MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Machine Learning

Lecture 3: NumPy

Introduction to NumPy

- NumPy is an open-source add-on module to Python that provide common mathematical and numerical routines in **pre-compiled**, fast functions.

- The NumPy (Numeric Python) package provides basic routines for **manipulating large arrays** and matrices of numeric data.

- At the core of the NumPy package, is the **Ndarray** object.
  - This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.
- NumPy arrays facilitate mathematical and other types of operations on large numbers of data.
  - Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
  - Minimize dependency on **loops**
- This link provides a good tutorial on getting started with NumPy.

NumPy – Slicing Arrays

- Array can be sliced just as with lists using the ‘**:** ‘notation within the square brackets

```
import numpy as np


arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = arr1[1:3]
print (arr2)
```

[45.6 3.2]

It is <u>important</u> to understand that a slice represents a view of the original array and references the data items in the original array.

NumPy – Slicing Arrays

```
arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = arr1[1:3]
print (arr2)
arr2[0] = 12
print (arr1)
```

```
[ 45.6  3.2]

[ 5.5 12.   3.2]
```

- Notice that the change made to the sliced NumPy array (arr1) is reflected in the original NumPy array (arr2).

NumPy – Multi-Dimensional Arrays

- Arrays can be multidimensional. Elements are accessed using [row, column] format inside bracket notation.
- Most of the time we will be working with 2D arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)
print (arr[0, 0])
print (arr[1, 2])
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

1.0

6.0
```

Slicing in 2D Arrays

- A single index value provided to a multi-dimensional array will refer to an entire row.
- We can use slicing to access an individual column by accessing all rows and specific columns

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)

print (arr)
print (arr[1])

print (arr[:, 0])

print (arr[:, [0,2]])
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[ 4.  5.  6.]

[ 1.  4.]

[[ 1.  3.]
 [ 4.  6.]]
```

Slicing in 2D Arrays

Of course we can also provide a start and stop index for a slice (just as we did with lists previously).

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], float)
print (arr)

arr2 = arr[1:3, 0:2]
print (arr2)
```

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

[[ 4.  5.]
 [ 7.  8.]]
```

Exact everything from row 1 and 2 for the column 0 and 1

NumPy – Broadcasting

- NumPy has a useful broadcasting functionality that allows us to apply operations to a entire subset of data items.

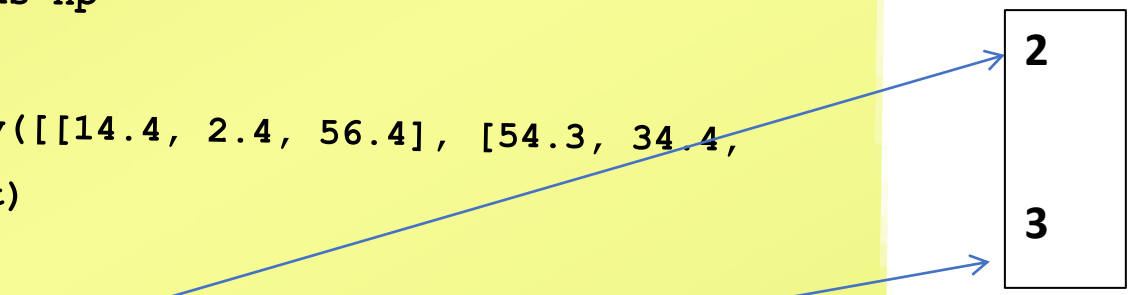
```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr[0, 1] = 12.2
print (arr)

arr[0] = 12.2
print (arr)

arr[:,0] *= 2
print (arr)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[[ 1.  12.2  3. ]
 [ 4.   5.   6. ]]

[[ 12.2  12.2  12.2]
 [ 4.   5.   6. ]]

[[ 24.4  12.2  12.2]
 [ 8.   5.   6. ]]
```

Use of len Function in M-D Arrays

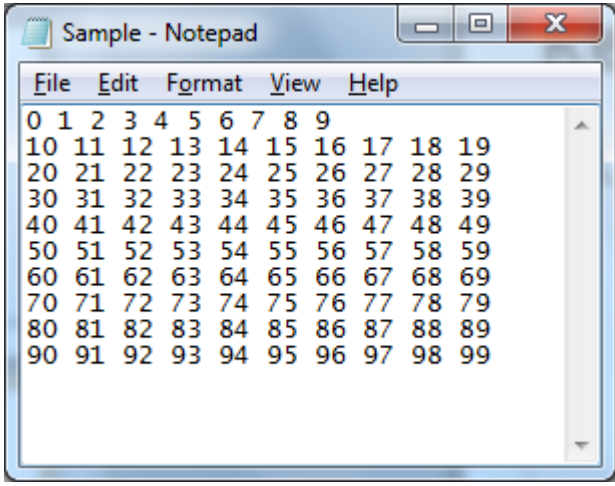
- len function can be used to obtain the number of rows or the number of columns
  - len of 2D array will return the **number of rows**
  - len of 2D row will return the **number of columns within that row**

```
import numpy as np

arr = np.array([[14.4, 2.4, 56.4], [54.3, 34.4,
98.22]], float)

print (len(arr))
print (len( arr[0] ) )
```



Reading Data From a File

- The code below shows how to read that data depicted in the file Sample.txt, into a two-dimension array
- ***np.genfromtxt*** uses *dtype=float* by default
- Large number of arguments that can specify delimiters, skipping headers, etc.




```
import numpy as np

data = np.genfromtxt('Sample.txt', dtype=int)
print (data[0,0])
print (data[1, 0])
```

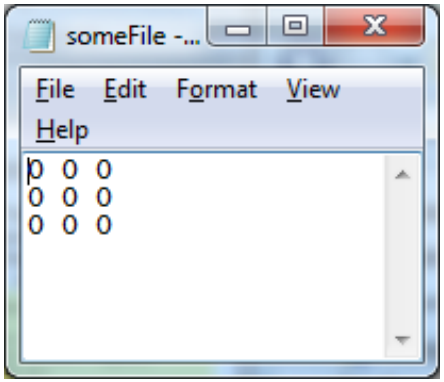
```
0
10
```

```
import numpy as np
data = np.genfromtxt('Sample.txt', dtype=int, delimiter = ',')
```

Writing Data To a File

- Use **savetxt** method to save data from an array to a file. The code below saves the 2D array data to the file called *someFile.txt*.
- Notice we can provide a format option (fmt).
  - %d indicates an integer
  - %f indicates a float
  - %s a string
- http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html

```
import numpy as np


data = np.zeros((3,3))
np.savetxt("someFile.txt", data, fmt="%d")
```



Notice above we use a function called zero. This is a straight-forward function that allows you to generate dummy MD arrays populated with zeros.

NumPy – Appending to MD Arrays

- We can add elements using append to MD arrays in NumPy
  - numpy.**append**(*arr*, *values*, ***axis=None***)
    - arr - Values are appended to a copy of this array.
    - values - These values are appended to a copy of arr. It must be of the correct shape
    - axis = The axis along which values are appended. **If axis is not given, both arr and values are flattened before use.**
      - In Numpy dimensions are called axes.

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [7, 8, 9])

print (arr1)
```

Notice the output array has been flattened. This is because no axis was specified

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

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis= ?)

print (arr1)
```

axis = 0 refers to the vertical axis

axis = 1 refers to the horizontal axis

Dimension of values being added must be same as the specific axis we are adding to

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis= 0)

print (arr1)
```

Add a row containing the values [7, 8, 9] to axis = 0

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

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

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis= 1)

print (arr1)
```

Add a column containing the values [7, 8, 9] to axis = 1

Generates an **error** specifying array dimensions don't match because each column only contains two values (not three)

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
```

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print arr

arr1 = np.append(arr, [[7],[8]], axis = 1)


print arr1
```

Notice we use two [] brackets, that is because we are adding a single column element to each row

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

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

Reshaping Arrays

- The ***arange*** function is similar to the range function but returns a NumPy array
- Only possible to create 1D array with arange
- Arrays can be reshaped by specifying new dimensions with reshape

```
import numpy as np


arr1 = np.arange(0,100, dtype=float)
arr2 = arr1.reshape((10, 10))
print (arr2)
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
 [50. 51. 52. 53. 54. 55. 56. 57. 58. 59.]
 [60. 61. 62. 63. 64. 65. 66. 67. 68. 69.]
 [70. 71. 72. 73. 74. 75. 76. 77. 78. 79.]
 [80. 81. 82. 83. 84. 85. 86. 87. 88. 89.]
 [90. 91. 92. 93. 94. 95. 96. 97. 98. 99.]]
```

Obtain Max or Min in an Array

- For multi-dimensional arrays we can specify the axis.
  - If we don't specify the axis it will determine the maximum for the entire array
- The way to understand it is whichever axis you are using will be 'collapsed' into the shape of the array. If axis is 0 the collapse is down to rows.

```
import numpy as np

arr1 = np.array([[10,20,30],[50, 60, 10]], float)
print (arr1)

print (np.amax(arr1))

print (np.amax(arr1, axis=0))

print (np.amax(arr1, axis=1))
```

```
[[ 10.  20.  30.]
 [ 50.  60.  10.]]

60.0

[ 50.  60.  30.]

[ 30.  60.]
```

Basic Array Operations

- Many functions exist for extracting whole-array properties.
- The items in an array can be summed or multiplied:
- These functions can be performed on multi-dimensional arrays
  - We can also provide an additional element of the axis we wish to access
- http://docs.scipy.org/doc/numpy/reference/routines.math.html
- http://docs.scipy.org/doc/numpy/reference/routines.statistics.html

```
import numpy as np

arr1 = np.array([[1, 2, 4],[3, 4, 2]], float)
print (np.sum(arr1))
print (np.product(arr1))
print (np.mean(arr1, axis = 0))
print (np.std(arr1, axis = 1))
```

```
16.0

192.0

[ 2.  3.  3.]

[ 1.24721913
0.81649658]
```

Example – Feature Selection

- In machine learning and statistics, feature selection involves selecting a subset of features (attributes) to employ in building your model.

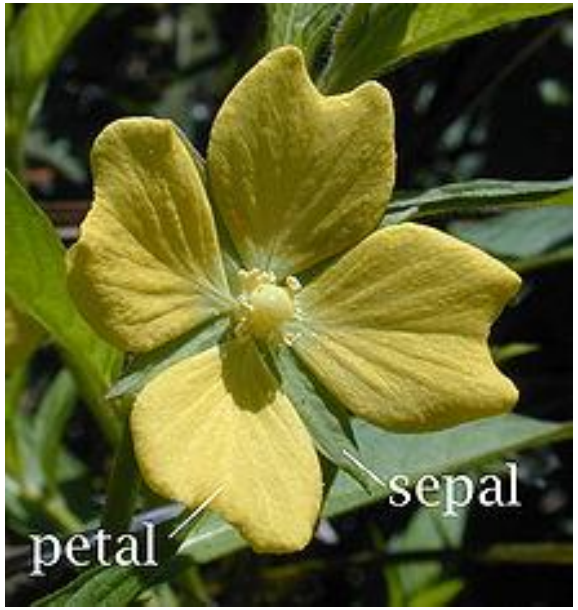
- Feature selection allows for the:
  - Provision of simpler machine learning models.
  - Shorter training times for models
  - Reduced variance in the model.

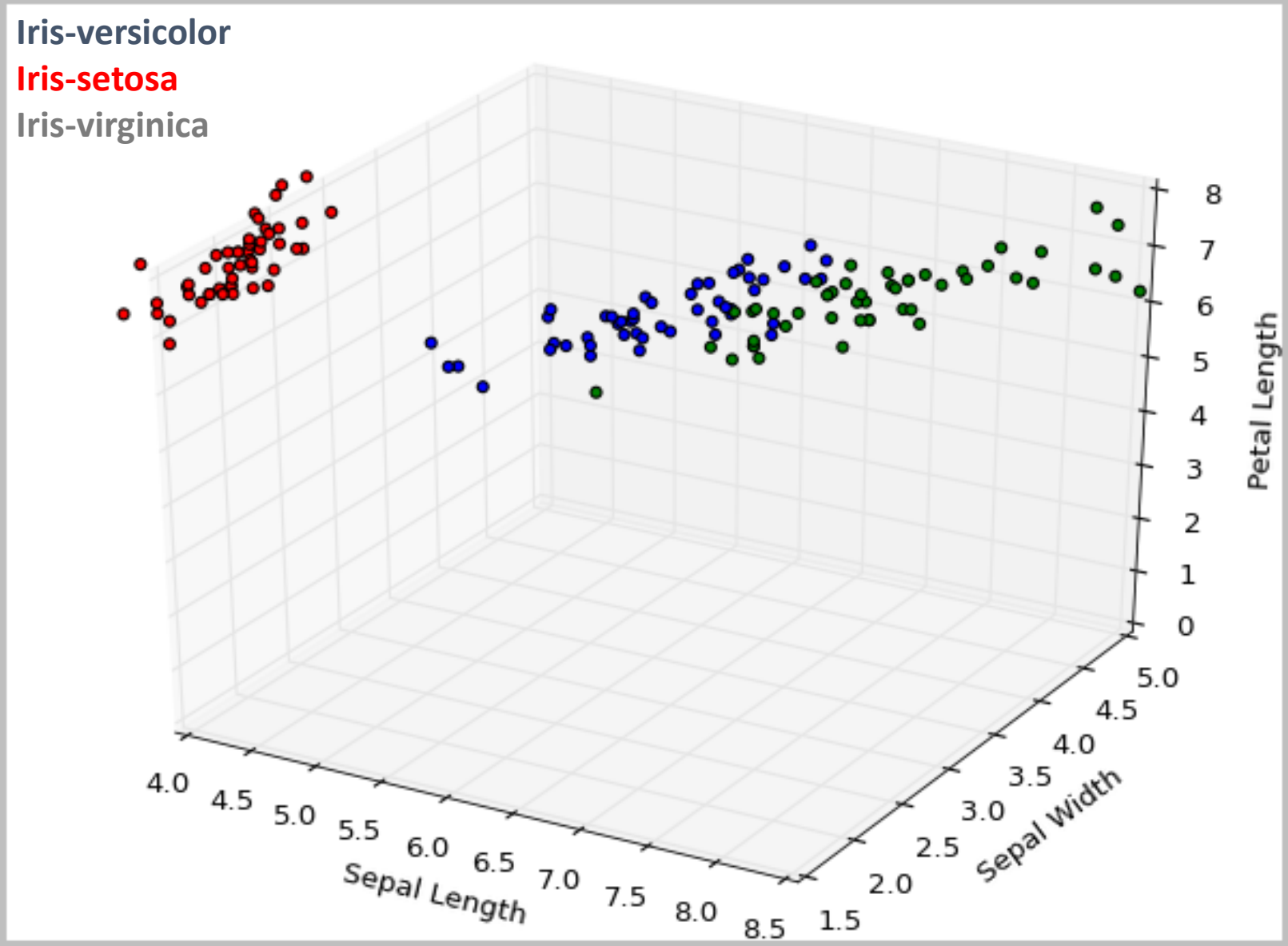
- One simple method of looking at feature selection is to look at the correlation between features in your dataset.

Iris Dataset

- The Iris dataset is perhaps the best known database to be found in the pattern recognition literature.
  - Predicted attribute: class of iris plant.
  - Number of Instances: 150 (50 in each of three classes)
  - Number of Attributes: 4 numeric

- Attribute Information:
- 1. sepal length in cm
- 2. sepal width in cm
- 3. petal length in cm
- 4. petal width in cm
- 5. class:
- -- Iris Setosa
- -- Iris Versicolour
- -- Iris Virginica




Iris-versicolor
Iris-setosa
Iris-virginica
Petal Length
Sepal Length
Sepal Width
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
1
2
3
4
5
6
7
8

Iris Dataset

- In the following code we use NumPy to look at the correlations between different variables using Pearson's coefficient.

```
import numpy as np

iris = np.genfromtxt("Iris.csv", delimiter=",")


## Correlation between sepal length and width
print (np.corrcoef(iris[:,0], iris[:,1]))

## Correlation between petal length and width
print (np.corrcoef(iris[:,2], iris[:,3]))
```

```
[[ 1.         -0.10936925]
 [-0.10936925  1.        ]]


[[ 1.        0.9627571]
 [ 0.9627571 1.       ]]
```

Deleting a column/row from a NumPy array

- To delete a column or row from an existing array we can used numpy.delete(arr, index, axis=None)

```
arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
arr1 = np.delete(arr, 1, axis=0)

print (arr1)
```

In the code above we remove a row from the vertical axis (the row with index 1).

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[[ 1  2  3  4]
 [ 9 10 11 12]]
```

Deleting Columns

- In the previous Iris example we noted a very high correlation between the features at index 2 and 3 (petal length and width)
- As such we may remove the feature at index 2 from the dataset as follows.

```
iris = np.genfromtxt("Iris.csv", delimiter=",")
newIris = np.delete(iris, 2, 1)
```

Array Mathematical Operations

- When **standard mathematical operations** are used with arrays, they are applied on an element by-element basis.
  - This means that the arrays should be the **same size** during addition, subtraction, etc
  - NumPy arrays support the typical range of operators +,-, *, /, %, **

```
import numpy as np

arr1 = np.array([[10,20], [30, 40]], float)
arr2 = np.array([[1,2], [3,4]], float)

print (arr1+arr2)
print ((arr1+arr2)*2)
```

```
[[ 11.  22.]
 [ 33.  44.]]

[[ 22.  44.]
 [ 66.  88.]]
```

Array Selectors

- We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. **Arrays also permit selection using other arrays**.

- That is, we can use **an array to filter** for specific subsets of elements of other arrays.
  - Before we look at this we must look at the result of using relational operators on NumPy arrays.

Comparison operators

- Boolean comparisons can be used to compare members element-wise on arrays of equal size.
- These operators (<,>, >=, <=, ==) return a **Boolean array** as a result

```
import numpy as np

arr1 = np.array([1, 3, 0], float)
arr2 = np.array([1, 2, 3], float)

resultArr = arr1>arr2
print (resultArr)
print (arr1== arr2)
```

[False True False]

[True False False]

Array Selectors

- We can use a Boolean array to **filter** the contents of another array.
- Below we use a Boolean array to select a subset of element from the NumPy array

```
import numpy as np

arr1 = np.array([45, 3, 2, 5, 67], float)

boolArr1 = [True, False, True, False, True]

print (arr1[boolArr1])
```

[45. 2. 67.]

Notice the program only returns the elements in arr1, where the corresponding element in the Boolean array is true

Comparison operators – Using relational operators to filter arrays

```
import numpy as np

arr1 = np.array([1, 3, 20, 5, 6, 78], float)

arr2 = np.array([1, 2, 3, 67, 56, 32], float)

resultArr = arr1>arr2
print (arr1[resultArr])
```

[3. 20. 78.]

Notice here we combine comparison operators and Boolean selection. This will print out all those values in arr1 that are greater than the corresponding value in arr2

```
arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]], float)
boolArr3 = np.array([True, False], bool)
print (arr2D[boolArr3])
```

If provided to a 2D array the boolean values refer to rows

If we provide a matching Boolean array it will select individual values and return a flat array

```
arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]], float)
boolArr3 = np.array([[True, False, True, False],[True, True,
False, True]], bool)
print (arr2D[boolArr3])
```

```
[[ 45.  3.  67.  34.]]

[ 45.  67.  12.  43.  36.]
```

The examples above illustrates the impact of using Boolean arrays to filter 2D arrays.

Selecting Columns from 2D Array

```
import numpy as np

arr2D = np.array([[45, 3, 67],[12, 43, 73]], float)

boolArr4 = np.array([True, False, True], bool)
print (arr2D[:,boolArr4])
```

```
[[ 45.  67.]
 [ 12.  73.]]
```

Here we use booleans to select particular columns from a 2D array. We specify all rows using : and we select the first and last column for selection

Comparison Operators

- The following code applies a conditional operator to the column with index 1 and returns the rows that satisfy this condition.

```
[[ 1.  2.  3.]
 [ 2.  4.  5.]
 [ 4.  5.  7.]
 [ 6.  2.  3.]]

[[ 1.  2.  3.]
 [ 6.  2.  3.]]
```

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5], [4, 5, 7], [6, 2, 3]], float)
print (data)

# return all rows in array where the element at index 1 in a row equals 2
newdata = data[ data[:,1] == 2 ]
print (newdata)
```

Returns all rows in the 2D array such that the value of the column with index 1 in that row contains the value 2

Logical Operators

- You can combine multiple conditions using logical operators.
- Unlike standard Python the logical operators used are **&** and **|**

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5],
[4, 5, 7], [6, 2, 3]], float)

resultA = data[:,0]>3
resultB = data[:,2]>6

print ( data [ resultA & resultB ] )
```

Notice in the code we combine two conditions using & (we could chain as many conditions as we wish)

[[4. 5. 7.]]

Back to Array Selectors

- In addition to Boolean selection, it is possible to select using integer arrays.
- In this example the new array *c* is composed by selecting the elements from *a* using the index specified by the elements of *b*.

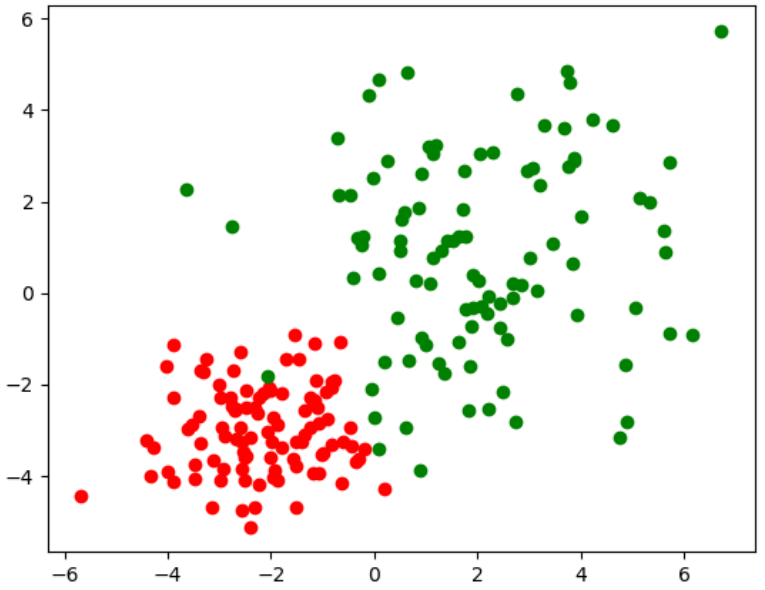
```
import numpy as np

a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)
c = a[b]
print (c)
```

[2. 2. 4. 8. 6. 4.]

Notice the array c is composed of index 0,0, 1, 3, 2, 1 of the array a

Visualising data

- 2d scatter plots



Import matplotlib

```
import matplotlib.pyplot as plt

data1 = [-2,-3] + np.random.randn(100,2)
data2 = [2,1] + 2*np.random.randn(100,2)

plt.figure()
plt.scatter(data1[:,0], data1[:,1], color='r')
plt.scatter(data2[:,0], data2[:,1], color='g')
```

Create a figure

Add a scatter plot

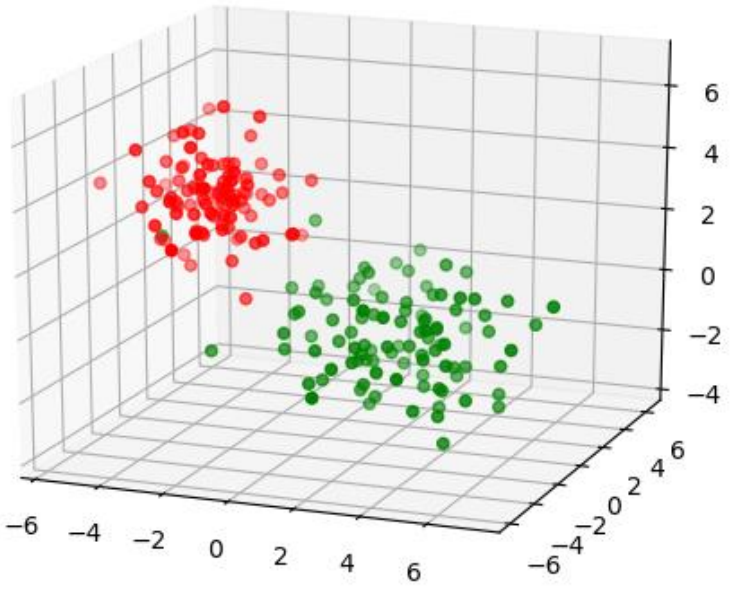
Colour of markers

Array of X-coordinates

Array of Y-coordinates

Visualising data

- 3d scatter plots



Import 3d axes

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(data1[0,:], data1[1,:], data1[2,:], color='r')
ax.scatter(data2[0,:], data2[1,:], data2[2,:], color='g')
```

Add 3d axes to plot

Call the scatter function on the 3d axes

Array of Z-coordiantes

Thank you for your attention