Plotting with Matplotlib

Matplotlib is a library built for creating high-quality plots. It provides control over several aspects of plots, can be used with Latex, can generate animations, and can output plots in various formats (like png, jpg and PDF). Furthermore, Matplotlib is a building block for several other libraries (HoloViews, Bokeh and seaborn are interesting projects).

1 Install

Matplotlib can be installed with `pip`. After activating your environment in the terminal, run:

```
pip install matplotlib
```

Now `matplotlib` is available for Python and can be imported with:

```
import matplotlib
```

All of the Matplotlib functionality can now be accessed via the object `matplotlib`. Matplotlib offers all of the plotting functionality in the sub-package `pyplot`, which can be imported as:

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

Which makes the `matplotlib.pyplot` functionality available via `plt`.

While using jupyter notebook, we will also execute a special command that starts with the percentage symbol. These type of commands are particular to jupyter and are not part of the Python language. These commands are used to configure how the notebook interacts with certain features of Python.

```
%matplotlib inline
```

This command enables the display of figures directly in the notebook in the result part of the code cell. An alternative, would be `%matplotlib`, which would open a window outside the notebook. You read more about it on the reference page.

2 Basics

Matplotlib offers two different styles for plotting: a high-level approach and a object-oriented approach. The high-level approach is convenient for quick plots:

```
import numpy as np
x = np.linspace(0, 2*np.pi, 100)
y = np.cos(x)

# Plot the graph of y
```

```
plt.plot(x, y, 'k-', linewidth=1.5)
plt.show()
```

The function `plot` takes care of creating the figure window, adding the axes and then plotting the curve. While this is simple and convenient in some cases, it also creates several variables in the `plt` namespace without alerting the user.

The second approach, and the one that is more Pythonic (to see Python guidelines type `import this`), makes the creation of the figure more explicit:

```
fig, ax = plt.subplots()
# subplots creates a window figure and the axes for plotting,
# and then returns both objects
ax.plot(x, y, 'k-', linewidth=1)
fig.show()
```

This second approach is object-oriented, since we use the methods of the object `ax` to create draw the plot on the figure. This approach is explicit on what it is doing, but also offers more fine control over the plot.

3 Scatter Plots

Let's generate some fake data to plot:

```
import numpy as np
noise = np.random.normal(size=(1000, 1))
x = np.random.normal(size=(1000, 1))
beta = np.random.uniform(size=(1, 1))
y = x@beta + noise
```

We will create a scatter plot while specifying many of the options that you might use in your own plots.

```
# Create figure and axis:
fig, ax = plt.subplots(nrows=1,
                       ncols=1,
                       figsize=(12, 6))
# nrows, ncols: used for multiple plots in a single figure
# figsize: changes figure window size (length, height)

# Scatter plot:
ax.scatter(x, y,
           color='black',       # color of points
           marker='o',          # marker style
           alpha=1,             # transparency
           label='Points: $(x,y)$')
# label: label for legend,
# notice the use of Latex works in matplotlib

# Add a single point:
ax.scatter(0, 0, s=50,          # s for marker size
           color=(1, 0, 0),     # color specified as RGB triple
           marker='*',          # marker style
           label='Special Point: $(0,0)$')  # notice the use of Latex
```

```
# Add title and labels:
ax.set(title='Scatter Plot Example',
       xlabel='Value of $x$',
       ylabel='Value of $y$')

# For more control over title and labels:
ax.set_title('Scatter Plot Example',
             fontdict={'fontsize': 18,  # in pixels
                       'fontweight': 'bold'})
# fontweight: can be an int, like 600, but varies with font
ax.set_xlabel('Value of $x$', fontdict={'fontsize': 12}, labelpad=12)
ax.set_ylabel('Value of $y$', fontdict={'fontsize': 12}, labelpad=12)

# Change range of values on axis:
ax.set_xlim((-6, 6))
ax.set_ylim((-10, 10))
# Same as:
ax.set_xlim(left=-6, right=6)
ax.set_ylim(bottom=-10, top=10)

# Add legends:
ax.legend(loc='best',
          fontsize=18)
# loc: 'upper right', 'center right', 'lower right', and so on

# Turn on a semi-transparent grid:
ax.grid(True, alpha=0.5)

# Save figure:
fig.savefig('scatter-plot.png', dpi=300)
fig.show()

# Close figure window:
plt.close(fig)
```

Notice the use of `$ $` on labels. Matplotlib understands Latex expressions without the need of any other adjustment.

4 Line Plots

Let's create a line plot of the data:

```
# Estimate linear model
beta_hat = np.linalg.inv(x.T@x)@x.T@y
x_min = 1.1*x.min()
x_max = 1.1*x.max()
x_values = np.linspace(x_min, x_max)
y_values = x_values
# Create figure and axis
fig, ax = plt.subplots(nrows=1,
                       ncols=1,
```

```
                        figsize=(12, 6))
# Line plot:
ax.plot(x_values, y_values,
        color='black',
        alpha=1,
        label='Points: $(x,y)$')
# add title and labels
ax.set(title='Scatter Plot Example',
       xlabel='Value of $x$',
       ylabel='Value of $y$')
# change range of values on axis
ax.set_xlim([-6, 6])
ax.set_ylim([-10, 10])
# add legends
ax.legend()
# turn on grid
ax.grid(True, alpha=0.5)
# Save figure:
fig.savefig('line-plot.png', dpi=300)
fig.show()                          # show the figure on a window
# Close figure:
# plt.close(fig)
```

5 Combining the Scatter and Line Plots

```
# Create figure and axis:
fig, ax = plt.subplots(nrows=1,
                       ncols=1,
                       figsize=(12, 6))
# Scatter plot:
ax.scatter(x, y, color=[0.8, 0.8, 0.8], marker='o', label='Sample')

# Add a single point:
ax.scatter(0, 0, s=200, color='red', marker='*',
           label='Special Point: $(0,0)$')

# Add the line plot:
ax.plot(x_values, y_values, linewidth=2, color='black',
        label='Estimated Model')

# Add title and labels:
ax.set(title='Linear Regression Example',
       xlabel='$x$',
       ylabel='$y$')
ax.legend(fontsize=18)
ax.grid(True, alpha=0.5)

# Save figure:
fig.savefig('scatter-plot.png', dpi=300)
fig.show()
```

```
# Close figure window:
plt.close(fig)
```

6 Multiple Plots in a Figure

We can create a single figure window that contains separate plots. That is, we can create a figure window with multiple separate axes.

Now, use `plt.subplots` to create a figure window and multiple axes:

```
fig, axes = plt.subplots(nrows=2,
                         ncols=1,
                         figsize=(12, 14))  # height is 14 now
# ax is now a Numpy array containing the two axis just created
# We can plot to each axis by using each of the objects
# in the array ax.

# Plot on the first axis:
axes[0].scatter(x, y, color='black', marker='o',
                label='Points: $(x,y)$')
# add a single point
axes[0].scatter(0, 0, s=200,
                color='red', marker='*',
                label='Special Point: $(0,0)$')
# add title and labels
axes[0].set(title='Scatter Plot Example',
            xlabel='Value of $x$',
            ylabel='Value of $y$')

# Plot on the second axis:
axes[1].plot(x, y, color='black',
             label='Points: $(x,y)$')
# add title and labels
axes[1].set(title='Scatter Plot Example',
            xlabel='Value of $x$',
            ylabel='Value of $y$')

# For turning on legends and the grid, we can use a loop:
for axis in axes:
    # add legends
    axis.legend()
    # turn on grid
    axis.grid(True, alpha=0.5)
fig.show()
```

Notice the change in the `nrows` option and in the `figsize` so that both plots have space in the rendered figure.

Let's create a figure with 4 axes:

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 12))
# Now, axes is a 2x2 numpy array, where each element represents one of
```

```
# the axis in the figure.
for row in axes:
    for axis in row:
        mean = np.random.uniform(-1, 1)
        std = np.random.uniform(1, 4)
        axis.hist(np.random.normal(mean, std, size=(2000, 1)), bins=50)
        axis.set_title(fr'$\mathcal{{N}}\left(\mu={mean:.2f}, \sigma={std:.
fig.show()
```

7 Surface Plots

While the focus of Matplotlib is on 2-dimensional plots, the package still offers some tools to create surface plots.

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

The `Axes3D` updates `plt` so that we can also create surface plots, and `cm` will be used to specify the color-map of the figure.

Define a function to plot:

```
def rosenbrock(x, y):
    a = 1
    b = 100
    return (a-x)**2+b*(y-x**2)**2
```

We need to create a grid of points where the function will be evaluated:

```
x_grid = np.linspace(-2, 2, 50)
y_grid = np.linspace(-2, 2, 50)
# help(np.meshgrid)
x, y = np.meshgrid(x_grid, y_grid)
z = rosenbrock(x, y)
```

The elements of `x` and `y` define a mesh of points in the plane. By getting an element of the matrix in `x` and an element in the same position in the matrix `y`, we can compute the value of `z`.

We can now create the figure and the surface plot. Here we need to go one level lower and first create the figure, and then add the axis separately.

```
fig = plt.figure(figsize=(12, 8))
axis = fig.add_subplot(1, 1, 1,     # nrows, ncols, index
                       projection='3d')
surf = axis.plot_surface(x, y, z,
                         rcount=50,     # downsamples input x
                         ccount=50,     # downsamples input y
                         alpha=0.7,     # transparency
                         cmap=cm.cool
                         )
# color-map: hot, winter, jet and
#            terrain are some of the many options

fig.colorbar(surf,              # inserts colorbar based on these values
```

```
                ax=axis,            # on this axis
                shrink=0.5,         # shrinks the height of the colorbar
                orientation='vertical')
fig.show()
```

See the reference page on colormaps to see all available options.

8 Customizing Defaults

There are a few ways of customizing the defaults on Matplotlib. A low-level customization of defaults is by altering the properties of the `matplotlib.rcParams`.

```
import matplotlib as mpl
# object that holds all default values
print(mpl.rcParams)
fig, ax = plt.subplots()
ax.plot(np.linspace(-5, 5), np.linspace(-5, 5), linewidth=1.5)
# change default value for linewidth
mpl.rcParams['lines.linewidth'] = 5
ax.plot(np.linspace(-5, 5), np.linspace(5, -5))
fig.show()
# Reset default values:
mpl.rcdefaults()
```

For more details on `rcParams` read this reference page.

There is a high-level way of customizing the defaults via style sheets. Style sheets are basically files where the `rcParams` have been set in a certain way, and you just use those settings.

```
def plotlines():
    fig, ax = plt.subplots()
    for i in range(5):
        x = np.linspace(0, 2*np.pi, 100)
        y = np.cos(x + i*np.pi/2)
        ax.plot(x, y)
    return fig, ax


fig, ax = plotlines()
fig.show()
# change style sheet
plt.style.use('ggplot')
fig, ax = plotlines()
fig.show()
```

Due to the number of people that have some type of color vision deficiency, we might want to use a more friendly color scheme:

```
plt.style.use('seaborn-colorblind')
fig, ax = plotlines()
fig.show()
# or a grayscale (which also works well for printing)
plt.style.use('grayscale')
```

```
fig, ax = plotlines()
fig.show()
```

The official style sheets reference page contains the names of the different built-in styles. This page also displays the different style sheets, but with different styles of plots.

9 Assignment

There are many other types of plots we have not covered, like the ones generated with the functions `bar` and `contour`. We will cover some other important tools for plotting in the assignment problems. However, a full documentation of the plotting facilities in Matplotlib is available on the Axes reference page.

Problem 1 *Use the housing data set to estimate a linear regression of the logarithm of the median house value on the logarithm of the total number of bedrooms. Given the beta estimate, compute* $\mathbb{E}[ln(\textit{Median House Value})\,|ln(\textit{Total Bedrooms})]$ *over a range of values for the total number of bedrooms (say from* 1 *to* 30*) and make a plot with the values.*

Problem 2 *Using the standard error estimates for* $\hat{\beta}$*, create a* 95% *confidence interval* $\mathbb{E}[ln(\textit{Median House Value})\,|ln(\textit{Total Bedrooms})]$*. Extend the plot from the previous problem with the confidence interval.*

Problem 3 *(Shading Confidence Intervals)*

Read the documentation of the `fill_between` *function. Use it to shade the area between the two curves of the confidence interval. Change the* `alpha` *property so that the curve in the middle is also visible.*

Problem 4 *(Shading Confidence Intervals)*

Consider a figure and its `axis` *object, in which one line was plotted. The* `axis.lines` *property returns a list with all lines plotted on that axis (in this case there is only one). Alternatively, you can use the* `axis` *method* `get_lines` *to also get the same list. Given a line object in the* `axis.lines` *list, you can recover its color with the method* `get_color`*. Use the* `get_color` *method to automatically set the color of the shaded confidence interval when calling* `fill_between`*, so that the shaded region has the same color as the line. Also use the* `alpha` *property so that the shaded region is transparent.*

Problem 5 *Download the following file:* `AAPL.csv`*. The file follows the .csv format and contains the stock price of the publicly traded company Apple Inc. The file has 3 columns (no headers): date, time and price. The first column contains the date of a given price in the* `YYYYMMDD` *format. For example, a date of* `20070103` *means January 3rd of 2007. The second column contains the time of a given price in the* `HHMM` *format. For example, a time of* `935` *means that the price in the 3rd column was recorded at 9:35 am. The last column contains the stock price in dollars at the given date and time.*

Load the data into Python.

Problem 6 *The geometric return of a stock over a time interval that starts at time* $t-1$ *and ends at time* t *is given by:*

$$r_t \equiv ln\left(\frac{P_t}{P_{t-1}}\right)$$

where P_t is the price of the stock at time t.

The stock market is open only during some hours of the day, closing at around 4 pm. The data you downloaded has prices sampled every 5-minutes of the day. We can use this data to compute intraday returns and overnight returns. Intraday returns are returns of a stock within a day. In this case, we can compute intraday returns for every 5 minutes interval. Overnight returns are the returns from the time the market closes at a given day, to the time the market opens on the next day.

How many days of data are in the data set? Find answer with code.

How many observations do you have each day? Find answer with code. Remember that you can do integer division with `//`.

Read the documentation of the `numpy.reshape` *function, and pay special attention to the* `order` *input. Also, read the documentation for* `numpy.diff`*, and pay special attention to the* `axis` *input.*

Compute the intraday geometric returns from the stock prices.

Problem 7 *Create a histogram of the intraday geometric returns.*

Problem 8 *(Optional)*

Read about Kernel density estimation. Use the function `scipy.stats.gaussian_kde` *to estimate the density of the distribution of the intraday returns. You will need to install the* `scipy` *package.*

Problem 9 *(Time Series Plot)*

Create a 2D-line plot of the intraday returns. The returns should be reshaped into a vector so that you plot a single time series. For the x-axis use a range of numbers.

Problem 10 *(Time Series Plot)*

To modify the x-axis so that it correctly displays timestamps, we need to use the first two columns of the data set to obtain the timestamps for each return.

Python has a built-in module named `datetime` *which supplies a class for manipulating dates and times. We will use the class* `datetime` *of the module* `datetime` *to create objects that represent the date (year, month and day) and time (hour, minute and second) of each return.*

We can create a `datetime.datetime` *instance in several different ways. The constructor of this class takes the date and time as inputs, and returns the instance:*

```
from datetime import datetime
# Create a date
print(datetime(2017, 11, 20))           # November 20th, 2017
# Create a date with time
print(datetime(2017, 11, 20, 9, 27))  # 9:27 AM (24 hour clock)
print(datetime(2017, 11, 20, 23, 55))  # 11:55 PM (24 hour clock)
# We can also create a datetime from a string
print(datetime.strptime('20171120 0927', '%Y%m%d %H%M'))
```

The string `'%Y%m%d %H%M'` *is a formatting string so that the method* `strptime` *can understand what the number* `'20171120 0927'` *means. You must read this section to understand what is the meaning of the format codes, like* `%Y`.

Why did I type `'0927'` *instead of* `'927'` *when I called the method* `strptime` *above?*

Use the first two columns of `AAPL.csv` *to generate the* `datetime` *instances for each price.*

Hint 1: How would you generate the **`datetime`** *instance for just one observation?*
Hint 2: To loop over two vectors we can the built-in function **`zip`**. *For example:*

```
dates = [20070102, 20070103, 20070104]
times = [930, 1234, 1635]
for date, time in zip(dates, times):
    print(date, time)
```

Problem 11 *(Time Series Plot)*

Since we are plotting the time series for the intraday returns, you need to recover the **`datetime`** *instances for only the appropriate returns.*

Plot the time series of intraday returns, but now use the **`datetime`** *objects for the x-axis. The returns and the dates should be vectors when calling the plot function.*