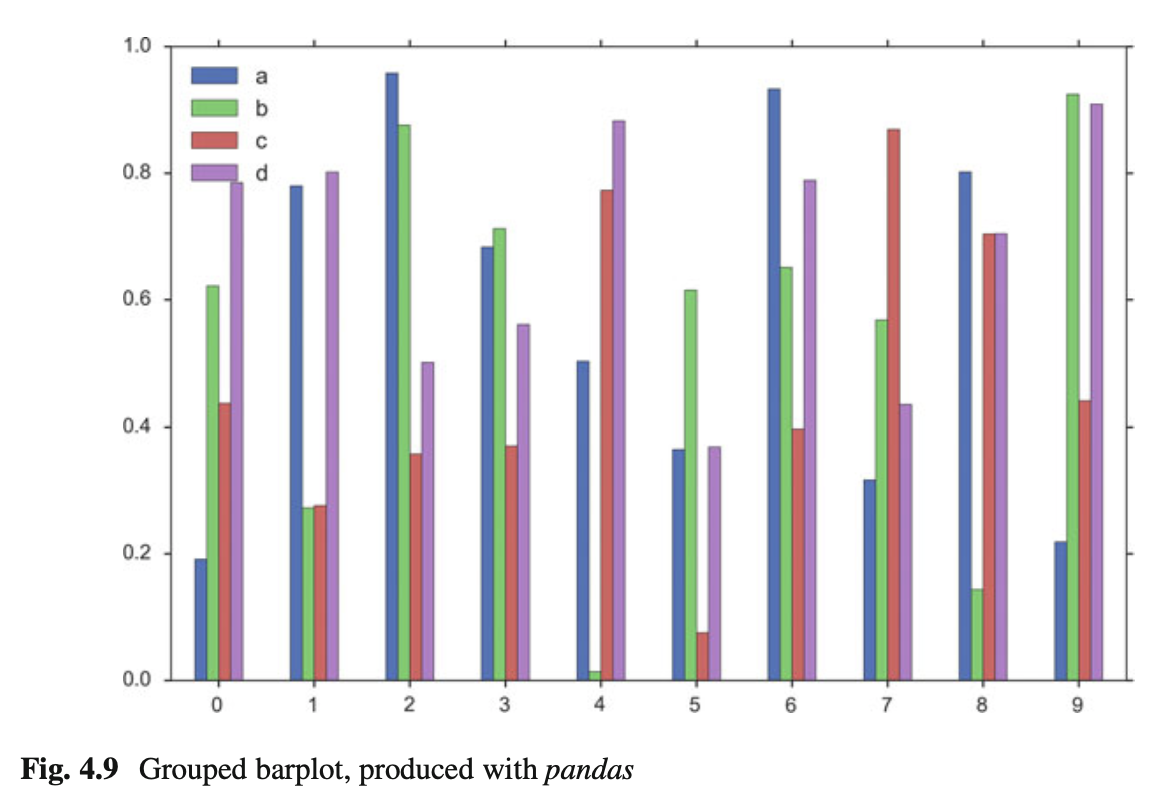
df = pd.DataFrame({'Girls':data, 'Boys':data2}) sns.violinplot(df)

**g) Grouped Bar Charts**

For some applications the plotting abilities of *pandas* can facilitate the generation of useful graphs, e.g., for grouped barplots (Figs. 4.9, 4.10, 4.11, and 4.12):

df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

df.plot(kind='bar', grid=False)



**h) Pie Charts**

*Pie charts* can be generated with a number of different options, e.g. import seaborn as sns

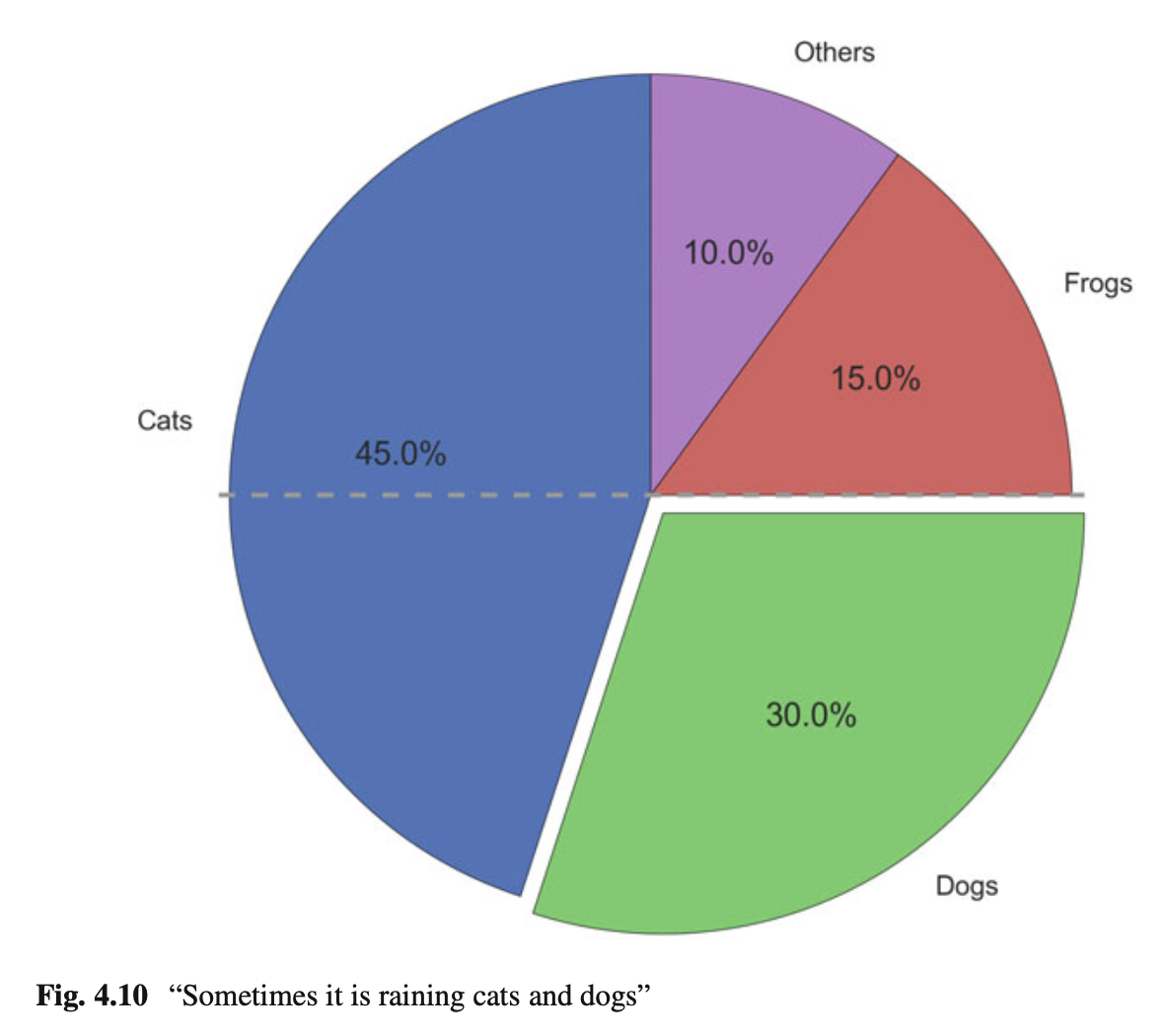
import matplotlib.pyplot as plt

txtLabels = 'Cats', 'Dogs', 'Frogs', 'Others' fractions = [45, 30, 15, 10]

offsets =(0, 0.05, 0, 0)

plt.pie(fractions, explode=offsets, labels=txtLabels, autopct='%1.1f%%', shadow=True, startangle=90, colors=sns.color\_palette('muted') )

plt.axis('equal')



**i) Programs: Data Display**

**Code:** “ISP\_showPlots.py”3 shows how the plots in this section have been generated.

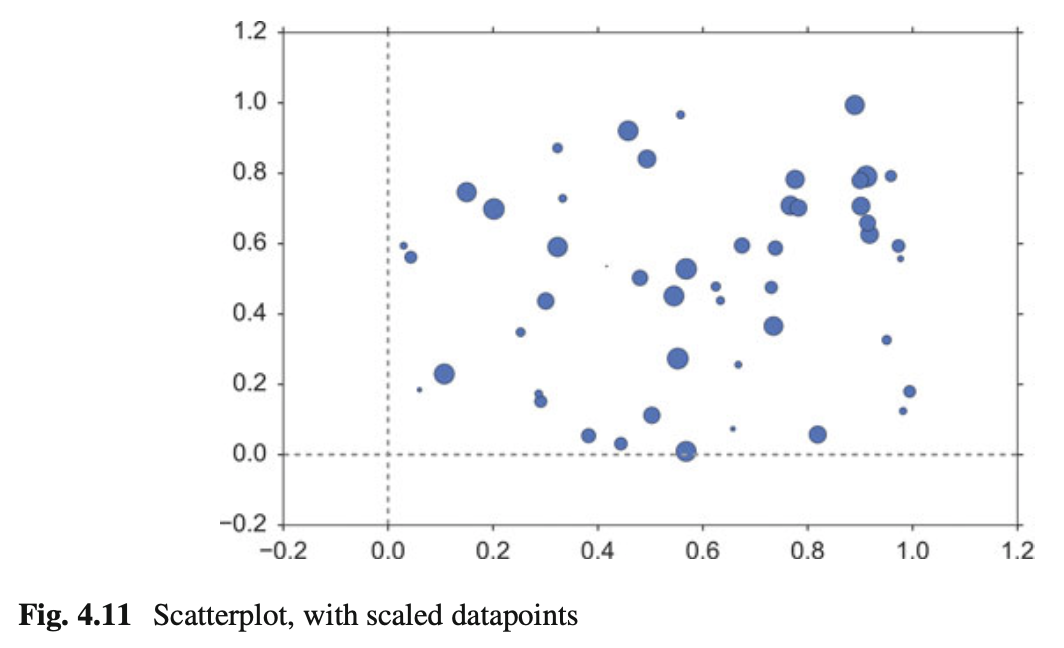
***4.3.2 Bivariate and Multivariate Plots***

**a) Bivariate Scatter Plots**

Simple scatter plots are trivial. But *pandas* also makes fancy scatter plots easy:

df2 = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])

df2.plot(kind='scatter', x='a', y='b', s=df['c']\*300);



**b) 3D Plots**

3D plots in *matplotlib* are a bit awkward, because separate modules have to be imported, and axes for 3D plots have to be explicitly declared. However, once the axis is correctly defined, the rest is straightforward. Here are two examples:

# imports specific to the plots in this example import numpy as np

from matplotlib import cm

from mpl\_toolkits.mplot3d.axes3d import get\_test\_data

# Twice as wide as it is tall.

fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot

# Note that the declaration "projection='3d'" # is required for 3d plots!

ax = fig.add\_subplot(1, 2, 1, projection='3d')

# Generate the grid

X = np.arange(-5, 5, 0.1) Y = np.arange(-5, 5, 0.1) X, Y = np.meshgrid(X, Y)

# Generate the surface data R = np.sqrt(X\*\*2 + Y\*\*2)

Z = np.sin(R)

# Plot the surface

surf = ax.plot\_surface(X, Y, Z, rstride=1, cstride=1,

cmap=cm.GnBu, linewidth=0, antialiased=False) ax.set\_zlim3d(-1.01, 1.01)

fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot

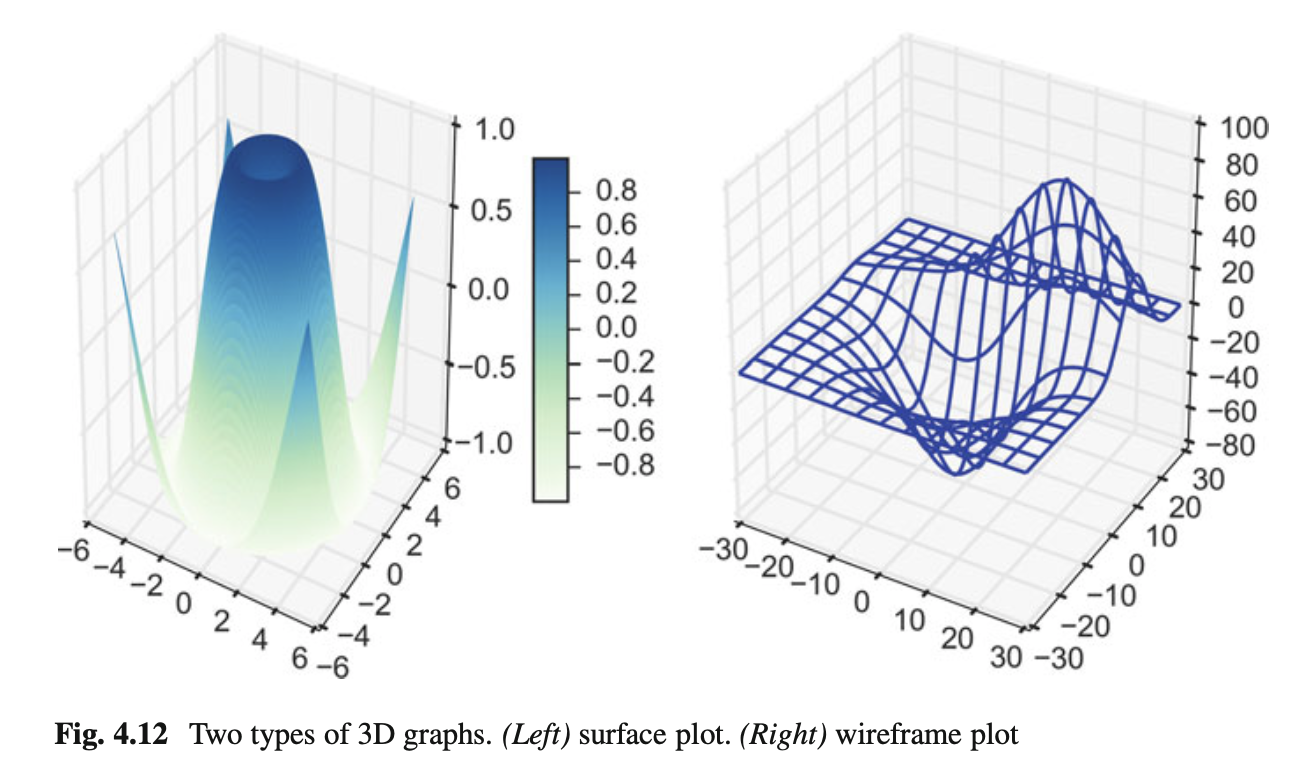
ax = fig.add\_subplot(1, 2, 2, projection='3d')

X, Y, Z = get\_test\_data(0.05)

ax.plot\_wireframe(X, Y, Z, rstride=10, cstride=10)

outfile = '3dGraph.png' plt.savefig(outfile, dpi=200)

print('Image saved to {0}'.format(outfile)) plt.show()



**4.4 Exercises**

**4.1 DataDisplay**

1. Read in the data from ‘Data\amstat\babyboom.dat.txt’.

2. Inspect them visually, and give a numerical description of the data.

3. Are the data normally distributed?