



**G H Raisoni College of Engineering and Management,
Pune**

(An Autonomous Institute Affiliated to Savitribai Phule Pune University)

Gate No. 1200, Domkhel Road, Wagholi, Pune-412207



**Department of
Computer Engineering
D-19
Lab Manual (2021-22)
Pattern-2019**

Class: TY B.Tech.

Term: VI

Machine Learning Lab (BCOP19310)

Faculty Name: Mrs. Gayatri Bedre/Miss Geeta Zawar

**G. H. Raison College of Engineering & Management, Wagholi, Pune –
412 207**

Department: Computer Engineering

Course: Machine Learning Lab (BCOP19310)

**Class: TY BTECH
Internal Marks:25
Credit:01**

**Division: A &B
Cont. Ass: 25 Marks
Ext.: 25 Marks
Total: 50 Marks**

Course Outcomes	
CO1	Use of different Machine learning models and methods of problem-solving
CO2	Apply feature selection to create an accurate predictive model
CO3	Analyze various machine learning models
CO4	Combine different machine learning models for improved accuracy

List of Experiment

Sr.No	List of Laboratory Assignments	Mapped Course Outcomes
1	Understanding data formats of Pandas: Series, Data frame, Panel; Creating, Appending, Deleting. Importing different types of Datasets. Working with Dimensions	CO1, CO2
2	Type conversions from different datatype into Series, Data frame and Panel, Necessary operations like renaming, traversing columns and indexes, Statistics on data formats of Pandas	CO1, CO2
3	Understanding data formats of NumPy: ndarrays (1D, 2D and 3D arrays), Array creation routines	CO1,2
4	Matplotlib plotting for Data visualization	CO1, CO2
5	Tweaking Colors, Symbols, Formulations. Plotting Categorical data, 3D Axes, Parametric Curves, Trigonometry functions, Histogram, Bar, Pie chart. Graph plotting using Pandas	CO2
6	Introduction to SciPy, Scikit-learn, Importing Algorithm Classes and creating objects with parametric values	CO2, CO3
7	Dataset selection: Dataset for Classification / Regression / Associative Rule Mining. and Analysis: Independent Variables, Dependent Variables, Handling Missing Values, Categorical data, and Feature Scaling	CO1,2,3
8	Regression: Performing Simple Linear Regression over a salary dataset and predict salaries according to their experience years	CO1,2,3,4
9	Regression & Data Valuation: Performing Multi-linear Regression (using appropriate Model) to evaluate with data which is useful for model training	CO4
10	Regression: Using Polynomial regression resolve bluff query for new employee salary	CO4
11	Classification: Using KNN (with WCSS), NB Predicting if a customer with certain age and Salary will purchase a product or not	CO4
12	Classification: Using DT and SVM Predicting if a customer with certain age and Salary will purchase a product or not	CO4
13	Clustering: Using K-Means clustering, determine Customers of a Mall according to Categories so as to launch a scheme for business growth a product or not for imbalanced data and determining Fitting issues and Sampling methods and Optimizing techniques	CO3, CO4

ASSIGNMENT NUMBER 1

Aim: Python Overview: Variables, String, List, Dictionary, Tuples, Comparison operators

Theory:

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example –

```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result –

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example –

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example –

```
var1 = 1  
var2 = 10
```

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is –

```
del var1[,var2[,var3[...varN]]]
```

You can delete a single object or multiple objects by using the del statement. For example –

```
del var
```

```
del var_a, var_b
```

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFA BCE CBDAECBFBAEI	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by $x + yj$, where x and y are the real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
#!/usr/bin/python

str = 'Hello World!'

print str      # Prints complete string
print str[0]   # Prints first character of the string
print str[2:5] # Prints characters starting from 3rd to 5th
print str[2:]  # Prints string starting from 3rd character
print str * 2  # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example –

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylst = [123, 'john']

print list      # Prints complete list
print list[0]   # Prints first element of the list
print list[1:3] # Prints elements starting from 2nd till 3rd
print list[2:]  # Prints elements starting from 3rd element
print tinylst * 2 # Prints list two times
print list + tinylst # Prints concatenated lists
```

This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
```



```

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists

```

This produce the following result –

```

('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')

```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list

```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```

#!/usr/bin/python

```

```
dict = {}

dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}

print dict['one']      # Prints value for 'one' key
print dict[2]          # Prints value for 2 key
print tinydict         # Prints complete dictionary
print tinydict.keys()  # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Python - Basic Operators:

Operators are the constructs which can manipulate the value of operands.

Consider the expression $4 + 5 = 9$. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look on all operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10$ to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$, $-11 // 3 = -4$, $-11.0 // 3 = -4.0$

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value

		of $a + b$ into c
<code>+=</code> Add AND	It adds right operand to the left operand and assign the result to left operand	$c += a$ is equivalent to $c = c + a$
<code>-=</code> Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	$c -= a$ is equivalent to $c = c - a$
<code>*=</code> Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	$c *= a$ is equivalent to $c = c * a$
<code>/=</code> Divide AND	It divides left operand with the right operand and assign the result to left operand	$c /= a$ is equivalent to $c = c / a$
<code>%=</code> Modulus AND	It takes modulus using two operands and assign the result to left operand	$c \% = a$ is equivalent to $c = c \% a$
<code>**=</code> Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	$c ** = a$ is equivalent to $c = c ** a$
<code>//=</code> Floor Division	It performs floor division on operators and assign value to the left operand	$c //= a$ is equivalent to $c = c // a$

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if $a = 60$; and $b = 13$;
Now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
Binary OR	It copies a bit if it exists in either operand.	(a b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)

>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)
-----------------------	--	----------------------------------

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below –

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).

is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).
--------	---	---

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description	
1	**	Exponentiation (raise to the power)
2	~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % //	Multiply, divide, modulo and floor division
4	+ -	Addition and subtraction
5	>> <<	Right and left bitwise shift
6	&	Bitwise 'AND'
7	^ 	Bitwise exclusive 'OR' and regular 'OR'
8	<= < > >=	Comparison operators
9	<> == !=	Equality operators
10	= %= /= //= -= += *= **=	Assignment operators
11	is is not	Identity operators

12	in not in	Membership operators
13	not or and	Logical operators

Exercise:

1. Write a Python program to display the first and last colors from the following list.

```
color_list = ["Red","Green","White" ,"Black"]
```

2. Write a Python program to display the examination schedule. (extract the date from exam_st_date).

```
exam_st_date = (11, 12, 2014)
```

Sample Output: The examination will start from : 11 / 12 / 2014

3. Write a Python program to print the following floating numbers upto 2 decimal places: 3.1415926, 12.9999

Conclusion: In this experiment, we studied Variables, String, List, Dictionary, Tuples, Comparison operators, and how to work on the dataset, using Numpy and Panda Libraries.

ASSIGNMENT NUMBER 2

Aim:

Understanding data formats of Pandas: Series, Dataframe, Panel; Creating, Appending, Deleting. Importing different types of Datasets.

Theory:

Pandas contain high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas are built on top of NumPy and make it easy to use in NumPy-centric applications.

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

pandas.Series

A pandas Series can be created using the following constructor –

```
pandas.Series( data, index, dtype, copy)
```

The parameters of the constructor are as follows –

S.No	Parameter & Description
1	data data takes various forms like ndarray, list, constants
2	index Index values must be unique and hashable, same length as data. Default np.arange(n) if no index is passed.
3	dtype dtype is for data type. If None, data type will be inferred
4	copy Copy data. Default False

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

Create an Empty Series

A basic series, which can be created is an Empty Series.

Example

```
#import the pandas library and aliasing as pd
```

```
import pandas as pd  
s = pd.Series()  
print s
```

Its **output** is as follows –

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

Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., [0,1,2,3.... **range(len(array))-1**].

Example 1

```
#import the pandas library and aliasing as pd  
import pandas as pd  
import numpy as np  
data = np.array(['a','b','c','d'])  
s = pd.Series(data)  
print s
```

Its **output** is as follows –

```
0  a  
1  b  
2  c  
3  d  
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

Example 2

```
#import the pandas library and aliasing as pd  
import pandas as pd  
import numpy as np  
data = np.array(['a','b','c','d'])
```

```
s = pd.Series(data,index=[100,101,102,103])  
print s
```

Its **output** is as follows –

```
100 a  
101 b  
102 c  
103 d  
dtype: object
```

We passed the index values here. Now we can see the customized indexed values in the output.

Create a Series from dict

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

Example 1

```
#import the pandas library and aliasing as pd  
import pandas as pd  
import numpy as np  
data = {'a': 0., 'b': 1., 'c': 2.}  
s = pd.Series(data)  
print s
```

Its **output** is as follows –

```
a 0.0  
b 1.0  
c 2.0  
dtype: float64
```

Observe – Dictionary keys are used to construct index.

Example 2

```
#import the pandas library and aliasing as pd  
import pandas as pd  
import numpy as np
```

```
data = {'a': 0., 'b': 1., 'c': 2.}
s = pd.Series(data, index=['b', 'c', 'd', 'a'])
print s
```

Its **output** is as follows –

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe – Index order is persisted and the missing element is filled with NaN (Not a Number).

Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print s
```

Its **output** is as follows –

```
0 5
1 5
2 5
3 5
dtype: int64
```

Accessing Data from Series with Position

Data in the series can be accessed similar to that in an **ndarray**.

Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

```
import pandas as pd
```

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
```

#retrieve the first element

```
print s[0]
```

Its **output** is as follows –

```
1
```

Example 2

Retrieve the first three elements in the Series. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index)

```
import pandas as pd
```

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
```

#retrieve the first three element

```
print s[:3]
```

Its **output** is as follows –

```
a 1
b 2
c 3
dtype: int64
```

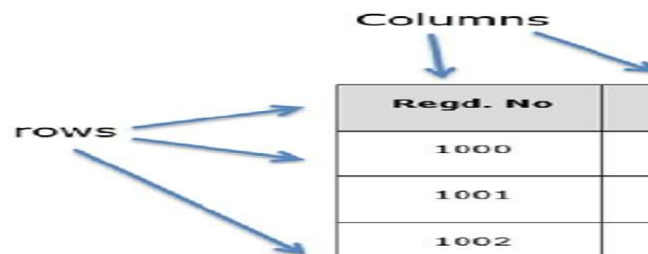
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Structure

Let us assume that we are creating a data frame with student's data.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor –

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows –

S.No	Parameter & Description
1	data data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
2	index For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.
3	columns For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.
4	dtype Data type of each column.

4

copy

This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like –

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print df
```

Its **output** is as follows –

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

```
import pandas as pd
```

```
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print df
```

Its **output** is as follows –

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

Example 2

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print df
```

Its **output** is as follows –

```
   Name  Age
0  Alex   10
1  Bob   12
2  Clarke 13
```

Example 3

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print df
```

Its **output** is as follows –

```
   Name  Age
0  Alex 10.0
1  Bob  12.0
2  Clarke 13.0
```

Note – Observe, the **dtype** parameter changes the type of Age column to floating point.

Create a DataFrame from Dict of ndarrays / Lists

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

Example 1

```
import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

	Age	Name
0	28	Tom
1	34	Jack
2	29	Steve
3	42	Ricky

Note – Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example 2

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])

print df
```

Its **output** is as follows –

	Age	Name
rank1	28	Tom
rank2	34	Jack
rank3	29	Steve
rank4	42	Ricky

Note – Observe, the **index** parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd

data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

	a	b	c
0	1	2	NaN
1	5	10	20.0

Note – Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd

data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data, index=['first', 'second'])

print df
```

Its **output** is as follows –

	a	b	c
first	1	2	NaN
second	5	10	20.0

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd

data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])

print df1
print df2
```

Its **output** is as follows –

```
#df1 output
   a  b
first  1  2
second 5 10

#df2 output
   a  b1
first  1 NaN
second 5 NaN
```

Note – Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
```

```
df = pd.DataFrame(d)
print df
```

Its **output** is as follows –

```
   one  two
a  1.0   1
b  2.0   2
c  3.0   3
d  NaN   4
```

Note – Observe, for the series one, there is no label ‘d’ passed, but in the result, for the **d** label, NaN is appended with NaN.

Let us now understand **column selection**, **addition**, and **deletion** through examples.

Column Selection

We will understand this by selecting a column from the DataFrame.

Example

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df ['one']
```

Its **output** is as follows –

```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

Column Addition

We will understand this by adding a new column to an existing data frame.

Example

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

# Adding a new column to an existing DataFrame object with column label by passing new
series

print ("Adding a new column by passing as Series:")

df['three']=pd.Series([10,20,30],index=['a','b','c'])

print df

print ("Adding a new column using the existing columns in DataFrame:")

df['four']=df['one']+df['three']

print df
```

Its **output** is as follows –

Adding a new column by passing as Series:

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Adding a new column using the existing columns in DataFrame:

	one	two	three	four
a	1.0	1	10.0	11.0
b	2.0	2	20.0	22.0

c	3.0	3	30.0	33.0
d	NaN	4	NaN	NaN

Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

Example

```
# Using the previous DataFrame, we will delete a column
```

```
# using del function
```

```
import pandas as pd
```

```
d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),  
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),  
     'three': pd.Series([10,20,30], index=['a','b','c'])}
```

```
df = pd.DataFrame(d)
```

```
print ("Our dataframe is:")
```

```
print df
```

```
# using del function
```

```
print ("Deleting the first column using DEL function:")
```

```
del df['one']
```

```
print df
```

```
# using pop function
```

```
print ("Deleting another column using POP function:")
```

```
df.pop('two')
```

```
print df
```

Its **output** is as follows –

Our dataframe is:

	one	three	two
a	1.0	10.0	1
b	2.0	20.0	2
c	3.0	30.0	3
d	NaN	NaN	4

Deleting the first column using DEL function:

	three	two
a	10.0	1
b	20.0	2
c	30.0	3
d	NaN	4

Deleting another column using POP function:

	three
a	10.0
b	20.0
c	30.0
d	NaN

Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

Selection by Label

Rows can be selected by passing row label to a **loc** function.

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df.loc['b']
```

Its **output** is as follows –

```
one 2.0
two 2.0
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df.iloc[2]
```

Its **output** is as follows –

```
one  3.0
two  3.0
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ‘:’ operator.

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df[2:4]
```

Its **output** is as follows –

```
   one  two
c   3.0   3
d   NaN   4
```

Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

print df
```

Its **output** is as follows –

```
a b
0 1 2
1 3 4
0 5 6
1 7 8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0
```

```
df = df.drop(0)
```

```
print df
```

Its **output** is as follows –

```
a b  
1 3 4  
1 7 8
```

In the above example, two rows were dropped because those two contain the same label 0.

A **panel** is a 3D container of data. The term **Panel data** is derived from econometrics and is partially responsible for the name pandas – **pan(el)-da(ta)-s**.

The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data. They are –

- **items** – axis 0, each item corresponds to a DataFrame contained inside.
- **major_axis** – axis 1, it is the index (rows) of each of the DataFrames.
- **minor_axis** – axis 2, it is the columns of each of the DataFrames.

pandas.Panel()

A Panel can be created using the following constructor –

```
pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)
```

The parameters of the constructor are as follows –

Parameter	Description
Data	Data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame
Items	axis=0
major_axis	axis=1
minor_axis	axis=2
Dtype	Data type of each column

Copy	Copy data. Default, false
------	----------------------------------

Create Panel

A Panel can be created using multiple ways like –

- From ndarrays
- From dict of DataFrames

From 3D ndarray

creating an empty panel

```
import pandas as pd
```

```
import numpy as np
```

```
data = np.random.rand(2,4,5)
```

```
p = pd.Panel(data)
```

```
print p
```

Its **output** is as follows –

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 4
```

Note – Observe the dimensions of the empty panel and the above panel, all the objects are different.

From dict of DataFrame Objects

#creating an empty panel

```
import pandas as pd
```

```
import numpy as np
```

```
data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),
        'Item2' : pd.DataFrame(np.random.randn(4, 2))}
```

```
p = pd.Panel(data)
print p
```

Its **output** is as follows –

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 4
```

Create an Empty Panel

An empty panel can be created using the Panel constructor as follows –

```
#creating an empty panel
import pandas as pd
p = pd.Panel()
print p
```

Its **output** is as follows –

```
<class 'pandas.core.panel.Panel'>
Dimensions: 0 (items) x 0 (major_axis) x 0 (minor_axis)
Items axis: None
Major_axis axis: None
Minor_axis axis: None
```

Selecting the Data from Panel

Select the data from the panel using –

- Items
- Major_axis
- Minor_axis

Using Items

```
# creating an empty panel
import pandas as pd
import numpy as np
data = {'Item1': pd.DataFrame(np.random.randn(4, 3)),
```

```

    'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p['Item1']

```

Its **output** is as follows –

	0	1	2
0	0.488224	-0.128637	0.930817
1	0.417497	0.896681	0.576657
2	-2.775266	0.571668	0.290082
3	-0.400538	-0.144234	1.110535

We have two items, and we retrieved item1. The result is a DataFrame with 4 rows and 3 columns, which are the **Major_axis** and **Minor_axis** dimensions.

Using major_axis

Data can be accessed using the method **panel.major_axis(index)**.

```

# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

        'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p.major_xs(1)

```

Its **output** is as follows –

	Item1	Item2
0	0.417497	0.748412
1	0.896681	-0.557322
2	0.576657	NaN

Using minor_axis

Data can be accessed using the method **panel.minor_axis(index)**.

```

# creating an empty panel

import pandas as pd

import numpy as np

```

```
data = {'Item1': pd.DataFrame(np.random.randn(4, 3)),  
        'Item2': pd.DataFrame(np.random.randn(4, 2))}  
p = pd.Panel(data)  
print p.minor_xs(1)
```

Its **output** is as follows –

	Item1	Item2
0	-0.128637	-1.047032
1	0.896681	-0.557322
2	0.571668	0.431953
3	-0.144234	1.302466

Note – Observe the changes in the dimensions.

Conclusion: In this experiment, we studied data formats of Pandas- the series, Dataframe, Panel and Creating, Appending, Deleting data. Also how to Import different types of Datasets.

ASSIGNMENT NUMBER 3

Aim

Understanding data formats of Numpy: ndarrays (1D, 2D and 3D arrays), Array creation routines

THEORY:

What are Numpy and numpy arrays

Python has built-in:

- containers: lists (costless insertion and append), dictionaries (fast lookup)
- high-level number objects: integers, floating point

Numpy is:

- extension package to Python for multidimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)

```
• import numpy as np
• >>> a = np.array([0, 1, 2, 3])
• >>> a
• array([0, 1, 2, 3])
```

Creating arrays

1-D:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

2-D, 3-D, ...:

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]]) # 2 x 3 array
>>> b
array([[0, 1, 2],
```

```

        [ 3,  4,  5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)      # returns the size of the first dimension
2

>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],
       [[3],
        [4]]])
>>> c.shape
(2, 2, 1)

```

Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```

import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2
rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]

```

```
print(a[0, 1])    # Prints "77"
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower
rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an
array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import numpy as np

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

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to
this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
```

```

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]]) # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]])) # Prints "[2 2]"

```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```

import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a) # prints "array([[ 1,  2,  3],
#                [ 4,  5,  6],
#                [ 7,  8,  9],
#                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b]) # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a) # prints "array([[11,  2,  3],
#                [ 4,  5, 16],
#                [17,  8,  9],
#                [10, 21, 12]])"

```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```

import numpy as np

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

bool_idx = (a > 2) # Find the elements of a that are bigger than
2;
# this returns a numpy array of Booleans of
the same # shape as a, where each slot of bool_idx
tells # whether that element of a is > 2.

```

```

print(bool_idx)      # Prints "[False False]
                        #           [ True  True]
                        #           [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])   # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])      # Prints "[3 4 5 6]"

```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should [read the documentation](#).

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```

import numpy as np

x = np.array([1, 2])    # Let numpy choose the datatype
print(x.dtype)          # Prints "int64"

x = np.array([1.0, 2.0]) # Let numpy choose the datatype
print(x.dtype)          # Prints "float64"

x = np.array([1, 2], dtype=np.int64) # Force a particular
                                     # datatype
print(x.dtype)          # Prints "int64"

```

You can read all about numpy datatypes [in the documentation](#).

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```

import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

```

```

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2          0.33333333]
#  [ 0.42857143  0.5         ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.         ]]
print(np.sqrt(x))

```

Conclusion: In this experiment, we studied data formats of **Numpy library** - ndarrays (1D, 2D and 3D arrays), and Array creation and Indexing and Slicing

ASSIGNMENT NUMBER 4,5

Aim: Matplotlib plotting for Data visualization: Tweaking Colors, Symbols, Formulations. Plotting Categorical data, 3D Axes, Parametric Curves, Trigonometry functions, Histogram, Bar, Pie chart. Graph plotting using Pandas

THEORY:

Introduction to Data Visualization

Data visualization is a key part of any data science workflow, but it is frequently treated as an afterthought or an inconvenient extra step in reporting the results of an analysis. Data visualization should really be part of your workflow from the very beginning, as there is a lot of value and insight to be gained from just looking at your data. Furthermore, the impact of an effective visualization is difficult to match with words and will go a long way toward ensuring that your work gets the recognition it deserves. In data visualization, there are three main types of variables:

1. **Quantitative:** These are numerical data and represent a measurement. Quantitative variables can be discrete (e.g., units sold in 2016) or continuous (e.g., average units sold per person).
2. **Categorical:** The values of these variables are names or labels. There is no inherent ordering to the labels. Examples of such variables are countries in a sales database and the names of products.
3. **Ordinal:** Variables that can take on values that are ranked on an arbitrary numerical scale. The numerical index associated with each value has no meaning except to rank the values relative to each other. Examples include days of the week, levels of satisfaction (not satisfied, satisfied, very satisfied), and customer value (low, medium, high).

When visualizing data, the most important factor to keep in mind is the purpose of the visualization. This is what will guide you in choosing the best plot type. It could be that you are trying to compare two quantitative variables to each other. Maybe you want to check for differences between groups. Perhaps you are interested in the way a variable is distributed. Each of these goals is best served by different plots and using the wrong one could distort your interpretation of the data or the message that you are trying to convey. Another critical guiding principle is that simpler is almost always better. Often, the most effective visualizations are those that are easily digested — because the clarity of your thought processes is reflected in the clarity of your work. Additionally, overly complicated visuals can be misleading and hard to interpret, which might lead your audience to tune out your results. For these reasons, restrict your plots to two dimensions (unless the need for a third one is absolutely necessary), avoid visual noise (such as unnecessary tick marks, irrelevant annotations and clashing colors), and make sure that everything is legible.

Introduction to Matplotlib :Matplotlib is the leading visualization library in Python. It is powerful, flexible, and has a dizzying array of chart types for you to choose from. Matplotlib is a python library used to create 2D graphs and plots by using python scripts. It has a module named pyplot which makes things easy for plotting by providing feature to control line styles, font properties, formatting axes etc. It supports a very wide variety of graphs and plots namely - histogram, bar charts, power spectra, error charts etc. It is used along with NumPy to provide an environment that is an effective open-source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython. Conventionally, the package is imported into the Python script by adding the following statement –

```
from matplotlib import pyplot as plt
```

Matplotlib Example

The following script produces the **sine wave plot** using matplotlib.

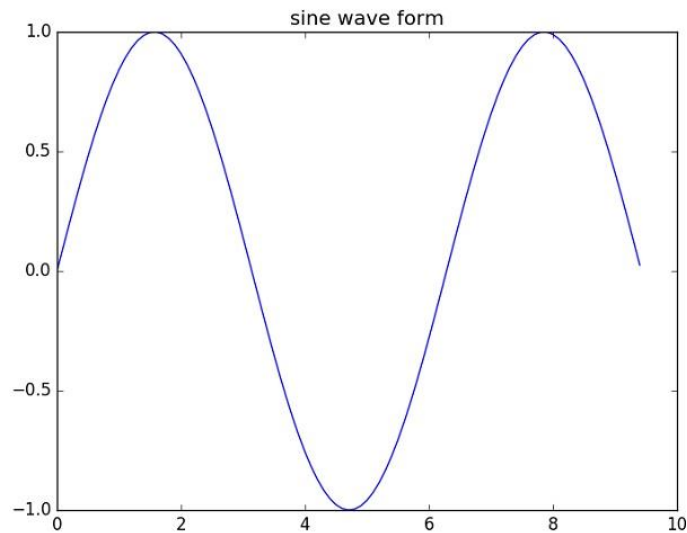
Example

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

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
plt.title("sine wave form")

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```

Its **output** is as follows –



Bar Plots:

For labeled, non-time series data, you may wish to produce a bar plot:

```
In [15]: plt.figure();
```

```
In [16]: df.iloc[5].plot.bar(); plt.axhline(0, color='k')
```

```
Out[16]: <matplotlib.lines.Line2D at 0x135da0588>
```

Calling a DataFrame's `plot.bar()` method produces a multiple bar plot:

```
In [17]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a',  
'b', 'c', 'd'])
```

```
In [18]: df2.plot.bar();
```

To produce a stacked bar plot, pass `stacked=True`:

```
In [19]: df2.plot.bar(stacked=True);
```

To get horizontal bar plots, use the `barh` method:

```
In [20]: df2.plot.barh(stacked=True);
```

Histograms

Histogram can be drawn by using the `DataFrame.plot.hist()` and `Series.plot.hist()` methods.

```
In [21]: df4 = pd.DataFrame({'a': np.random.randn(1000) + 1,
    'b': np.random.randn(1000),
    ....:                    'c': np.random.randn(1000) - 1},
    columns=['a', 'b', 'c'])
    ....:

In [22]: plt.figure();

In [23]: df4.plot.hist(alpha=0.5)

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x136364da0>
```

Histogram can be stacked by `stacked=True`. Bin size can be changed by `bins` keyword.

```
In [24]: plt.figure();

In [25]: df4.plot.hist(stacked=True, bins=20)

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x136353390>
```

The existing interface `DataFrame.hist` to plot histogram still can be used.

```
In [28]: plt.figure();

In [29]: df['A'].diff().hist()

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1369e4a90>
```

`DataFrame.hist()` plots the histograms of the columns on multiple subplots:

```
In [30]: plt.figure()
```

```
Out[30]: <Figure size 640x480 with 0 Axes>
```

```
In [31]: df.diff().hist(color='k', alpha=0.5, bins=50)
```

```
Out[31]:
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at  
0x136b7e5c0>,
```

```
        <matplotlib.axes._subplots.AxesSubplot object at  
0x136bb21d0>],
```

```
        [<matplotlib.axes._subplots.AxesSubplot object at  
0x136be4e80>,
```

```
        <matplotlib.axes._subplots.AxesSubplot object at  
0x136c22a20>]], dtype=object)
```

Pie plot

You can create a pie plot with `DataFrame.plot.pie()` or `Series.plot.pie()`. If your data includes any `NaN`, they will be automatically filled with 0. A `ValueError` will be raised if there are any negative values in your data.

```
In [73]: series = pd.Series(3 * np.random.rand(4), index=['a',  
'b', 'c', 'd'], name='series')
```

```
In [74]: series.plot.pie(figsize=(6, 6))
```

```
Out[74]: <matplotlib.axes._subplots.AxesSubplot at 0x114fc5518>
```

For pie plots it's best to use square figures, one's with an equal aspect ratio. You can create the figure with equal width and height, or force the aspect ratio to be equal after plotting by calling `ax.set_aspect('equal')` on the returned `axes` object. Note that pie plot with **DataFrame** requires that you either specify a target column by the `y` argument or `subplots=True`. When `y` is specified, pie plot of selected column will be drawn. If `subplots=True` is specified, pie plots for each column are drawn as subplots. A legend will be drawn in each pie plots by default; specify `legend=False` to hide it.

```
In [75]: df = pd.DataFrame(3 * np.random.rand(4, 2), index=['a',  
'b', 'c', 'd'], columns=['x', 'y'])
```

```
In [76]: df.plot.pie(subplots=True, figsize=(8, 4))
```

Out[76]:

```
array([<matplotlib.axes._subplots.AxesSubplot object at  
0x130db1f98>,  
  
       <matplotlib.axes._subplots.AxesSubplot object at  
0x12da33710>], dtype=object)
```

```
# Importing libraries  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd
```

```
# Plotting a Line  
plt.plot([1,2,3,4], [1,2,3,4])  
plt.xlabel('Some numbers')  
plt.ylabel('Some more numbers')  
plt.show()
```

```
# Plotting using X axis only  
plt.plot([4,3,8,9])  
plt.xlabel('Some numbers')  
plt.ylabel('Some more numbers')  
plt.show()
```

```
# Plotting with Tweaking Colors and Symbols  
plt.plot([1,2,6,8], [9,16,2,3], 'c')  
plt.xlabel('Some numbers')  
plt.ylabel('Some more numbers')  
plt.show()
```

```
plt.plot([1,2,6,8], [9,16,2,3], 'm+')
```

```

plt.xlabel('Some numbers')
plt.ylabel('Some more numbers')
plt.show()

# red dashes, blue squares and green triangles
t = np.arange(0., 5., 0.2)

plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()

# ----- By using Formula

data = np.arange(0, 10, 0.5)
y = 1 * data + 5
plt.plot(data, y, '*')
plt.show()

# ----- By Dictionary, Color and Size
data = {
    'a': np.arange(50),
    'color': np.random.randint(0, 50, 50),
    'size': np.random.randn(50)
}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['size'] = np.abs(data['size']) * 100

plt.scatter('a', 'b', c = 'color', s = 'size', data=data)
plt.xlabel('Entries for A')
plt.ylabel('Entries of B')
plt.show()

# ----- Plotting with Categorical Data
names = ['GroupA', 'GroupB', 'GroupC']
values = [1, 10, 100]

plt.figure(1, figsize=(10,4))

plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()

# ----- Plotting over 3D Axes
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# ----- Parametric Curve

```

```

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)

z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)

ax.plot(x, y, z, label='Parametric Curve')
ax.legend()
plt.show()

# ----- Simple Trigonometry plots

x = np.arange(1, 5 * np.pi, 0.01)
y = np.sin(x)
plt.title('Sine Wave')
plt.plot(x, y)
plt.show()

# ----- Subplots of Sine and Cosine
x = np.arange(0, 3* np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

plt.subplot(1, 2, 1)
plt.plot(x, y_sin)
plt.title('Sine Wave')

plt.subplot(1, 2, 2)
plt.plot(x, y_cos)
plt.title('Cosine Wave')
plt.suptitle('Waveforms')
plt.show()

# ----- Plotting of Histogram
a = np.array([22, 87, 5, 43, 56, 73, 55, 54, 11, 20, 51, 5, 79, 31, 27])
plt.hist(a, bins=[0, 20, 40, 60, 80, 100])
plt.show()

# ----- Plotting using BAR Graph
x1 = [5, 8, 10]
y1 = [12, 16, 6]

x2 = [6, 9, 11]
y2 = [6, 15, 7]

plt.bar(x1, y1, color = 'b')
plt.bar(x2, y2, color = 'g', align='center')

plt.title('Bar Graph')
plt.ylabel('Y Axis')

```

```

plt.xlabel('X Axis')
plt.show()

# ----- PIE Chart Plotting

labels = ['Politics', 'Science', 'History', 'Heritage']
interest = [15, 30, 45, 10]

fig1, ax1 = plt.subplots()
ax1.pie(interest, labels=labels)
plt.show()

# ----- MATPLOTLIB with PANDAS

ds1 = pd.read_csv('1_WaterWellDev.csv')
ds1.plot()
ds1 = ds1.dropna()
ds1.plot.bar(subplots=True)
ds1.plot.bar(stacked=True)

ds1 = ds1.T
ds1.iloc[1:, :].plot()
ds1.plot.box()

```

Conclusion: Hence in this experiment, we studied the important machine learning library **Matplotlib**. Matplotlib is used to plot different graphs like Pie Plot, Histograms, BarPlot etc. for Data visualization

ASSIGNMENT NO.6

Aim: Introduction to Scipy, Scikit learn, importing algorithm classes and creating objects with parametric values.

Theory

Scipy:

Scipy, I/O package, has a wide range of functions for work with different files format which are Matlab, Arff, Wave, Matrix Market, IDL, NetCDF, TXT, CSV and binary format.

```
import numpy as np
```

```
from scipy import io as sio
```

```
array = np.ones((4, 4))
```

```
sio.savemat('example.mat', {'ar': array})
```

```
data = sio.loadmat('example.mat', struct_as_record=True)
```

```
data['ar']
```

Output:

```
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Scikit-Learn:

APIs of scikit-learn objects

To have a uniform API, we try to have a common basic API for all the objects. In addition, to avoid the proliferation of framework code, we try to adopt simple conventions and limit to a minimum the number of methods an object must implement.

Elements of the scikit-learn API are described more definitively in the [Glossary of Common Terms and API Elements](#).

Different objects

The main objects in scikit-learn are (one class can implement multiple interfaces):

Estimator

The base object, implements a `fit` method to learn from data, either:


```
estimator = estimator.fit(data, targets)
```

or:

```
estimator = estimator.fit(data)
```

Predictor

For supervised learning, or some unsupervised problems, implements:

```
prediction = predictor.predict(data)
```

Classification algorithms usually also offer a way to quantify certainty of a prediction, either using `decision_function` or `predict_proba`:

```
probability = predictor.predict_proba(data)
```

Transformer

For filtering or modifying the data, in a supervised or unsupervised way, implements:

```
new_data = transformer.transform(data)
```

When fitting and transforming can be performed much more efficiently together than separately, implements:

```
new_data = transformer.fit_transform(data)
```

Model

A model that can give a [goodness of fit](#) measure or a likelihood of unseen data, implements (higher is better):

```
score = model.score(data)
```

Estimators

The API has one predominant object: the estimator. An estimator is an object that fits a model based on some training data and is capable of inferring some properties on new data. It can be, for instance, a classifier or a regressor. All estimators implement the fit method:

```
estimator.fit(X, y)
```

All built-in estimators also have a `set_params` method, which sets data-independent parameters (overriding previous parameter values passed to `__init__`).

All estimators in the main scikit-learn codebase should inherit from `sklearn.base.BaseEstimator`.

Instantiation

This concerns the creation of an object. The object's `__init__` method might accept constants as arguments that determine the estimator's behavior (like the `C` constant in SVMs). It should not, however, take the actual training data as an argument, as this is left to the `fit()` method:

```
clf2 = SVC(C=2.3)
```

```
clf3 = SVC([[1, 2], [2, 3]], [-1, 1]) # WRONG!
```

The arguments accepted by `__init__` should all be keyword arguments with a default value. In other words, a user should be able to instantiate an estimator without passing any arguments to it. The arguments should all correspond to hyperparameters describing the model or the optimisation problem the estimator tries to solve. These initial arguments (or parameters) are

always remembered by the estimator. Also note that they should not be documented under the “Attributes” section, but rather under the “Parameters” section for that estimator.

In addition, **every keyword argument accepted by `__init__` should correspond to an attribute on the instance**. Scikit-learn relies on this to find the relevant attributes to set on an estimator when doing model selection.

To summarize, an `__init__` should look like:

```
def __init__(self, param1=1, param2=2):
    self.param1 = param1
    self.param2 = param2
```

There should be no logic, not even input validation, and the parameters should not be changed. The corresponding logic should be put where the parameters are used, typically in `fit`. The following is wrong:

```
def __init__(self, param1=1, param2=2, param3=3):
    # WRONG: parameters should not be modified
    if param1 > 1:
        param2 += 1
    self.param1 = param1
    # WRONG: the object's attributes should have exactly the name of
    # the argument in the constructor
    self.param3 = param2
```

The reason for postponing the validation is that the same validation would have to be performed in `set_params`, which is used in algorithms like `GridSearchCV`.

Fitting

The next thing you will probably want to do is to estimate some parameters in the model. This is implemented in the `fit()` method.

The `fit()` method takes the training data as arguments, which can be one array in the case of unsupervised learning, or two arrays in the case of supervised learning.

Note that the model is fitted using `X` and `y`, but the object holds no reference to `X` and `y`. There are, however, some exceptions to this, as in the case of precomputed kernels where this data must be stored for use by the `predict` method.

Parameters

`X` array-like of shape (n_samples, n_features)

`y` array-like of shape (n_samples,)

`kwargs` optional data-dependent parameters

`X.shape[0]` should be the same as `y.shape[0]`. If this requisite is not met, an exception of type `ValueError` should be raised.

`y` might be ignored in the case of unsupervised learning. However, to make it possible to use the estimator as part of a pipeline that can mix both supervised and unsupervised transformers, even unsupervised estimators need to accept a `y=None` keyword argument in the second position that is just ignored by the estimator. For the same

reason, `fit_predict`, `fit_transform`, `score` and `partial_fit` methods need to accept a `y` argument in the second place if they are implemented.

The method should return the object (`self`). This pattern is useful to be able to implement quick one liners in an IPython session such as:

```
y_predicted = SVC(C=100).fit(X_train, y_train).predict(X_test)
```

Depending on the nature of the algorithm, `fit` can sometimes also accept additional keywords arguments. However, any parameter that can have a value assigned prior to having access to the data should be an `__init__` keyword argument. **fit parameters should be restricted to directly data dependent variables.** For instance a Gram matrix or an affinity matrix which are precomputed from the data matrix `X` are data dependent. A tolerance stopping criterion `tol` is not directly data dependent (although the optimal value according to some scoring function probably is).

When `fit` is called, any previous call to `fit` should be ignored. In general, calling `estimator.fit(X1)` and then `estimator.fit(X2)` should be the same as only calling `estimator.fit(X2)`. However, this may not be true in practice when `fit` depends on some random process, see [random_state](#). Another exception to this rule is when the hyperparameter `warm_start` is set to `True` for estimators that support it. `warm_start=True` means that the previous state of the trainable parameters of the estimator are reused instead of using the default initialization strategy.

Estimated Attributes

Attributes that have been estimated from the data must always have a name ending with trailing underscore, for example the coefficients of some regression estimator would be stored in a `coef_` attribute after `fit` has been called.

The estimated attributes are expected to be overridden when you call `fit` a second time.

Optional Arguments

In iterative algorithms, the number of iterations should be specified by an integer called `n_iter`.

Pairwise Attributes

An estimator that accepts `X` of shape `(n_samples, n_samples)` and defines a `_pairwise` property equal to `True` allows for cross-validation of the dataset, e.g. when `X` is a precomputed kernel matrix. Specifically, the `_pairwise` property is used by `utils.metaestimators._safe_split` to slice rows and columns.

Deprecated since version 0.24: The `_pairwise` attribute is deprecated in 0.24. From 1.1 (renaming of 0.26) onward, the `pairwise` estimator tag should be used instead.

Universal attributes

Estimators that expect tabular input should set a `n_features_in_` attribute at `fit` time to indicate the number of features that the estimator expects for subsequent calls to `predict` or `transform`. See [SLEP010](#) for details.

Rolling your own estimator

If you want to implement a new estimator that is scikit-learn-compatible, whether it is just for you or for contributing it to scikit-learn, there are several internals of scikit-learn that you

should be aware of in addition to the scikit-learn API outlined above. You can check whether your estimator adheres to the scikit-learn interface and standards by running `check_estimator` on an instance. The `parametrize_with_checks` pytest decorator can also be used (see its docstring for details and possible interactions with `pytest`):

```
>>>
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from sklearn.svm import LinearSVC
>>> check_estimator(LinearSVC()) # passes
```

The main motivation to make a class compatible to the scikit-learn estimator interface might be that you want to use it together with model evaluation and selection tools such as `model_selection.GridSearchCV` and `pipeline.Pipeline`.

Before detailing the required interface below, we describe two ways to achieve the correct interface more easily.

Project template:

We provide a [project template](#) which helps in the creation of Python packages containing scikit-learn compatible estimators. It provides:

- an initial git repository with Python package directory structure
- a template of a scikit-learn estimator
- an initial test suite including use of `check_estimator`
- directory structures and scripts to compile documentation and example galleries
- scripts to manage continuous integration (testing on Linux and Windows)
- instructions from getting started to publishing on [PyPi](#)

BaseEstimator and mixins:

We tend to use “duck typing”, so building an estimator which follows the API suffices for compatibility, without needing to inherit from or even import any scikit-learn classes.

However, if a dependency on scikit-learn is acceptable in your code, you can prevent a lot of boilerplate code by deriving a class from `BaseEstimator` and optionally the mixin classes in `sklearn.base`. For example, below is a custom classifier, with more examples included in the scikit-learn-contrib [project template](#).

```
>>>
>>> import numpy as np
>>> from sklearn.base import BaseEstimator, ClassifierMixin
>>> from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
>>> from sklearn.utils.multiclass import unique_labels
>>> from sklearn.metrics import euclidean_distances
>>> class TemplateClassifier(BaseEstimator, ClassifierMixin):
...
...     def __init__(self, demo_param='demo'):
...         self.demo_param = demo_param
...
... 
```

```

... def fit(self, X, y):
...
...     # Check that X and y have correct shape
...     X, y = check_X_y(X, y)
...     # Store the classes seen during fit
...     self.classes_ = unique_labels(y)
...
...     self.X_ = X
...     self.y_ = y
...     # Return the classifier
...     return self
...
... def predict(self, X):
...
...     # Check is fit had been called
...     check_is_fitted(self)
...
...     # Input validation
...     X = check_array(X)
...
...     closest = np.argmin(euclidean_distances(X, self.X_), axis=1)
...     return self.y_[closest]

```

get_params and set_params

All scikit-learn estimators have `get_params` and `set_params` functions. The `get_params` function takes no arguments and returns a dict of the `__init__` parameters of the estimator, together with their values.

It must take one keyword argument, `deep`, which receives a boolean value that determines whether the method should return the parameters of sub-estimators (for most estimators, this can be ignored). The default value for `deep` should be `True`. For instance considering the following estimator:

```

>>>
>>> from sklearn.base import BaseEstimator
>>> from sklearn.linear_model import LogisticRegression
>>> class MyEstimator(BaseEstimator):
...     def __init__(self, subestimator=None, my_extra_param="random"):
...         self.subestimator = subestimator
...         self.my_extra_param = my_extra_param

```

The parameter `deep` will control whether or not the parameters of the `subestimator` should be reported. Thus when `deep=True`, the output will be:

```

>>>
>>> my_estimator = MyEstimator(subestimator=LogisticRegression())
>>> for param, value in my_estimator.get_params(deep=True).items():
...     print(f"/{param}/ -> {value}")
my_extra_param -> random

```

```

subestimator__C -> 1.0
subestimator__class_weight -> None
subestimator__dual -> False
subestimator__fit_intercept -> True
subestimator__intercept_scaling -> 1
subestimator__l1_ratio -> None
subestimator__max_iter -> 100
subestimator__multi_class -> auto
subestimator__n_jobs -> None
subestimator__penalty -> l2
subestimator__random_state -> None
subestimator__solver -> lbfgs
subestimator__tol -> 0.0001
subestimator__verbose -> 0
subestimator__warm_start -> False
subestimator -> LogisticRegression()

```

Often, the subestimator has a name (as e.g. named steps in a [Pipeline](#) object), in which case the key should become `<name>__C`, `<name>__class_weight`, etc.

While when `deep=False`, the output will be:

```

>>>
>>> for param, value in my_estimator.get_params(deep=False).items():
...     print(f"/{param} -> {value}")
my_extra_param -> random
subestimator -> LogisticRegression()

```

The `set_params` on the other hand takes as input a dict of the form `'parameter': value` and sets the parameter of the estimator using this dict. Return value must be estimator itself.

While the `get_params` mechanism is not essential (see [Cloning](#) below), the `set_params` function is necessary as it is used to set parameters during grid searches.

The easiest way to implement these functions, and to get a sensible `__repr__` method, is to inherit from `sklearn.base.BaseEstimator`. If you do not want to make your code dependent on scikit-learn, the easiest way to implement the interface is:

```

def get_params(self, deep=True):
    # suppose this estimator has parameters "alpha" and "recursive"
    return {"alpha": self.alpha, "recursive": self.recursive}

def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self

```

Parameters and init

As [model_selection.GridSearchCV](#) uses `set_params` to apply parameter setting to estimators, it is essential that calling `set_params` has the same effect as setting parameters using the `__init__` method. The easiest and recommended way to accomplish this is to **not do any**

parameter validation in `__init__`. All logic behind estimator parameters, like translating string arguments into functions, should be done in `fit`.

Also it is expected that parameters with trailing `_` are **not to be set inside the `__init__` method**. All and only the public attributes set by `fit` have a trailing `_`. As a result the existence of parameters with trailing `_` is used to check if the estimator has been fitted.

Cloning

For use with the `model_selection` module, an estimator must support the `base.clone` function to replicate an estimator. This can be done by providing a `get_params` method. If `get_params` is present, then `clone(estimator)` will be an instance of `type(estimator)` on which `set_params` has been called with clones of the result of `estimator.get_params()`.

Objects that do not provide this method will be deep-copied (using the Python standard function `copy.deepcopy`) if `safe=False` is passed to `clone`.

Pipeline compatibility

For an estimator to be usable together with `pipeline.Pipeline` in any but the last step, it needs to provide a `fit` or `fit_transform` function. To be able to evaluate the pipeline on any data but the training set, it also needs to provide a `transform` function. There are no special requirements for the last step in a pipeline, except that it has a `fit` function. All `fit` and `fit_transform` functions must take arguments `X, y`, even if `y` is not used. Similarly, for `score` to be usable, the last step of the pipeline needs to have a `score` function that accepts an optional `y`.

Estimator types

Some common functionality depends on the kind of estimator passed. For example, cross-validation in `model_selection.GridSearchCV` and `model_selection.cross_val_score` defaults to being stratified when used on a classifier, but not otherwise. Similarly, scorers for average precision that take a continuous prediction need to call `decision_function` for classifiers, but `predict` for regressors. This distinction between classifiers and regressors is implemented using the `_estimator_type` attribute, which takes a string value. It should be "classifier" for classifiers and "regressor" for regressors and "clusterer" for clustering methods, to work as expected. Inheriting from `ClassifierMixin`, `RegressorMixin` or `ClusterMixin` will set the attribute automatically. When a meta-estimator needs to distinguish among estimator types, instead of checking `_estimator_type` directly, helpers like `base.is_classifier` should be used.

Specific models

Classifiers should accept `y` (target) arguments to `fit` that are sequences (lists, arrays) of either strings or integers. They should not assume that the class labels are a contiguous range of integers; instead, they should store a list of classes in a `classes_` attribute or property. The order of class labels in this attribute should match the order in which `predict_proba`, `predict_log_proba` and `decision_function` return their values. The easiest way to achieve this is to put:

```
self.classes_ = np.unique(y, return_inverse=True)
```

in `fit`. This returns a new `y` that contains class indexes, rather than labels, in the range `[0, n_classes)`.

A classifier's `predict` method should return arrays containing class labels from `classes_`. In a classifier that implements `decision_function`, this can be achieved with:

```
def predict(self, X):  
    D = self.decision_function(X)  
    return self.classes_[np.argmax(D, axis=1)]
```

In linear models, coefficients are stored in an array called `coef_`, and the independent term is stored in `intercept_`. `sklearn.linear_model._base` contains a few base classes and mixins that implement common linear model patterns.

The `sklearn.utils.multiclass` module contains useful functions for working with multiclass and multilabel problems.

Estimator Tags

Warning

The estimator tags are experimental and the API is subject to change.

Scikit-learn introduced estimator tags in version 0.21. These are annotations of estimators that allow programmatic inspection of their capabilities, such as sparse matrix support, supported output types and supported methods. The estimator tags are a dictionary returned by the method `_get_tags()`. These tags are used in the common checks run by the `check_estimator` function and the `parametrize_with_checks` decorator. Tags determine which checks to run and what input data is appropriate. Tags can depend on estimator parameters or even system architecture and can in general only be determined at runtime.

The current set of estimator tags are:

allow_nan (default=False)

whether the estimator supports data with missing values encoded as `np.NaN`

binary_only (default=False)

whether estimator supports binary classification but lacks multi-class classification support.

multilabel (default=False)

whether the estimator supports multilabel output

multioutput (default=False)

whether a regressor supports multi-target outputs or a classifier supports multi-class multi-output.

multioutput_only (default=False)

whether estimator supports only multi-output classification or regression.

no_validation (default=False)

whether the estimator skips input-validation. This is only meant for stateless and dummy transformers!

non_deterministic (default=False)

whether the estimator is not deterministic given a fixed `random_state`

pairwise (default=False)

This boolean attribute indicates whether the data (`X`) `fit` and similar methods consists of pairwise measures over samples rather than a feature representation for each sample. It is usually `True` where an estimator has a `metric` or `affinity` or `kernel` parameter with value 'precomputed'. Its primary purpose is that when a `meta-estimator` extracts a sub-sample of data intended for a pairwise estimator, the data needs to be indexed on both axes, while other data is indexed only on the first axis.

preserves_dtype (default='[np.float64]')

applies only on transformers. It corresponds to the data types which will be preserved such that `X_trans.dtype` is the same as `X.dtype` after calling `transformer.transform(X)`. If this list is empty, then the transformer is not expected to preserve the data type. The first value in the list is considered as the default data type, corresponding to the data type of the output when the input data type is not going to be preserved.

poor_score (default=False)

whether the estimator fails to provide a "reasonable" test-set score, which currently for regression is an R^2 of 0.5 on a subset of the boston housing dataset, and for classification an accuracy of 0.83 on `make_blobs(n_samples=300, random_state=0)`. These datasets and values are based on current estimators in sklearn and might be replaced by something more systematic.

requires_fit (default=True)

whether the estimator requires to be fitted before calling one of `transform`, `predict`, `predict_proba`, or `decision_function`.

requires_positive_X (default=False)

whether the estimator requires positive `X`.

requires_y (default=False)

whether the estimator requires `y` to be passed to `fit`, `fit_predict` or `fit_transform` methods. The tag is `True` for estimators inheriting from `~sklearn.base.RegressorMixin` and `~sklearn.base.ClassifierMixin`.

requires_positive_y (default=False)

whether the estimator requires a positive `y` (only applicable for regression).

_skip_test (default=False)

whether to skip common tests entirely. Don't use this unless you have a *very good* reason.

`_xfail_checks (default=False)`

dictionary `{check_name: reason}` of common checks that will be marked as XFAIL for pytest, when using `parametrize_with_checks`. These checks will be simply ignored and not run by `check_estimator`, but a `SkipTestWarning` will be raised. Don't use this unless there is a *very good* reason for your estimator not to pass the check. Also note that the usage of this tag is highly subject to change because we are trying to make it more flexible: be prepared for breaking changes in the future.

`stateless (default=False)`

whether the estimator needs access to data for fitting. Even though an estimator is stateless, it might still need a call to `fit` for initialization.

`X_types (default=['2darray'])`

Supported input types for X as list of strings. Tests are currently only run if '2darray' is contained in the list, signifying that the estimator takes continuous 2d numpy arrays as input. The default value is `['2darray']`. Other possible types are 'string', 'sparse', 'categorical', dict, '1dlabels' and '2dlabels'. The goal is that in the future the supported input type will determine the data used during testing, in particular for 'string', 'sparse' and 'categorical' data. For now, the test for sparse data do not make use of the 'sparse' tag.

It is unlikely that the default values for each tag will suit the needs of your specific estimator. Additional tags can be created or default tags can be overridden by defining a `_more_tags()` method which returns a dict with the desired overridden tags or new tags. For example:

```
class MyMultiOutputEstimator(BaseEstimator):  
  
    def _more_tags(self):  
        return {'multioutput_only': True,  
                'non_deterministic': True}
```

Any tag that is not in `_more_tags()` will just fall-back to the default values documented above.

Even if it is not recommended, it is possible to override the method `_get_tags()`. Note however that **all tags must be present in the dict**. If any of the keys documented above is not present in the output of `_get_tags()`, an error will occur.

In addition to the tags, estimators also need to declare any non-optional parameters to `__init__` in the `_required_parameters` class attribute, which is a list or tuple. If `_required_parameters` is only `["estimator"]` or `["base_estimator"]`, then the estimator will be instantiated with an instance of `LogisticRegression` (or `RidgeRegression` if the estimator is a regressor) in the tests. The choice of these two models is somewhat idiosyncratic but both should provide robust closed-form solutions.

Coding guidelines

The following are some guidelines on how new code should be written for inclusion in scikit-learn, and which may be appropriate to adopt in external projects. Of course, there are special cases and there will be exceptions to these rules. However, following these rules when submitting new code makes the review easier so new code can be integrated in less time.

Uniformly formatted code makes it easier to share code ownership. The scikit-learn project tries to closely follow the official Python guidelines detailed in [PEP8](#) that detail how code should be formatted and indented. Please read it and follow it.

In addition, we add the following guidelines:

- Use underscores to separate words in non class names: `n_samples` rather than `nsamples`.
- Avoid multiple statements on one line. Prefer a line return after a control flow statement (if/for).
- Use relative imports for references inside scikit-learn.
- Unit tests are an exception to the previous rule; they should use absolute imports, exactly as client code would. A corollary is that, if `sklearn.foo` exports a class or function that is implemented in `sklearn.foo.bar.baz`, the test should import it from `sklearn.foo`.
- **Please don't use `import *` in any case.** It is considered harmful by the [official Python recommendations](#). It makes the code harder to read as the origin of symbols is no longer explicitly referenced, but most important, it prevents using a static analysis tool like [pyflakes](#) to automatically find bugs in scikit-learn.
- Use the [numpy docstring standard](#) in all your docstrings.

A good example of code that we like can be found [here](#).

Input validation

The module `sklearn.utils` contains various functions for doing input validation and conversion. Sometimes, `np.asarray` suffices for validation; do *not* use `np.asanyarray` or `np.atleast_2d`, since those let NumPy's `np.matrix` through, which has a different API (e.g., `*` means dot product on `np.matrix`, but Hadamard product on `np.ndarray`).

In other cases, be sure to call `check_array` on any array-like argument passed to a scikit-learn API function. The exact parameters to use depends mainly on whether and which `scipy.sparse` matrices must be accepted.

For more information, refer to the [Utilities for Developers](#) page.

Random Numbers

If your code depends on a random number generator, do not use `numpy.random.random()` or similar routines. To ensure repeatability in error checking, the routine should accept a keyword `random_state` and use this to construct a `numpy.random.RandomState` object. See `sklearn.utils.check_random_state` in [Utilities for Developers](#).

Here's a simple example of code using some of the above guidelines:

```
from sklearn.utils import check_array, check_random_state

def choose_random_sample(X, random_state=0):
```

"""Choose a random point from X.

Parameters

X : array-like of shape (n_samples, n_features)

An array representing the data.

random_state : int or RandomState instance, default=0

The seed of the pseudo random number generator that selects a random sample. Pass an int for reproducible output across multiple function calls.

See :term:`Glossary` <random_state>`.

Returns

x : ndarray of shape (n_features,)

A random point selected from X.

"""

```
X = check_array(X)
```

```
random_state = check_random_state(random_state)
```

```
i = random_state.randint(X.shape[0])
```

```
return X[i]
```

If you use randomness in an estimator instead of a freestanding function, some additional guidelines apply.

First off, the estimator should take a `random_state` argument to its `__init__` with a default value of `None`. It should store that argument's value, **unmodified**, in an attribute `random_state`. `fit` can call `check_random_state` on that attribute to get an actual random number generator. If, for some reason, randomness is needed after `fit`, the RNG should be stored in an attribute `random_state_`. The following example should make this clear:

```
class GaussianNoise(BaseEstimator, TransformerMixin):
```

```
    """This estimator ignores its input and returns random Gaussian noise.
```

```
    It also does not adhere to all scikit-learn conventions,  
    but showcases how to handle randomness.
```

```
    """
```

```
def __init__(self, n_components=100, random_state=None):
```

```
    self.random_state = random_state
```

```
    self.n_components = n_components
```

```
    # the arguments are ignored anyway, so we make them optional
```

```
def fit(self, X=None, y=None):
```

```
    self.random_state_ = check_random_state(self.random_state)
```

```
def transform(self, X):
```

```
    n_samples = X.shape[0]
```

```
    return self.random_state_.randn(n_samples, self.n_components)
```

The reason for this setup is reproducibility: when an estimator is fit twice to the same data, it should produce an identical model both times, hence the validation in fit, not `__init__`.

Conclusion: We studied Scipy, and Scikit and Sklearn library for importing algorithm and implementing classes and creating objects with parametric values.

ASSIGNMENT NUMBER 7

Aim

Dataset selection: Dataset for Classification / Regression / Associative Rule Mining. and Analysis: Independent Variables, Dependent Variables, Handling Missing Values, Categorical data, and Feature Scaling

Importing Data with `read_csv()`

The first step to any data science project is to import your data. Often, you'll work with data in Comma Separated Value (CSV) files and run into problems at the very start of your workflow. In this tutorial, you'll see how you can use the `read_csv()` function from `pandas` to deal with common problems when importing data and see why loading CSV files specifically with `pandas` has become standard practice for working data scientists today.

The filesystem

Before you can use `pandas` to import your data, you need to know where your data is in your filesystem and what your current working directory is. You'll see why this is important very soon, but let's review some basic concepts:

Everything on the computer is stored in the filesystem. "Directories" is just another word for "folders", and the "working directory" is simply the folder you're currently in. Some basic Shell commands to navigate your way in the filesystem:

- The `ls` command lists all content in the current working directory.
- The `cd` command followed by:
 - the name of a sub-directory allows you to change your working directory to the sub-directory you specify.
 - `..` allows you to navigate back to the parent directory of your current working directory.

- The `pwd` command prints the path of your current working directory.

IPython allows you to execute Shell commands directly from the IPython console via its magic commands. Here are the ones that correspond to the commands you saw above:

- `!ls` in IPython is the same as `ls` in the command line.
- `%cd` in IPython is the same as `cd` in the command line.
- `!pwd` in IPython is the same as `pwd` in the command line. The working directory is also printed after changing into it in IPython, which isn't the case in the command line.

In your filesystem, there's a file called `cereal.csv` that contains nutrition data on 80 cereals. Enter the magic commands one-by-one in the IPython Shell, and see if you can locate the dataset!

Loading your data

Now that you know what your current working directory is and where the dataset is in your filesystem, you can specify the file path to it. You're now ready to import the CSV file into Python using `read_csv()` from `pandas`:

```
import pandas as pd

cereal_df = pd.read_csv("/tmp/tmp07wuam09/data/cereal.csv")

cereal_df2 = pd.read_csv("data/cereal.csv")
```

```
# Are they the same?

print(pd.DataFrame.equals(cereal_df, cereal_df2))
```

```
True
```

Feature Selection for Machine Learning

This section lists 4 feature selection recipes for machine learning in Python

This post contains recipes for feature selection methods.

1. Univariate Selection

Statistical tests can be used to select those features that have the strongest relationship with the output variable.

The scikit-learn library provides the SelectKBest class that can be used with a suite of different statistical tests to select a specific number of features.

2. Recursive Feature Elimination

The Recursive Feature Elimination (or RFE) works by recursively removing attributes and building a model on those attributes that remain.

It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

3. Principal Component Analysis

Principal Component Analysis (or PCA) uses linear algebra to transform the dataset into a compressed form.

Generally this is called a data reduction technique. A property of PCA is that you can choose the number of dimensions or principal component in the transformed result.

4. Feature Importance

Bagged decision trees like Random Forest and Extra Trees can be used to estimate the importance of features.

Feature Scaling

Feature scaling is the method to limit the range of variables so that they can be compared on common grounds. It is performed on continuous variables. Lets plot the distribution of all the continuous variables in the data set.

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

```
>> X_train[X_train.dtypes[(X_train.dtypes=="float64")|(X_train.dtypes=="int64")]  
        .index.values].hist(figsize=[11,11])
```


Data Preprocessing:

Data Preprocessing for Machine learning in Python

- Pre-processing refers to the transformations applied to our data before feeding it to the algorithm.
- Data Preprocessing is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not feasible for the analysis.

Need of Data Preprocessing

- For achieving better results from the applied model in Machine Learning projects the format of the data has to be in a proper manner. Some specified Machine Learning model needs information in a specified format, for example, Random Forest algorithm does not support null values, therefore to execute random forest algorithm null values have to be managed from the original raw data set.
- Another aspect is that data set should be formatted in such a way that more than one Machine Learning and Deep Learning algorithms are executed in one data set, and best out of them is chosen.

Get the Dataset:

- In any Machine Learning Model, we are going to use some independent variables to predict the dependent variables
- Example as per the Data.csv

Importing the Libraries and Dataset

Library is a tool that you can use to make a specific job. You just have to give some inputs then the library will do job for you & it returns the output that you exactly looking for

```
import numpy as np
```

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

```
import pandas as pd
```

Missing Data

- Method1: Remove the lines or observations of missing data

- Dangerous?
- Method2: Averaging/Mean: Take the mean of columns and place the missing value by mean of all columns
- Imputer class allows us to take out the missing data and transform replaces the missing values by mean of columns

Categorical Data:

- Since machine learning models are based on mathematical equations, it will cause some problem if we keep text in the categorical variables in the equations. As we want only numbers in the equation, encode the text into numbers
- As $1 > 0$ & $2 > 1$, the equation in the model thinks that e.g. Spain has higher value than Germany & France & Germany has higher value than France.
- As there is no relational order, we have to prevent the machine learning equation from thinking this
- To prevent this, dummy variables are used. Instead of one column, we have three columns and no. of columns=no of categories

Categorical Data

DUMMY ENCODING

Country	France	Germany	Spain
France	1	0	0
Spain	0	0	1
Germany	0	1	0
Spain	0	0	1
Germany	0	1	0
France	1	0	0
Spain	0	0	1
France	1	0	0
Germany	0	1	0
France	1	0	0

Feature Scaling

- Varying nature of Data
 - AGE: 27 to 47
 - SALARY: 40K to 80K
- Lose of Scaling
- ML are based on Euclidean distances

Euclidean

ju:'klɪdɪən | adjective

is Two data points is the Sq root of Sum of the squared co-ordinates

Feature Scaling

EUREKA!

- Scale values from -1 to +1 to get both the AXES in same range
- Eliminate Domination

Program:

```
# Data Pre-Processing
# IMPORT LIBRARIES
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# IMPORTING DATASET
dataset = pd.read_csv("LinearRegression/Data.csv")
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 3].values

print("FIRST X ____\n", X)
print("FIRST y ____\n", y)

# Taking care of Missing Data
from sklearn.preprocessing import Imputer

imputer = Imputer(missing_values= 'NaN', strategy='mean', axis = 0)
imputer1 = imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])

print("IMPUTED X ____\n",X)

# Encoding Categorical data
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
# Learn the Output
# PROBLEM for Country & Purchase
# France > Spain > Germany! Well... WTH is that!

print("TRANSFORMED X____\n",X)

# Solution
# COUNTRY
onehotencoder = OneHotEncoder(categorical_features= [0])
X = onehotencoder.fit_transform(X).toarray()
# PURCHASE
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
```

```

print("CATEGORICAL y",y)

# Splitting Data into Training & Testing
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 0)

print("PRE-SCALED VALUES")
print(X_train)
print(X_test)
print("_"*40)
print(y_train)
print(y_test)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)

print("POST-SCALED VALUES")
print(X_train)
print(X_test)
# DO YOU NEED TO SCALE DUMMY VARIABLES?
# IT DEPENDS on your models
# DO YOU NEED TO SCALE Y Variables?
# NO, Cz IT is a dependent variable

```

Conclusion: In this experiment we studied supervised learning important concept Classification, Regression, Associative Rule Mining and also Independent Variables, Dependent Variables, Handling Missing Values, Categorical data, and Feature Scaling

ASSIGNMENT NUMBER 8

Aim: Regression: Performing Simple Linear Regression over a salary dataset and predict salaries according to their experience years

Theory:

Linear regression is a statistical approach for modelling relationship between a dependent variable with a given set of independent variables.

Simple Linear Regression

Simple linear regression is an approach for predicting a **response** using a **single feature**. It is assumed that the two variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature or independent variable(x).

Let us consider a dataset where we have a value of response y for every feature x:

For generality, we define:

x as **feature vector**, i.e $x = [x_1, x_2, \dots, x_n]$,
y as **response vector**, i.e $y = [y_1, y_2, \dots, y_n]$
for n observations (in above example, n=10).
A scatter plot of above dataset looks like:-

Now, the task is to find a **line which fits best** in above scatter plot so that we can predict the response for any new feature values. (i.e a value of x not present in dataset)

This line is called **regression line**.

The equation of regression line is represented as:

Here,

- $h(x_i)$ represents the **predicted response value** for ith observation.
- b_0 and b_1 are regression coefficients and represent **y-intercept** and **slope** of regression line respectively.

To create our model, we must “learn” or estimate the values of regression coefficients b_0 and b_1 . And once we’ve estimated these coefficients, we can use the model to predict responses

Get the Dataset:

- Find out the correlation between salary & years of experience

- For this we will create a model i.e. Simple Linear Regression Model which will tell us what is the best fitting line for this relationship

$$Y = b_0 + b_1 \cdot X_1$$

What is Regression ? How does it work ?

Regression is a parametric technique used to predict continuous (dependent) variable given a set of independent variables. It is parametric in nature because it makes certain assumptions (discussed next) based on the data set. If the data set follows those assumptions, regression gives incredible results. Otherwise, it struggles to provide convincing accuracy. Don't worry. There are several tricks (we'll learn shortly) we can use to obtain convincing results.

Mathematically, regression uses a linear function to approximate (predict) the dependent variable given as:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where, Y - Dependent variable

X - Independent variable

β_0 - Intercept

β_1 - Slope

ϵ - Error

β_0 and β_1 are known as coefficients. This is the equation of simple linear regression. It's called 'linear' because there is just one independent variable (X) involved. In multiple regression, we have many independent variables (Xs). If you recall, the equation above is nothing but a line equation ($y = mx + c$) we studied in schools. Let's understand what these parameters say:

Y - This is the variable we predict

X - This is the variable we use to make a prediction

β_0 - This is the intercept term. It is the prediction value you get when $X = 0$

β_1 - This is the slope term. It explains the change in Y when X changes by 1 unit. ϵ - This represents the residual value, i.e. the difference between actual and predicted values.

Error is an inevitable part of the prediction-making process. No matter how powerful the algorithm we choose, there will always remain an (ϵ) irreducible error which reminds us that the "future is uncertain."

Yet, we humans have a unique ability to persevere, i.e. we know we can't completely eliminate the (ϵ) error term, but we can still try to reduce it to the lowest. Right? To do this, regression uses a technique known as **Ordinary Least Square(OLS)**.

So the next time when you say, I am using *linear /multiple regression*, you are actually referring to the *OLS technique*. Conceptually, OLS technique tries to reduce the sum of squared errors $\sum [\text{Actual}(y) - \text{Predicted}(y')]^2$ by finding the best possible value of regression coefficients (β_0 , β_1 , etc).

Is OLS the only technique regression can use? No! There are other techniques such as Generalized Least Square, Percentage Least Square, Total Least Squares, Least absolute deviation, and many more. Then, why OLS? Let's see.

1. It uses squared error which has nice mathematical properties, thereby making it easier to differentiate and compute gradient descent.
2. OLS is easy to analyze and computationally faster, i.e. it can be quickly applied to data sets having 1000s of features.
3. Interpretation of OLS is much easier than other regression techniques.

Let's understand OLS in detail using an example:

We are given a data set with 100 observations and 2 variables, namely Height and Weight. We need to predict weight(y) given height(x1). The OLS equation can be written as:

$$Y = \beta_0 + \beta_1(\text{Height}) + \epsilon$$

When using R, Python or any computing language, you don't need to know how these coefficients and errors are calculated. As a matter of fact, most people don't care. But you must know, and that's how you'll get close to becoming a master.

The formula to calculate these coefficients is easy. Let's say you are given the data, and you don't have access to any statistical tool for computation. Can you still make any prediction? Yes!

The most intuitive and closest approximation of Y is **mean of Y**, i.e. even in the worst case scenario our predictive model should at least give higher accuracy than mean prediction. The formula to calculate coefficients goes like this:

$$\beta_1 = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \text{ where } i = 1 \text{ to } n \text{ (no. of obs.)}$$

$$\beta_0 = \bar{y} - \beta_1(\bar{x})$$

Now you know ymean plays a crucial role in determining regression coefficients and furthermore accuracy. In OLS, the error estimates can be divided into three parts:

Residual Sum of Squares (RSS) - $\sum [\text{Actual}(y) - \text{Predicted}(y)]^2$

Explained Sum of Squares (ESS) - $\sum [\text{Predicted}(y) - \text{Mean}(\bar{y})]^2$

Total Sum of Squares (TSS) - $\sum [\text{Actual}(y) - \text{Mean}(\bar{y})]^2$

The most important use of these error terms is used in the calculation of the Coefficient of Determination (R^2).

$$R^2 = 1 - (\text{SSE}/\text{TSS})$$

R^2 metric tells us the amount of variance explained by the independent variables in the model. In the upcoming section, we'll learn and see the importance of this coefficient and more metrics to compute the model's accuracy.

What are the assumptions made in regression ?

As we discussed above, regression is a parametric technique, so it makes assumptions. Let's look at the assumptions it makes:

1. There exists a **linear** and **additive** relationship between dependent (DV) and independent variables (IV). By linear, it means that the change in DV by 1 unit change in IV is constant. By additive, it refers to the effect of X on Y is independent of other variables.
2. There must be no correlation among independent variables. Presence of correlation in independent variables lead to **Multicollinearity**. If variables are correlated, it becomes extremely difficult for the model to determine the true effect of IVs on DV.
3. The error terms must possess constant variance. Absence of constant variance leads to **heteroskedestacity**.
4. The error terms must be uncorrelated i.e. error at t must not indicate the error at $t+1$. Presence of correlation in error terms is known as **Autocorrelation**. It drastically affects the regression coefficients and standard error values since they are based on the assumption of uncorrelated error terms.
5. The dependent variable and the error terms must possess a **normal distribution**.

Presence of these assumptions make regression quite restrictive. By **restrictive** I meant, the performance of a regression model is conditioned on fulfillment of these assumptions.

Steps:

Prepare your data preprocessing template

Co-relate salaries with experience

Carry out prediction

Verify the values of prediction

Prediction on test set

Program:

```
# Simple Linear Regression

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
```



```

X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values

# Splitting the dataset into the Training set and Test set
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3,
random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train)"""

# Fitting Simple Linear Regression to the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Predicting the Test set results
y_pred = regressor.predict(X_test)

# Visualising the Training set results
plt.scatter(X_train, y_train, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Training set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()

# Visualising the Test set results
plt.scatter(X_test, y_test, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Test set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()

```

Conclusion: In this experiment we studied Regression and how to Perform Simple Linear Regression over a salary dataset and predict salaries according to their experience years

ASSIGNMENT NUMBER 9

Aim: Regression & Data Valuation: Performing Multi-linear Regression (using appropriate Model) to evaluate with data which is useful for model training

Theory:

Multiple Linear Regression

Multiple Linear Regression is a type of Linear Regression when the input has multiple features (variables).

An extension of simple linear regression

In simple linear regression there is a one-to-one relationship between the input variable and the output variable. But in multiple linear regression, as the name implies there is a many-to-one relationship, instead of just using one input variable, you use several.

New considerations

Adding more input variables does not mean the regression will be better, or offer better predictions. Multiple and simple linear regression have different use cases, one is not superior. In some cases adding more input variables can make things worse, this is referred to as over-fitting.

Multicollinearity

When you add more input variables it creates relationships among them. So not only are the input variables potentially related to the output variable, they are also potentially related to each other, this is referred to as multicollinearity. The optimal scenario is for all of the input variables to be correlated with the output variable, but not with each other.

The model

The model for multiple linear regression is as you would expect, very similar to the one for simple linear regression. It goes as follows:
 $f(X) = a + (B_1 * X_1) + (B_2 * X_2) \dots + (B_p * X_p)$.
Where **X** is the input variables and **X_p** is a specific input variable, **B_p** is the coefficient (slope) of the input variable **X_p** and **a** is the intercept. Let's test this out with an example!

Preparation before doing multiple regression

1. Collect a list of potential input variables and a potential output variable.
2. Collect data on the variables.
3. check the correlation between each input variable and the output variable.
4. Check the correlation among the input variables.
5. Use the non-redundant input variables in the analysis to find the best fitting model.

Program:

```
# Multiple Linear Regression

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('50_Startups.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 4].values

# Encoding categorical data
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder = LabelEncoder()
X[:, 3] = labelencoder.fit_transform(X[:, 3])
onehotencoder = OneHotEncoder(categorical_features = [3])
X = onehotencoder.fit_transform(X).toarray()

# Avoiding the Dummy Variable Trap
```

```

X = X[:, 1:]

# Splitting the dataset into the Training set and Test set
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 0)

# Feature Scaling
"""from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train)"""

# Fitting Multiple Linear Regression to the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Predicting the Test set results
y_pred = regressor.predict(X_test)

```

Conclusion: In this experiment we studied multi-Linear regression & Data Valuation. we performed multi linear regression (using appropriate Model) to evaluate with data which is useful for model training

ASSIGNMENT NUMBER 10

Aim: Regression: Using Polynomial regression resolve bluff query for new employee salary

Theory:

Polynomial regression, a special case of multiple linear regression that adds terms with degrees greater than one to the model. The real-world curvilinear relationship is captured when you transform the training data by adding polynomial terms, which are then fit in the same manner as in multiple linear regression.

Historically, polynomial models are among the most frequently used empirical models for fitting functions. These models are popular for the following reasons: • In mathematical analysis, the Weierstrass approximation theorem states that every continuous function defined on an interval $[a,b]$ can be uniformly approximated as closely as desired by a polynomial function. They have a simple form, and well known and understood properties. They have a moderate flexibility of shapes, and they are computationally easy to use.

- Polynomial functions are a closed family. Linear transformations in the data result in a polynomial model being mapped to another polynomial model. That means that the polynomial models are not dependent on the underlying metric. Polynomial models also have the following limitations:

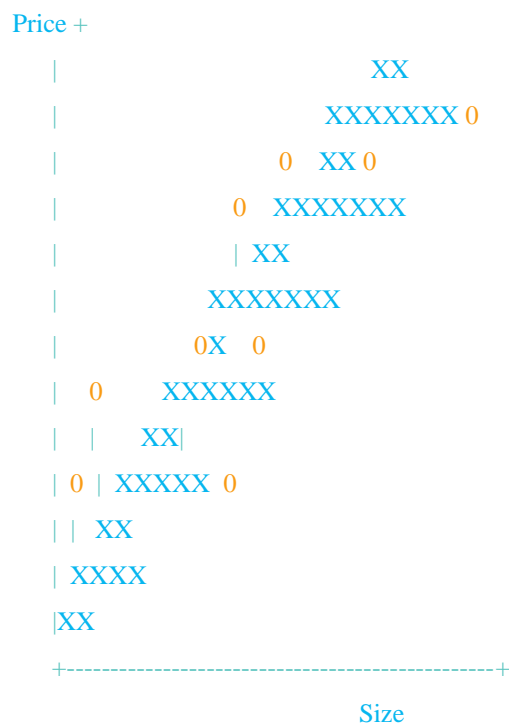
- Polynomial models have poor interpolatory and extrapolatory properties. High-degree polynomials are known for oscillation at the edges of an interval³ producing poor interpolatory properties. While polynomials may provide good fits within the range of data, the degree of fit frequently deteriorates rapidly outside the range of the data resulting in poor extrapolatory properties.
- Polynomial models have poor asymptotic properties. They have a finite response for finite values and have an infinite response if some variable takes an infinite value. Thus polynomials may not model asymptotic phenomena very well.
- Polynomial models have a shape/degree tradeoff. In order to model data with a complicated structure, the degree of the model must be high, indicating that the associated number of parameters to be estimated will also be high. This can result in highly unstable models. Polynomial regression is a form of linear regression in which the relationship between the input variables x and the output variable y is modeled as a polynomial. Although polynomial regression fits a nonlinear model to the data, as a statistical estimation problem it is linear, in the sense that the regression function is linear in the unknown parameters that are estimated from the data. For this reason, polynomial regression is considered to be a special case of linear regression.

Polynomial regression comes into play when your correlation of data is nonlinear and thus a linear model type isn't fitting anymore. Rather than using a straight line, so a **linear model** to estimate the predictions, it could be for instance a **quadratic model** or **cubic model** with a curved line. Polynomial regression is a form of linear regression that allows you to predict a single y variable by

decomposing the x variable into a n-th order polynomial. It can have any form of the following function for the hypothesis function.

$$h(x) \Rightarrow \text{thetaZero} + \text{thetaOne} * x + \text{thetaTwo} * x^2 + \text{thetaThree} * x^3 \dots \text{thetaK} * x^k$$

As the successive powers of x are added to the equation, the regression line changes its shape. By selecting a fitting model type, you can reduce your costs over time by a significant amount. In the following diagram, the regression line is better fitting than the previously linear regression line.

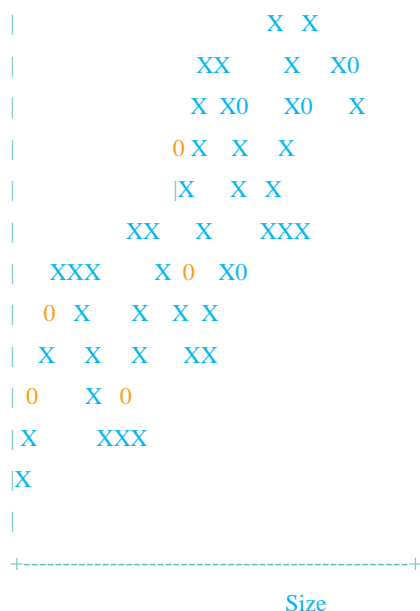


Polynomial regression can reduce your costs returned by the cost function. It gives your regression line a curvilinear shape and makes it more fitting for your underlying data. By applying a higher order polynomial, you can fit your regression line to your data more precisely. But isn't there any problem with the approach of using more complex polynomials to perfectly fit the regression line?

Over-fitting in Polynomial Regression

There is one crucial aspect when using polynomial regression. By selecting models for your regression problem, you want to determine which of these models is the most parsimonious. What does a parsimonious model mean? Generally speaking, you need to care more about a parsimony model rather than a best-fitting model. A complex model could **over-fit** your data. It becomes an **over-fitting problem** or in other words the algorithm has a **high variance**. For instance, you might find that a quadratic model fits your trainings set reasonably well. On the other hand, you find out about a very high order polynomial that goes almost perfectly through each of your data points.

Price +XXX



Even though this model fits perfectly, it will be terrible at making future predictions. It fits the data too well, so it's over-fitting. It's about balancing the model's complexity with the model's explanatory power. That's a parsimonious

model. It is a model that accomplishes a desired level of explanation or prediction with as few predictor variables as possible. In conclusion, you want to have a best-fitting prediction when using low order polynomials. There is no point in finding the best-fitting regression line that fits all of your data points.

Program:

```
# POLYNOMIAL LINEAR REGRESSION
```

```
# IMPORT LIBRARIES
```

```
import numpy as np
```

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

```
import pandas as pd
```

```
# IMPORTING DATASET
```

```
dataset = pd.read_csv("1_Regression/Position_Salaries.csv")
```

```
X = dataset.iloc[:, 1:2].values
```

```
y = dataset.iloc[:, 2].values
```

```
# Splitting Data into Training & Testing
```

```
# WE WILL NOT SPLIT DATA HERE Since the size of dataset is very small
```

```
# and that'll be ridiculous to do that with distinct values
```

```
# Fitting Linear Regression to the Dataset
```

```
from sklearn.linear_model import LinearRegression
```

```
lin_reg = LinearRegression()
```

```
lin_reg.fit(X, y)
```

```
# Fitting Polynomial Regression to the Dataset
```

```
from sklearn.preprocessing import PolynomialFeatures
```

```
poly_reg = PolynomialFeatures(degree=4)
```

```
X_poly = poly_reg.fit_transform(X)
```

```
lin_reg_2 = LinearRegression()
```

```
d = lin_reg_2.fit(X_poly, y)
```

```
# Visualising the Linear Regression results
```

```
plt.scatter(X, y, color='red')
plt.plot(X, lin_reg.predict(X), color='black')
plt.title("Truth or Bluff (Linear Regression)")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```

```
# Visualising the Polynomial Regression
X_grid = np.arange(min(X), max(X), 0.1)
X_grid = X_grid.reshape(len(X_grid), 1)
plt.scatter(X, y, color='red')
plt.plot(X_grid, lin_reg_2.predict(poly_reg.fit_transform(X_grid)), color='black')
plt.title("<?> Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```

```
# Predicting a new Result with Linear Regression
o = lin_reg.predict(X =6.5)
```

```
# Predictig a new Result with Polynomial Regression
o1 = lin_reg_2.predict(poly_reg.fit_transform(6.5))
```

Conclusion: In this experiment, we studied how to implement polynomial regression algorithm using python machine learning library. ,It is a special case of multiple linear regression that adds terms with degrees greater than one to the model.

ASSIGNMENT NUMBER 11

Aim: Classification: Use KNN (with WCSS) to Predict if a customer with certain age and Salary will purchase a product or not

Theory:

When do we use KNN algorithm?

KNN can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique we generally look at 3 important aspects:

1. Ease to interpret output
2. Calculation time
3. Predictive Power

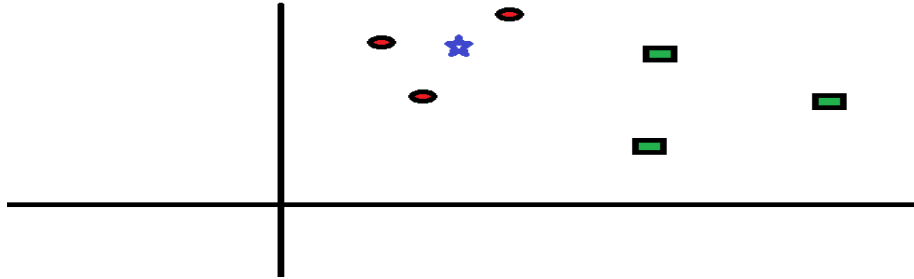
Let us take a few examples to place KNN in the scale :

	Logistic Regression	CART	Random Forest	KNN
1. Ease to interpret output	2	3	1	3
2. Calculation time	3	2	1	3
3. Predictive Power	2	2	3	2

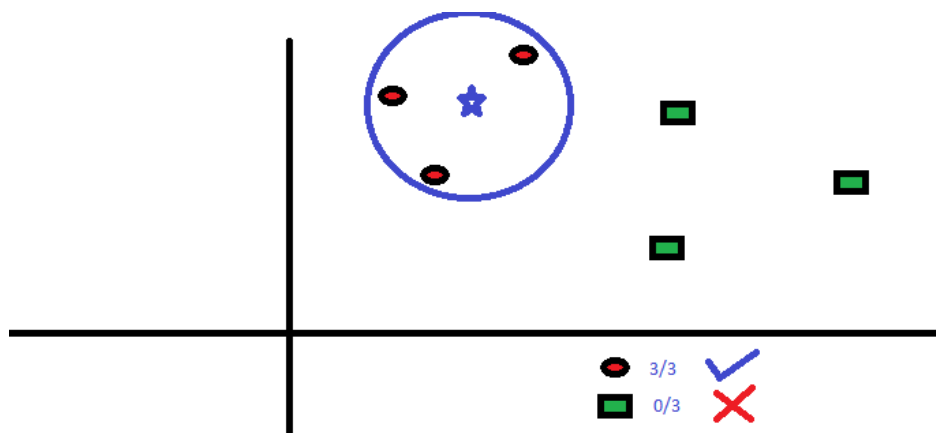
KNN algorithm fares across all parameters of considerations. It is commonly used for its easy of interpretation and low calculation time.

How does the KNN algorithm work?

Let's take a simple case to understand this algorithm. Following is a spread of red circles (RC) and green squares (GS):



You intend to find out the class of the blue star (BS). BS can either be RC or GS and nothing else. The “K” in KNN algorithm is the nearest neighbors we wish to take vote from. Let's say $K = 3$. Hence, we will now make a circle with BS as center just as big as to enclose only three datapoints on the plane. Refer to following diagram for more details:

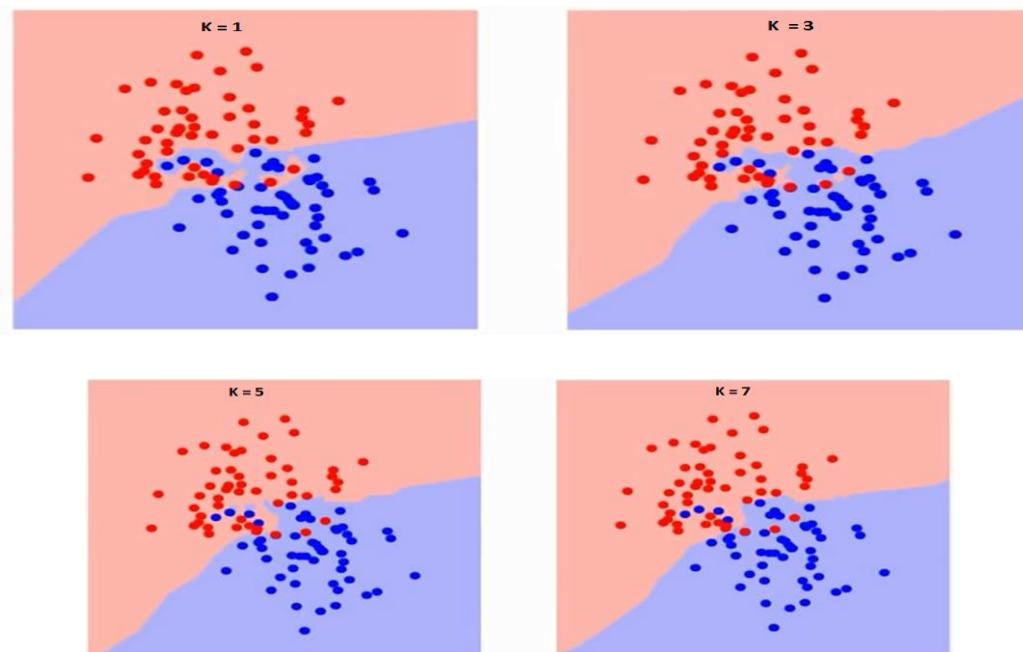


The three closest points to BS are all RC. Hence, with good confidence level we can say that the BS should belong to the class RC. Here, the choice became very

obvious as all three votes from the closest neighbor went to RC. The choice of the parameter K is very crucial in this algorithm. Next we will understand what are the factors to be considered to conclude the best K .

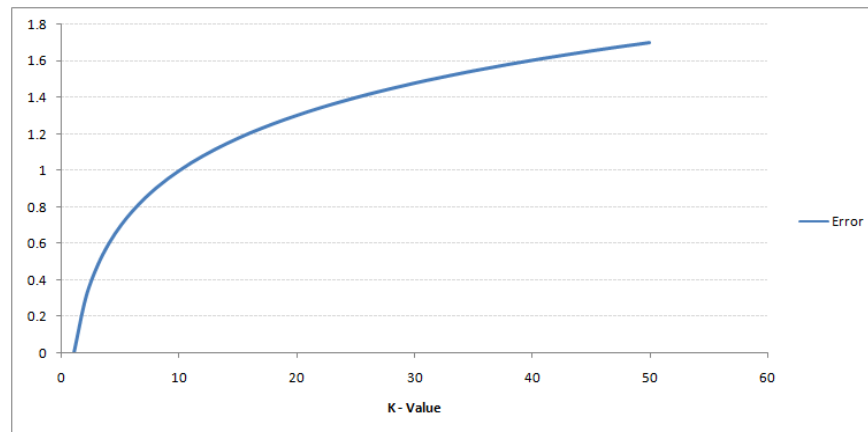
How do we choose the factor K ?

First let us try to understand what exactly K influences in the algorithm. If we see the last example, given that all the 6 training observation remain constant, with a given K value we can make boundaries of each class. These boundaries will segregate RC from GS. The same way, let's try to see the effect of value " K " on the class boundaries. Following are the different boundaries separating the two classes with different values of K .

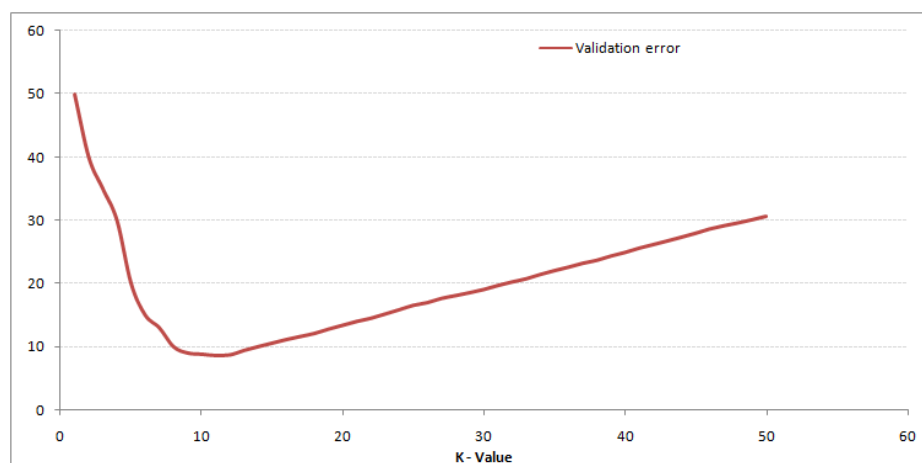


If you watch carefully, you can see that the boundary becomes smoother with increasing value of K . With K increasing to infinity it finally becomes all blue or all red depending on the total majority. The training error rate and the validation

error rate are two parameters we need to access on different K-value. Following is the curve for the training error rate with varying value of K:



As you can see, the error rate at K=1 is always zero for the training sample. This is because the closest point to any training data point is itself. Hence the prediction is always accurate with K=1. If validation error curve would have been similar, our choice of K would have been 1. Following is the validation error curve with varying value of K:



This makes the story more clear. At K=1, we were overfitting the boundaries. Hence, error rate initially decreases and reaches a minima. After the minima

point, it then increase with increasing K. To get the optimal value of K, you can segregate the training and validation from the initial dataset. Now plot the validation error curve to get the optimal value of K. This value of K should be used for all predictions.

Breaking it Down – Pseudo Code of KNN

We can implement a KNN model by following the below steps:

1. Load the data
2. Initialise the value of k
3. For getting the predicted class, iterate from 1 to total number of training data points
 1. Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric since it's the most popular method. The other metrics that can be used are Chebyshev, cosine, etc.
 2. Sort the calculated distances in ascending order based on distance values
 3. Get top k rows from the sorted array
 4. Get the most frequent class of these rows
 5. Return the predicted class

Program:

```
# K NEAREST NEIGHBORS
```

```
# IMPORT LIBRARIES  
import numpy as np
```

```

import matplotlib.pyplot as plt
import pandas as pd

# IMPORTING DATASET
dataset = pd.read_csv("2_Classification/Social_Network_Ads.csv")
X = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, 4].values

# Splitting Data into Training & Testing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)

# Fitting Classifier to the Training set
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

# for i in range(len(y_pred)):
#     print(X_test[i, :], y_pred[i])

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step
= 0.01),
np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
c = ListedColormap(('red', 'green'))(i), label = j)
plt.title ('KNN (Train set)')
plt.xlabel ('Age')

```



```

plt.ylabel ('Estimated Salary')
plt.legend ()
plt.show ()

# visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid (np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step
= 0.01),
    np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict (np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
    alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter (X_set[y_set == j, 0], X_set[y_set == j, 1],
        c = ListedColormap(('red', 'green'))(i), label = j)
plt.title ('KNN (Test set)')
plt.xlabel ('Age')
plt.ylabel ('Estimated Salary')
plt.legend ()
plt.show ()

```

Conclusion: In this experiment we applied KNN algorithm (with WCSS) to Predict if a customer with certain age and Salary will purchase a product or not.

ASSIGNMENT NUMBER 12-1

Aim: Classification: Use Decision Tree to Predict if a customer with certain age and Salary will purchase a product or not

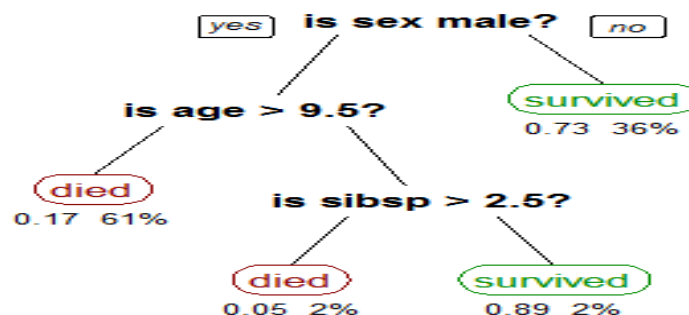
Theory:

Decision Trees in Machine Learning

A tree has many analogies in real life, and turns out that it has influenced a wide area of **machine learning**, covering both **classification and regression**. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions. Though a commonly used tool in data mining for deriving a strategy to reach a particular goal, its also widely used in machine learning, which will be the main focus of this article.

How can an algorithm be represented as a tree?

For this let's consider a very basic example that uses titanic data set for predicting whether a passenger will survive or not. Below model uses 3 features/attributes/columns from the data set, namely sex, age and sibsp (number of spouses or children along).



A decision tree is drawn upside down with its root at the top. In the image on the left, the bold text in black represents a condition/internal node, based on which

the tree splits into branches/ **edges**. The end of the branch that doesn't split anymore is the decision/**leaf**, in this case, whether the passenger died or survived, represented as red and green text respectively.

Although, a real dataset will have a lot more features and this will just be a branch in a much bigger tree, but you can't ignore the simplicity of this algorithm. The **feature importance is clear** and relations can be viewed easily. This methodology is more commonly known as **learning decision tree from data** and above tree is called **Classification tree** as the target is to classify passenger as survived or died. **Regression trees** are represented in the same manner, just they predict continuous values like price of a house. In general, Decision Tree algorithms are referred to as CART or Classification and Regression Trees.

So, what is actually going on in the background? Growing a tree involves deciding on **which features to choose** and **what conditions to use** for splitting, along with knowing when to stop. As a tree generally grows arbitrarily, **you will need to trim it down** for it to look beautiful. Lets start with a common technique used for splitting.

Recursive Binary Splitting



In this procedure all the features are considered and different split points are tried and tested using a cost function. The split with the best cost (or lowest cost) is selected.

Consider the earlier example of tree learned from titanic dataset. In the first split or the root, all attributes/features are considered and the training data is divided into groups based on this split. We have 3 features, so will have 3 candidate splits. Now we will **calculate how much accuracy each split will cost us, using a function. The split that costs least is chosen**, which in our example is sex of the passenger. This **algorithm is recursive in nature** as the groups formed can be sub-

divided using same strategy. Due to this procedure, this algorithm is also known as the **greedy algorithm**, as we have an excessive desire of lowering the cost. **This makes the root node as best predictor/classifier.**

Cost of a split

Lets take a closer look at **cost functions used for classification and regression**. In both cases the cost functions try to **find most homogeneous branches, or branches having groups with similar responses**. This makes sense we can be more sure that a test data input will follow a certain path.

Regression : $\text{sum}(y - \text{prediction})^2$

Lets say, we are predicting the price of houses. Now the decision tree will start splitting by considering each feature in training data. The mean of responses of the training data inputs of particular group is considered as prediction for that group. The above function is applied to all data points and cost is calculated for all candidate splits. *Again the split with lowest cost is chosen.* Another cost function involves reduction of standard deviation.

*Classification : $G = \text{sum}(pk * (1 - pk))$*

A Gini score gives an idea of how good a split is by how mixed the response classes are in the groups created by the split. Here, pk is proportion of same class inputs present in a particular group. A perfect class purity occurs when a group contains all inputs from the same class, in which case pk is either 1 or 0 and $G = 0$, where as a node having a 50–50 split of classes in a group has the worst purity, so for a binary classification it will have $pk = 0.5$ and $G = 0.5$.

When to stop splitting?

You might ask ***when to stop growing a tree?*** As a problem usually has a large set of features, it results in large number of split, which in turn gives a huge tree. Such trees are *complex and can lead to overfitting*. So, we need to know when to stop? One way of doing this is to **set a minimum number of training inputs to use on each leaf**. For example we can use a minimum of 10 passengers to reach a decision(died or survived), and ignore any leaf that takes less than 10 passengers. Another way is to set **maximum depth** of your model. **Maximum depth refers to the the length of the longest path from a root to a leaf.**

Pruning

The performance of a tree can be further increased by **pruning**. It involves **removing the branches that make use of features having low importance**. This way, we reduce the complexity of tree, and thus increasing its predictive power by reducing overfitting.

Pruning can start at either root or the leaves. The simplest method of pruning starts at leaves and removes each node with most popular class in that leaf, this change is kept if it doesn't deteriorate accuracy. Its also called **reduced error pruning**. More sophisticated pruning methods can be used such as **cost complexity pruning** where a learning parameter (alpha) is used to weigh whether nodes can be removed based on the size of the sub-tree. This is also known as **weakest link pruning**.

Advantages of CART

- Simple to understand, interpret, visualize.
- Decision trees *implicitly perform variable screening or feature selection*.
- Can *handle both numerical and categorical data*. Can also handle *multi-output problems*.
- Decision trees require *relatively little effort from users for data preparation*.
- *Nonlinear relationships between parameters do not affect tree performance*.

Disadvantages of CART

- Decision-tree learners *can create over-complex trees* that do not generalize the data well. This is called *overfitting*.
- Decision trees can be unstable because *small variations in the data might result in a completely different tree being generated*. This is called *variance*, which needs to be *lowered by methods like bagging and boosting*.
- Greedy algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees, where the features and samples are randomly sampled with replacement.

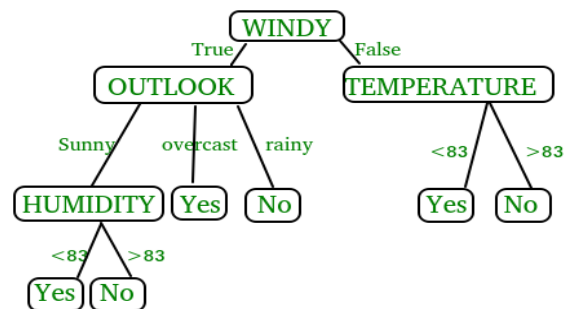
- Decision tree learners create *biased trees if some classes dominate*. It is therefore recommended to balance the data set prior to fitting with the decision tree.

Assumptions we make while using Decision tree :

- At the beginning, we consider the whole training set as the root.
- Attributes are assumed to be categorical for information gain and for gini index, attributes are assumed to be continuous.
- On the basis of attribute values records are distributed recursively.
- We use statistical methods for ordering attributes as root or internal node.

Pseudocode :

- Find the best attribute and place it on the root node of the tree.
- Now, split the training set of the dataset into subsets. While making the subset make sure that each subset of training dataset should have the same value for an attribute.
- Find leaf nodes in all branches by repeating 1 and 2 on each subset.



While implementing the decision tree we will go through the following two phases:

- Building Phase
 - Preprocess the dataset.
 - Split the dataset from train and test using Python sklearn package.
 - Train the classifier.
- Operational Phase
 - Make predictions.
 - Calculate the accuracy.

Data Import :

- To import and manipulate the data we are using the *pandas* package provided in python.
- Here, we are using a URL which is directly fetching the dataset from the UCI site no need to download the dataset. When you try to run this code on your system make sure the system should have an active Internet connection.
- As the dataset is separated by “,” so we have to pass the sep parameter’s value as “,”.
- Another thing is notice is that the dataset doesn’t contain the header so we will pass the Header parameter’s value as none. If we will not pass the header parameter then it will consider the first line of the dataset as the header.

Data Slicing :

- Before training the model we have to split the dataset into the training and testing dataset.
- To split the dataset for training and testing we are using the sklearn module *train_test_split*
- First of all we have to separate the target variable from the attributes in the dataset.

```
X = balance_data.values[:, 1:5]
```

```
Y = balance_data.values[:,0]
```

- Above are the lines from the code which separte the dataset. The variable X contains the attributes while the variable Y contains the target variable of the dataset.
- Next step is to split the dataset for training and testing purpose.

```
X_train, X_test, y_train, y_test = train_test_split(
```

```
X, Y, test_size = 0.3, random_state = 100)
```

- Above line split the dataset for training and testing. As we are splitting the dataset in a ratio of 70:30 between training and testing so we are pass *test_size* parameter’s value as 0.3.
- *random_state* variable is a pseudo-random number generator state used for random sampling

s

Program:

```
# Decision Tree Classifier

# IMPORT LIBRARIES
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# IMPORTING DATASET
dataset = pd.read_csv("2_Classification/Social_Network_Ads.csv")
X = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, 4].values

# Splitting Data into Training & Testing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
# DT is not based on Euclidean Distances
# We need some interpretations
# We need actual values and regions

# Fitting Logistic Regression to the Training set
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

for i in range(len(y_pred)):
    print(y_test[i], y_pred[i])

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop =
X_set[:, 0].max() + 1, step = 0.01),
                    np.arange(start = X_set[:, 1].min() - 1, stop =
X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
            alpha = 0.75, cmap = ListedColormap(('red', 'green')))
```



```

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Decision Tree (Train set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop =
X_set[:, 0].max() + 1, step = 0.01),
                    np.arange(start = X_set[:, 1].min() - 1, stop =
X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Decision Tree (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

```

Conclusion: In this experiment we performed Classification Using Decision Tree algorithm to Predict if a customer with certain age and Salary will purchase a product or not.

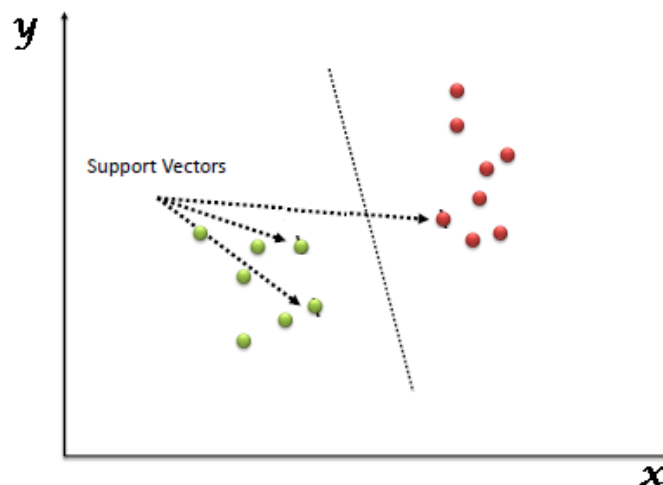
ASSIGNMENT NUMBER 12-2

Aim: Classification: Using SVM Predict if a customer with certain age and Salary will purchase a product or not

Theory:

What is Support Vector Machine?

“Support Vector Machine” (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well (look at the below snapshot).



Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/line).

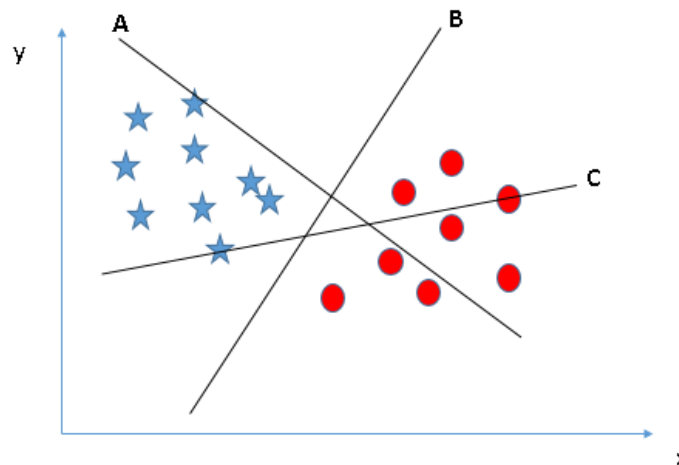
You can look at [definition of support vectors](#) and a few examples of its working here.

How does it work?

Above, we got accustomed to the process of segregating the two classes with a hyper-plane. Now the burning question is “How can we identify the right hyper-plane?”. Don’t worry, it’s not as hard as you think!

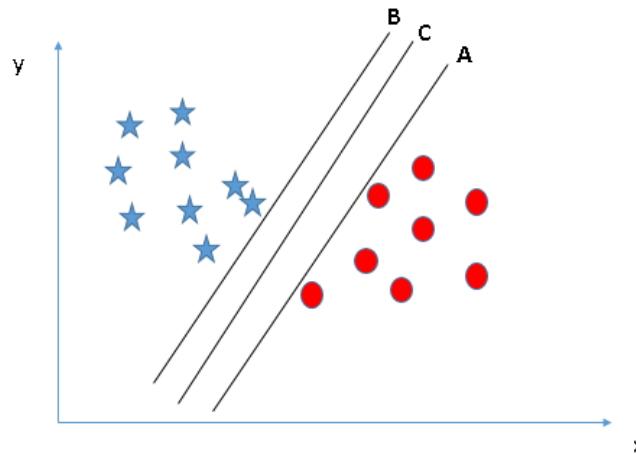
Let’s understand:

- **Identify the right hyper-plane (Scenario-1):** Here, we have three hyper-planes (A, B and C). Now, identify the right hyper-plane to classify star and circle.

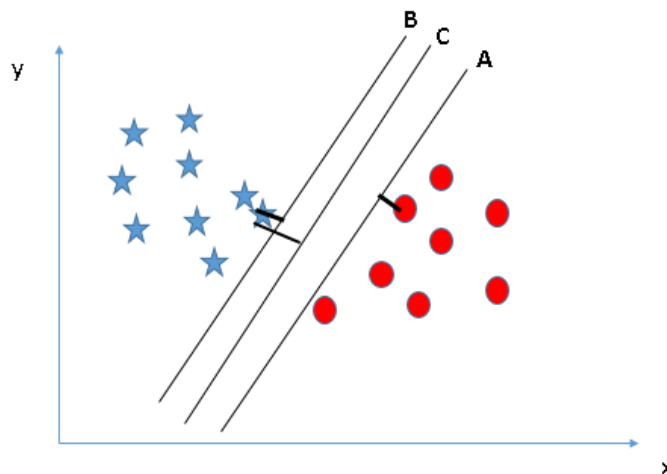


You need to remember a thumb rule to identify the right hyper-plane: “Select the hyper-plane which segregates the two classes better”. In this scenario, hyper-plane “B” has excellently performed this job.

- **Identify the right hyper-plane (Scenario-2):** Here, we have three hyper-planes (A, B and C) and all are segregating the classes well. Now, How can we identify the right hyper-plane?



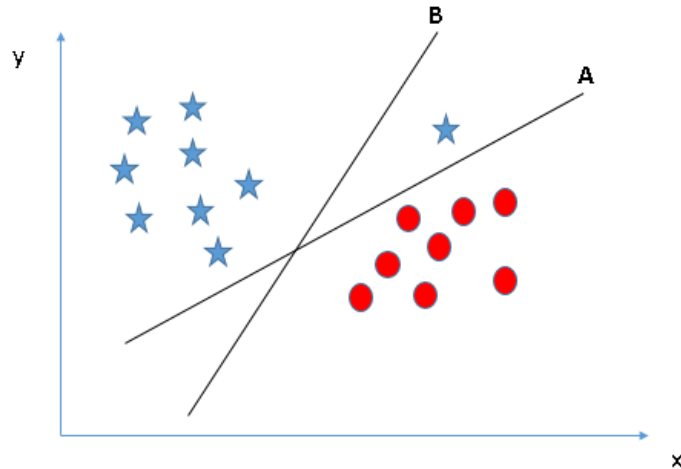
Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as **Margin**. Let's look at the below



snapshot:

Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is high chance of miss-classification.

- **Identify the right hyper-plane (Scenario-3):**Hint: Use the rules as discussed in previous section to identify the right hyper-plane



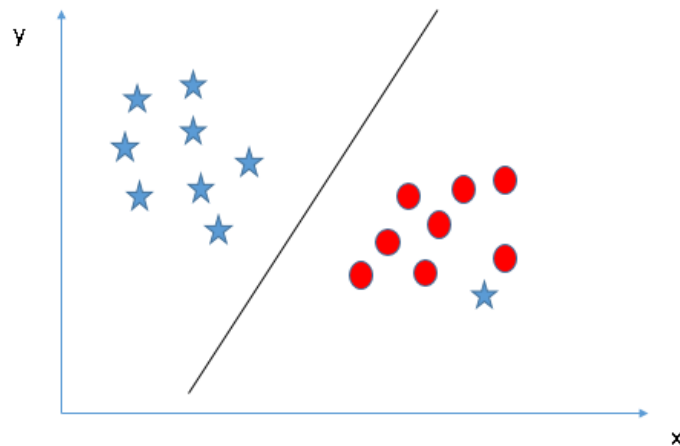
Some of you may have selected the hyper-plane **B** as it has higher margin compared to **A**. But, here is the catch, SVM selects the hyper-plane which classifies the classes accurately prior to maximizing margin. Here, hyper-plane B has a classification error and A has classified all correctly. Therefore, the right hyper-plane is **A**.

- **Can we classify two classes (Scenario-4)?**: Below, I am unable to segregate the two classes using a straight line, as one of star lies in the territory of other(circle) class as an outlier.

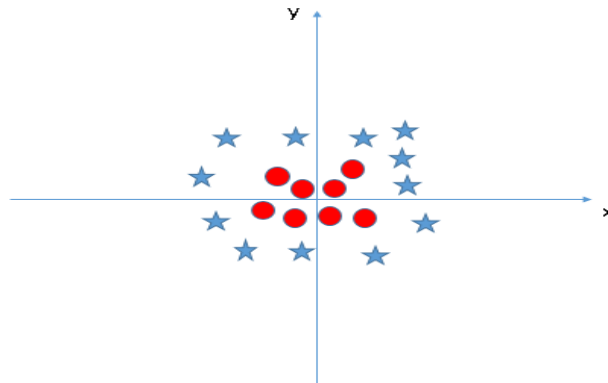


As I have already mentioned, one star at other end is like an outlier for star class. SVM has a feature to ignore outliers and find the hyper-plane that has

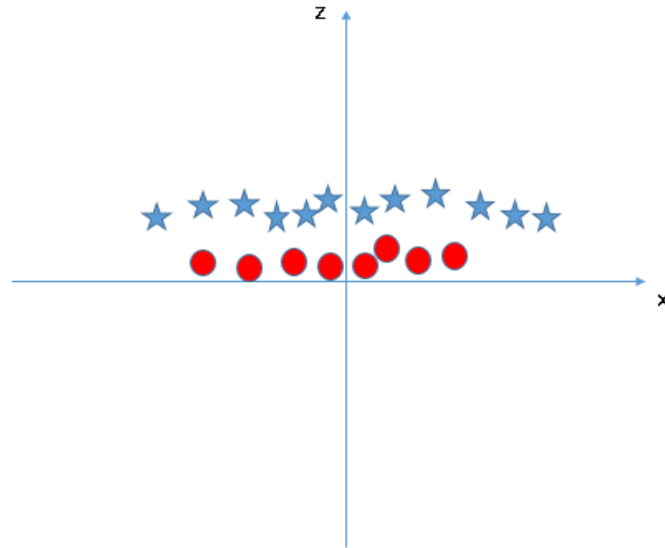
maximum margin. Hence, we can say, SVM is robust to outliers.



- **Find the hyper-plane to segregate to classes (Scenario-5):** In the scenario below, we can't have linear hyper-plane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyper-plane.



SVM can solve this problem. Easily! It solves this problem by introducing additional feature. Here, we will add a new feature $z = x^2 + y^2$. Now, let's plot the data points on axis x and z:

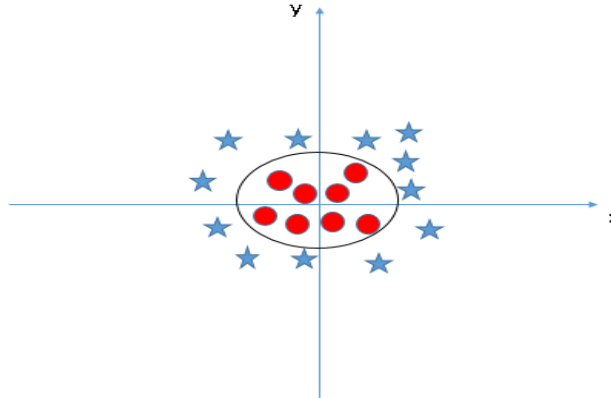


In above plot, points to consider are:

- All values for z would be positive always because z is the squared sum of both x and y
- In the original plot, red circles appear close to the origin of x and y axes, leading to lower value of z and star relatively away from the origin result to higher value of z .

In SVM, it is easy to have a linear hyper-plane between these two classes. But, another burning question which arises is, should we need to add this feature manually to have a hyper-plane. No, SVM has a technique called the **kernel trick**. These are functions which takes low dimensional input space and transform it to a higher dimensional space i.e. it converts not separable problem to separable problem, these functions are called kernels. It is mostly useful in non-linear separation problem. Simply put, it does some extremely complex data transformations, then find out the process to separate the data based on the labels or outputs you've defined.

When we look at the hyper-plane in original input space it looks like a circle:



How to implement SVM in Python?

In Python, scikit-learn is a widely used library for implementing machine learning algorithms, SVM is also available in scikit-learn library and follow the same structure (Import library, object creation, fitting model and prediction).

Program:

```
# SVM
# IMPORT LIBRARIES
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# IMPORTING DATASET
dataset = pd.read_csv("2_Classification/Social_Network_Ads.csv")
X = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, 4].values

# Splitting Data into Training & Testing
from sklearn.cross_validation import train_test_split
```



```

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)

# Fitting SVM to the Training set
from sklearn.svm import SVC
classifier = SVC(kernel='rbf', random_state=0)
# C = Penalty Parameter
# Kernel = RBF, LINEAR, POLY, SIGMOID
# Degree = If you choose POLY
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

# for i in range(len(y_pred)):
#     print(X_test[i, :], y_pred[i])

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop =
X_set[:, 0].max() + 1, step = 0.01),
                    np.arange(start = X_set[:, 1].min() - 1, stop =
X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
            alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('SVM (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop =
X_set[:, 0].max() + 1, step = 0.01),
                    np.arange(start = X_set[:, 1].min() - 1, stop =
X_set[:, 1].max() + 1, step = 0.01))

```

```

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
               c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('SVM (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

```

Conclusion: In this experiment we studied SVM algorithm for Classification to Predict if a customer with certain age and Salary will purchase a product or not. We used scikit-learn library for implementing SVM algorithm.

ASSIGNMENT NUMBER 13

Aim: Clustering: Using K-Means clustering, determine Customers of a Mall according to Categories so as to launch a scheme for business growth a product or not for imbalanced data and determining Fitting issues and Sampling methods and Optimizing techniques.

Theory:

K-Means clustering is an unsupervised learning algorithm that, as the name hints, finds a fixed number (k) of clusters in a set of data. A *cluster* is a group of data points that are grouped together due to similarities in their features. When using a K-Means algorithm, a cluster is defined by a *centroid*, which is a point (either imaginary or real) at the center of a cluster. Every point in a data set is part of the cluster whose centroid is most closely located. To put it simply, K-Means finds k number of centroids, and then assigns all data points to the closest cluster, with the aim of keeping the centroids small.

The Algorithm

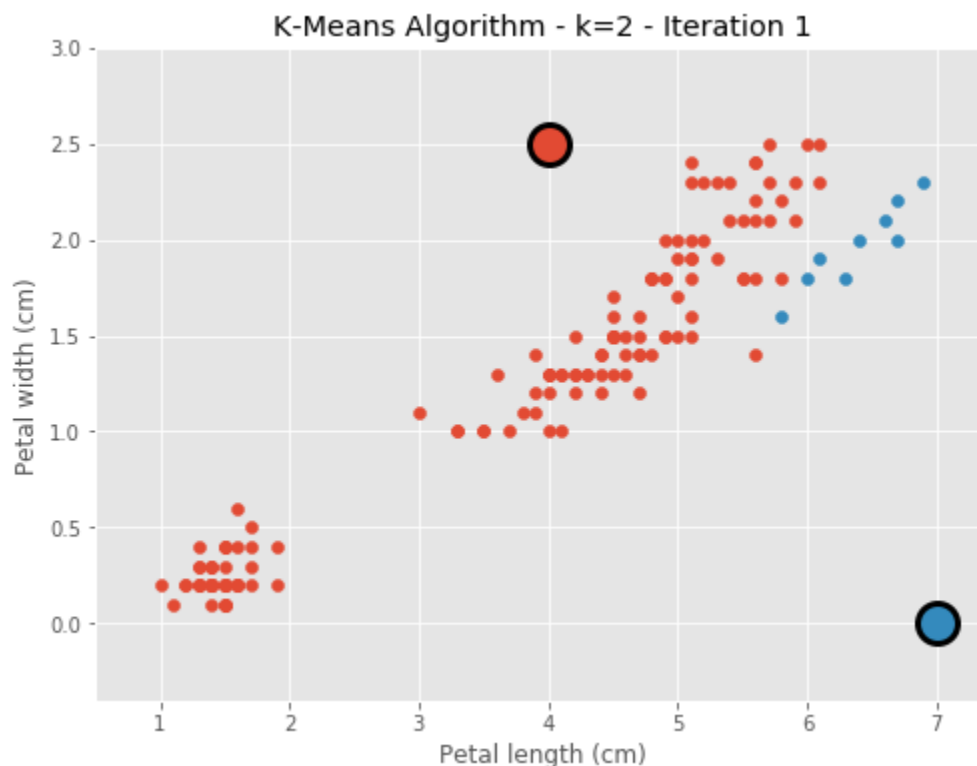
K-Means starts by randomly defining k centroids. From there, it works in iterative (repetitive) steps to perform two tasks:

1. Assign each data point to the closest corresponding centroid, using the standard Euclidean distance. In layman's terms: the straight-line distance between the data point and the centroid.
2. For each centroid, calculate the mean of the values of all the points belonging to it. The mean value becomes the new value of the centroid.

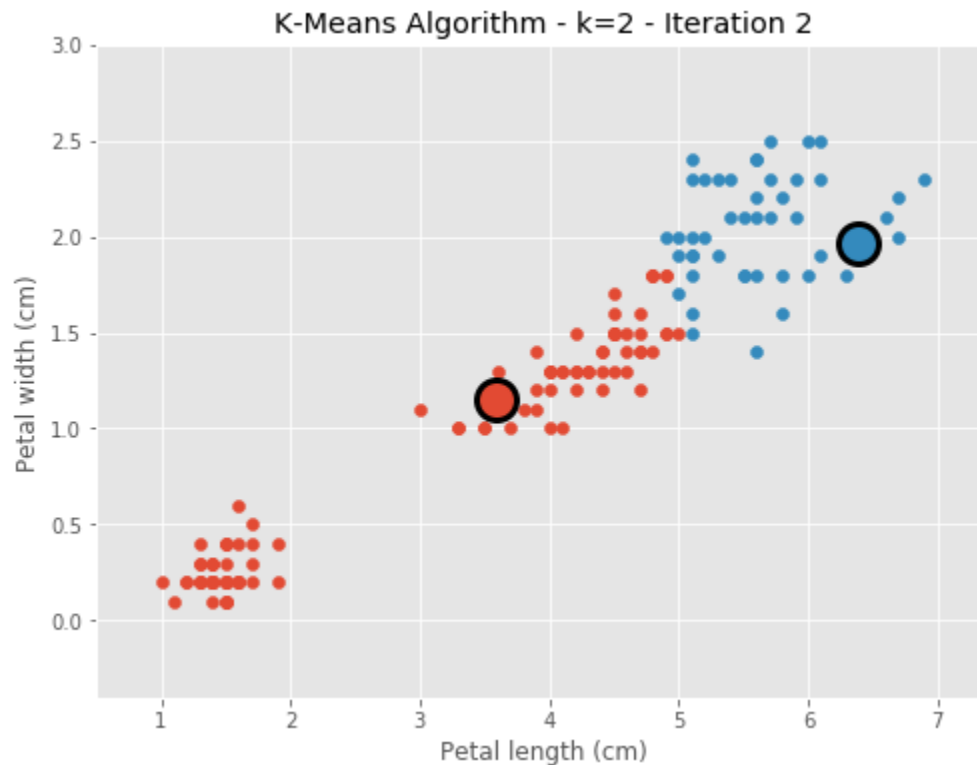
Once step 2 is complete, all of the centroids have new values that correspond to the means of all of their corresponding points. These new points are put through steps one and two producing yet another set of centroid values. This process is repeated over and over until there is no change in the centroid values, meaning that they have been accurately grouped. Or, the process can be stopped when a previously determined maximum number of steps has been met.

Applying the K-Means Algorithm

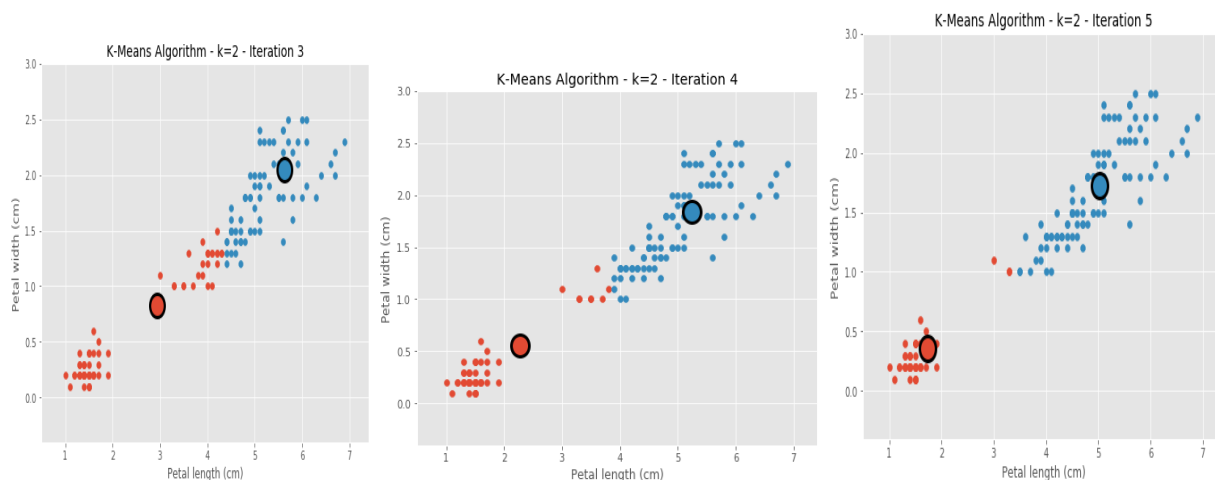
Iteration 1: First, we create two randomly generated centroids and assign each data point to the cluster of the closest centroid. In this case, because we are using two centroids, our k value is 2.



Iteration 2: As you can see above, the centroids are not evenly distributed. In the second iteration of the algorithm, the average values of each of the two clusters are found and become the new centroid values.



Iterations 3-5: We repeat the process until there is no further change in the value of the centroids.



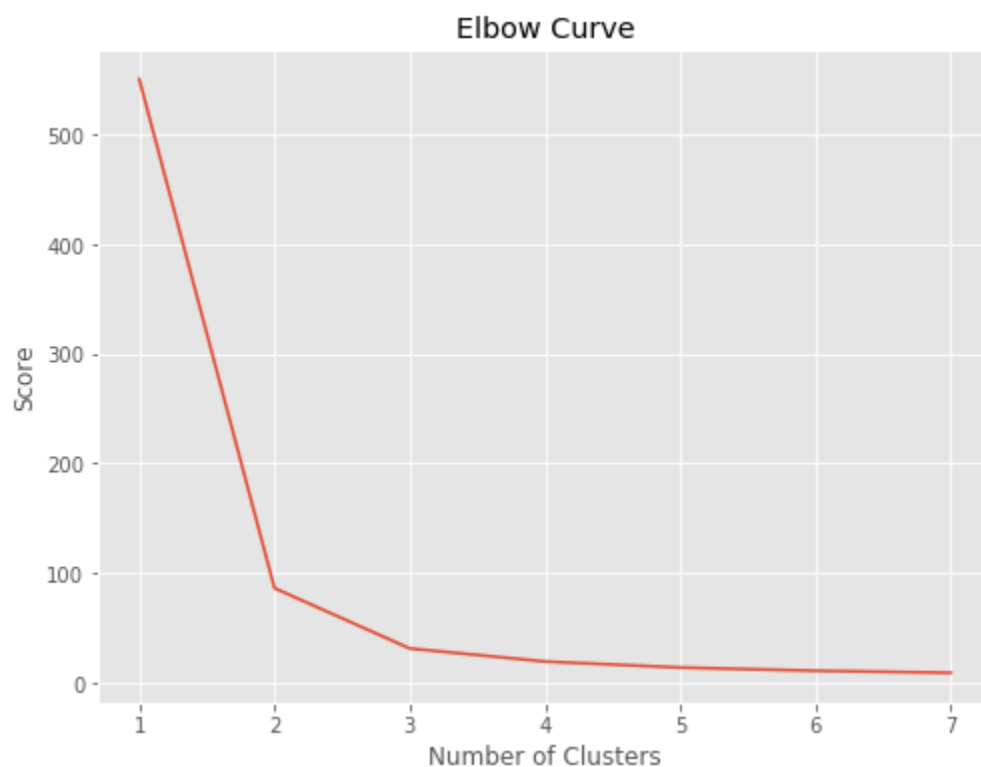
Finally, after iteration 5, there is no further change in the clusters.

Choosing K

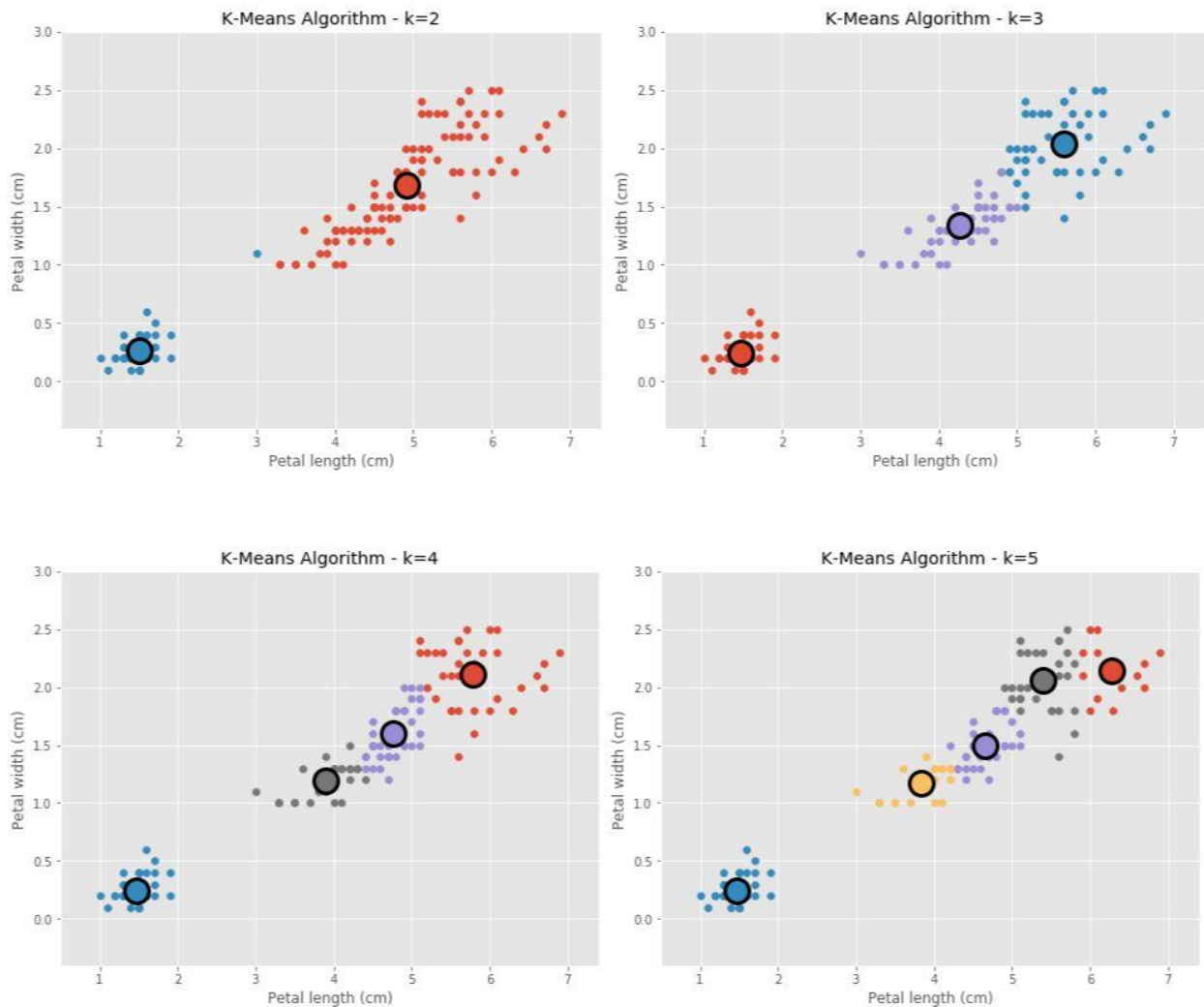
The algorithm explained above finds clusters for the number k that we chose. So, how do we decide on that number?

To find the best k we need to measure the quality of the clusters. The most traditional and straightforward method is to start with a random k , create centroids, and run the algorithm as we explained above. A sum is given based on the distances between each point and its closest centroid. As an increase in clusters correlates with smaller groupings and distances, this sum will always decrease when k increases; as an extreme example, if we choose a k value that is equal to the number of data points that we have, the sum will be zero.

The goal with this process is to find the point at which increasing k will cause a very small decrease in the error sum, while decreasing k will sharply increase the error sum. This sweet spot is called the “elbow point.” In the image below, it is clear that the “elbow” point is at $k=3$.



Now, let's take a look at the visual differences between using two, three, four, or five clusters.



The different graphs also show that three is the most appropriate k value, which makes sense when taking into account that the data contains three types of iris flowers.

The K-Means algorithm is a great example of a simple, yet powerful algorithm. For example, it can be used to cluster phishing attacks with the objective of

discovering common patterns and even discovering new phishing kits. It can also be used to understand fraudulent patterns in financial transactions.

Advantages of K-Means:

- Widely used method for cluster analysis
- Easy to understand
- Trains quickly

Disadvantages of K-Means:

- Euclidean distance is not ideal in many applications
- Performance is (generally) not competitive with the best clustering methods
- Small variations in the data can result in a completely different clusters (high variance)
- Clusters are assumed to have a spherical shape and be evenly sized

Program:

```
# K MEANS CLUSTERING
```

```
# Importing the Libraries
```

```
import numpy as np
```

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

```
import pandas as pd
```



```

# Import Mall Dataset
dataset = pd.read_csv('3_Clustering/Mall_Customers.csv')
# Clients that subscribe to Membership card
# Maintains the Purchase history
# Score is Dependent on INCOME,
# No. times in week the show up in Mall, total expense in same mall
# YOU ARE!!
# TO Segment Clients into Different Groups based on Income & Score
# CLUSTERING PROBLEM
X = dataset.iloc[:, [3, 4]].values
# We have no Idea to look for
# We don't know the Optimal no. of Clusters

# USE ELBOW METHOD
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    # Fit values into KM
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,
random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title("ELBOW METHOD")
plt.xlabel("No. of Clus")
plt.ylabel("WCSS")
plt.plot()

# Applying K-means to Mall
kmeans = KMeans(n_clusters=5, init='k-means++', max_iter=300, n_init=10,
random_state=0)
y_kmeans = kmeans.fit_predict(X)

```

```
# Visualising the Clusters
```

```
plt.scatter(X[y_kmeans==0, 0], X[y_kmeans==0, 1], s= 100, c = 'red', label='Cluster1')  
plt.scatter(X[y_kmeans==1, 0], X[y_kmeans==0, 1], s= 100, c = 'blue', label='Cluster2')  
plt.scatter(X[y_kmeans==2, 0], X[y_kmeans==0, 1], s= 100, c = 'yellow',  
label='Cluster3')  
plt.scatter(X[y_kmeans==3, 0], X[y_kmeans==0, 1], s= 100, c = 'green', label='Cluster4')  
plt.scatter(X[y_kmeans==4, 0], X[y_kmeans==0, 1], s= 100, c = 'cyan', label='Cluster5')  
plt.scatter(kmeans.cluster_centers_[0, 0], kmeans.cluster_centers_[0, 1], s=300, c = 'red',  
label='Cluster1')  
plt.title('Clusters of customers')  
plt.xlabel('Annual Income (k$)')  
plt.ylabel('Spending Score (1-100)')  
plt.legend()  
plt.show()
```

Conclusion: In this experiment we studied Clustering algorithm from unsupervised learning. Using K-Means clustering, we determined Customers of a Mall according to Categories, to launch a scheme for business growth. Determining Fitting issues and Sampling methods and Optimizing techniques to achieve clustering.