Machine Learning Basics Tutorials

NumPy and Matplotlib

Contents

1 Acknowledgements

This tutorial for Machine Learning basics has been created by Danyal Saqib under the supervision of Dr. Wajahat Hussain for the School of Electrical Engineering and Computer Sciences (SEECS) at the National University of Sciences and Technology, NUST. Reporting any errors or omissions within this text shall be appreciated.

The github repository for this Machine Learning Basics tutorial series can be found at: https://github.com/danyalsaqib/Machine-Learning-Basics

2 Introduction

This main purpose of this whole text is to take you through the basics of some of the most popular programming packages used in Python for various tasks. This particular section gives a brief overview of each of 2 very important Python packages.

2.1 NumPy

NumPy (Numerical Python) is a Python library that deals with arrays. It helps in creation of array objects, and contains functions and routines for dealing and processing these array objects. The operations that can be performed on these array objects may be mathematical or logical, which come in very handy during machine learning implementation.

2.2 Matplotlib

Matplotlib is a very popular python package used for data visualization. It integrates seamlessly with both Python in itself, and the NumPy library. It can make highly customizable 2D and 3D graphs and charts, and is an extremely useful tool for visualization of your data.

3 Numpy

3.1 Numpy Basics

Now that we have a basic idea of what to expect from each of the libraries, we shall start the actual tutorial by first looking at NumPy. Learning NumPy first will allow us to study Matplotlib in a much more comprehensive manner. We assume that you already have python setup on your device. If you are using Google Colab for this, the environment shall already be setup for you. If not, look up a guide for installing these libraries on your device.

The first thing to do here is to import NumPy. You can import NumPy as it is, or import it as a shorthand version of its name:

```
import numpy
import numpy as np
```

When we import numpy as np, we can basically use np wherever we want to refer to numpy. Thus, np acts as a sort of a shorthand name, or 'nickname' for the library.

As NumPy deals with arrays, the most basic task to do in NumPy after importing it is to create an array. In NumPy, the array object is called 'ndarray', or N-Dimensional array. We will start by creating a one dimensional array:

```
import numpy as np
a = np.array([1,2,3])
print a
```

Output

```
[1, 2, 3]
```

NumPy arrays also have an optional parameter called 'dtype', that can be used to specify the data type:

Code: basicArrays.py

```
import numpy as np

a = np.array([1,2,3], dtype = int)
print("a: ", a)

b = np.array([1,2,3], dtype = float)
print("b: ", b)

c = np.array([1,2,3], dtype = complex)
print("c: ", c)
```

Output

```
a:  [1 2 3]
b:  [1. 2. 3.]
c:  [1.+0.j 2.+0.j 3.+0.j]
```

We will now look at multidimensional arrays, and how to get some useful information about these arrays:

Code: multiDimArrays.py

```python
import numpy as np

# Define a function to get array information
def getarrayinfo(arr):        # Takes an array as input
   print("\n**********")
   print("array:\n", arr)        # Prints array
   print("dimensions: ", arr.ndim) # Prints array dimensions
   print("shape: ", arr.shape)   # Prints array shape
   print("datatype: ", arr.dtype) # Prints array datatype

a = np.array([1,2,3])
getarrayinfo(a)

b = np.array([[1,2,3], [4,5,6]])
getarrayinfo(b)

c = np.array([[1,2,3], [4,5,6], [7,8,9]], dtype = float)
getarrayinfo(c)

d = np.array([[[1,2,3]], [[4,5,6]], [[7,8,9]]], dtype = np.int32)
getarrayinfo(d)
```

Output

```
**********
array:
 [1 2 3]
dimensions:  1
shape:  (3,)
datatype:  int64
```

```
**********
array:
 [[1 2 3]
 [4 5 6]]
dimensions:  2
shape:  (2, 3)
datatype:  int64

**********
array:
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
dimensions:  2
shape:  (3, 3)
datatype:  float64

**********
array:
 [[[1 2 3]]

 [[4 5 6]]

 [[7 8 9]]]
dimensions:  3
shape:  (3, 1, 3)
datatype:  int32
```

3.2 Indexing and Special Arrays

To extract or find a particular item in the array, we use indexing. The syntax for the index is [row, col]. We can use our previously created 'getarrayinfo' function to better understand this concept. So to demonstrate this in code:

Code: indexing.py

```python
# Creating an array
d = np.array([[1,2,3], [4,5,6], [7,8,9]], dtype = np.int32)
getarrayinfo(d)

# Remember that indexes in Python start from 0
item12 = d[0, 1]
print("First Row, Second Column: ", item12)
```

```
item33 = d[2, 2]
print("Third Row, Third Column: ", item33)
item23 = d[1, 2]
print("Second Row, Third Column: ", item23)

row2 = d[1, :]              # Using ':' gets all the elements in a row or column
print("Second Row: ", row2)
col3 = d[:, 2]
print("Third Column: ", col3) # Extracting a column using the ':' operator
```

Output

```
**********
array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
dimensions:  2
shape:  (3, 3)
datatype:  int32
First Row, Second Column:  2
Third Row, Third Column:  9
Second Row, Third Column:  6
Second Row:  [4 5 6]
Third Column:  [3 6 9]
```

Similarly for multidimensional arrays, the syntax for indexing is [dim, row, col]. Go ahead and try this for yourself.

We will now look at some very useful functions for creating arrays with predefined values:

Code: specialArrayFunctions.py

```
import numpy as np

# NumPy function for creating arrays with all zeros
a = np.zeros(5)
print("a: \n", a)
b = np.zeros((3, 3))
print("\nb: \n", b)

# NumPy function for creating arrays with all ones
c = np.ones(10)
print("c: \n", c)
d = np.ones((3, 3, 2))
print("\nd: \n", d)
```

```python


# NumPy function for creating identity matrix
i = np.identity(4)
print("i: \n", i)
```

Output

```
a:
 [0. 0. 0. 0. 0.]

b:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
c:
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

d:
 [[[1. 1.]
  [1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]
  [1. 1.]]]
i:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Finally, if you want to make a copy of an array, you have to be careful:

Code: copyCommand.py

```python
import numpy as np

# Trying to create a copy
a = np.array([1, 2, 3])
```

```
b = a
b[0] = 100
print("b: ", b)
print("a: ", a)
print("\nThis is because the '=' operator does not actually make a copy, but just
    tells NumPy that b points to the same thing that a does. Thus to make a proper
    copy, we have to use the 'copy()' command:\n")
# Making copy the proper way
a = np.array([1, 2, 3])
b = a.copy()
b[0] = 100
print("b: ", b)
print("a: ", a)
```

Output

```
b:  [100   2   3]
a:  [100   2   3]

This is because the '=' operator does not actually make
a copy, but just tells NumPy that b points to the same
thing that a does. Thus to make a proper copy, we have
to use the 'copy()' command:

b:  [100   2   3]
a:  [1  2  3]
```

3.3 Mathematical and Logical Operators

Performing Element-Wise operations on NumPy arrays is very easy:

Code: elementOperations.py

```
import numpy as np

# Creating an Array
a = np.array([1, 2, 3, 4, 5], dtype = np.float)
print("a: \n", a)

# Printing various element-wise operations
print("\na + 2: \n", a + 2)
print("\na - 4: \n", a - 4)
print("\na * 3: \n", a * 3)
print("\na / 5: \n", a / 5)
```

```python

# Matrix element wise addition
b = np.ones(5)
print("\nb: \n", b)
print("\na + b: \n", a + b)

# Matrix element wise multiplication
c = np.array([2, 4, 6, 8, 10])
print("\nc: \n", c)
print("\na * c: \n", a * c)
```

Output

```

a:
 [1. 2. 3. 4. 5.]

a + 2:
 [3. 4. 5. 6. 7.]

a - 4:
 [-3. -2. -1.  0.  1.]

a * 3:
 [ 3.  6.  9. 12. 15.]

a / 5:
 [0.2 0.4 0.6 0.8 1. ]

b:
 [1. 1. 1. 1. 1.]

a + b:
 [2. 3. 4. 5. 6.]

c:
 [ 2  4  6  8 10]

a * c:
 [ 2.  8. 18. 32. 50.]
```

You are encouraged to take a look at the many other operations to be performed on arrays, such as taking exponents, and computing sinusoids.

Linear Algebra computations are an important part of Machine Learning and Data Science in general. Let's take a look at some of the most commonly used functions:

Code: linearAlgebra.py

```python
import numpy as np

a = np.array([[1,2,3], [4,5,6], [7,8,9]])
print("a:\n", a)

b = np.array([[5,4], [3,2], [1,0]])
print("\nb:\n", b)

# Remember that for matrix multiplication, the number of columns of first matrix
    should be equal to the number of rows of second matrix
c = np.matmul(a, b)
print("\nc:\n", c)

# Multiplying with identity matrix
i = np.identity(3)
print("\ni:\n", i)
m = np.matmul(a, i)
print("\nm:\n", m)
```

Output

```
a:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

b:
 [[5 4]
 [3 2]
 [1 0]]

c:
 [[14  8]
 [41 26]
 [68 44]]

i:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
m:
 [[1.  2.  3.]
 [4.  5.  6.]
 [7.  8.  9.]]
```

Once again, you are encouraged to look up more Linear Algebra functions on your own.

3.4 Statistics and Reorganization

Statistical commands are very commonly used, and NumPy makes their implementation very simple:

```
import numpy as np

a = np.array([[1,2,3], [4,5,6], [7,8,9]])
print("a:\n", a)

# Performing and Printing Statistical Operations
print("\nmin: ", np.min(a))
print("\nmax: ", np.max(a))
print("\nsum: ", np.sum(a))
```

Output

```
a:
 [[1  2  3]
 [4  5  6]
 [7  8  9]]

min:  1

max:  9

sum:  45
```

A lot of reshaping is done on arrays during Machine Learning. The implementation in NumPy is given below:

Code: reshaping.py

```
import numpy as np

```

```python
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,10,12], [13,14,15,16]])
print("a:\n", a)
print("\nshape: ", a.shape)

#Remember when transforming an array into a different shape, the total number of
    elements don't change
b = a.reshape(2, 8)
print("\nb:\n", b)
print("\nshape: ", b.shape)

c = a.reshape(8, 2)
print("\nc:\n", c)
print("\nshape: ", c.shape)

d = a.reshape(1, 16)
print("\nd:\n", d)
print("\nshape: ", d.shape)

e = a.reshape(2, 2, 2, 2)
print("\ne:\n", e)
print("\nshape: ", e.shape)
```

Output

```
a:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 10 12]
 [13 14 15 16]]

shape:  (4, 4)

b:
 [[ 1  2  3  4  5  6  7  8]
 [ 9 10 10 12 13 14 15 16]]

shape:  (2, 8)
```

```
c:
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [10 12]
 [13 14]
 [15 16]]

shape:  (8, 2)

d:
 [[ 1  2  3  4  5  6  7  8  9 10 10 12 13 14 15 16]]

shape:  (1, 16)

e:
 [[[[ 1  2]
   [ 3  4]]

  [[ 5  6]
   [ 7  8]]]


 [[[ 9 10]
   [10 12]]

  [[13 14]
   [15 16]]]]

shape:  (2, 2, 2, 2)
```

With this, the NumPy section of this tutorial comes to an end. By now, you hopefully have a grasp of the fundamentals of this library. NumPy can be slightly daunting at first, but rest assured, practice makes perfect!
NumPy is almost certainly the most challenging part of this tutorial, but once you become proficient, the rest of the libraries become very easy to use.

In the next section, we will take a look at Matplotlib, and how it can be integrated seamlessly with NumPy.

4 Matplotlib

4.1 Matplotlib Basics

Once again, the first thing we will do is import both NumPy and Matplotlib. Note that we will mostly be working with a collection of functions within Matplotlib called Pyplot. Go ahead, and import these 2 as shown below.

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

Note that we use the the ‘plt’ shorthand when importing Pyplot, as is convention.

We will now learn how to make basic graphs in Matplotlib. For this example, we are going to use a NumPy array, and plot it against it’s sine graph:

Code: basicPlot.py

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

# Creating an array that ranges from 0 to 5, and has an increment of 0.1 for each
    element
x = np.arange(0, 5, 0.1)

# Taking sine of array x
y = np.sin(x)

# Plotting and showing results
plt.plot(x, y)
plt.show()
```

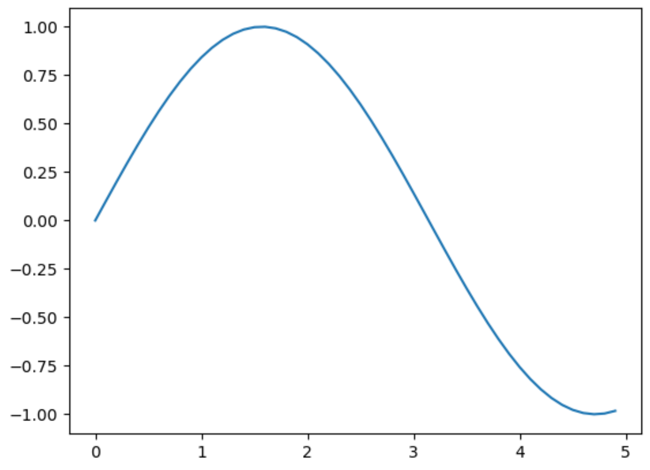

Figure 1: Sinusoidal Graph

That was easy! But we may want to properly label our axes, and give our plot a title. That is easily done:

Code: plotLabels.py

```python
# Plotting and showing results
plt.plot(x, y, label = 'sin(x)')

# Adding labels to axes, and adding a title
plt.title('Sine Graph')
plt.xlabel('x Axis')
plt.ylabel('Y Axis')
plt.legend()

plt.show()
```

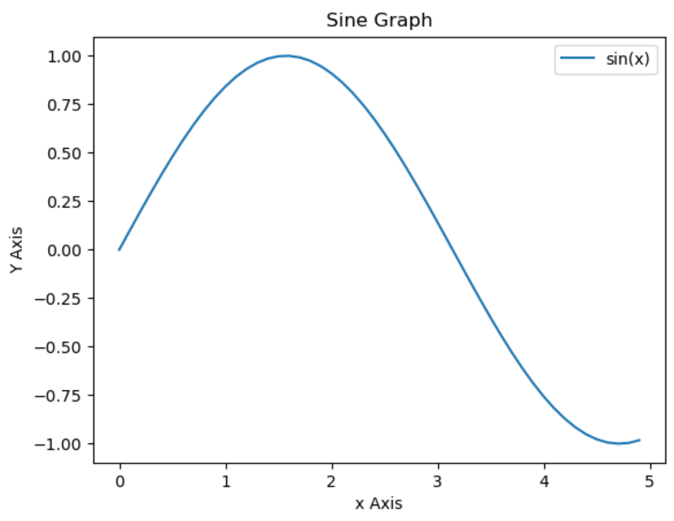

Figure 2: Labelled Sinusoidal Graph

4.2 Multiple Graphs and Plots

We will now see how to plot multiple graphs on the same plot, while also customizing each line's color and style:

Code: multiGraphs.py

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

# Creating an array that ranges from 0 to 5, and has an increment of 0.1 for each
    element
```

```
x = np.arange(0, 5, 0.1)

# Taking sine of array x
y = np.sin(x)

# Plotting Sine of x
plt.plot(x, y, label = 'sin(x)', color = 'blue', linestyle = '-')

# Plotting Cosine of x
z = np.cos(x)
plt.plot(x, z, label = 'cos(x)', color = 'red', linestyle = '--')

plt.title('Sine and Cos Graphs')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

plt.legend()
plt.show()
```

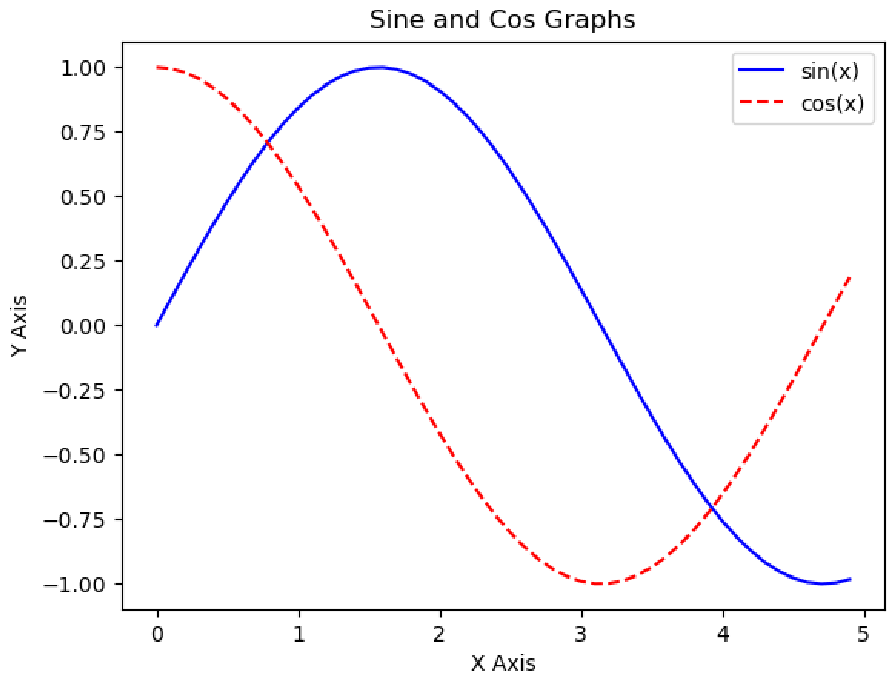

Figure 3: Two Graphs on a single plot

The figure shown has two plots, each labelled with their own different function name, and each with their own style. Note that we have customized the color and linestyle of each of the graphs. There is much more customization available ranging from font styles and sizes to marker styles and linewidths. As always, you are encouraged to look through the official documentation for further guidance.

We will now take a look at how to add multiple plots to the same figure. The code is again very intuitive, and the basic idea remains the same:

Code: multiPlots.py

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

# Creating an array that ranges from 0 to 5, and has an increment of 0.1 for each
    element
x = np.arange(0, 20, 0.1)

# Defining some functions of x
y1 = np.sin(x) # Sine Function
y2 = np.cos(x) # Cos Function
y3 = x        # Graph of y = x
y4 = x**2     # Graph of y = xˆ2

fig, axes = plt.subplots(2, 2)   # Defining our figure’s columns and rows

axes[0, 0].plot(x, y1, color = 'blue')
axes[0, 0].set(xlabel = 'X Axis', ylabel = 'Sine Function')

axes[0, 1].plot(x, y2, color = 'red')
axes[0, 1].set(xlabel = 'X Axis', ylabel = 'Cos Function')

axes[1, 0].plot(x, y3, color = 'green')
axes[1, 0].set(xlabel = 'X Axis', ylabel = 'Linear Function')

axes[1, 1].plot(x, y4, color = 'black')
axes[1, 1].set(xlabel = 'X Axis', ylabel = 'Quadratic Function')

plt.legend()
plt.show()
```

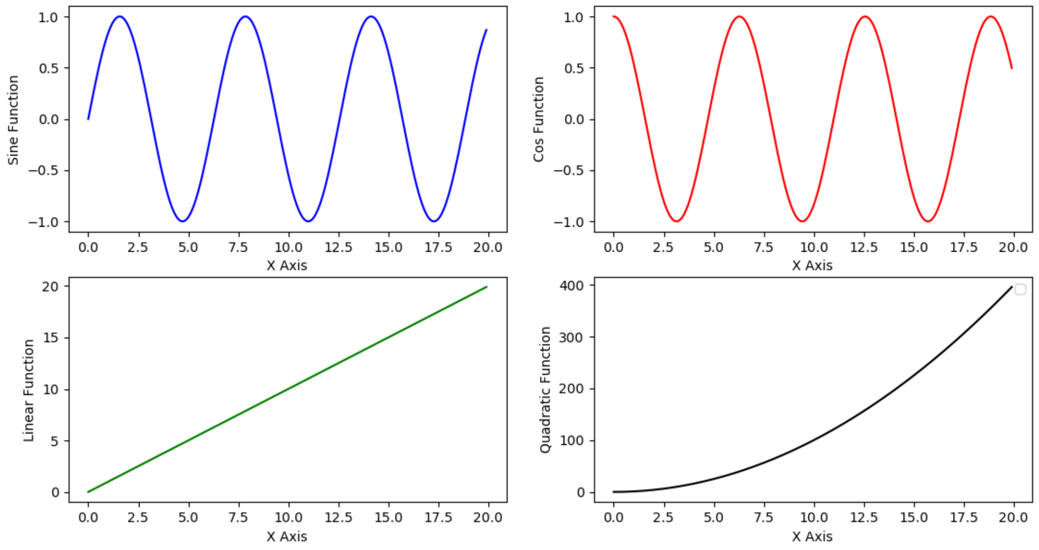

Figure 4: Four plots on a single figure

4.3 Other Important Graphs

What we just created in the previous sections were line plots. We will now take a look at some other important types of graphs.

A very commonly used graph is the stem plot:

Code: stemPlot.py

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

# Creating an array that ranges from 0 to 5, and has an increment of 0.1 for each
    element
x = np.arange(0, 20, 0.4)

# Plotting Sine function as stem plot
y = np.sin(x)
plt.stem(x, y, label = 'sin(x)')

plt.title('Sine Graph - Stem Plot')
plt.xlabel('x Axis')
plt.ylabel('Y Axis')

plt.legend()
plt.show()
```

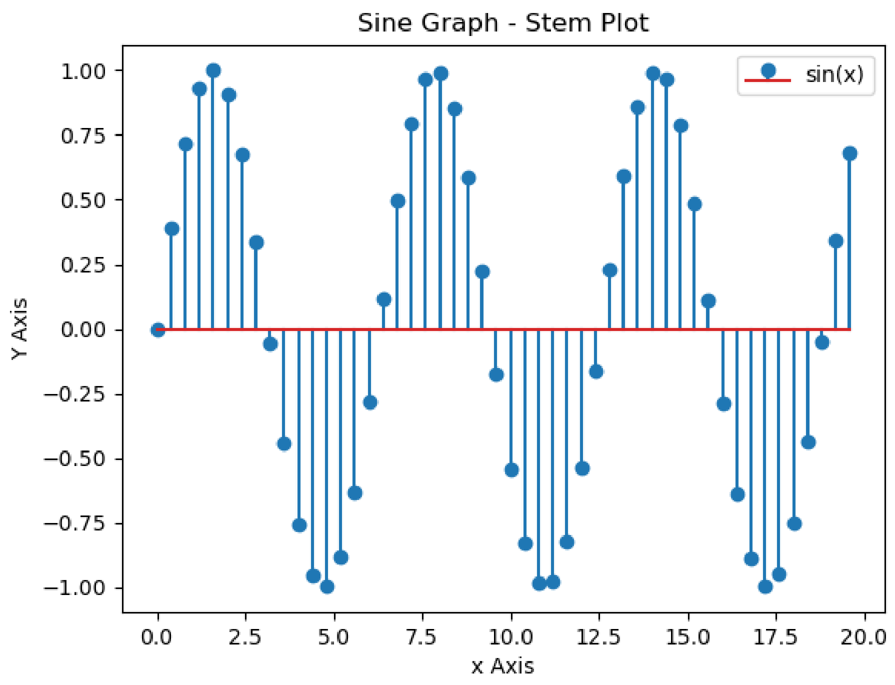

Figure 5: Stem Plot

Another commonly used plot is the scatter plot. This is especially useful in visualizing problems relating to Machine Learning, such as Linear Regression and Classification problems. We will see its utility via an example.

Let's say that we have the data for students of a particular school. It shows how much a particular students scored in a Math test, and how much they scored in an English Test. The data could look something like:

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

math_scores = np.array([78, 93, 54, 91, 82, 98])
english_scores = np.array([81, 65, 70, 68, 79, 94])
```

The array shows corresponding scores for students. A very convenient way to visualize this data would be through a scatter plot, as it would show the data for both math and english scores for every student.
The code to generate a scatter plot is given below:

Code: scatterPlot.py

```python
# Scatter Plot
plt.scatter(math_scores, english_scores, color = 'red')

```

```
# Setting axis ranges from 60 to 100
plt.xlim(60, 100)
plt.ylim(60, 100)

# Making it look good
plt.title('Student Scores')
plt.xlabel('Math Scores')
plt.ylabel('English Scores')

plt.show()
```

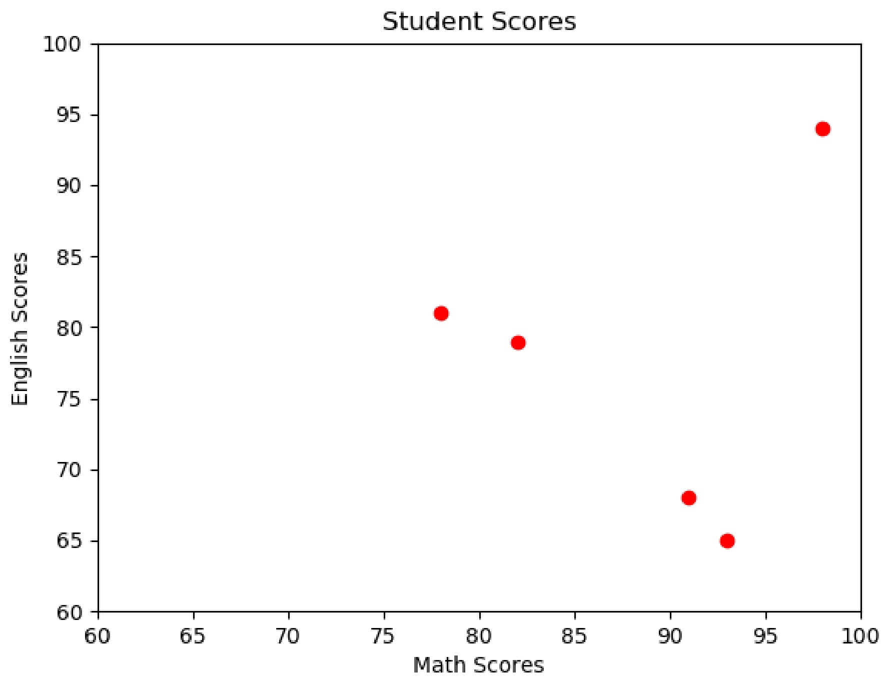

Figure 6: Scatter Plot for Student Scores

You see how it is very easy to visualize this data through a scatter plot.

Now, let us say that there is another school, School 2, which also has data for it's students' math and english scores. We can plot another scatter plot on the same diagram very easily:

Code: schoolExample.py

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

# Scores for School 1
math_scores_1 = np.array([78, 93, 54, 91, 82, 98])
```

```
english_scores_1 = np.array([81, 65, 70, 68, 79, 94])

# Scores for School 2
math_scores_2 = np.array([80, 72, 86, 71, 62, 84])
english_scores_2 = np.array([83, 80, 89, 98, 87, 95])

# Scatter Plot for both schools
plt.scatter(math_scores_1, english_scores_1, label = 'School 1', color = 'red',
    marker = 'o')
plt.scatter(math_scores_2, english_scores_2, label = 'School 2', color = 'blue',
    marker = 'x')

# Setting axis ranges from 60 to 100
plt.xlim(60, 100)
plt.ylim(60, 100)

# Making it look good
plt.title('Student Scores')
plt.xlabel('Math Scores')
plt.ylabel('English Scores')

plt.legend(loc = 'lower left')
plt.show()
```

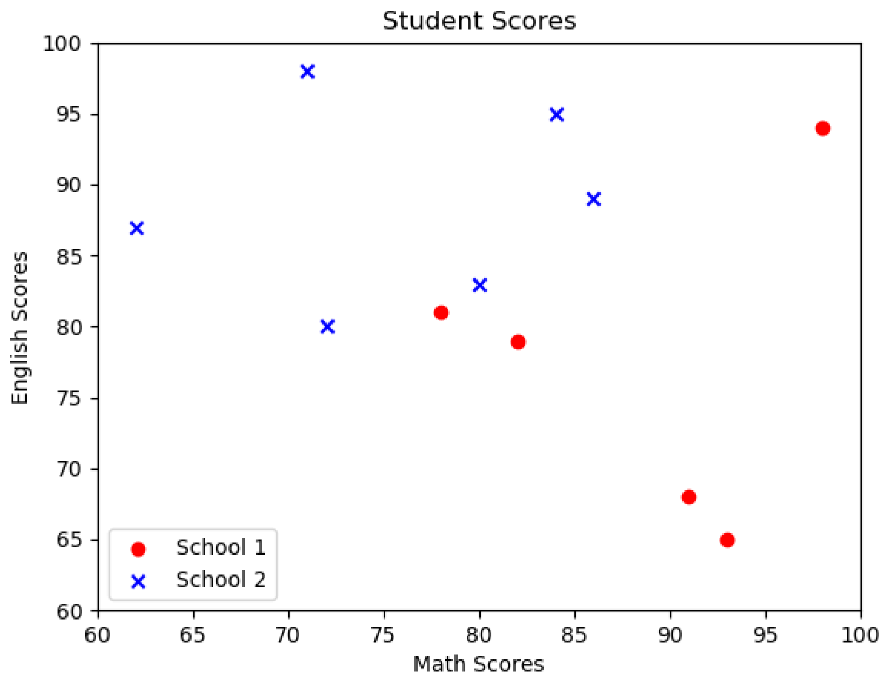

Figure 7: Scatter Plot for both Schools

Seeing this figure, you should immediately start noticing a trend; Students of School 1 did much better than students of School 2 in Maths, whereas students of School 2 outperformed students of School 1 in English. If you were a data scientist working on a survey of different schools, you could discern a pattern and tell each school what their strengths and weaknesses were. Such is the power of Data Visualization.

This brings us to the end of the Matplotlib section of the tutorial. You should be at home with both NumPy and Matplotlib by now, and should be able to code comfortably with these libraries.

5 Concluding Remarks

We now conclude this particular Machine Learning basics tutorial. While it is not possible to cover every single thing about these libraries in these tutorials, they will hopefully give you a strong starting point in getting comfortable with them. These tutorial documents combined with the assignments will hopefully be a very effective and valuable learning experience for you.