经济大数据与 Python 应用

Introduction to Matplotlib

陈洲
湘潭大学商学院
2025

Press Space for next page →

Content

- Simple Line Plot
- Simple Scatter Plot
- Density and Contour Plots

Visualization with Matplotlib

Importing Matplotlib

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Demo: $f(x) = sin(x)$ and $f(x) = cos(x)$

```
import numpy as np
x = np.linspace(0, 10, 100)

fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```

Saving Figures to File

Saving a figure can be done using the `savefig` command.

For example, to save the previous figure as a PNG file, we can run this:

```
fig.savefig('my_figure.png')
```

File format is inferred from the **extension** of the given filename.

The list of supported file types

```
fig.canvas.get_supported_filetypes()
```

Two Interfaces for the Price of One

- a convenient MATLAB-style state-based interface
- a powerful object-oriented interface

MATLAB-style Interface

```
plt.figure()  # create a plot figure

# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```

Object-oriented interface

To re-create the previous plot using this style of plotting

```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```

Simple Line Plots

Setting up the notebook for plotting

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

Creating a figure and axes.

```
fig = plt.figure()
ax = plt.axes()
```

- **figure**: a single container that contains all the objects representing axes, graphics, text, and labels
- **axes**: a bounding box with ticks, grids, and labels

Use the `ax.plot` method to plot some data

```
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```

The semicolon at the end of the last line

- suppresses the textual representation of the plot from the output.

Alternatively, the MATLAB-style

```
plt.plot(x, np.sin(x));
```

Create a single figure with multiple lines (see the following figure)

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

Adjusting the Plot: Line Colors and Styles

```
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name
plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')        # grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # hex code (RRGGBB, 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # HTML color names supported
```

Similarly, the line style can be adjusted using the `linestyle` keyword (see the following figure):

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');
```

```
# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```

Combining these `linestyle` and `color` codes into a single non-keyword argument to the `plt.plot` function

```
plt.plot(x, x + 0, '-g')   # solid green
plt.plot(x, x + 1, '--c')  # dashed cyan
plt.plot(x, x + 2, '-.k')  # dashdot black
plt.plot(x, x + 3, ':r');  # dotted red
```

Labeling Plots

The labeling of plots: titles, axis labels, and simple legends.

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```

Create a plot legend (图例) that labels each line type.

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')
```

```
plt.legend();
```

Matplotlib Gotchas

MATLAB-style to Object-oriented style

- `plt.xlabel` → `ax.set_xlabel`
- `plt.ylabel` → `ax.set_ylabel`
- `plt.xlim` → `ax.set_xlim`
- `plt.ylim` → `ax.set_ylim`
- `plt.title` → `ax.set_title`
- use the `ax.set` method to set all these properties at once

```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2),
       xlabel='x', ylabel='sin(x)',
       title='A Simple Plot');
```

Simple Scatter Plots

Setup

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

Scatter Plots with plt.plot

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```

Marker styles

```
# just run, skip explanation
rng = np.random.default_rng(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.random(2), rng.random(2), marker, color='black',
             label="marker='{0}'".format(marker))
plt.legend(numpoints=1, fontsize=13)
plt.xlim(0, 1.8);
```

These character codes can be used together with line and color codes

```
plt.plot(x, y, '-ok');
```

Additional keyword arguments to `plt.plot` specify a wide range of properties of the lines and markers

```
plt.plot(x, y, '-p', color='gray',
         markersize=15, linewidth=4,
         markerfacecolor='white',
         markeredgecolor='gray',
         markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

Scatter Plots with plt.scatter

```
plt.scatter(x, y, marker='o');
```

Create scatter plots where the **properties of each individual point**

```
rng = np.random.default_rng(0)
x = rng.normal(size=100)
y = rng.normal(size=100)
colors = rng.random(100)
sizes = 1000 * rng.random(100)
```

```
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)
plt.colorbar();  # show color scale
```

For example

- the Iris dataset from Scikit-Learn
- three types of flowers

```
from sklearn.datasets import load_iris
iris = load_iris()
features = iris.data.T

plt.scatter(features[0], features[1], alpha=0.4,
            s=100*features[3], c=iris.target, cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1]);
```

sepal 花萼

Visualizing Uncertainties

Errors example: %timeit result

Basic Errorbars

Setup

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

```
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)
```

```
plt.errorbar(x, y, yerr=dy, fmt='.k');
```

In crowded plots, to make the errorbars lighter than the points themselves (see the following figure):

```
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
             ecolor='lightgray', elinewidth=3, capsize=0);
```

Density and Contour Plots

There are three Matplotlib functions:

- `plt.contour` for contour plots
- `plt.contourf` for filled contour plots
- and `plt.imshow` for showing images.

Setup

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

Visualizing a Three-Dimensional Function

A contour plot using a function $z = f(x, y)$

$$f(x, y) = \sin^{10}(x) + \cos(10 + y \times x) \times \cos(x)$$

```
def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

A contour plot can be created with the `plt.contour` function.

```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

```
plt.contour(X, Y, Z, colors='black');
```

Notice that when a single color is used

- *negative* values are represented by *dashed* lines and positive values by solid lines.

lines can be color-coded by specifying a colormap with the `cmap` argument.

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```

The `RdGy` (short for *Red–Gray*) colormap

- More color map `plt.cm.<TAB>`

Switching to a filled contour plot using the `plt.contourf` function

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```

Add a `plt.colorbar` command, which creates an additional axis

Histograms, Binnings, and Density

Setup

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

rng = np.random.default_rng(1701)
data = rng.normal(size=1000)
```

Histogram

```
plt.hist(data);
```

More customized histogram

```
plt.hist(data, bins=30, density=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```

Multiple histogram

```
x1 = rng.normal(0, 0.8, 1000)
x2 = rng.normal(-2, 1, 1000)
x3 = rng.normal(3, 2, 1000)
```

```
kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

Two-Dimensional Histograms and Binnings

Setup

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = rng.multivariate_normal(mean, cov, 10000).T
```

plt.hist2d: Two-dimensional histogram

```
plt.hist2d(x, y, bins=30)
cb = plt.colorbar()
cb.set_label('counts in bin')
```

plt.hexbin: Hexagonal binnings

```
plt.hexbin(x, y, gridsize=30)
cb = plt.colorbar(label='count in bin')
```

Multiple Subplots

In this chapter we'll explore four routines for creating subplots in Matplotlib.

We'll start by importing the packages we will use:

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

plt.subplot: Simple Grids of Subplots

This command takes three integer arguments

- the number of rows, the number of columns, and the index of the plot
- runs from the upper left to the bottom right (see the following figure):

```
for i in range(1, 7):
    plt.subplot(2, 3, i)
    plt.text(0.5, 0.5, str((2, 3, i)),
             fontsize=18, ha='center')
```

The command `plt.subplots_adjust` can be used to adjust the spacing between these plots.

- the space is 40% of the subplot width and height

The equivalent object-oriented command, `fig.add_subplot`; the following figure shows the result:

```
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(1, 7):
    ax = fig.add_subplot(2, 3, i)
    ax.text(0.5, 0.5, str((2, 3, i)),
           fontsize=18, ha='center')
```

plt.subplots: The Whole Grid in One Go

Let's create a 2×3 grid of subplots, where all axes in the same row share their y-axis scale, and all axes in the same column share their x-axis scale (see the following figure):

```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```

Text and Annotation

Setup

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import pandas as pd
```

Example: Effect of Holidays on US Births

Data in birthrate data

Create dataframe `births_by_date` (直接复制)

```
from datetime import datetime

births = pd.read_csv('data/births.csv')

quartiles = np.percentile(births['births'], [25, 50, 75])
mu, sig = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')

births['day'] = births['day'].astype(int)

births.index = pd.to_datetime(10000 * births.year +
                              100 * births.month +
                              births.day, format='%Y%m%d')
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.index = [datetime(2012, month, day)
                        for (month, day) in births_by_date.index]
```

Create plot

```
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax);
```

- Useful to annotate certain features of the plot to draw the reader's attention
- with the `plt.text` / `ax.text` functions

```
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Add labels to the plot
style = dict(size=10, color='gray')

ax.text('2012-1-1', 3950, "New Year's Day", **style)
ax.text('2012-7-4', 4250, "Independence Day", ha='center', **style)
ax.text('2012-9-4', 4850, "Labor Day", ha='center', **style)
ax.text('2012-10-31', 4600, "Halloween", ha='right', **style)
ax.text('2012-11-25', 4450, "Thanksgiving", ha='center', **style)
ax.text('2012-12-25', 3850, "Christmas ", ha='right', **style)

# Label the axes
ax.set(title='USA births by day of year (1969-1988)',
       ylabel='average daily births')

# Format the x-axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
```

The `ax.text` method

- takes an *x* position, a *y* position
- a string, and
- optional keywords specifying the color, size, style, alignment, and other properties of the text.
- `ha='right'` and `ha='center'`, where `ha` is short for *horizontal alignment*.

See the docstrings of `plt.text` and `mpl.text.Text` for more information on the available options.

Annotation

Along with tickmarks and text, another useful annotation mark is the simple arrow.

```
fig, ax = plt.subplots()

x = np.linspace(0, 20, 1000)
ax.plot(x, np.cos(x))
ax.axis('equal')

ax.annotate('local maximum', xy=(6.28, 1), xytext=(10, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -6),
            arrowprops=dict(arrowstyle="->"));
```

Let's demonstrate several of the possible options using the birthrate plot from before (see the following figure):

```
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Add labels to the plot
ax.annotate("New Year's Day", xy=('2012-1-1', 4100),  xycoords='data',
            xytext=(50, -30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3,rad=-0.2"))

ax.annotate("Independence Day", xy=('2012-7-4', 4250),  xycoords='data',
            bbox=dict(boxstyle="round", fc="none", ec="gray"),
            xytext=(10, -40), textcoords='offset points', ha='center',
            arrowprops=dict(arrowstyle="->"))
```

```
ax.annotate('Labor Day Weekend', xy=('2012-9-4', 4850), xycoords='data',
            ha='center', xytext=(0, -20), textcoords='offset points')
ax.annotate('', xy=('2012-9-1', 4850), xytext=('2012-9-7', 4850),
            xycoords='data', textcoords='data',
            arrowprops={'arrowstyle': '|-|,widthA=0.2,widthB=0.2', })

ax.annotate('Halloween', xy=('2012-10-31', 4600),  xycoords='data',
            xytext=(-80, -40), textcoords='offset points',
            arrowprops=dict(arrowstyle="fancy",
                            fc="0.6", ec="none",
                            connectionstyle="angle3,angleA=0,angleB=-90"))

ax.annotate('Thanksgiving', xy=('2012-11-25', 4500),  xycoords='data',
            xytext=(-120, -60), textcoords='offset points',
            bbox=dict(boxstyle="round4,pad=.5", fc="0.9"),
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle,angleA=0,angleB=80,rad=20"))
```

```
ax.annotate('Christmas', xy=('2012-12-25', 3850),  xycoords='data',
            xytext=(-30, 0), textcoords='offset points',
            size=13, ha='right', va="center",
            bbox=dict(boxstyle="round", alpha=0.1),
            arrowprops=dict(arrowstyle="wedge,tail_width=0.5", alpha=0.1));



# Label the axes
ax.set(title='USA births by day of year (1969-1988)',
       ylabel='average daily births')

# Format the x-axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));

ax.set_ylim(3600, 5400);
```

Customizing Ticks

Major and Minor Ticks

Within each axes,

- there is the concept of a *major* tickmark, and a *minor* tickmark
- major ticks are usually bigger or more pronounced
- minor ticks are usually smaller

Setup

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

Ticks

```
ax = plt.axes(xscale='log', yscale='log')
ax.set(xlim=(1, 1E3), ylim=(1, 1E3))
ax.grid(True);
```

Hiding Ticks or Labels

This can be done using `plt.NullLocator` and `plt.NullFormatter`, as shown here (see the following figure):

```
ax = plt.axes()
rng = np.random.default_rng(1701)
ax.plot(rng.random(50))
ax.grid()

ax.yaxis.set_major_locator(plt.NullLocator())
ax.xaxis.set_major_formatter(plt.NullFormatter())
```

Reducing or Increasing the Number of Ticks

```
fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)
```

- quite difficult to decipher

Adjust this is with `plt.MaxNLocator`

```
# For every axis, set the x and y major locator
for axi in ax.flat:
    axi.xaxis.set_major_locator(plt.MaxNLocator(3))
    axi.yaxis.set_major_locator(plt.MaxNLocator(3))
fig
```

Fancy Tick Formats

Consider this plot of a sine and a cosine curve (see the following figure):

```
# Plot a sine and cosine curve
fig, ax = plt.subplots()
x = np.linspace(0, 3 * np.pi, 1000)
ax.plot(x, np.sin(x), lw=3, label='Sine')
ax.plot(x, np.cos(x), lw=3, label='Cosine')

# Set up grid, legend, and limits
ax.grid(True)
ax.legend(frameon=False)
ax.axis('equal')
ax.set_xlim(0, 3 * np.pi);
```

Set ticks and gridlines in multiples of π

```
ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 4))
fig
```

Customizing Matplotlib: Stylesheets

Stylesheets

A newer mechanism for adjusting overall chart styles is via Matplotlib's `style` module

A list of the available styles

```
plt.style.available[:5]
```

The standard way to switch to a stylesheet is to call `style.use`:

```
plt.style.use('stylename')
```

Alternatively, use the style context manager:

```
with plt.style.context('stylename'):
    make_a_plot()
```

To demonstrate these styles, let's create a function that will make two basic types of plot:

```
def hist_and_lines():
    np.random.seed(0)
    fig, ax = plt.subplots(1, 2, figsize=(11, 4))
    ax[0].hist(np.random.randn(1000))
    for i in range(3):
        ax[1].plot(np.random.rand(10))
    ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

Built-in styles

Default style

```
with plt.style.context('default'):
    hist_and_lines()
```

Grayscale style

```
with plt.style.context('grayscale'):
    hist_and_lines()
```

Three-Dimensional Plotting in Matplotlib

Three-dimensional plots are enabled by importing the `mplot3d` toolkit

```
from mpl_toolkits import mplot3d
```

A three-dimensional axes can be created by passing the keyword `projection='3d'`

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

```
fig = plt.figure()
ax = plt.axes(projection='3d')
```

Three-Dimensional Points and Lines

Using the `ax.plot3D` and `ax.scatter3D` functions

```
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

Three-Dimensional Contour Plots

Meshgrid

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

ax.contour3D

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 40, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```

`view_init` method to set the elevation and azimuthal angles

- 60 degrees above the x-y plane
- 35 degrees counter-clockwise about the z-axis

```
ax.view_init(60, 35)
fig
```

Wireframes and Surface Plots

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z)
ax.set_title('wireframe');
```

A surface plot is like a wireframe plot

- but each face of the wireframe is a filled polygon.

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z,
                cmap='viridis', edgecolor='none')
ax.set_title('surface');
```

Task

使用销售额的模拟数据

- 绘制销售额随时间变化的折线图
- 日期为时间数据类型

```
import pandas as pd

data = {
    'date': ['2023年01月15日', '2023年02月20日', '2023年03月25日',
             '2023年04月10日', '2023年05月05日', '2023年06月18日'],
    'sales': [120, 150, 180, 135, 210, 195]  # 万元
}

df = pd.DataFrame(data)
```

END