

## Python Packages

### **B.1 Introduction to NumPy**

NumPy is a library of Python and it is a shorthand form of Numerical Python. NumPy, along with other python packages SciPy and Matplotlib, aims ~~is aiming~~ to replace Matlab, another popular development environment, for implementing scientific ~~data-science~~ applications.

NumPy provides an array of data structure and helps in numerical analysis. NumPy is used to manipulate arrays. The manipulation includes mathematical and logical operations. It can be used for variety of tasks like shape manipulation such as Fourier analysis, and linear algebra operations.

Python, though provides list data structure that provides a mechanism of storing homogeneous and heterogeneous items, has some serious limitations. Python core list by default is a one-dimensional array. The multidimensional array needs to be implemented as a nested list. Also, the core python list does not provide operations for elementwise operations. Also, core python stores elements in non-contiguous manner making it slow.

NumPy is better than the Python list as it reduces coding time, has faster execution and uses less memory. NumPy array stores data in a continuous manner and consumes less memory. It also executes fast. It is also very convenient to use it in data applications.

### ***NumPy Data Structures***

The important characteristics of defining a NumPy array are listed below:

- Data type
- Item size
- Shape —dimensions
- Data

Data type:

Data types are integers, int, float, complex other data types are Boolean, string, datetime and Python objects.

Item size is the memory requirement of data elements in bytes.

Shape is the dimension of the array.

Data are the elements of a NumPy array.

#### **B.1.1 Installation**

NumPy can be installed as:

pip install NumPy.

If it is already installed, then it can be upgraded using this following command:

```
pip install NumPy—upgrade
```

Once NumPy is installed, then one can check whether it is installed or not by executing the command in the python command shell as follows:

```
Import NumPy as np
```

If there is no error thrown up, one can concluded that NumPy is installed correctly.

One can check version of installed NumPy as:

```
>> import NumPy as np  
>> print(__version__)
```

Also, it should be noted that NumPy must be installed before other packages of python such as scilab, Pandas and Matplotlib.

### **B.1.2 Basic Commands on NumPy**

One can create a NumPy array of elements 1,2,3,4 and 5 as follows:

```
>>> import NumPy as np  
>>> x= np. array ([1,2,3,4,5])
```

The elements of the created array x can be displayed by typing the name of the array.

```
>>> x
```

Immediately, the command would display the elements of the array x as:

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

The command,

```
>>> print(type(x))
```

Prints the datatypes of array x.

The command,

```
>>>x.shape()
```

Prints the dimensions of the array x.

There is another way of creating an array using the command arange. The following command creates an array with elements in the range 1 to 10. The range would be divided by 5 and two elements would be displayed.

```
>>> np.arange (1,10,5)  
array([1, 6])
```

The result displayed is array([1, 6]).

Another way to create a NumPy array is to use the command linspace. It creates a range with the starting value to ending value with exactly specified elements. The following command creates a range as follows:

```
>>>np.linspace(1,10,5)
array([ 1. ,  3.25,  5.5 ,  7.75, 10.  ])
```

The result shows that the range 1 to 10 is divided evenly to create 5 elements. It can be observed that this is a float array.

One can read the values for a NumPy array by loading from a file also as shown below:

```
X = np.loadtxt( "sample.txt", dtype=np.uint8, delimiter="," , skiprows=1')
```

This command specifies that the data file is sample.txt, data type in uint, delimiter is comma and skiprows indicates that the header needs to be skipped.

These are some of the command used to manipulate the created array

```
x= np.array ([1,2,3,4,5,6])
```

### B.1: Array Operations of NumPy

S.No	Command	Results	Remarks
1.	x.reshape(2,3)	array([[1, 2, 3], [4, 5, 6]])	Command is used to rearrange the elements into the specified dimension 2 and 3.
2.	x.size	6	Returns the number of elements
3.	x.shape	6	Explains the dimensions of the array
4.	x.dtype	dtype('int32')	Explains the datatype of the created array
5.	x.itemsize	4	Returns the value of memory size of the elements. It is 4 bytes.
6.	np.zeros([4,5])	array([[0., 0., 0., 0., 0.], [0., 0., 0., 0., 0.], [0., 0., 0., 0., 0.], [0., 0., 0., 0., 0.]])	Creates a zero array with specified elements
7..	np.ones(4,5)	array([[1., 1., 1., 1., 1.], [1., 1., 1., 1., 1.], [1., 1., 1., 1., 1.], [1., 1., 1., 1., 1.]])	Creates a one array with specified elements
8.	a=	array([[0.69679581,	Creates a random array

	<code>np.random.random((4, 5))</code>	0.67029325, 0.43743738, 0.42247693, 0.34274265], [0.53140071, 0.63648804, 0.20716326, 0.37178029, 0.81598079], [0.91025997, 0.03569654, 0.19755513, 0.83251418, 0.5821717 ], [0.29830542, 0.73955474, 0.65891879, 0.40882818, 0.53910358]])	with the specified elements
9.	<code>x= np.random.randint(0,10,5)</code>	<code>array([9, 9, 2, 3, 6])</code>	Creates an array of integer elements in the specified range, in this case, from 0 to 10 with exactly 5 elements
10.	<code>np.random.shuffle(x)</code>	Values will change as the shuffle is random.	Shuffles the value of the array randomly
11.	<code>np.random.choice(a)</code>	Values will appear in random	Returns a value randomly

### ***Slicing Operations***

Indexing and slicing operations are used to access the elements of the array. Slicing is constructed by specifying start, stop and step parameters.

For example, consider the program segment

```
>> import NumPy as np
>> a = np.arange(15)
>> a[3:9:2]
```

The command `a[3:9:2]` would slice the array from 3 to 8 with step 2. The result would be `[3,5,7]`.

We can also mention from start as `a[3:]`. In this case all the elements starting from 3 to the end, in this case 14 would be printed. `A[3:8]` would slice between the indexes 3 and 8.

### **B.1.3 Arithmetic and Statistical Operations on NumPy**

One can create an array and apply the following commands to perform statistical operations. Let us create an array `x = [1,2,3,4,5,6]` with the command `x = np.array([1,2,3,4,5,6])`. Similarly, let us create another array `y = [5,6,7,8,9,10]`. Then the arithmetic operations on arrays can be done as follows:

```
>>> print(x+y)
```

```
>>> print(x-y)
```

```
>>> print(x*y)
```

```
>>> print(x/y)
```

Similarly, the statistical operations can be done as follows:

```
>>> print(np.mean(x))
```

```
>>> print(np.median(x))
```

```
>>>print(np.max(x))
```

```
>>>print(np.min(x))
```

Or simply by specifying the statistical operators as follows as shown in the Table A.2.

Example, >>> x.sum() returns the sum of created x array.

### **B.2: Array Operations of NumPy**

<b>S.N o.</b>	<b>Comma nd</b>	<b>Results</b>	<b>Remarks</b>
1.	x.sum()	21	Returns the sum of the array
2.	x.min()	1	Returns the minimum of the array
3.	x.max()	6	Returns the maximum of the array
4.	x.mean()	3.5	Returns the average of the array
5.	x.var()	2.9166666666666666 65	Returns variance of the array x
6.	x.std	1.70782512765993 3	Returns the standard deviation of the array

### **B.1.4 2D Arrays in NumPy**

Using the same logic, 2d arrays can be created as follows:

x = np.array([[1,2,3],[2,2,2]]) would create a 2D matrix as follows:

```
array([[1, 2, 3],  
       [2, 2, 2]])
```

The command,

```
x.sum(axis=1)
```

would create a result array([6, 6]), by adding the elements of the row wise. It can be observed that it is 1+2+3=6 and 2+2+2 = 6.

Array operations can be done as shown as follows:

```
>>> from NumPy import array
>>>x=array([4,5,6])
>>>print(x)
```

It can be observed that the tag np. Is missing. Similarly, all vector operations can be done

```
>>> from NumPy import array
>>>x=array([4,5,6])
>>>y=([10,10,10])
>>>c = x * y
>>>print(c)
```

Similarly, matrix can be created as follows:

Printing Arrays in 2D

```
>>>from NumPy import array
>>>x=([1,0,0],[0,1,0])
>>>print(x)
```

Print(x) would print the contents of 2D matrix x.

```
>>>y=([1,1,1],[1,1,1])
>>>print(y)
>>>c=x+y
>>>print(c)
```

The following code segment explains the manipulation of 2D arrays:

```
>> import NumPy as np
>> x = np.array([[2,3],[6,7]])
>>y= np.array([[5,6],[8,7]])
>>print(np.add(x,y))
```

More matrix commands are given in Table A.3.

The command,

```
x.sum(axis=0)
```

would add the elements in column wise.

## ***Ravel***

The command `ravel` is used to flatten the data to one-dimensional arrays and is very useful in data science applications. The syntax is `np.ravel(a,order)`. The options are 'F' by default. The option 'C' can be used for row-major ordering and 'A' for column major ordering.

```
>> import NumPy as np
```

```
>> a1 = np.arange(15).reshape(3, 5)
```

The displaying of `a1` would result as

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The flattening of the array can be done as follows:

```
>> a1.ravel()
```

This would display as

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

## **B.3: Matrix Operations of NumPy**

S.No	Command	Remarks
1.	<code>np.add(x, y)</code>	Returns the addition of two matrices x and y
2.	<code>np.sub(x, y)</code>	Returns the subtraction of two matrices x and y
3.	<code>np.matmul(x,y)</code>	Returns the multiplication of two matrices x and y
4.	<code>np.divide(x,y)</code>	Returns the division of two matrices x and y
5.	<code>np.matlib.empty((3,3))</code>	Creates an empty matrix with dimensions 3,3 with random elements
6.	<code>np.matlib.zeros((3,3))</code>	Creates a zero matrix with dimensions 3,3
7.	<code>np.matlib.ones((2,2))</code>	Creates a ones matrix with dimensions 3,3
8.	<code>np.matlib.identity(3, dtype = float)</code>	Creates an identity matrix of dimensions 3 X 3.

9.	<code>np.matlib.rand(3, 3)</code>	Creates a 3 by 3 matrix with random elements
10.	<code>np.linalg.det(x)</code>	Returns the determinant of a matrix x

### B.2.1 Matplotlib and Seaborn

One of the most useful packages for data visualization is Matplotlib. It makes use of NumPy of python. It helps to create charts for machine learning. Pylab is a procedural interface for Matplotlib. John.D.Hunter designed Matplotlib in 2003.

Matplotlib can be installed using the command:

```
pip install matplotlib
```

Matplotlib along with Pyplot is equivalent to an environment like MATLAB.

Let us create a simple plot for sine wave. This can be accomplished as follows:

Import matplotlib.pyplot as plt.

This is how pyplot of matplotlib is imported. plt is an alias created for matplotlib.pyplot. the value 0 to  $2\pi$  is created with a function.

```
x= np.arange(0,math.pi*2,0.1)
```

The y value is  $y = \sin(x)$ . therefore,

```
y= np.sin(x)
```

The values can be plotted as:

```
plt.plot(x,y)
```

Plot is the simple command for plotting. For example a function  $y=x^2$  can be plotted as below:

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

```
import numpy as np
```

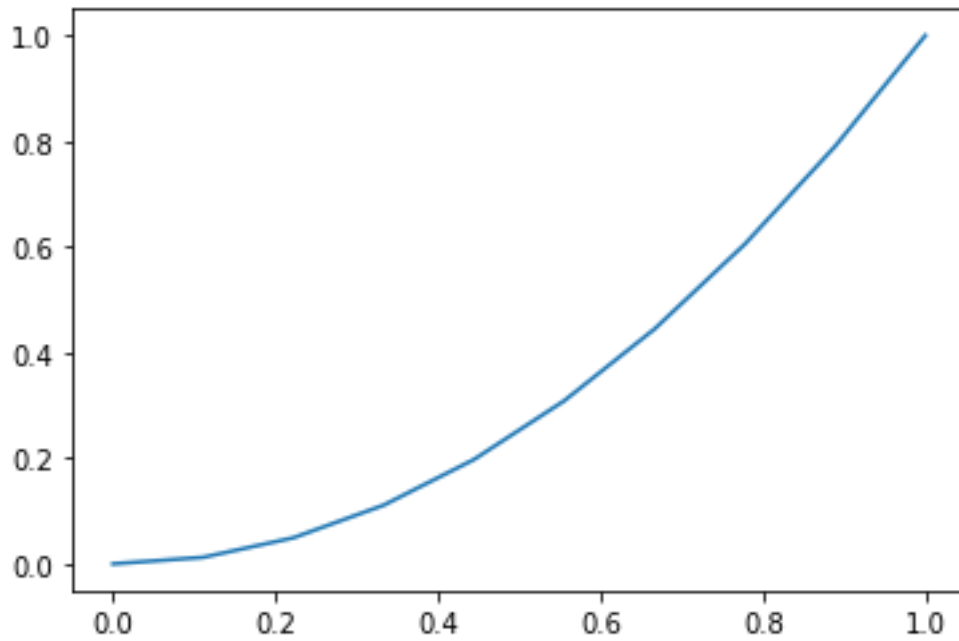
```
x= np.linspace(0,1,10)
```

```
y=x**2
```

```
plt.plot(x,y)
```

This would produce a image as shown below in Fig. B.1.

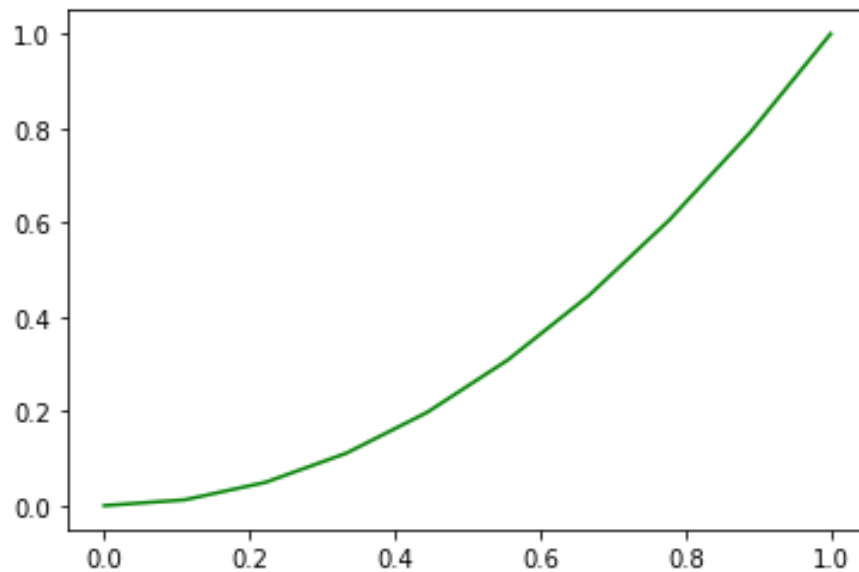




**Figure B.1:** Sample Plot

The symbols can be changed and colour can also be changed:

`plt.plot(x,y,'g')` can produce plots in bold dots as shown in Figure B.2.



**Figure B.2:** Sample Plot in Bold

The title can be created as follows:

#### **B.4: Matplotlib Commands**

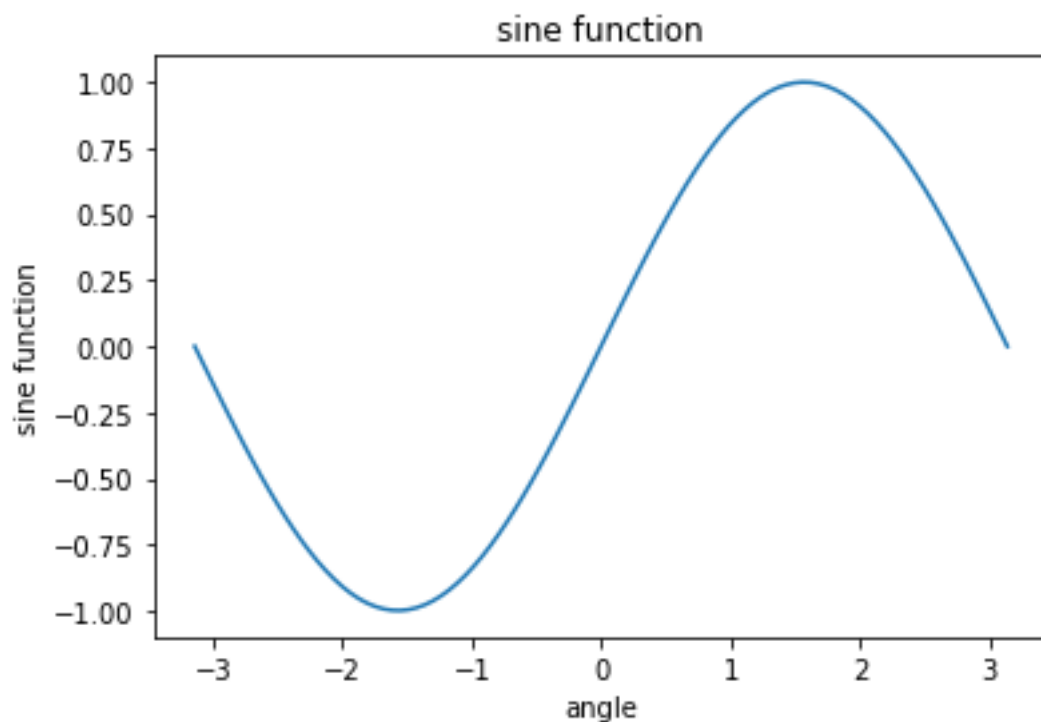
Comman	Remarks
--------	---------

<b>ds</b>	
plt.xlabel	Creates a label for x axis
plt.ylabel	Creates a label for y axis
plt.title	Creates a label for the graph

The complete program can be given as:

```
from matplotlib import pyplot as plt
import numpy as np
import math
x = np.linspace(-np.pi,np.pi,100)
y=np.sin(x)
plt.plot(x,y)
plt.xlabel('angle')
plt.ylabel('sine function')
plt.title('sine function')
plt.show()
```

This program results in a graph like this as shown in Figure B.3.



**Figure B.3:** Sample Sine Plot

In Jupyter notebook, % Matplotlib in line can be used to create graph in the notebook itself. Many graphs can be created in a single plot. The command

```
plt.subplot(nrows,ncols,Index)
```

can be used to create a grid with the specified rows and columns.

For example, the command `plt.subplot(211)` creates space for the 1<sup>st</sup> plot and `plt.subplot(212)` creates space for 2<sup>nd</sup> plot.

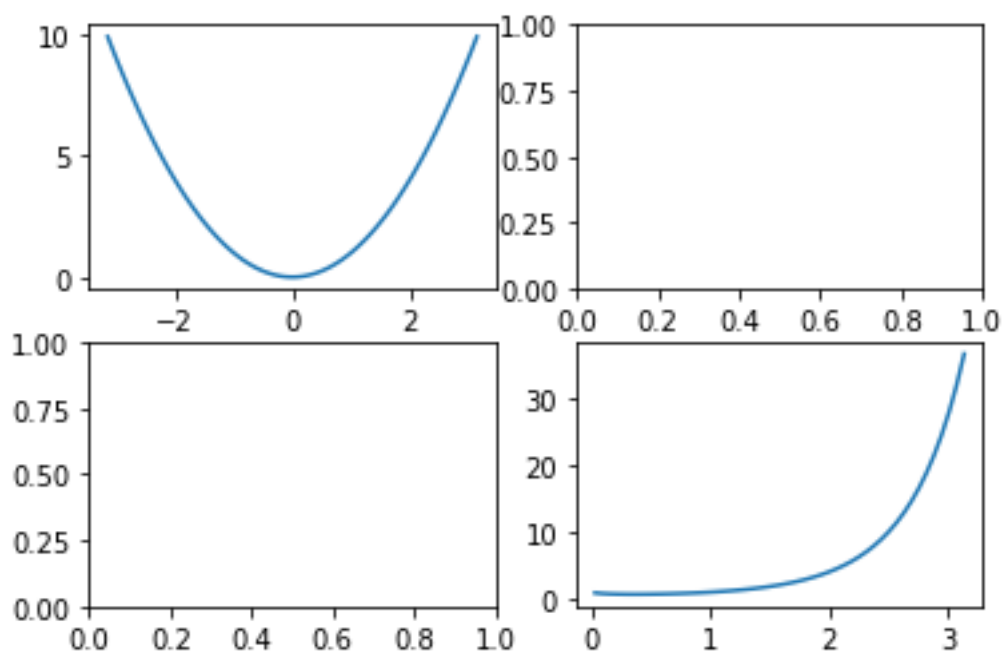
The same thing can be created using `subplots()` function also.

```
fig,a=plt.subplots(2,2)
```

`a[0][0].plot(x,x*x)` //creates a plot in the first grid of the 2\*2matrix

`a[1][1].plot(x,x**x)` //creates a plot in the last grid of the 2\*2 matrix

This results in a graph as shown in Figure B.4.



**Figure B.4:** Sample Plot

It can be noted that (0,1) and (1,0) grids are empty.

The grid can be displayed by using the command:

```
axes.grid(true)
```

Matplotlib automatically, takes care of the spacing of points on axes. This can be changed by using the command ticks:

```
ax.set_xticks([1,3,5,7]).
```

This command marks the point as per the list. The labels for the tick marks can be done by using the command:

```
ax.set_xlabel(['one','three','five','seven'])
```

Some of the examples of Matplotlib are shown as follows:

**Table B.5:** Axes Commands

Commands	Syntax
ax.bar	ax.bar(x,height,width,bottom,align)
ax.hist	ax.hist(x,bins)
ax.pie	ax.pie(x,labels,colours,autopct)
ax.scatter	ax.scatter(x,y,color='r')
ax.boxplot	ax.boxplot(data)
ax.violinplot	ax.violinplot(data)

Seaborn is an alternative to Matplotlib. It is a wrapper for Matplotlib to created plots.

- ❖ Visualizing univariate data
- ❖ Visualization of bivariate data
- ❖ Visualization of linear model
- ❖ Plotting of graphs

Using pip installation:

```
pip install seaborn
```

Seaborn has many data sets. The following commands loads a predefined dataset tips:

```
import seaborn as sb
df= sb.load_dataset('tips')
print df.head
```

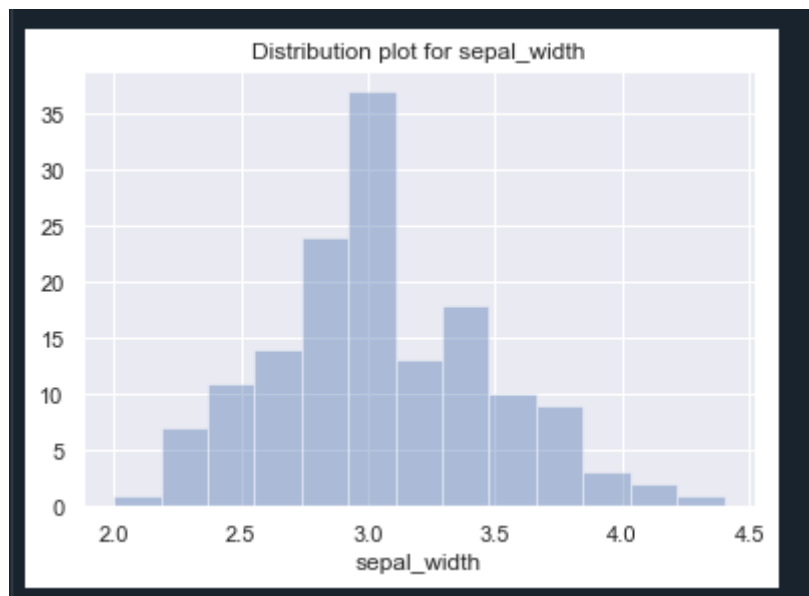
## Plots with Seaborn

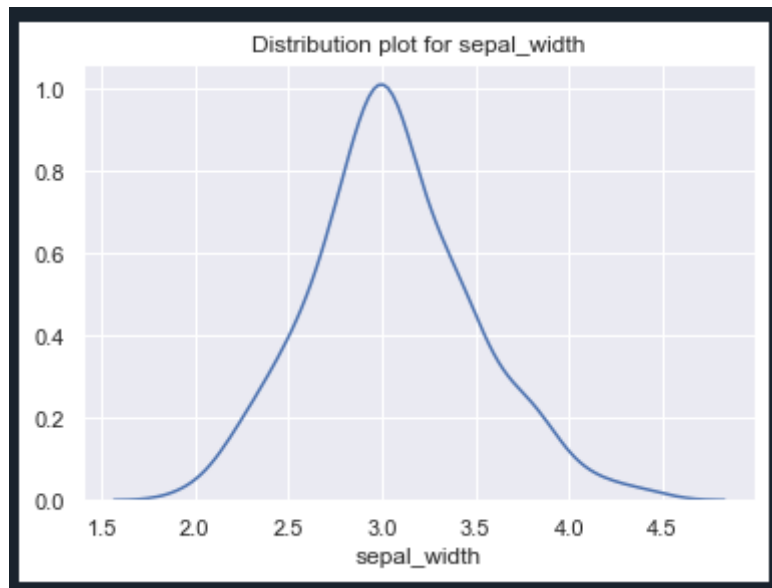
### Distplot or Density Plots

Distplot is used to plot univariate distribution. The following program is useful for plotting a distribution of sepal-length.

```
sns.distplot(df1['sepal_width'],kde = False).set_title('Distribution plot for sepal_width')  
plt.show()  
  
sns.distplot(df1['sepal_width'],hist = False).set_title('Distribution plot for sepal_width')  
plt.show()
```

Instead of Kde=false, setting hist as false result in Kernel density estimates plot of the attribute. The corresponding plots are shown Figure B.5.





**Figure B.5:** Distribution Plot

### ***Jointplot***

Jointplot is used to visualize bivariate distribution.

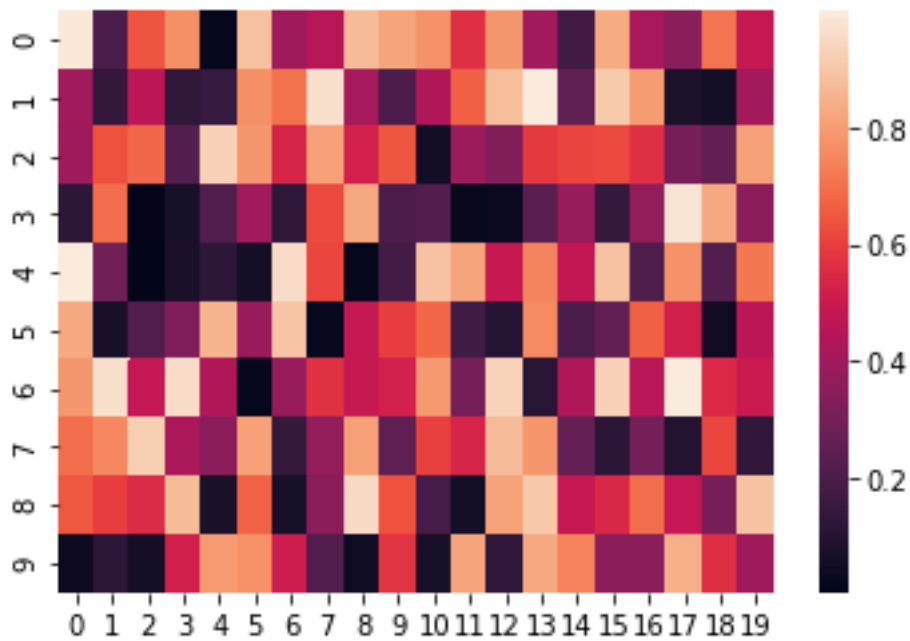
```
sb.jointplot(x='petal-length',y='petal-width',data=df)
```

### ***Heatmaps***

Heatmaps are useful as they colour different numbers in different colours. Therefore, it is better to visualize the data. The following code segment is used to display the heatmap for the uniformly distrusted random data in the range 10,20.

```
>> import numpy as np
>> import seaborn as sns
>> uniform_dt = np.random.rand(10,20)
>> ax = sns.heatmap(uniform_dt)
```

This would display the heatmap as shown in Fig. B.6.



**Figure B.6:** Heatmap

### ***Pairplots***

Pairplot is used to plot multiple pairwise bivariate distributions. The matrix has diagonal univariate plot and the rest of the matrix as pairwise bivariate distribution.

The syntax is given as:

```
seaborn.pairplot(df,hue,palette,kind,diag-kind)
```

df -> data frame

hue -> refers to the variable

palette -> sat of colours

kind -> this can be scatter or reg

diag-kind -> can be either histogram or kernel dist.estimations

### ***Stripplot***

This is used for plotting categorical data

### ***Box Plot and Violin Plot***

These plots are useful for distribution of data. Boxplots are useful for representing five-point summary and violin plot represents box plot with kde.

### **B.3.1 Pandas**

Pandas is a name from “panel data” and was designed by Wes McKinney in 2008.

Pandas is used for data manipulation and analysis. Pandas can be used for:

1. One can load data
2. Data preparation for data analysis can be done
3. Data manipulation can be done
4. Statistical models creation
5. Analyzing data

Pandas can be created using the command

```
pip install pandas
```

Pandas provide data structure like series, data frame and panel for processing one dimensional, two dimensional and three-dimensional data, respectively. Higher dimensional data structure are containers of lower dimensional data.

One dimensional series can be created using NumPy array as follows:

```
import pandas as pd
import numpy as np
data=np.array([1,2,3,4,5])
s=pd.Series(data)
Print s
```

The output of this would be

```
0    1
1    2
2    3
3    4
4    5
```

### B.3.2 Input-Output Tools

Python Pandas can read an .CSV file as follows:

Import Pandas as pd

```
df=pd.read_csv("sample.csv")
print df
```

Similarly, Pandas can read from different sources from data as

```
read_csv()
read_excel()
read_json()
read_json()
```



```
read_sql()
```

One can index it as follows:

```
df=pd.read_csv("Sample.csv",index col=['SNO'])
```

```
Print df
```

One can create a header name as follows:

```
df=pd.read_csv("Sample.csv",names=['mark1','mark2'])
```

```
print df
```

One can skip the rows, say 3, using the following command

```
df=pd.read_CSV("Sample.CSV",skiprows=3)
```

```
print df
```

### B.3.3 Basic Functionalities of Data Frames

The accessing of an element of the created dataframe in Pandas is done through the command `iloc` and `loc`. The rows of the dataframe can be accessed using `iloc` method. For example, `df.iloc[0]` returns the first row of the dataframe.

`df.iloc[:]` returns all the elements of the dataframe.

`Df.iloc[row:column]` returns the elements of the row to column specified.

`loc` is similar to `iloc`. But, `loc` allows to index the column items or labels. The details are given in the following Table B.6.

**Table B.6:** Access Operations in Pandas

Command	Results
<code>df.loc['row']</code>	Pass row number to <code>.loc</code> to select that row
<code>df.iloc['row']</code>	Pass row number to select integer location
<code>df.df[2:3]</code>	Select multiple rows using <code>:</code> operator
<code>df.append(row)</code>	Append a row to the existing data frame <code>df</code>
<code>df.drop(label)</code>	Drop the row with the label

`head()` returns the first `n` rows. This can be used as follows to print first 5 elements of the series

Import Pandas as `pd`

```
s=pd.Series(np.random.randn(10))
```

```
print s(5)
```

Similarly, these functions can be used.

tail(n) - returns the last n elements of the series or data frames

empty - returns true if series is empty

Ndim - returns number of dimensions

size - returns the number of elements

values- returns the series as N-dimensional array

The index can be changed using this command as follows:

```
s=pd.Series(data,index=[100,110,120,130,140])
```

Alternatively, series can be created from dictionary as follows:

```
data={'a':100,'b':200,'c':300}
```

Then the output would be like with the dictionary key is used as an index.

A 100

B 200

C 300

A scalar array can be created as follows:

```
s=pd.Series(10,index=[a,b,c])
```

to create a scalar array as

A 10

B 10

C 10

Assuming a series

```
s=pd.Series([1,2,3,4,5],index=[100,200,300,400,500])
```

The print operations are shown in Table B.7.

**Table B.7:** Print Operations in Series

Command	Result
Print s[0]	Returns the first elements of the array
Print s[:2]	Retrieve the first two elements of the array
Print s[-2:]	Retrieve the last two elements of the array
Print s['100']	Retrieve the element whose index is 100

A data frame is a 2D structure that is useful for data analysis. For example, the following Table B.8 can be created as follows:

**Table B.8:** Sample Data

Reg.n o	Marks
100	37
101	40
102	42
103	57
104	60

```
data=[[100,37],[101,40],[102,42],[103,57],[105,60]]
df=pd.DataFrame( data,columns=['reg-no','marks'],dtype=float )
print df
```

Data frame can be created using dictionary also

```
import Pandas as pd
data=[(100:37),(101,40),(102,42),(103,57),(104,60)]
df=pd.DataFrame(data)
print df
```

The following table can be constructed as a set of Pandas series as shown in Table B.9.

**Table B.9:** Matrix Operations of NumPy

Roll no.	Marks 1	Marks 2
100	37	40
101	40	42
102	42	43
103	57	58
104	60	67

```
import Pandas as pd
dict= {'marks1': pd.Series([37,40,42,57,60],index=[100,101,102,103,104]), \
'marks2': pd.Series([40,42,43,58,67],index=[100,101,102,103,104])}
df = pd.DataFrame.from_dict(dict)
print(df['marks1'])
```

A new column total can be added as follows:

```
df['total']= df['marks1']+df['marks2']
```

Similarly, a column, say marks1 can be deleted using the command

```
del df['marks1']
```

### **B.3.4 Descriptive Statistics**

The statistics can be created as follows:

```
df.sum() returns the sum of the values
```

```
df.sum(0 returns the sum of the values with axis= 1
```

```
df.mean() returns the average values of the data frame
```

```
df.std() returns the standard deviation of the numerical columns
```

```
df.mode() returns the mode of values
```

```
df.min() returns the minimum value
```

```
df.max() returns the maximum value
```

```
df.prod() returns the product of the values
```

```
df.cumsum() returns the cumulative sum
```

```
df.cumprod() returns the cumulative product
```

```
df.describe(include='all') returns the descriptions of the entire  
data frame
```

### ***Covariance***

Covariance of two series can be computed as follows:

```
import Pandas as pd
```

```
import NumPy as np
```

```
s1= pd.Series([1,2,3,4,5])
```

```
s2= pd.Series([4,3,6,7,8])
```

```
print (s1.cov(s2))
```

Similarly, the correlation may be computed by using the command `s1.corr(s2)`

### ***Aggregation***

One can group using these commands.

```
df.groupby('key') group the items using the key
```

`df.groupby('key1','key2')` group the items using keys, key1 and key2

### ***Pandas Visualization***

Let us assume the marks for three subjects for ten students, let us create the data using random number. Then the bar plot can be created as below:

```
df=pd.dataframe(np.random.rand(15,2), columns=['mark1','mark2'])  
df.plot.bar()
```

The following commands can be used to create plots as shown in Table B.10.

**Table B.10:** Plot Commands in Pandas

Command	Remarks
<code>df.plot.bar(Stacked=true)</code>	Creates a bar h chart
<code>df.plot.hist(bins=5)</code>	Create a histogram with five bins
<code>df.plot.box()</code>	Create a box plot to visualize the distributions
<code>df.plot.area()</code>	Create an area plot
<code>df.plot.scatter(x='column1',y='column2')</code>	Create a scatter plot
<code>df.plot.pie(subplot=true)</code>	Create a pie chart

### **B.4.1 SciPy Package**

SciPy is another related python package that can be used for many linear algebra applications. SciPy is helpful in linear algebra applications. It can be useful to find determinant and inverse of matrices.

```
>>> from SciPy import linalg
```

```
>>> import NumPy as np
```

```
>>> myarray=np.array([[1,2],[3,4]])
```

```
>>> linalg.det(myarray)
```

Reurns the determinant value of myarray. In this case, it is -2.

Similarly, the matrix inverse can be obtained as below:

```
>>>linalg.inv(myarray)
```

The following code illustrates the method of computing Chisquare test using SciPy.

```
import Pandas as pd
import seaborn as sns
import NumPy as np
import SciPy as sy
from SciPy.stats import chisquare
from SciPy.stats import ttest_ind
x=chisquare([40,10,20,30],f_exp=[30,20,30,20])
print(x)
```

### **B.5.1 Scikit-Learn**

Scikit-learn or Sklearn is a popular python package for implementing machine learning algorithms. Scikit-learn was developed by David Cournapeau as a Google summer project. Scikit-Learn can be installed using pip command as

```
pip install -U scikit-learn
```

or

```
conda install -c anaconda scikit-learn
```

The Scikit-Learn is built on NumPy, SciPy and Matplotlib, and Pandas. Scikit-Learn can implement supervised learning algorithms, unsupervised learning algorithms like clustering algorithm. This entire lab manual is implemented using Scikit-Learn package. Scikit-Learn comes with some ready datasets like iris and digits. The dataset can be loaded as follows:

```
From sklearn.datasets import load_iris
```

```
Iris= load_iris()
```

### **B.5.2 Scikit-Learn Cycle**

The steps of Scikit-Learn programs are as follows

1. Arranging data as x and y
2. Choose a model. this is done by importing the class

3. The model parameters can be chosen. These parameters are called hyper parameters
4. Model fitting. This can be done using .fit() command
5. Apply model. This can be done by predict() or transform() to get the labels of the unknown data.

The dataset can be split into training and testing dataset as follows:

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=1)

joblib.load('sample\_model.joblib')

### **B.5.3 Scikit-Learn Preprocessing**

Mean removal

Input\_data = np.array([-3,-2,3] [-2,1,4] [4,3,8])

data\_scaled= preprocessing.scale(input\_data)

The command applies of threshold value of 0.6 and outputs data.

output\_data= preprocessing.Binarizer (threshold=0.6). transform  
(input\_data)

The command MinMaxscaler applies the scaling of input for the given input data. The command is given as follows:

data\_scaler= preprocessing.MinMaxscaler (feature\_range=(0,1))

data\_scaler= data\_scaler.fit\_transform(input\_data)

The command normalize can scale the input data to a common scale. The L1 and L2 normalization can be implemented using the following commands.

data\_output= preprocessing.normalize(input\_data, norm='l1')

data\_output= preprocessing.normalize(input\_data, norm='l2')

### **B.5.4 Model Fitting**

Model fitting can be done using .fit command. For example, the following commands are used to create KNN classifier,

class\_knn= KNeighborsClassifier (n=4)

```
class_Knn_fit(X_train, y_train)
```

The constructed model can be stored with the help of joblib package,

```
joblib.dump(classifier_knn, 'sample_model.joblib')
```

and can be loaded as follows:

A simple constructed SVM classifier is as shown below:

```
from sklearn import svm
from sklearn.svm import SVC
X = [[0, 1], [1, 2]]
y = [0, 1]
clf = svm.SVC()
clf.fit(X, y)
SVC()
print(clf.predict([[2., 2.]])
```

SVM classifier should return the result class 1.

### **B.6.1 Keras**

Keras is an API that solves the problem of lack of support of scikit learn for creating neural and deep learning network. Keras helps to create deep neural network and is built on the frameworks like tensor flow, CNTK and Theano. Some of the advantages of Keras are as follows:

1. It is user friendly and modular.
2. Many advanced functions like optimizers, layers are built into Keras.
3. Keras is specialized for neural networks
4. It can work with python and is an open source.

The input can be given using NumPy or Pandas. The following code shows the way of reading data using NumPy:

```
ds = loadtxt('Iris.csv', delimiter=',')
```



the read data can be split into input and target as follows:

```
X = ds[:,0:4]
```

```
y = ds[:,4]
```

The simplest model structure is sequential model. It can be imported as follows:

```
from keras.models import Sequential
```

The Sequential model can take layers as parameters as shown here:

```
model= Sequential ([  
    # All layers description  
])
```

Or it can be done by adding layers one by one to form a neural network structure. Keras core API has a concept of layers. This can be done using a set of

```
model.add commands
```

A layer can be visualized as a set of nodes. Some of the layers that are provided by Keras are:

### ***Dense Layer***

Dense layer is directly connected to the inputs and output, it is a fully connected layer. A dense layer specify the number of neurons or nodes as an argument and an activation function. The activation function can be sigmoid, tanh or Relu. The following layer creates a dense layer with its neurons with activation function Relu

```
model.add(Dense(12,input_dim=4,activation='relu'))
```

Let us assume that input data has 4 features and three classes. Then a Keras model can be created as follows:

```
from Keras.models import sequential  
  
from tensorflow import random  
  
model= sequential()
```

One hidden layer of 10 neurons with one activation function tanh for 4 features can be created as follows:

```
from keras layers import Dense, activation  
  
model.add(dense(10,activation='tanh',input_dim=4))
```

The general format is as follows:

```
model.add(dense(units,input_dim=input_shape.'Activation Function'))
```

Keras provides many activation functions such as step, linear, sigmoid, tanh and ReLU. A sample keras tanh function can be given as follows: `model.add(Activation('tanh'))`. In the case of convolutional neural networks(CNN), the `model.add` command can create additional layers like below:

### ***Convolutional Layer***

Convolutional layer has many filters or kernels. Kernels can be of any dimensions. Kernels are convolved with input image to produce many features.

### ***Pooling Layers***

Pooling is another layer and can be either max pooling or average pooling.

### ***Recurrent Layers***

Recurrent layers are used to process sequential data like time series data or natural language constructs.

### ***Model Compile***

Model compile requires specification of additional parameters like loss functions, optimizers and metrics. The loss function for binary classification problems is Adam. Adam is based on stochastic gradient methods that gives the best results using auto tuning. The Keras command can be like

```
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
```

Adam is a default optimizer. The loss functions may be either `binary_crossentropy` for two class problems or `categorical_crossentropy` for multi-class problems. Many metrics can be used. Accuracy is one metric that is commonly used.

### ***Model Fitting***

Once the model is created, then the model can be executed. The execution can be done as follows:

```
model.fit(X,y,epochs=50,batch_size=10)
```

Once the model is compiled, then the data is fit into it. The parameters of this command is as follows:

X- the feature array

y- the target data

epochs- number of times the model should run. An epoch may have one or more batches. The batch size indicates the samples after updating of weight is done.

batch-size – the number of samples

validation-split – the proportion of data used for validation

shuffle- flag to indicate whether shuffling of data is required before epoch.

Keras evaluate command can be used to evaluate the model using test data. The evaluation of the model is done using the command evaluate(). This can predict the values of the input. This can be done as follows:

```
_,accuracy = model.evaluate(X,y)
```

The prediction for the unseen or test data is done as follows:

```
Predict = model.predict_classes(X)
```

Similarly, many hidden layers can be added.

The constructed model can be visualized as

```
model.summary()
```

The constructed model can be evaluated as follows:

```
result= model.fit(X,y,epochs=50,batch_size=10,verbose=1,validation-  
split=0.2,shuffle=false)
```

The loss functions can be plotted as

```
plt.plot(result.result['loss'])
```

The accuracy values can be plotted as

```
plt.plot(result.result['accuracy'])
```

A constructed simple neural network is given as follows:

```
model = Sequential()  
model.add(Dense(12, input_dim=8,activation='relu'))  
model.add(Dense(8, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))  
# compile model  
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])  
# Fit the model  
model.fit(X, y, epochs=150, batch_size=10, verbose=0)  
# evaluate the model  
scores = model.evaluate(X, Y, verbose=0)
```

```
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```