Data Visualization Techniques - Scatter, Line, and Histogram

We'll cover the following

Visualization Techniques

1. Scatter Plots

Scatter plots are deceptively simple and commonly used, but simple doesn't mean that they aren't useful!

In a scatter plot, data points are represented individually with a dot, circle, or some other shape. These plots are great for showing the relationship between two variables as we can directly see the raw distribution of the data.

To create a scatter plot in Matplotlib we can simply use the `scatter` method. Let's see how by creating a scatter plot with randomly generated data points of many colors and sizes.

First, let's generate some random data points, *x* and *y*, and set random values for colors and sizes because we want a pretty plot:

A complete "runnable" example is at the end.

```
# Generating Random Data
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000 * np.random.rand(
```

Now that we have our two variables *(x, y)* and the colors and sizes of the points, we can call the scatter method like this:

```
plt.scatter(x, y, c=colors, s=sizes, alpha=0.2, cmap='viridis')
```

`alpha = 0.2`: The optional alpha input parameter (optional) of the scatter method allows us to adjust the transparency level so that we can view overlapping data points. You will understand this concept better, visually, in the final output.

`cmap='viridis'`: Matplotlib provides a wide range of colormaps, an optional parameter, for creating visually appealing plots. For example, you can set `cmap=plt.cm.Blues`, `cmap=plt.cm.autumn`, or `cmap=plt.cm.gist_earth`.

The color argument, `c`, is automatically mapped to a color scale, and the size argument, `s`, is in pixels. To view the color scale next to the plot on the right-hand side, we can use the `colobar()` command:

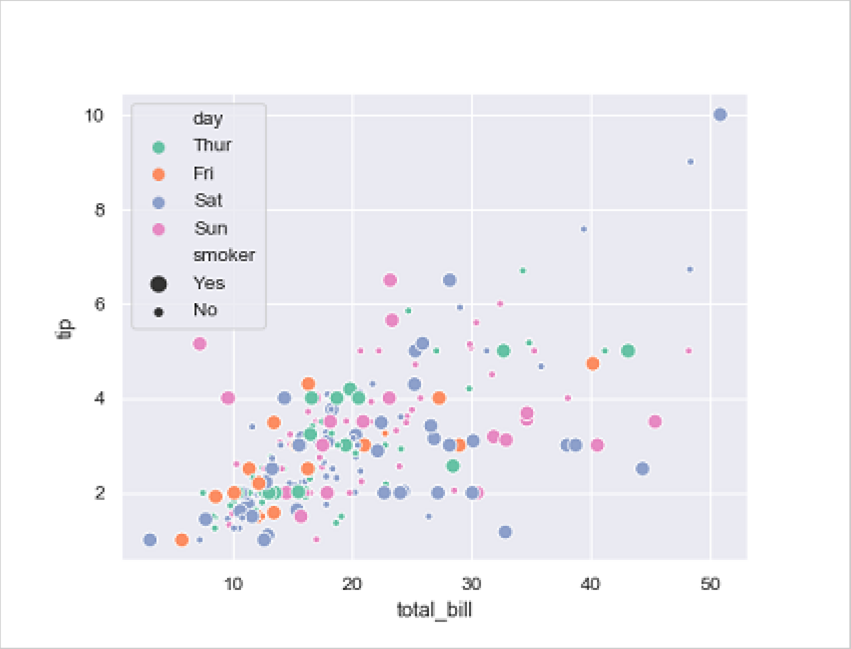
```
plt.colorbar()
```

Now let's put it all together and run the full code to see the output. Note that we will save the output to a file to display the plot with Educative's code widget.

As an exercise, try changing the parameter values to understand their effect on the output.

```
# 1. Importing modules
import matplotlib.pyplot as plt
import numpy as np

# 2. Generating some random data
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

# 3. Create a scatterplot using the a colormap.
plt.scatter(x, y, c=colors, s=sizes, alpha=0.2,cmap='viridis')
plt.colorbar();  # To show the color scale

plt.savefig("output/scatter.png")
```

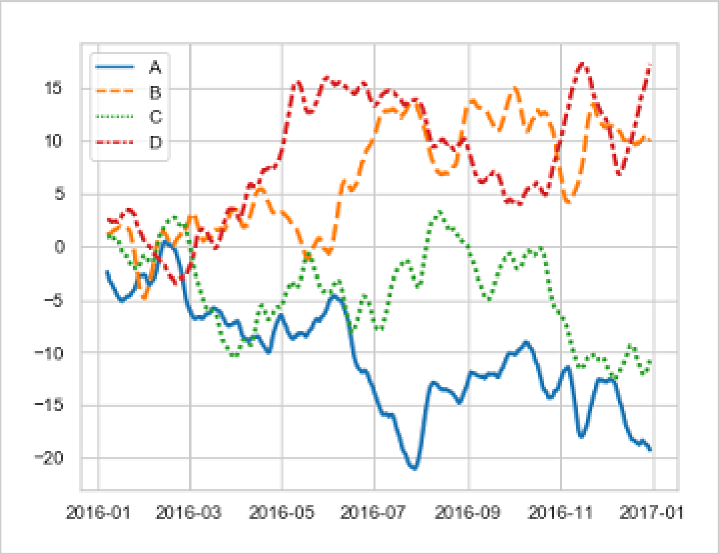
We can see that this scatter plot has given us the ability to **simultaneously explore four different dimensions of the data**. If this was a real data set, the *x*, *y* location of each point can be used to represent two distinct features. The size of the points can be used to represent a third feature, and the color mapping can be used to represent different classes or groups of the data points. **Multicolor and multifeature scatter plots** like this are very useful both for exploring and presenting data. Here is an interesting example:



Example of Scatter Plot Analysis

2. Line Plots

A line plot displays data points connected by straight line segments, instead of showing them individually. This type of plot is useful for finding relationships between datasets .We can easily see if values for one dataset are affected by the variations in the other. This will tell us if they are correlated or not. For example in the plot below, *D* and *C* don't seem to be going on own paths independently:



Example of Line Plots Analysis

Say we have a dataset with the population of two cities over time and we want to see if there is some correlation in their population sizes. We can use the `plt.plot()` function to plot population against time. We would put time on the *x-axis* and population on the *y-axis*.

In order to compare the population sizes of the two cities, we want the population data for both to be represented on the same plot. To create a single figure with multiple lines, we can just call the plot function multiple times. And then to distinguish between the two, we can adjust the colors using the `color` keyword.

Finally, to label the plot, title and axis labels, we can simply call:

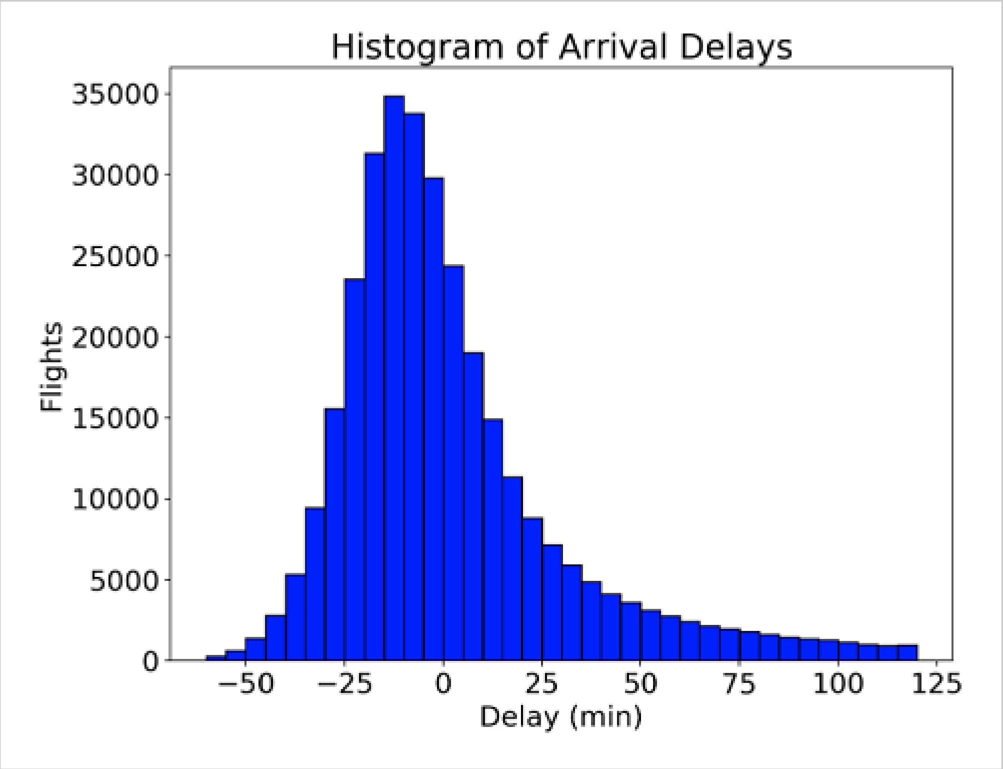
```
plt.title("Some cool title for the plot")
plt.xlabel("Some x-axis label")
plt.ylabel("Some y-axis label");
```

Let’s look at this from start to finish:

```
# 1. Importing modules
import matplotlib.pyplot as plt

# 2. Getting the Data
year = [1970, 1980, 1990, 2000, 2010, 2020]
pop_A = [4.9, 5.3, 7.1, 10.7, 13.5, 15.6]
pop_B = [44.4, 55.6, 69.7, 87.1, 95.4, 100.5]

# 3. Visualising the Data
plt.plot(year, pop_A, color='g')
plt.plot(year, pop_B, color='r')
plt.xlabel('Countries')
plt.ylabel('Population in million')

plt.title('Populations over the years')

plt.savefig("output/line.png")
```

3. Histograms

Histograms are useful for understanding the distribution of data points. Basically, a histogram is a plot where data is split into bins. A bin is a range of values or intervals. For each bin, we get the count of how many values fall into it. The *x-axis* represents bin ranges while *y-axis* shows frequency. The bins are usually specified as consecutive, non-overlapping intervals of a variable. For example, in the plot below, we have information about how many flights were delayed for a range of time intervals:



Example Histogram

We can plot histograms using the `hist()` function:

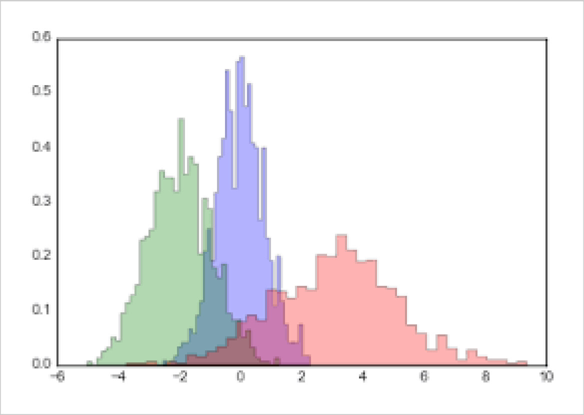
```
import numpy as np
import matplotlib.pyplot as plt
```

```
data = np.random.randn(1000)
plt.hist(x=data);


plt.savefig("output/hist1.png")
```

hist() has many options that can be used to tune both the calculation and the display. Let’s see an example of a more customized histogram for the same data:

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

plt.savefig("output/hist2.png")
```

To create multiple histograms in the same plot, we can either call the *hist()* function multiple times or pass all the input values to it at once. The `x` parameter of *hist()* can take both a single array or a sequence of arrays as input.



0.6
0.5
0.4
0.3
0.2
0.1
0.0
−6
−4
−2
0
2
4
6
8
10