Python for Machine Learning and Data Analysis

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Spring 2019

Data Visualization with Matplotlib

Lecture 5

Contents

- Introduction to Matplotlib
- Simple Line Plots
- Adjust Plots
- Simple Scatter Plots
- Visualizing Errors
- Density and Contour Plots
- Histograms and Binning
- Three-Dimensional Plotting in Matplotlib

Introduction to Matplotlib

- Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.
- One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends.
- General Matplotlib Tips
  - Importing Matplotlib

    ```
    import matplotlib as mpl

    import matplotlib.pyplot as plt
    ```

  - Setting Styles

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

How to Display Your Plots

- How you view your Matplotlib plots depends on the context.
- The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in
  - a script,
  - a Jupyter shell, or
  - a Jupyter notebook (JupyterLab)

- Plotting from a script:
  - The plt.show() command does a lot under the hood, as it must interact with your system's interactive graphical backend.
  - One thing to be aware of: the plt.show() command should be used only 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 behavior, and should mostly be avoided.
- Plotting from a Jupyter shell
  - %matplotlib → magic command
  - Using plt.show() in Matplotlib mode is not required.
- Plotting from a Jupyter notebook
  - %matplotlib inline → it will lead to static images of your plot embedded in the notebook
  - %matplotlib notebook → it will lead to interactive plots embedded within the notebook

- Saving Figures to File
  - Saving a figure can be done using the savefig() command.
  - In savefig(), the file format is inferred from the extension of the given filename.

    fig.canvas.get_supported_filetypes() → list the supported file types
- Two Interfaces for the Price of One
  - MATLAB-style Interface
    - The interface is *stateful*: it keeps track of the "current" figure and axes.
    - While this stateful interface is fast and convenient for simple plots, it is easy to run into problems.
  - Object-oriented interface
    - The plotting functions are *methods* of explicit Figure and Axes objects.
  - In most cases, the difference is as small as switching plt.plot() to ax.plot(), but there are a few gotchas that we will highlight as they come up in the following sections.

Simple Line Plots

- Perhaps the simplest of all plots is the visualization of a single function *y=f(x)*:
- Creating a figure and an axes for starting all Matplotlib plots
- Use the *ax.plot* or *plt.plot* function to plot some data
- Create a single figure with multiple lines, simply call the plot function multiple times.

Figure vs. Axes

- In Matplotlib, the *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels.
- The *axes* (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.

Adjusting the Plot: Line Colors and Styles

- Function *plt.plot()* takes additional arguments that can be used to specify these by the *color* keyword, which accepts a string argument.
- If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines.
- Similarly, the line style can be adjusted using the *linestyle* keyword.
- To be extremely terse, these *linestyle* and *color* codes can be combined into a single non-keyword argument to the *plt.plot()* function:
  - These single-character color codes reflect the standard abbreviations in the RGB (Red/Green/Blue) and CMYK (Cyan/Magenta/Yellow/blacK) color systems, commonly used for digital color graphics.
  - For details, viewing the docstring of the plt.plot() function using Jupyter's help tools

Adjusting the Plot: Axes Limits

- Adjust axis limits is to use the *plt.xlim()* and *plt.ylim()* methods:
- Display either axis in reverse, simply reverse the order of the arguments.
- A useful related method is *plt.axis():*
  - Set the x and y limits by specifying [xmin, xmax, ymin, ymax]
  - “tight” -- automatically tighten the bounds around the current plot
  - “equal” – ensure an equal aspect ratio so that on your screen, one unit in x is equal to one unit in y
  - More details, using *plt.axis?* to view docstring

Labeling Plots

- Titles and axis labels are the simplest such labels.
  - plt.title() → gives the title label;
  - plt.xlabel() and plt.ylabel() → give the labels of x axis and y axis.
- The position, size, and style of these labels can be adjusted using optional arguments to the function. See the Matplotlib documentation and the docstrings of each of these functions.
- Function plt.legend() to show legend by specifying the label of each line using the label keyword of the plot function.
  - the plt.legend() function keeps track of the line style and color, and matches these with the correct label.

Simple Scatter Plots

- Scatter Plots with plt.plot
  - Same as producing line plots.
- Scatter Plots with plt.scatter
  - more powerful method of creating scatter plots
- The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Exercise:

- Use the Iris data from Scikit-Learn, where each sample is one of three types of flowers that has had the size of its petals and sepals carefully measured.
- Require:
  - 1. give the labels of x and y axes;
  - 2. give the title label as "Relation of Iris Petals and Sepals";
  - 3. try to adjust alpha=0.3, 0.5, s as 200* or 400*
  - and cmap as 'plasma' or 'magma'

Exercise → Summary

- Scatter plot has given us the ability to simultaneously explore four different dimensions of the data:
  - the (x, y) location of each point corresponds to the sepal length and width,
  - the size of the point is related to the petal width, and
  - the color is related to the particular species of flower.
  - Multi-color and multi-feature scatter plots like this can be useful for both exploration and presentation of data.

plot Versus scatter: A Note on Efficiency

- For small amounts of data, both work well.
- As datasets get larger than a few thousand points, plt.plot can be noticeably more efficient than plt.scatter.
  - The reason is that plt.scatter has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually.
  - In plt.plot, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data.
- For large datasets, the difference between these two can lead to vastly different performance, and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

Visualizing Errors

- For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself.
- In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.
- Basic Errorbars
  - plt.errorbar(), with basic options;
  - many additional options to fine-tune the outputs
  - → easily customize the aesthetics of your errorbar plot;
  - → to make the errorbars lighter than the points themselves, especially in crowded plots
  - See docstring of plt.errorbar for more information.

Density and Contour Plots

- Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions.
- There are three Matplotlib functions that can be helpful for this task:
  - plt.contour for contour plots,
  - plt.contourf for filled contour plots, and
  - plt.imshow for showing images.
- Demonstrate a contour plot using a function *z=f(x,y)*

- A contour plot can be created with the *plt.contour* function. It takes three arguments: a grid of x values, a grid of y values, and a grid of z values.
- The x and y values represent positions on the plot, and the z values will be represented by the contour levels.
- Perhaps the most straightforward way to prepare such data is to use the *np.meshgrid* function, which builds two-dimensional grids from one-dimensional arrays.
- Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines.
- Alternatively, the lines can be color-coded by specifying a colormap with the *cmap* argument.

- Function plt.contour(): a filled contour plot
- Function plt.colorbar(): automatically creates an additional axis with labeled color information for the plot.
- Function plt.imshow():
  - A bit “splotchy”: the color steps are discrete rather than continuous.
  - A remedy by setting the number of contours to a very high number is rather inefficient.
  - A better way is to use plt.imshow() function, which interprets a two-dimensional grid of data as an image.

- There are a few potential gotchas with imshow(), however:
  - plt.imshow() doesn't accept an x and y grid, so you must manually specify the extent [xmin, xmax, ymin, ymax] of the image on the plot.
  - plt.imshow() by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.
  - plt.imshow() will automatically adjust the axis aspect ratio to match the input data; this can be changed by setting, for example, plt.axis(aspect='image') to make x and y units match.

Histograms, Binnings, and Density

- A simple histogram can be a great first step in understanding a dataset: *plt.hist()*
- The hist() function has many options to tune both the calculation and the display. See details in *plt.hist* docstring.
- If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the *np.histogram()* function is available.

Two-Dimensional Histograms and Binnings

- plt.hist2d: Two-dimensional histogram
  - create a tessellation of squares across the axes
- plt.hexbin: Hexagonal binnings
  - represent a two-dimensional dataset binned within a grid of hexagons.
  - plt.hexbin has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).
- Kernel density estimation
  - Another common method of evaluating densities in multiple dimensions is kernel density estimation (KDE).
  - KDE can be thought of as a way to "smear out" the points in space and add up the result to obtain a smooth function.
  - One extremely quick and simple KDE implementation exists in the *scipy.stats* package.

Three-Dimensional Plotting in Matplotlib

- Three-dimensional plots are enabled by importing the mplot3d toolkit, included with the main Matplotlib installation.

```
from mpl_toolkits import mplot3d
```

- Once this submodule is imported, a three-dimensional axes can be created by passing the keyword *projection='3d'* to any of the normal axes creation routines.
- 3-D plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook.

Three-dimensional Points and Lines

- The most basic three-dimensional plot is a line or collection of scatter plot created from sets of *(x, y, z)* triples.
- In analogy with the more common two-dimensional plots discussed earlier, these can be created using the *ax.plot3D* and *ax.scatter3D* functions.
- The call signature for these is nearly identical to that of their two-dimensional counterparts, so you can refer to Simple Line Plots and Simple Scatter Plots for more information on controlling the output.
- Here we'll plot a trigonometric spiral, along with some points drawn randomly near the line.

Three-dimensional Contour Plots

- “mplot3d” contains tools to create 3-D relief plots using the same inputs as in Density and Contour Plots.
- Like 2-D *ax.contour* plots, *ax.contour3D* requires all the input data to be in the form of 2-D regular grids, with the *Z* data evaluated at each point.
- Here we'll show a three-dimensional contour diagram of a three-dimensional sinusoidal function:
- Use the *view_init* method to set the elevation and azimuthal angles if default viewing angle is not optimal.

Wireframes and Surface Plots

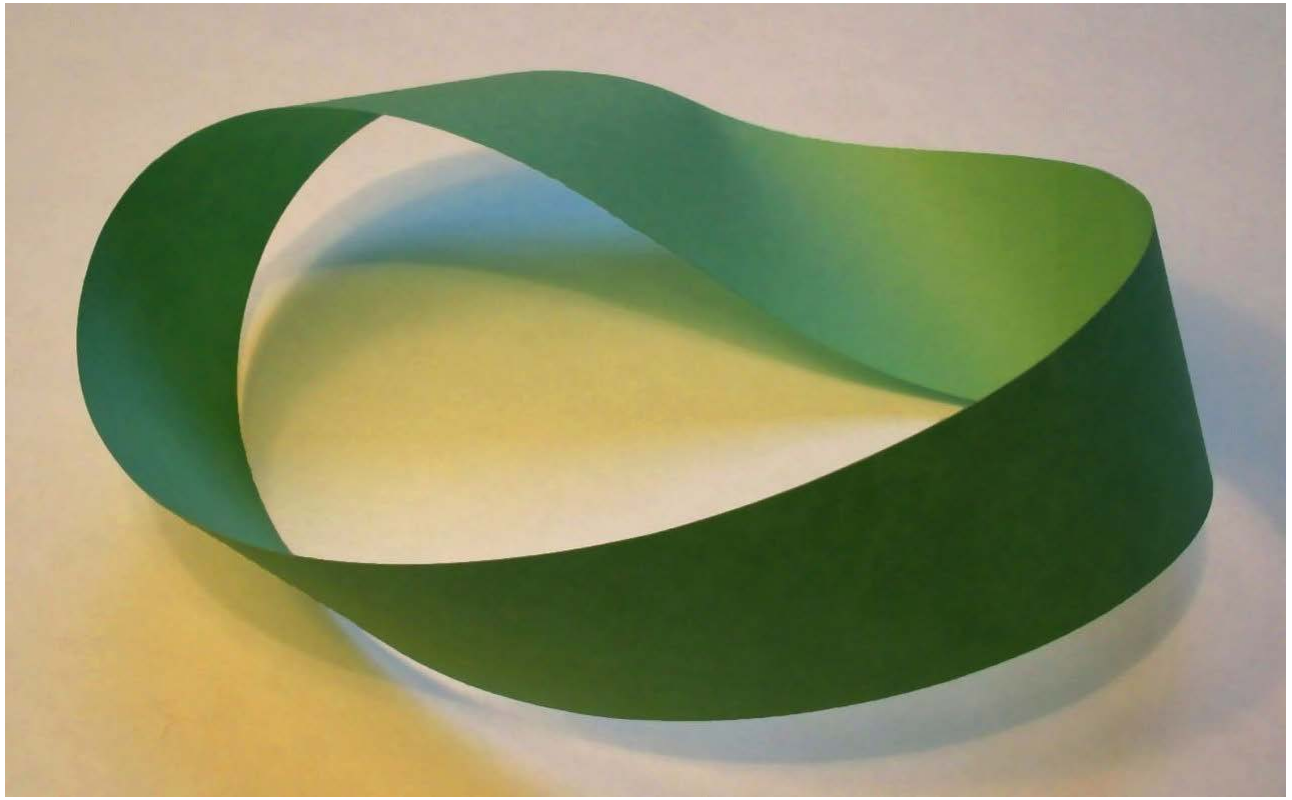
- Wireframes and surface plots take a grid of values and project it onto the specified three-dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize.
- A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized.
- Note that though the grid of values for a surface plot needs to be two-dimensional, it need not be rectilinear. And the surface3D plot can give us a slice into the function for visualizing.

Surface Triangulations

- For some applications, the evenly sampled grids required by the above routines is overly restrictive and inconvenient.
- In these situations, the triangulation-based plots can be very useful.
- What if rather than an even draw from a Cartesian or a polar grid, we instead have a set of random draws?
- We could create a scatter plot of the points to get an idea of the surface we're sampling from: *ax.scatter()*
- Function *ax.plot_trisurf()* creates a surface by first finding a set of triangles formed between adjacent points
  - remember that x, y, and z here are one-dimensional arrays.
- Not clean, the flexibility of such a triangulation allows for some really interesting three-dimensional plots.

Example: Visualizing a Möbius strip

- A Möbius strip is similar to a strip of paper glued into a loop with a half-twist.
- Topologically, it's quite interesting because despite appearances it has only a single side!



- Here we will visualize such an object using Matplotlib's three-dimensional tools.
- The key to creating the Möbius strip is to think about it's parametrization:
  - it's a two-dimensional strip, so we need two intrinsic dimensions. Let's call them θ, which ranges from 0 to 2π around the loop, and *w* which ranges from -1 to 1 across the width of the strip:
- Now from this parametrization, we must determine the *(x, y, z)* positions of the embedded strip.
- Thinking about it, we might realize that there are two rotations happening:
  - one is the position of the loop about its center (called θ), while
  - the other is the twisting of the strip about its axis (call this $\emptyset$).
  - For a Möbius strip, we must have the strip makes half a twist during a full loop, or $\Delta\emptyset = \Delta\theta/2$

- Define *r* → the distance of each point from the center, and use this to find the embedded *(x,y,z)* coordinates:
- Finally, to plot the object, make sure the triangulation is correct. The best way is to:
  - define the triangulation *within the underlying parametrization*, and then
  - let Matplotlib project this triangulation into the three-dimensional space of the Möbius strip.

- Combining all of these techniques, it is possible to create and display a wide variety of three-dimensional objects and patterns in Matplotlib.

Further Resources: Matplotlib Resources

- Docstring
- Matplotlib online documentation: https://matplotlib.org/
- See in particular the Matplotlib gallery:
  https://matplotlib.org/gallery.html
- it shows thumbnails of hundreds of different plot types, each one linked to a page with the Python code snippet used to generate it. In this way, you can visually inspect and learn about a wide range of different plotting styles and visualization techniques.

Other Python Graphics Libraries

- Bokeh: https://bokeh.pydata.org/en/latest/
  - Bokeh is a JavaScript visualization library with a Python frontend that creates highly interactive visualizations capable of handling very large and/or streaming datasets.
- Plotly: https://plot.ly/
  - Plotly is the eponymous open source product of the Plotly company, and is similar in spirit to Bokeh.
- VisPy: http://vispy.org/
  - Vispy is an actively developed project focused on dynamic visualizations of very large datasets. Because it is built to target OpenGL and make use of efficient graphics processors in your computer, it is able to render some quite large and stunning visualizations.
- Vega: https://vega.github.io/
- Vega-Lite: https://vega.github.io/vega-lite/
  - They are declarative graphics representations, and are the product of years of research into the fundamental language of data visualization.

The End!

Assignment:

1) Practice the example code I ran in this class.

2) Homework will be assigned by the end of Feb. 10th ; and should be finished by Feb. 19th.

3) Tell your myNEU username by the end of Feb. 10th for creating the accounts on Discovery Cluster.