Unit - II Python libraries suitable for Machine Learning

Subject Name: Introduction to Machine Learning

Unit No: 2

Subject Code: 4350702

2.1 Numpy:

What is NumPy?

- ✓ NumPy stands for Numerical Python.
- ✓ NumPy is a Python library used for working with arrays.
- ✓ It also has functions for working in domain of linear algebra, Fourier transform, and matrices.
- ✓ NumPy was created in 2005 by Travis Oliphant. It is an open-source project and you can use it freely.

Why Use NumPy?

- ✓ In Python we have lists that serve the purpose of arrays, but they are slow to process.
- ✓ NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- ✓ The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- ✓ Arrays are very frequently used in data science, where speed and resources are very important.
- ✓ NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- ✓ This behavior is called locality of reference in computer science.
- ✓ This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Creating Array: array()

- ✓ NumPy is used to work with arrays. The array object in NumPy is called ndarray.
- ✓ We can create a NumPy ndarray object by using the array() function.

Example:

```
import numpy as np
```

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

Output:

```
[12345]
<class 'numpy.ndarray'>
```

- ✓ type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.
- ✓ To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Dimensions in Arrays: A dimension in arrays is one level of array depth (nested arrays).

- ✓ nested array: are arrays that have arrays as their elements.
- ✓ 0-D Arrays
- 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.
- Example: Create a 0-D array with value 42

```
import numpy as np
arr = np.array(42)
print(arr)
```

Output: 42

- ✓ 1-D Arrays
- An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
- These are the most common and basic arrays.
- Example: Create a 1-D array containing the values 1,2,3,4,5

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Output: [12345]

- ✓ 2-D Arrays
- An array that has 1-D arrays as its elements is called a 2-D array.
- These are often used to represent matrix or 2nd order tensors.
- NumPy has a whole sub module dedicated towards matrix operations called numpy.mat
- Example: Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

Output:

```
[[123]
[456]]
```

✓ 3-D arrays

- An array that has 2-D arrays (matrices) as its elements is called 3-D array.
- These are often used to represent a 3rd order tensor.
- Example: Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6

```
import numpy as np
arr= np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Output:

```
[[[123]
[456]]
[[123]
[456]]]
```

Accessing Array: by referring to its index number:

✓ Access Array Elements

- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc..
- Example: Get the second element from the following array.

```
import numpy as np
arr= np.array([1, 2, 3, 4])
print(arr[1])
```

Output: 2

✓ Access 2-D Arrays

- To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.
- Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.
- Example: Access the element on the first row, second column:

```
import numpy as np
arr = np.array([[1,2,3,4.5], [6.7,8,9,10]])
print('2nd element on 1st row:', arr[0, 1])
```

Output:

```
2nd element on 1st dim: 2
```

✓ Access 3-D Arrays

- To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.
- Example: Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Output:

```
6
```

✓ Negative Indexing

- Use negative indexing to access an array from the end.
- Example: Print the last element from the 2nd dim:

```
import numpy as np
arr= np.array([[1.2.3.4.5], [6,7,8,9.10]])
print('Last element from 2nd dim:', arr[1, -1])
```

Output:

```
Last element from 2nd dim: 10
```

Stacking & Splitting: stack(), array_split()

stack()

- stack() is used for joining multiple NumPy arrays. Unlike, concatenate(), it joins arrays along a new axis. It returns a NumPy array.
- to join 2 arrays, they must have the same shape and dimensions. (e.g. both (2,3)-> 2 rows,3 columns)
- stack() creates a new array which has 1 more dimension than the input arrays. If we stack 2 1-D arrays, the resultant array will have 2 dimensions.
- ***Syntax:*** *`numpy.stack(arrays, axis=0, out=None)`*

  Where,

  - arrays: Sequence of input arrays (required)
  - axis: Along this axis, in the new array, input arrays are stacked. Possible values are 0 to (n-1) positive integer for n-dimensional output aray.
  - out: The destination to place the resultant array.
  - Example: stacking two 1d arrays

```
import numpy as np
# input array
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
```

```
#Stacking 2 1-d arrays
e= np.stack((a, b).axis=0)
print(c)
```
Output:

```
array([[1, 2, 3],
[4, 5, 6]])
```

array_split()

- Splitting is reverse operation of Joining.
- Joining merges multiple arrays into one and Splitting breaks one array into multiple.
- We use array split() for splitting arrays, we pass it the array we want to split and the number of splits.
- Example: Split the array in 3 parts

```
import numpy as np
arr = np.array([1, 2, 3, 7,8])
newarr = np.array_split(arr, 3)
print(newarr)
Output:
[array([1, 2]), array([3, 4]), array([5, 6])]
```

- Splitting 2-D Arrays
- Example: Split the 2-D array into three 2-D arrays. nto three 2-D

```
import numpy as np
arr = np.array([[1.2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
newarr = np.array_split(arr. 3)
print(newarr)
```
Output:

```
[array([[1, 2], [3, 4]]),
 array([[5, 6], [7, 8]]),
 array([[9, 10],[11,12]])]
```

- In addition, you can specify which axis you want to do the split around.
- The example below also returns three 2-D arrays, but they are split along the row (axis=1).
  - Example:Split the 2-D array into three 2-D arrays along rows.

```
import numpy as np
arr= np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])
newarr = np.array_split(arr, 3, axis=1)
print(newarr)
```
Output:

```
[array([[ 1],
[4].
[7],
[10].
[13].
[16]]), array([[ 2].
[5].
[8].
[11].
[14].
[17]]), array([[ 3].
[6].
[9].
[12],
[15].
[18]])]
```

- An alternate solution is using hsplit() opposite of hstack()
- Example: Use the hsplit() method to split the 2-D array into three 2-D arrays along rows.

```
import numpy as np.
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])
newarr np.hsplit(arr, 3)
print(newarr)
```

Output:

```
[array([[ 1].
[4].
[7].
[10].
[13].
[16]]), array([[ 2].
[5].
[8].
[11].
[14].
[17]]), array([[ 3],
[6].
[9].
[12].
[15].
[18]])]
```

Maths Functions: add(), subtract(), multiply(), divide(), power(), mod()

- ✓ NumPy provides a wide range of arithmetic functions to perform on arrays.
- ✓ Here's a list of various arithmetic functions along with their associated operators:

Operation	Arithmetic Function	Operator
Addition	add()	+
Subtraction	subtract()	-
Multiplication	multiply()	*
Division	divide()	/
Exponentiation	power()	**
Modulus	mod()	%

- Example 1:

```
import numpy as np
first_array = np.array([1, 3, 5, 7])
second_array = np.array([2, 4, 6, 8])
#using the add() function
result2 = np.add(first_array, second_array)
print("Using the add() function:",result2)
```

output:

```
Using the add() function: [371115]
```

- Example 2:

```
import numpy as np
print("Add:")
print(np.add(1.0, 4.0))
print("Subtract:")
print(np.subtract(1.0, 4.0))
print("Multiply:")
print(np.multiply(1.0, 4.0))
print("Divide:")
print(np.divide(1.0, 4.0))
```

output:

```
Add:
5.0
Subtract:
-3.0
Multiply:
4.0
Divide:
```

```
0.25
```

Statistics Functions: amin(), amax(), mean(), median(), std(), var(), average(), ptp()

- ✓ The NumPy package contains a number of statistical functions which provides all the functionality required for various statistical operations.
- ✓ It includes finding mean, median, average, standard deviation, variance and percentile etc from elements of a given array. Below mentioned are the most frequently used statistical functions:

Function	Description
mean()	Computes the arithmetic mean along the specified axis
median()	Computes the median along the specified axis.
average()	Computes the weighted average along the specified axis.
std()	Compute the standard deviation along the specified axis.
var()	Compute the variance along the specified axis.
amax()	Returns the maximum of an array or maximum along an axis.
amin()	Returns the minimum of an array or minimum along an axis.
ptp()	Return range of values (maximum minimum) of an array or along an axis.
percentile()	Computes the specified percentile of the data along the specified axis.

numpy.mean() function

The numpy.mean() function is used to compute the arithmetic mean along the specified axis. The mean is calculated over the flattened array by default, otherwise over the specified axis.

Syntax:

```
numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)
```

- a: Required. Specify an array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.
- axis: Optional. Specify axis or axes along which the means are computed. The default is to compute the mean of the flattened array.
- dtype: Optional. Specify the data type for computing the mean. For integer inputs, the default is float64. For floating point inputs, it is same as the input dtype.
- out:Optional. Specify output array for the result. The default is None. If provided, it must have the same shape as output.
- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. With default value, the keepdims will not be passed through to the mean method of sub-classes of ndarray, but any non-default value will be. If the sub-class method does not implement keepdims the exceptions will be raised.

Example:

```
import numpy as np
Arr = np.array([[10,20,30],[70,80,90]])
print("Array is:")
print(Arr)

#mean of all values
print("\nMean of all values:", np.mean(Arr))

#mean along axis=0
print("\nMean along axis=0") print(np.mean(Arr, axis=0))

#mean along axis=1
print("\nMean along axis=1")
print(np.mean(Arr,
```

output:

```
Array is:
[[10 20 30] [ 70 80 90]]
Mean of all values: 50.0
Mean along axis=0
[40. 50. 60.]
Mean along axis=1
[20. 80.]
```

numpy.median() function

The numpy.median() function is used to compute the median along the specified axis. The median is calculated over the flattened array by default, otherwise over the specified axis.

Syntax:

```
numpy.median(a, axis=None, out=None, overwrite_input=False,
keepdims=False)
```

Parameters:

- a:Required. Specify an array (array_like) containing numbers whose median is desired.
- axis: Optional. Specify axis or axes along which the medians are computed. The default is to compute the median of the flattened array.
- out:Optional. Specify output array for the result. The default is None. If provided, it must have the same shape as output.
- overwrite_input: Optional. If True, the input array will be modified. If overwrite_input is True and a is not already an ndarray, an error will be raised. Default is False.

- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example:

In the example below, median() function is used to calculate median of all values present in the array. When axis parameter is provided, median is calculated over the specified axes.

```
import numpy as np
Arr = np.array([[10,20,500],[30,40,400], [100,200,300]])
print("Array is:")
print(Arr)

#median of all values
print("\nMedian of values:", np.median(Arr))

#median along axis zeta = 0
print("\nMedian along axis=0")
print(np.median(Arr, axis=(0))

#median along axis=1
print("\nMedian along axis=1")
print(np.median(Arr, axis=1))
```

output:

```
Array is:
[[ 10 20 500]
[30 40 400]
[100 200 3001]
Median of values: 100.0
Median along axis=0
[30.40.400.]
Median along axis=1
[20.40.200.]
```

numpy.average() function

The numpy.average() function is used to compute the weighted average along the specified axis. The syntax for using this function is given below:

Syntax:

```
numpy.average(a, axis=None, weights=None, returned=False)
```

Parameters:

- a: Required. Specify an array containing data to be averaged. If a is not an array. a conversion is attempted.
- axis: Optional. Specify axis or axes along which to average a. The default, axis=None, will average over all of the elements of the input array. If axis is negative it counts from the last to the first axis.
- weight: Optional. Specify an array of weights associated with the values in a. The weights array can either be 1-D (in which case its length must be the size of a along the given axis) or of the same shape as a. If weights=None, then all data in a are assumed to have a weight equal to one.
- returned: Optional. Default is False. If True, the tuple (average, sum_of_weights) is returned, otherwise only the average is returned.

Example:

In the example below, average() function is used to calculate average of all values present in the array. When axis parameter is provided, averaging is performed over the specified axes.

```
import numpy as np
Arr = np.array([[10,20,30],[70,80,90]])
print("Array is:") print(Arr)
#average of all values
print("\nAverage of values:", np.average(Arr ))
#averaging along axis=0
print("\nAverage along axis=0”)
print(np.average(Arr, axis=0))
#averaging along axis = 1
print("\nAverage along axis=1”)
print(np.average(Arr, axis=1))
```

Output:

```
Array is :
[10 20 30]
[70 80 90]]
Average along axis=0
[40. 50. 60.]
Average along axis=1
[20. 80.]
```

numpy.std() function

The numpy.std() function is used to compute the standard deviation along the specified axis.

The standard deviation is defined as the square root of the average of the squared deviations from the mean. Mathematically, it can be represented as:

```
std=sqrt(mean(abs(x-x.mean())**2))
```

Syntax:

```
numpy.std(a, axis=None, dtype=None, out=None, keepdims=<no value>)
```

Parameters:

- a: Required. Specify the input array.
- axis: Optional. Specify axis or axes along which the standard deviation is calculated. The default, axis=None, computes the standard deviation of the flattened array.
- dtype: Optional. Specify the type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type..
- out: Optional. Specify the output array in which to place the result. It must have the same shape as the expected output.
- keepdims : Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example:

Here, std() function is used to calculate standard deviation of all values present in the array. But, when axis parameter is provided, standard deviation is calculated over the specified axes as shown in the example below.

```
import numpy as np
Arr = np.array([[10,20,30],[70,80,90]])
print("Array is:")
print(Arr)
#standard deviation of all values
print("\nStandard deviation of all values:", np.std(Arr))
#standard deviation along axis=0
print("\nStandard deviation along axis i = 0 deg )
print(np.std(Arr, axis=0))
#standard deviation along axis i = 1
print("\nStandard deviation along axis=1")
print(np.std(Arr, axis=1))
```

Output:

```
Array is:
[[10 20 30]
[70 80 901]
Standard deviation of all values: 31.09126351029605
```

```
Standard deviation along axis=(0
[30. 30. 30.]
Standard deviation along axis=1
[8.16496581 8.16496581]
```

The numpy.var() function

The `numpy.var()` function is used to compute the variance along the specified axis. The variance is a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Syntax:

```
numpy.var(a, axis=None, dtype=None, out=None, keepdims=<no value>)
```

Parameters:

- a: Required. Specify the input array.
- axis: Optional. Specify axis or axes along which the variance is calculated. The default, `axis=None`, computes the variance of the flattened array.
- dtype: Optional. Specify the type to use in computing the variance. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.
- out: Optional. Specify the output array in which to place the result. It must have the same shape as the expected output.
- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example:

In the example below, `var()` function is used to calculate variance of all values present in the array. When axis parameter is provided, variance is calculated over the specified axes,

```
import numpy as np
Arr = np.array([[10,20,30],[70,80,90]])
print("Array is:")
print(Arr)
#variance of all values print("\nVariance of all values:",
np.var(Arr))
#variance along axis=0 print("\n Variance along axis=(0")
print(np.var(Arr, axis=(0))
#variance along axis=1
print("\n Variance along axis=1")
print(np.var(Arr, axis=1))
```

Output:

```
Array is:
[[10 20 30]
[70 80 901]
Variance of all values: 966.6666666666666
Variance along axis=0
[900, 900, 900.]
Variance along axis=1
[66.66666667 66.66666667]
```

numpy.amax() function

The NumPy `amax()` function returns the maximum of an array or maximum along the specified axis.

Syntax:

```
numpy.amax(a, axis=None, out=None, keepdims=<no value>)
```

Parameters:

- a: Required. Specify the input array.
- axis: Optional. Specify axis or axes along which to operate. The default, axis=None, operation is performed on flattened array.
- out: Optional. Specify the output array in which to place the result. It must have the same shape as the expected output.
- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example:

In the example below, amax() function is used to calculate maximum of an array. When axis parameter is provided, maximum is calculated over the specified axes.

```
import numpy as np
Arr = np.array([[10,20,30],[70,80,90]])
print("Array is:")
print(Arr)

#maximum of all values
print("\nMaximum of all values:", np.amax(Arr))
(s = 0 ^ n)

#maximum along axis=1
print("\nMaximum along axis=1")
```

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

Output:

```
Array is:
[[10 20 30]
[70 80 90]]
Maximum of all values: 90
Maximum along axis=0
[70 80 90]
Maximum along axis=1
[30 90]
```

numpy.amin() function

The NumPy `amin()` function returns the minimum of an array or minimum along the specified axis.

Syntax:

```
numpy.amin(a, axis=None, out=None, keepdims=<no value>)
```

Parameters:

- a: Required. Specify the input array.
- axis: Optional. Specify axis or axes along which to operate. The default, axis=None, operation is performed on flattened array.
- out: Optional. Specify the output array in which to place the result. It must have the same shape as the expected output.
- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example:

In the example below, amin() function is used to calculate minimum of an array. When axis parameter is provided, minimum is calculated over the specified axes.

```
import numpy as np
Arr = np.array([[10,20,30],[70,80,90]])
print("Array is:")
print(Arr)

#minimum of all values
```

```
print("\nMinimum of all values:", np.amin(Arr))

#minimum along axis=0
print("\nMinimum along axis=0")
print(np.amin(Arr, axis s = (0) )

#minimum along axis=1
print("\nMinimum along axis=1")
print(np.amin(Arr, axis omega = 1 ))
```

output:

```
Array is:
[[10 20 30]
70 80 90]]
Minimum of all values: 10
Minimum along axis = 0
[10 20 30]
Minimum along axis = 1
[10 70]
```

numpy.ptp() function

- The NumPy `ptp()` function returns range of values (maximum minimum) of an array or range of values along the specified axis.
- The name of the function comes from the acronym for peak to peak.

Syntax:

```
numpy.ptp(a, axis=None, out=None, keepdims=sno value>)
```

Parameters:

- a: Required. Specify the input array.
- axis: Optional. Specify axis or axes along which to operate. The default, axis None, operation is performed on flattened array.
- out: Optional. Specify the output array in which to place the result. It must have the same shape as the expected output.
- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example: In the example below, `ptp()` function is used to calculate the range of values present in the array. When axis parameter is provided, it is calculated over the specified axes.

```
import numpy as np
```

```
Arr = np.array([[10,20,30],[70,80,901])
print("Array is:")
print(Arr)
#range of values
print("\nRange of values:", np.ptp(Arr))
#Range of values along axis=0
print("\nRange of values along axis=0")
print(np.ptp(Arr. axis=0))
#Range of values along axis=1
print("\nRange of values along axis=1")
print(np.ptp(Arr, axis=1))
```

Output:

```
Array is:
[[10 20 30]
[70 80 90]]

Range of values: 80

Range of values along axis=0
[60 60 60]
Range of values along axis=1
[20 20]
```

numpy.percentile() function

The NumPy `percentile()` function returns the q-th percentile of the array elements or q-th percentile the data along the specified axis.

Syntax:

```
numpy.percentile(a, q. axis=None, out=None, interpolation=linear',
keepdims=False)
```

Parameters:

- a: Required. Specify the input array (array_like).
- q: Required. Specify percentile or sequence of percentiles to compute, which must be between 0 and 100 inclusive (array_like of float).
- axis: Optional. Specify axis or axes along which to operate. The default, `axis=None`, operation is performed on flattened array.
- out: Optional. Specify the output array in which to place the result. It must have the same shape as the expected output.

- interpolation: Optional. Specify the interpolation method to use when the desired percentile lies between two data points. It can take value from (`linear', 'lower', 'higher', 'midpoint', 'nearest'`)
- keepdims: Optional. If this is set to True, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

Example: In the example below, percentile() function returns the maximum of all values present in the array. When axis parameter is provided, it is calculated over the specified axes.

```
import numpy as np
Arr = np.array([[10,20, 30],[40, 50, 60]])
print("Array is:")
print(Arr)
print()

#calculating 50th percentile point
print("50th percentile:", np.percentile(Arr, 50)).
print()

#calculating (25, 50, 75) percentile points
print("[25, 50, 75] percentile:\n", np.percentile(Arr, (25, 50, 75)))
print()

#calculating 50th percentile point along axis=0
print("50th percentile (axis=0);", np.percentile(Arr, 50, axis=0))

#calculating 50th percentile point along axis=1
print("50th percentile (axis=1);", np.percentile(Arr, 50, axis=1))
```

Output:

```
Array is:
[[10 20 30]
[40 50 60]]
50th percentile: 35.0

[25, 50, 75] percentile:
[22.5 35. 47.5]

50th percentile (axis=0): [25. 35. 45.]
```

```
50th percentile (axis=1): [20. 50.]
```

2.2 Pandas

History of development

- In 2008, pandas development began at AQR Capital Management. By the end of 2009 it had been open sourced, and is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible.
- Since 2015, pandas is a NumFOCUS sponsored project. This will help ensure the success of development of pandas as a world-class open-source project.
- Timeline
  - 2008: Development of pandas started
  - 2009: pandas becomes open source
  - 2012: First edition of Python for Data Analysis is published
  - 2015: pandas becomes a NumFOCUS sponsored project
  - 2018: First in-person core developer sprint
- Library Highlights
  - A fast and efficient DataFrame object for data manipulation with integrated indexing:
  - Tools for reading and writing data between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format;
  - Intelligent data alignment and integrated handling of missing data: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form;
  - Flexible reshaping and pivoting of data sets;
  - Intelligent label-based slicing, fancy indexing, and subsetting of large data sets;
  - Columns can be inserted and deleted from data structures for size mutability;
  - Aggregating or transforming data with a powerful group by engine allowing split-apply-combine operations on data sets,
  - High performance merging and joining of data sets:
  - Hierarchical axis indexing provides an intuitive way of working with high-dimensional data in a lower-dimensional data structure;
  - Time series-functionality: date range generation and frequency conversion, moving
  - window statistics, date shifting and lagging. Even create domain-specific time offsets. and join time series without losing data;
  - Highly optimized for performance, with critical code paths written in Cython or C.
  - Python with pandas is in use in a wide variety of academic and commercial domains, including Finance, Neuroscience, Economics, Statistics, Advertising. Web Analytics, and more.

Series: Series()

- The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types.
- We can easily convert the list, tuple, and dictionary into series using "series' method. The row labels of series are called the index.
- A Series cannot contain multiple columns. It has the following parameter:
  - data: It can be any list, dictionary, or scalar value.
  - index: The value of the index should be unique and hashable. It must be of the same length
  - as data. If we do not pass any index, default np.arrange(n) will be used.
  - dtype: It refers to the data type of series.
  - copy: It is used for copying the data.

Creating a Series:

We can create a Series in two ways:

1. Create an empty Series
2. Create a Series using inputs.

Create an Empty Series:

We can easily create an empty series in Pandas which means it will not have any value.

The syntax that is used for creating an Empty Series:

```
<series object = pandas Series()
```

The below example creates an Empty Series type object that has no values and having default datatype, Le.. float64.

Example:

```
import pandas as pd
x = 1 pd.Series()
print (x)
```

Output:

```
Series([], dtype: float64)
```

Creating a Series using inputs:

We can create Series by using various inputs:

- Array
- Dict
- Scalar value

Creating Series from Array:

Before creating a Series, firstly, we have to import the numpy module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default index of range(n) is being passed where n defines the length of an array, i.e., [0,1,2,... range(len(array))-1].

Example:

```
import pandas as pd
import numpy as np
info = np.array(['P.and.a.s'])
a = pd.Series(info)
print(a)
```

output:

```
0 P
1 a
2 n
3 d
4 a
5 s
dtype: object
```

Create a Series from dict

We can also create a Series from dict. If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index.

If index is passed, then values correspond to a particular label in the index will be extracted from the dictionary.

Example:

```
#import the pandas library
import pandas as pd
import numpy as np
info['x': 0., 'y: 1, 2: 2.1 apd.Series(info)
print (a)
```

Output:

```
x 0.0
y 1.0
z 20
dtype: float64
```

Series Functions

There are some functions used in Series which are as follows:

Functions	Description
Pandas_Series.map()	Map the values from two series that have a common column
Pandas_Series.std()	Calculate the standard deviation of the given set of numbers, DataFrame, column, and rows
Pandas_Series.to_frame()	Convert the series object to the dataframe.
Pandas_Series.value_counts()	Returns a Series that contain counts of unique values.

Dataframes: DataFrames()

- Pandas DataFrame is a widely used data structure which works with a two-dimensional array with labeled axes (**`rows`** and **`columns`**).
- DataFrame is defined as a standard way to store data that has two different indexes, i.e., **`row index`** and **`column index`**. It consists of the following properties:
- The columns can be heterogeneous types like int, bool, and so on.
- It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.
- Parameter & Description:
  - data: it consists of different forms like ndarray, series, map, constants, lists, array.
  - index: The Default np.arrange(n) index is used for the row labels if no index is passed.
  - columns: The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.
  - dtype: It refers to the data type of each column.
  - copy(): It is used for copying the data

	Columns		
Rows	**Regd. No**	**Name**	**Percentage of Marks**
	100	John	74.5
	101	Smith	87.2
	102	Parker	92
	103	Jones	70.6
	104	William	87.5

Create a DataFrame

We can create a DataFrame using following ways:

- dict
- Lists
- Numpy ndarrrays
- Series

Create an empty DataFrame

Example:

```
#importing the pandas library
import pandas as pd
df pd.DataFrame()
print (df)
```

Output:

```
Empty DataFrame
columns:[]
index:[]
```

Create a DataFrame using List:

We can easily create a DataFrame in Pandas ndas using list.

Example:

```
#importing the pandas library
import pandas as pd
# a list of strings
x = 1 Python', 'Pandas]
#Calling DataFrame constructor on list
df=pd.DataFrame(x)
print(df)
```

Output:

```
0
0 Python
1 Pandas
```

Create a DataFrame from Dict of ndarrays/ Lists

Example:

```
#importing the pandas library
import pandas as pd
info= {'ID':[101, 102, 103], ‘Department’: [‘B.Sc’, ‘B.Tech’,
‘M.Tech’,]}
df=pd.DataFrame(info)
print(df)
```

Output:

	ID	Department
	101	B.Sc.
1	102	B.Tech
2	103	M.Tech

Create a DataFrame from Dict of Series :

Example:

```
#importing the pandas library
import pandas as pd
info={'one': pd.Series([1, 2, 3, 4, 5, 6], index=['a', b, c, d,
'e','f']),
'two': pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c',
'd', 'e', 'f', 'g','h'])}
d1= pd.DataFrame(info)
print (d1)
```

Output:

	One	Two
A	1.0	1
B	2.0	2
C	3.0	3
D	4.0	4
E	5.0	5
F	6.0	6
G	NaN	7
H	NaN	8

Read CSV File: read_csv()

- To read the csv file as pandas. DataFrame, use the pandas function read_csv() or read_table().
- The difference between read csv() and read table() is almost nothing. In fact, the same function is called by the source:
  - read_csv() delimiter is a comma character
  - read_table() is a delimiter of tab \t.
- The pandas function read_csv() reads in values, where the delimiter is a comma character.
- You can export a file into a csv file in any modern office suite including Google Sheets,

Use the following csv data as an example.

Example:

```
name, age, state, point
```

```
Alice, 24.NY,64
Bob,42.CA,92
Charlie, 18,CA,70
Dave,68,TX,70
Ellen 24.CA.88
Frank 30.NY,57
Alice, 24,NY,64
Bob,42,CA,92
Charlie, 18.CA.70
Dave,68,TX,70
Ellen, 24,CA,88
Frank,30,NY.5T
```

You can load the CSV like this:

```
#Load pandas
import pandas as pd
#Read CSV file into DataFrame df
df = pd.read_csv("sample.csv", index_col=0)
#Show dataframe
print(df)
```

output:

```
#         age state point
#name
#Alice   24 NT 04
#Bob     42 CA 92
#Charlie 18 CA 70
#Dove    68 TX 70
#Ellen   24 CA 88
#Frank   30 NY 57
```

Cleaning Empty Cells: dropna()

Empty cells can potentially give you a wrong analyze data.

- Remove Rows

One way to deal with empty cells is to remove rows that contain empty cells.

This is usually OK, since data sets can be very big, und removing a few rows will not have a big impact on the result.

Example: Return a new Data Frame with no empty cells:

```
import pardas as pd
```

```
df=pd.read_csv('data.csv')
new_df=df.dropna()
print(new_df.to_string())
```

Cleaning Wrong Data: drop()

- Removing Rows

Another way of handling wrong data is to remove the rows that contains wrong data.
This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses

Example: Delete rows where "Duration" is higher than 120:

```
for x in df.index:
      if df.loc[x, "Duration"]> 120:
            df.drop(x, inplace=True)
```

Removing Duplicates: duplicated()

The duplicated() method returns a Series with True and False values that describe which rows in the DataFrame are duplicated and not.
Use the subset parameter to specify which columns to include when looking for duplicates.
By default, all columns are included.
By default, the first occurrence of two or more duplicates will be set to False.
Set the keep parameter to False to also set the first occurrence in True
Syntax:

```
dataframe.duplicated(subset, keep)
```

Parameter	Value	Description
subset	columm label(s)	Optional. A String, or a list, of the column names to include when looking for duplicates. Default subset None (meaning no subset is specified, and all columns should be included.
Keep	'first' 'last' False	Optional, default 'first'. Specifies how to deal with duplicates: first' means set the first occurrence to False, the rest to True. last' means set the last occurrence to False, the rest to True. False means set all occurrences to True.

Example:

Only include the columns "name" and "age":

```
s =  df.duplicated(subset=["name", "age"])
print(s)
```

output:

```
0      False
1      False
2      True
3      False
4      True
dtype: bool
```

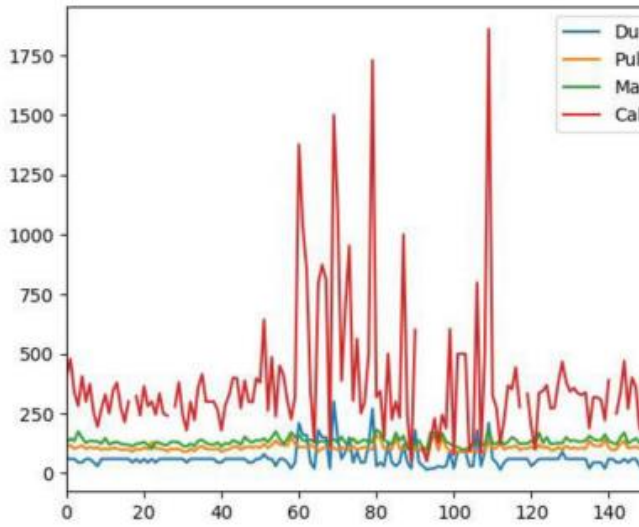
Pandas Plotting: plot()

Pandas uses the plot() method to create diagrams.

We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.

Example:

```
import pandas as pd
import matplotlib.pyplot as plt
dfpd.read_csv('data.csv')
df.plot()
plt.show()
```



2.3 Matplotlib

What is Matplotlib?

- Matplotlib is a low level graph plotting library in python that serves as a visualization utility.
- Matplotlib was created by John D. Hunter.
- Matplotlib is open source and we can use it freely.
- Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

➢ `Pyplot.plot: plot() and Show: show()`

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

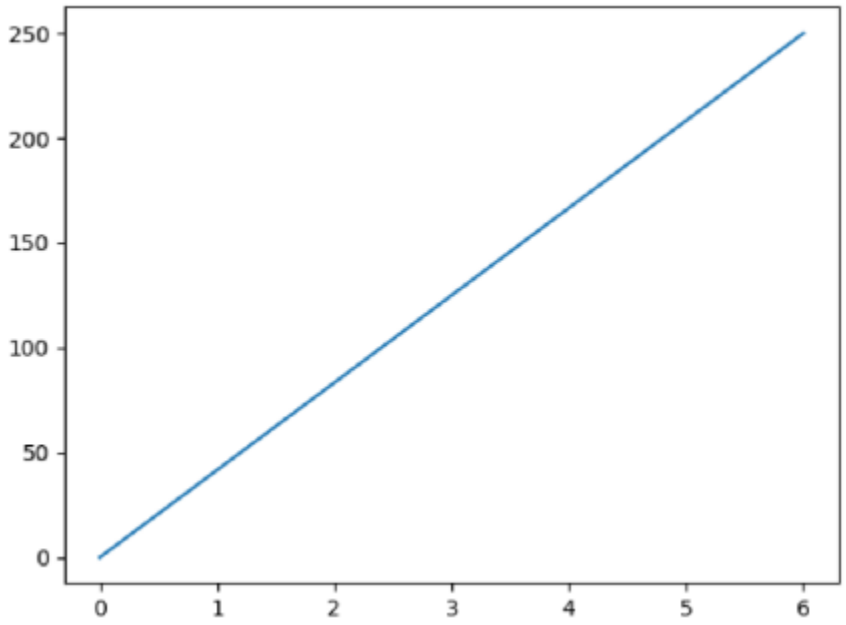
```
import matplotlib.pyplot as plt
```

Now the Pyplot package can be referred to as plt.

Example:

Draw a line in a diagram from position (0.0) to position (6,250):

```
import matplotlib.pyplot as plt import numpy as np
xpoints = np.array([0, 6])
ypoints= np.array([0, 250])
plt.plot(xpoints, ypoints)
plt.show()
```



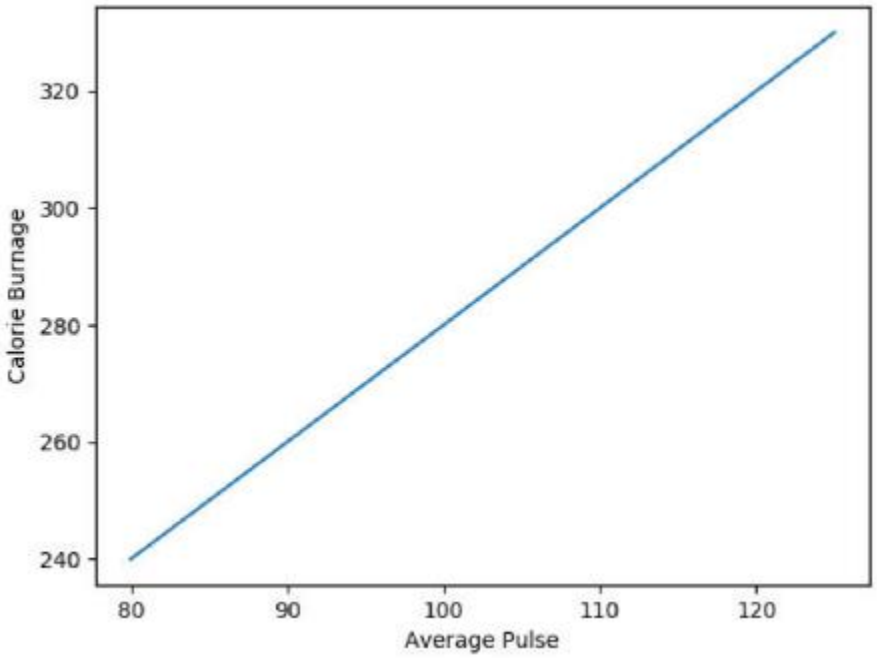
➢ **Labels: xlabel(), ylabel()**

With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

Example:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125]) y =
np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
```

```
plt.plot(x, y)
plt.xlabel("Average Pulse") plt.ylabel("Calorie Burnage")
plt.show()
```



- **Grid: grid()**

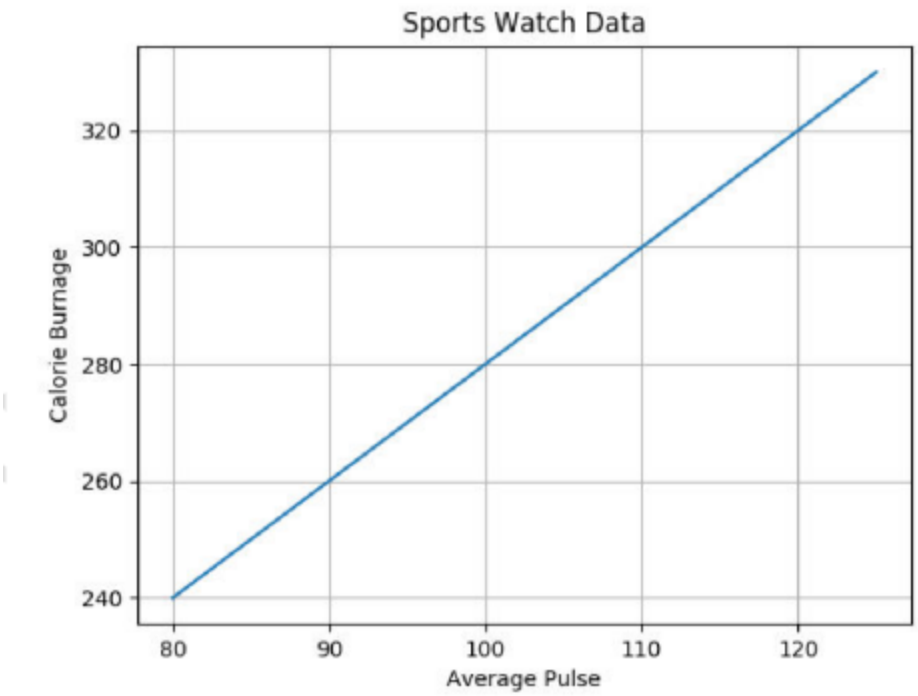
With Pyplot, you can use the grid() function to add grid lines to the plot.

Example:

Add grid lines to the plot:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125]) y =
np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.plot(x, y)
plt.grid()
plt.show()
```

Result:



- **Bars: bar()**

With Pyplot, you can use the bar() function to draw bar graphs:

Example:

Draw 4 bars:

```
import matplotlib.pyplot as plt
import numpy as np
x= np.array(["A", "B", "C", "D"]) ynp.array([3, 8, 1. 101)
plt.bar(x,y)
plt.show()
```

{{{{ Add Graph }}}}

- **Histogram: hist()**

A histogram is a graph showing *frequency* distributions.
It is a graph showing the number of observations within each given interval.
Example: Say you ask for the height of 250 people, you might end up with a histogram like this:
In Matplotlib, we use the hist() function to create histograms.
The `hist()` function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

Example:

```
import matplotlib.pyplot as plt import numpy as np
x = np.random.normal(170, 10, 250)
plt.hist(x) plt.show()
```

{{{{ Add Graph }}}}

- **Subplot: subplot()**

With the subplot() function you can draw multiple plots in one figure:

Example: Draw 2 plots

```
import matplotlib.pyplot as plt import numpy as np
#plot 1:
x = np.array([0, 1, 2, 3]) y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x,y)
#plot 2:
x= np.array([0, 1, 2, 3]) y= np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.show()
```

Result:

{{{{ Add Graph }}}}

- **pie chart: pie()**

With Pyplot, you can use the pie() function to draw pie charts:

Example:

A simple pie chart:

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
plt.pie(y)
plt.show()
```

Result:

➢ **Save the plotted images into pdf: savefig()**

Using plt.savefig("myImagePDF.pdf", format="pdf", bbox_inches="tight") method, we can save a figure in PDF format.

Steps

- Create a dictionary with Column 1 and Column 2 as the keys and Values are like i and i*i, where i is from 0 to 10, respectively.
- Create a data frame using pd. DataFrame(d), d created in step 1.
- Plot the data frame with 'o' and 'rx' style.
- To save the file in PDF format, use savefig() method where the image name is myImagePDF.pdf, format = "pdf".
- To show the image, use the plt.show() method.

Example:

```
import pandas as pd
from matplotlib import pyplot as plt
d = ('Column 1': [i for i in range(10)]. Column 2': [i* i for i in
range(10)]}
df = pd.DataFrame(d)
df.plot(style=['o', 'rx'])
plt.savefig("myImagePDF.pdf", format="pdf", bbox_inches="tight")
plt.show()
```

output:

2.4 sklearn

What is Sklearn?

- An open-source Python package to implement machine learning models in Python is called Scikit-learn.
- This library supports modern algorithms like KNN, random forest, XGBoost, and SVC.
- It aids in various processes of model building, like model selection, regression, classification, clustering, and dimensionality reduction (parameter selection).

Key concepts and features

- Supervised Learning algorithms Almost all the popular supervised learning algorithms. like Linear Regression, Support Vector Machine (SVM), Decision Tree etc., are the part of scikit-learn.
- Unsupervised Learning algorithms On the other hand, it also has all the popular
- unsupervised learning algorithms from clustering, factor analysis, PCA (Principal Component Analysis) to unsupervised neural networks.
- Clustering
- This model is used for grouping unlabeled data
- Cross Validation
- It is used to check the accuracy of supervised models on unseen data.
- Dimensionality Reduction It is used for reducing the number of attributes in data which can be further used for summarisation, visualisation and feature selection.
- Ensemble methods As name suggest, it is used for combining the predictions of multiple supervised models.
- Feature extraction It is used to extract the features from data to define the attributes in image and text data.
- Feature selection
- It is used to identify useful attributes to create supervised models.
- Open Source It is open source library and also commercially usable under BSD license.

Steps to Build a Model in Sklearn:

- Step 1: Load a dataset
- Step 2: Splitting the dataset
- Step 3: Training the model

Example: using KNN (K nearest neighbors) classifier.

```
#load the iris dataset as an example
from sklearn.datasets import load_iris
iris = load_iris()

# store the feature matrix (X) and response vector (y)
```

```
X = iris.data
y = iris.target

# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.4, random_state z = 1

#training the model on training set
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

#making predictions on the testing set
y_pred knn.predict(X_test)

# comparing actual response values (y_test) with predicted response
values (y_pred)
from sklearn import metrics
print("kNN model accuracy:", metrics.accuracy_score(y_test,
y_pred))

#making prediction for out of sample data
sample = [[3, 5, 4, 2], [ 2, 3, 5, 4]]
preds = knn.predict(sample)
pred_species = [iris.target_names[p] for p in preds]

#saving the model
from sklearn.externals import joblib
joblib.dump(knn, ‘iris_knn.pkl’)
```

Output:

```
KNN model accuracy: 0.983333333333
Predictions: ['versicolor', 'virginica']
```