LAB # 1: Introduction to Anaconda/Python.

Anaconda is a FREE enterprise-ready Python distribution for data analytics, processing, and scientific computing. Anaconda comes with Python 2.7 or Python 3.4 and 100+ cross-platform tested and optimized Python packages. All of the usual Python ecosystem tools work with Anaconda.

Additionally, Anaconda can create custom environments that mix and match different Python versions (2.6, 2.7, 3.3 or 3.4) and other packages into isolated environments and easily switch between them using conda, our innovative multi-platform package manager for Python and other languages.

INSTALLATION

Anaconda installs cleanly into a single directory, does not require Administrator or root privileges, does not affect other Python installs on your system, or interfere with OSX Frameworks.

System Requirements

System Requirements		
Linux	Windows	Mac OSX
32/64 bit Intel processor	32/64 bit Intel processor	64-bit Intel processor

Download Anaconda

Linux / Mac OS X command-line

After downloading the installer, in the shell execute, bash <downloaded file> Mac OS X (graphical installer)

After downloading the installer, double click the .pkg file and follow the instructions on the screen.

Windows

After downloading the installer, double click the .exe file and follow the instructions on the screen. Detailed Anaconda Installation Instructions are available at

http://docs.continuum.io/anaconda/install.html

Why Python?

1. Very Simple Language to learn.

   Get data (simulation, experiment control), Manipulate and process data, visualize results, quickly to understand, but also with high quality figures, for reports or publications.

2. Include best packages for Artificial Intelligence.

   Matplotlib, scikitlearn, scipy, numpy and tenserflow etc.

3. Extensively used in industry.

   General purpose Language,

How does Python compare to other solutions?

Compiled languages: C, C++, Fortran…

Pros: Very fast. For heavy computations, it's difficult to outperform these languages.

Cons: Painful usage: no interactivity during development, mandatory compilation steps, verbose syntax, manual memory management. These are difficult languages for non-programmers.

Matlab scripting language

Pros: Very rich collection of libraries with numerous algorithms, for many different domains. Fast execution because these libraries are often written in a compiled language. Pleasant development environment: comprehensive and help, integrated editor, etc. Commercial support is available.

Cons: Base language is quite poor and can become restrictive for advanced users. Not free.

Julia

Pros: Fast code, yet interactive and simple. Easily connects to Python or C.

Cons: Ecosystem limited to numerical computing. Still young.

Other scripting languages: Scilab, Octave, R, IDL, etc.

Pros: Open-source, free, or at least cheaper than Matlab. Some features can be very advanced (statistics in R, etc.)

Cons: Fewer available algorithms than in Matlab, and the language is not more advanced. Some software are dedicated to one domain. Ex: Gnuplot to draw curves. These programs are very powerful, but they are restricted to a single type of usage, such as plotting.

Python

Pros: Very rich scientific computing libraries. Well thought out language, allowing to write very readable and well structured code: we "code what we think". Many libraries beyond scientific computing (web server, serial port access, etc.). Free and open-source software, widely spread, with a vibrant community. A variety of powerful environments to work in, such as IPython, Spyder, Jupyter notebooks, Pycharm.

Cons: Not all the algorithms that can be found in more specialized software or toolboxes.

Starting Python

Start the Ipython shell (an enhanced interactive Python shell): by typing "ipython" from a Linux/Mac terminal, or from the Windows cmd shell, or by starting the program from a menu, e.g. in the Python(x,y) or EPD menu if you have installed one of these scientific-Python suites.

If you don't have Ipython installed on your computer, other Python shells are available, such as the plain Python shell started by typing "python" in a terminal, or the Idle interpreter. However, we advise to use the Ipython shell because of its enhanced features, especially for interactive scientific computing.

SPYDER (The Scientific Python Development Environment.)

Spyder is a free and open source scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. It features a unique combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package.

Python Keyword.

Keywords are the reserved words in python. We can't use a keyword as variable name, function name or any other identifier. Keywords are case sensitive.

```
import keyword
print(keyword.kwlist)
print("\nTotal number of keywords ", len(keyword.kwlist))
```

```
In [3]: runfile('C:/Users/ALI/.spyder-py3/temp.py', wdir='C:/Users/ALI/.spyder-py3')
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']

Total number of keywords  35
```

Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for Writing Identifiers:

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.

```
abc_12=12
```

```
In [7]: 1a=1
  File "<ipython-input-7-fbf1bec92ff8>", line 1
    1a=1
     ^
SyntaxError: invalid syntax
```

```
In [5]: global=1
  File "<ipython-input-5-89339e2d91ea>", line 1
    global=1
           ^
SyntaxError: invalid syntax
```

We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
In [6]: abc@=12
Traceback (most recent call last):

  File "<ipython-input-6-418b82dab90c>", line 1, in <module>
    abc@=12

NameError: name 'abc' is not defined
```

Python Comments

Comments are lines that exist in computer programs that are ignored by compilers and interpreters. Including comments in programs makes code more readable for humans as it provides some information or explanation about what each part of a program is doing.

In general, it is a good idea to write comments while you are writing or updating a program as it is easy to forget your thought process later on, and comments written later may be less useful in the long term. In Python, we use the hash (#) symbol to start writing a comment.

```
#Print Hello, world to console
print("Hello, world")
```

Multi Line Comments

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line.

```
#This is a long comment
#and it extends
#Multiple lines
```

Another way of writing multiline is to use triple quotes, either ''' or """.

```
"""This is also a
perfect example of
multi-line comments"""
```

```
In [7]: """This is also a
   ...: perfect example of
   ...: multi-line comments"""
Out[7]: 'This is also a\nperfect example of\nmulti-line comments'
```

Python Indentation

1. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
2. A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.
3. Generally, four whitespaces are used for indentation and is preferred over tabs.

```
for i in range(10):
    print (i)
```

```
In [3]: for i in range(10):
   ...:     print(i)
   ...:
0
1
2
3
4
5
6
7
8
9

In [4]:
```

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable.

```
if True:
    print "Machine Learning"
    c = "AAIC"

if True: print "Machine Learning"; c = "AAIC"
```

Python Statement

Instructions that a Python interpreter can execute are called statements.

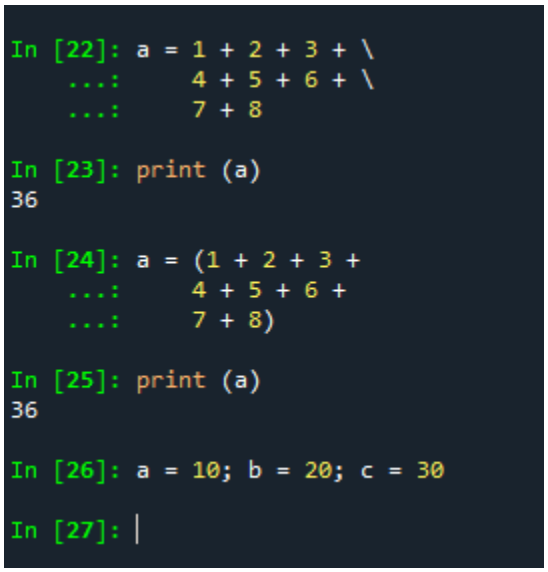
Examples:

```
a = 1  #single statement
```

Multi-Line Statement

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character ().

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8
print (a)

#another way is
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8)
print a
a = 10; b = 20; c = 30
#put multiple statements in a single line using ;
```



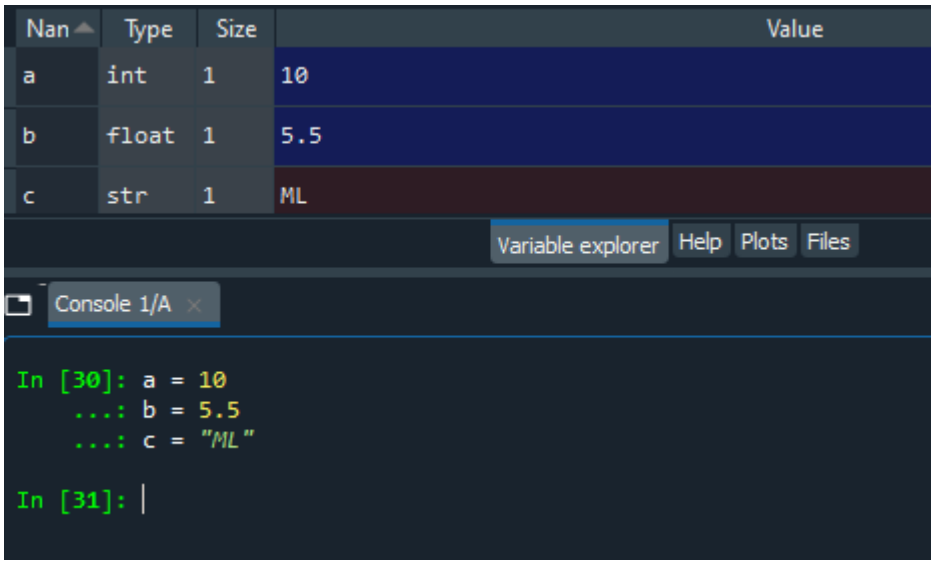
Variables

A variable is a location in memory used to store some data (value). They are given unique names to differentiate between different memory locations. The rules for writing a variable name is same as the rules for writing identifiers in Python.

We don't need to declare a variable before using it. In Python, we simply assign a value to a variable and it will exist. We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable.

Variable Assignments

```
#We use the assignment operator (=) to assign values to a variable
a = 10
b = 5.5
c = "ML"
```



Multiple Assignments

```
a, b, c = 10, 5.5, "ML"
a = b = c = "AI" #assign the same value to multiple variables at once
```

Storage Locations

```
x = 3
print(id(x))                #print address of variable x

y = 3
print(id(y))                #print address of variable y
```

Observation:
x and y points to same memory location

```
y = 2
print(id(y))                    #print address of variable y
```

```
In [32]: x = 3
    ...: print(id(x))
140710114441056

In [33]: y = 3
    ...: print(id(y))
140710114441056

In [34]: y = 2
    ...: print(id(y))
140710114441024
```

Data Types

Every value in Python has a datatype. Since everything is an object in Python programming, data types are classes and variables are instance (object) of these classes.

Numbers

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.

We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
a = 5                   #data type is implicitly set to integer
print(a, " is of type", type(a))
a = 2.5                 #data type is changed to float
print(a, " is of type", type(a))
```

```
In [1]: a = 5               #data type is implicitly set to integer
   ...: print(a, " is of type", type(a))
5  is of type <class 'int'>

In [2]: a = 2.5             #data type is changed to float
   ...: print(a, " is of type", type(a))
2.5  is of type <class 'float'>
```

```
a = 1 + 2j              #data type is changed to complex number
print(a, " is complex number?")
print(isinstance(1+2j, complex))
```

```
In [3]: a = 1 + 2j            #data type is changed to complex number
   ...: print(a, " is complex number?")
   ...: print(isinstance(1+2j, complex))
(1+2j)  is complex number?
True
```

Boolean

Boolean represents the truth values False and True

```
a = True                                #a is a boolean type

print(type(a))
```

```
In [4]: a = True                        #a is a boolean type
   ...: print(type(a))
<class 'bool'>
```

Conversion between Datatypes

```
>>> float(1)
1.0
```

Integer division

In Python2:

```
>>> 3 / 2
1
```

In Python 3:

```
>>> 3 / 2
1.5
```

To be safe: use floats:

```
>>> 3 / 2.
1.5
>>> a = 3
>>> b = 2
>>> a / b # In Python 2
1
>>> a / float(b)
1.5
```

Future behavior: to always get the behavior of Python3

```
>>> 3 / 2
1.5
```

If you explicitly want integer division use //:

```
>>> 3.0 // 2
1.0
```

The behavior of the division operator has changed in Python 3.

Lab Tasks:

1. Print your name and Reg No.
2. Take two variables and perform different mathematical operations on them and also use type
3. Write a Python program to print the following string in a specific format (see the output).
   Sample String : "Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are"

   Output :

Twinkle, twinkle, little star,
How I wonder what you are!
Up above the world so high,
Like a diamond in the sky.
Twinkle, twinkle, little star,
How I wonder what you are

4. Write a Python program to get the Python version you are using
5. Write a Python program to display the current date and time.
   Sample Output :
   Current date and time :
   2014-07-05 14:34:14
6. 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
7. Write a Python program that accepts an integer (n) and computes the value of n+nn+nnn.
   Sample value of n is 5
   Expected Result : 615
8. Write a Python program which accepts the radius of a circle from the user and compute the area.
   Sample Output :
   r = 1.1
   Area = 3.8013271108436504
9. Write a Python program which accepts the user's first and last name and print them in reverse order with a space between them.
10. Write a Python program to print the calendar of a given month and year.
    Note : Use 'calendar' module.

LAB # 2: Data types, Containers, Input/output, and Operators in Python.

Python Strings

String is sequence of Unicode characters. We can use single quotes or double quotes or even triple quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """. A string in Python consists of a series or sequence of characters - letters, numbers, and special characters. Strings can be indexed - often synonymously called subscripted as well. Similar to C, the first character of a string has the index 0.

Different string syntaxes (simple, double or triple quotes):

```
>>>s = 'Hello, how are you?'
s = "Hi, what's up"
>>>s = '''Hello, # tripling the quotes allows the
how are you''' # string to span more than one line
>>>s = """Hi,
what's up?"""
```

```
In [1]: s = 'Hello, how are you?'

In [2]: s = "Hi, what's up"

In [3]: s = '''Hello, # tripling the quotes allows the
   ...: how are you''' # string to span more than one line

In [4]: s = """Hi,
   ...: what's up?"""

In [5]: print(s)
Hi,
what's up?

In [6]: print(type(s))
<class 'str'>

In [7]:
```

The newline character is \n, and the tab character is \t. Strings are collections like lists. Hence they can be indexed and sliced, using the same syntax and rules.

Indexing:

```
>>>a = "hello"
>>> a[0]
>>> a[1]
```

```
>>> a[-1]
```

(Remember that negative indices correspond to counting from the right end.)

```
In [8]: a = "hello"

In [9]: a[0]
Out[9]: 'h'

In [10]: a[1]
Out[10]: 'e'

In [11]: a[-1]
Out[11]: 'o'
```

```
In [17]: print(a[len(a)-1])
o
```

Slicing:

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
>>> a[2:10:2] # Syntax: a[start:stop:step]
>>> a[::3] # every three characters, from beginning to end
```

```
In [20]: a = "hello, world!"

In [21]: a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
Out[21]: 'lo,'

In [22]: a[2:10:2] # Syntax: a[start:stop:step]
Out[22]: 'lo o'

In [23]: a[::3] # every three characters, from beginning to end
Out[23]: 'hl r!'
```

Accents and special characters can also be handled in Unicode strings. A string is an immutable object and it is not possible to modify its contents. One may however create new strings from the original one.

```
>>>a = "hello, world!"
>>>a[2] = 'z'
>>>a.replace('l', 'z', 1)
>>>a.replace('l', 'z')
```

```
In [28]: a = "hello, world!"

In [29]: print(a)
hello, world!

In [30]: a[2] = 'z'
Traceback (most recent call last):

  File "<ipython-input-30-1252fe4739cb>", line 1, in <module>
    a[2] = 'z'

TypeError: 'str' object does not support item assignment


In [31]: a.replace('l', 'z', 1)
Out[31]: 'hezlo, world!'

In [32]: a.replace('l', 'z')
Out[32]: 'hezzo, worzd!'
```

Strings have many useful methods, such as a.replace as seen above. Remember the a.object-oriented notation and use tab completion or help(str) to search for new methods. See also: Python offers advanced possibilities for manipulating strings, looking for patterns or formatting. The interested reader is referred to https://docs.python.org/library/stdtypes.html#stringmethodsand https://docs.python.org/library/string.html#new-string-formatting

String formatting:

```
>>> 'An integer: %i; a float: %f; another string: %s' % (1,
0.1, 'string')
>>> i = 102
>>> filename = 'processing_of_dataset_%d.txt' % i
```

```
In [35]: 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string')
Out[35]: 'An integer: 1; a float: 0.100000; another string: string'

In [36]: i = 102

In [37]: filename = 'processing_of_dataset_%d.txt' % i

In [38]: print(filename)
processing_of_dataset_102.txt
```

Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type. Declaring a list is , Items separated by commas are enclosed within brackets [ ].

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> type(colors)
```

```
In [41]: colors = ['red', 'blue', 'green', 'black', 'white']

In [42]: type(colors)
Out[42]: list
```

Indexing: accessing individual objects contained in the list:

```
In [43]: colors[2]
Out[43]: 'green'

In [44]: colors[-1]
Out[44]: 'white'

In [45]: colors[-2]
Out[45]: 'black'
```

Indexing starts at 0 (as in C), not at 1 (as in Fortran or Matlab)!,

Slicing: obtaining sublists of regularly-spaced elements. Note that colors[start:stop] contains the elements with indices i such as start<= i < stop (i ranging from start to stop-1). Therefore, colors[start:stop] has (stop - start) elements. Slicing syntax: colors[start:stop:stride] All slicing parameters are optional.

```
In [47]: colors
Out[47]: ['red', 'blue', 'green', 'black', 'white']

In [48]: colors[2:4]
Out[48]: ['green', 'black']

In [49]: colors[3:]
Out[49]: ['black', 'white']

In [50]: colors[:3]
Out[50]: ['red', 'blue', 'green']

In [51]: colors[::2]
Out[51]: ['red', 'green', 'white']
```

Lists are mutable objects and can be modified:

```
In [53]: colors[0] = 'yellow'

In [54]: colors
Out[54]: ['yellow', 'blue', 'green', 'black', 'white']

In [55]: colors[2:4] = ['gray', 'purple']

In [56]: colors
Out[56]: ['yellow', 'blue', 'gray', 'purple', 'white']
```

Note: The elements of a list may have different types:

```
In [59]: colors = [3, -200, 'hello']

In [60]: colors
Out[60]: [3, -200, 'hello']

In [61]: colors[1], colors[2]
Out[61]: (-200, 'hello')
```

Add and remove elements:

```
In [63]: colors = ['red', 'blue', 'green', 'black', 'white']

In [64]: colors.append('pink')

In [65]: colors
Out[65]: ['red', 'blue', 'green', 'black', 'white', 'pink']

In [66]: colors.pop() # removes and returns the last item
Out[66]: 'pink'

In [67]: colors
Out[67]: ['red', 'blue', 'green', 'black', 'white']

In [68]: colors.extend(['pink', 'purple']) # extend colors, in-place

In [69]: colors
Out[69]: ['red', 'blue', 'green', 'black', 'white', 'pink', 'purple']

In [70]: colors = colors[:-2]

In [71]: colors
Out[71]: ['red', 'blue', 'green', 'black', 'white']
```

Reverse:

```
In [79]: colors = ['red', 'blue', 'green', 'black', 'white']

In [80]: rcolors = colors[::-1]

In [81]: rcolors
Out[81]: ['white', 'black', 'green', 'blue', 'red']

In [82]: rcolors2 = list(colors)

In [83]: rcolors2
Out[83]: ['red', 'blue', 'green', 'black', 'white']

In [84]: rcolors2.reverse() # in-place

In [85]: rcolors2
Out[85]: ['white', 'black', 'green', 'blue', 'red']
```

Concatenate and repeat lists:

```
In [87]: rcolors + colors
Out[87]:
['white',
 'black',
 'green',
 'blue',
 'red',
 'red',
 'blue',
 'green',
 'black',
 'white']

In [88]: rcolors * 2
Out[88]:
['white',
 'black',
 'green',
 'blue',
 'red',
 'white',
 'black',
 'green',
 'blue',
 'red']
```

Sort:

```
In [90]: sorted(rcolors) # new object
Out[90]: ['black', 'blue', 'green', 'red', 'white']

In [91]: rcolors
Out[91]: ['white', 'black', 'green', 'blue', 'red']

In [92]: rcolors.sort() # in-place

In [93]: rcolors
Out[93]: ['black', 'blue', 'green', 'red', 'white']
```

Methods and Object-Oriented Programming

The notation rcolors.method() (e.g. rcolors.append(3) and colors.pop()) is our first example of object- oriented programming (OOP). Being a list, the object *rcolors* owns the *method function* that is called using the notation.

```
>>>rcolors.<tab> # views the list of OOP function that can be performed
on this object.
```

Python Tuple

Tuples are basically immutable lists. The elements of a tuple are written between parentheses, or just separated by commas

```
In [1]: t = 12345, 54321, 'hello!'

In [2]: t[0]
Out[2]: 12345

In [3]: t[0]=321
Traceback (most recent call last):

  File "<ipython-input-3-fd234f9da6c0>", line 1, in <module>
    t[0]=321

TypeError: 'tuple' object does not support item assignment
```

Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }.

Items in a set are not ordered.

```
In [7]: s = set(('a', 'b', 'c', 'a'))

In [8]: a={10,20,30,40,50}

In [9]: type(s)
Out[9]: set

In [10]: type(a)
Out[10]: set

In [11]: s = {10, 20, 20, 30, 30, 30}
    ...: print(s)                    #automatically set won't consider duplicate elements
{10, 20, 30}

In [12]: print(s[1]) #we can't print particular element in set because
    ...:             #it's unorder collections of items
Traceback (most recent call last):

  File "<ipython-input-12-3e2f312e6983>", line 1, in <module>
    print(s[1]) #we can't print particular element in set because

TypeError: 'set' object is not subscriptable
```

Python Dictionary

Dictionary is an unordered collection of key-value pairs. In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
In [16]: tel = {'emmanuelle': 5752, 'sebastian': 5578}

In [17]: tel['francis'] = 5915 # add an entry

In [18]: tel
Out[18]: {'emmanuelle': 5752, 'sebastian': 5578, 'francis': 5915}

In [19]: tel['sebastian']
Out[19]: 5578

In [20]: tel.keys()
Out[20]: dict_keys(['emmanuelle', 'sebastian', 'francis'])

In [21]: tel.values()
Out[21]: dict_values([5752, 5578, 5915])

In [22]: 'francis' in tel
Out[22]: True

In [23]: tel['hammer']
Traceback (most recent call last):

  File "<ipython-input-23-a6171a5b2f96>", line 1, in <module>
    tel['hammer']
```

Conversion between Datatypes

We can convert between different data types by using different type conversion functions like int(), float(), str() etc.

```
>>>float(5)      #convert interger to float using float() method
Out[25]: 5.0
>>>int(100.5)    #convert float to integer using int() method
Out[26]: 100
>>>str(20)       #convert integer to string
Out[27]: '20'
>>>user = "Amir"
>>>lines = 100
>>>print("Congratulations, " + user + "! You just wrote " + str(lines) +
" lines of code" )
#remove str and gives error
```

```
 Congratulations, Amir! You just wrote 100 lines of code
 >>>a = [1, 2, 3]
 >>>print(type(a))       #type of a is list
<class 'list'>
 >>>s = set(a)           #convert list to set using set() method
 >>>print(type(s))       #now type of s is set
<class 'set'
>>>list("Hello")        #convert String to list using list() method
Out[30]: ['H', 'e', 'l', 'l', 'o']
```

Python Input and Output

Python Output

We use the print() function to output data to the standard output device

```
In [39]: print("Hello World")
Hello World

In [40]: a = 10

In [41]: print("The value of a is", a)
The value of a is 10

In [42]: a = 10; b = 20 #multiple statements in single line.
    ...:
    ...: print("The value of a is {} and b is {}".format(a, b))    #default
The value of a is 10 and b is 20

In [43]: a = 10; b = 20  #multiple statements in single line
    ...:
    ...: print("The value of b is {1} and a is {0}".format(a, b)) #specify position of
arguments
The value of b is 20 and a is 10

In [44]: #we can use keyword arguments to format the string
    ...: print("Hello {name}, {greeting}".format(name="Amir", greeting="Good Morning"))
Hello Amir, Good Morning

In [45]: #we can combine positional arguments with keyword arguments
    ...: print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred',
    ...:                                                  other='Georg'))
The story of Bill, Manfred, and Georg
```

Python Input

If we want to take the input from the user, In Python, we have the input() function to allow this

```
In [49]: num = input("Enter a number: ")

Enter a number: 10

In [50]: print num
  File "<ipython-input-50-f59b73ad6832>", line 1
    print num
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(num)?


In [51]: print (num)
10
```

Operators

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Operator Types

1. Arithmetic operators
2. Comparison (Relational) operators
3. Logical (Boolean) operators
4. Bitwise operators
5. Assignment operators
6. Special operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

+ , -, *, /, %, //, ** are arithmetic operators

```
In [61]: x, y = 10, 20

In [62]: print(x + y) # addition
30

In [63]: print(x - y) # subtraction
-10

In [64]: print(x * y) # multiplication
200

In [65]: print(x / y) # division
0.5

In [66]: print(x % y) # #modulo division (%)
10

In [67]: print(x // y) #Floor Division (//)
0

In [68]: print(x ** y) #Exponent (**)
100000000000000000000
```

Comparison Operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

>, <, ==, !=, >=, <= are comparison operators

```
In [74]: a, b = 10, 20

In [75]: print(a < b)  #check a is less than b
True

In [76]: print(a > b) #check a is greater than b
False

In [77]: print(a == b)#check a is equal to b
False

In [78]: print(a != b)#check a is not equal to b (!=)
True

In [79]: print(a >= b)#check a greater than or equal to b
False

In [80]: print(a <= b)#check a less than or equal to b
True
```

Logical Operators

Logical operators are **and, or, not** operators.

```
In [92]: a, b = True, False

In [93]: print(a and b) #print a and b
False

In [94]: print(a or b) #print a or b
True

In [95]: print(not b) #print not b
True
```

Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit

&, |, ~, ^, >>, << are Bitwise operators

```
In [106]: a, b = 10, 4
     ...: print(a & b) #Bitwise AND
     ...: print(a | b) #Bitwise OR
     ...: print(~ b)#Bitwise NOT
     ...: print(a ^ b)#Bitwise XOR
     ...: print(a >> b)#Bitwise rightshift
     ...: print(a << b)#Bitwise Leftshift
0
14
-5
14
0
160
```

Assignment operators

Assignment operators are used in Python to assign values to variables. a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

```
In [107]: a = 10
     ...:
     ...: a += 10         #add AND
     ...: print(a)
     ...: a -= 10 #subtract AND (-=)
     ...: print(a)
     ...: a *= 10 #Multiply AND (*=)
     ...: print(a)
     ...:
     ...: a /= 10 #Divide AND (/=)
     ...: print(a)
     ...:
     ...: a %= 10  #Modulus AND (%=)
     ...: print(a)
     ...:
     ...: a //= 10  #Floor Division (//=)
     ...: print(a)
     ...:
     ...: a **= 10  #Exponent AND (**=)
     ...: print(a)
20
10
100
10.0
0.0
0.0
0.0
```

Special Operators

Identity Operators

is and is not are the identity operators in Python.

They are used to check if two values (or variables) are located on the same part of the memory.

```
In [110]: a = 5
     ...: b = 5
     ...: print(a is b)    #5 is object created once both a and b points to same object
     ...: print(a is not b)  #check is not
True
False
```

Membership Operators

in and not in are the membership operators in Python.

They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

```
In [113]: lst = [1, 2, 3, 4]

In [114]: print(1 in lst)       #check 1 is present in a given list or not
True

In [115]: print(5 in lst) #check 5 is present in a given list
False

In [116]: d = {1: "a", 2: "b"}

In [117]: print(1 in d)
True
```

Lab Tasks:

1. Write a Python program to sum all the items in a list.
2. Write a Python program to get the largest number from a list.
3. Write a Python program to remove duplicates from a list
4. Write a Python program to convert list to list of dictionaries.
   Sample lists: ["Black", "Red", "Maroon", "Yellow"], ["#000000", "#FF0000", "#800000", "#FFFF00"]
   Expected Output: [{'color_name': 'Black', 'color_code': '#000000'}, {'color_name': 'Red', 'color_code': '#FF0000'}, {'color_name': 'Maroon', 'color_code': '#800000'}, {'color_name': 'Yellow', 'color_code': '#FFFF00'}]
5. Write a Python program to read a matrix from console and print the sum for each column. Accept matrix rows, columns and elements for each column separated with a space(for every row) as input from the user.
   Input rows: 2

Input columns: 2
Input number of elements in a row (1, 2, 3):
1 2
3 4
sum for each column:
4 6

6. Write a Python program to Zip two given lists of lists.
Original lists:
[[1, 3], [5, 7], [9, 11]]
[[2, 4], [6, 8], [10, 12, 14]]
Zipped list:
[[1, 3, 2, 4], [5, 7, 6, 8], [9, 11, 10, 12, 14]]

7. Write a Python program to extract the nth element from a given list of tuples.
Original list:
[('Greyson Fulton', 98, 99), ('Brady Kent', 97, 96), ('Wyatt Knott', 91, 94), ('Beau Turnbull', 94, 98)]
Extract nth element (n = 0) from the said list of tuples:
['Greyson Fulton', 'Brady Kent', 'Wyatt Knott', 'Beau Turnbull']
Extract nth element (n = 2) from the said list of tuples:
[99, 96, 94, 98]

8. Write a Python program to remove additional spaces in a given list.
Original list:
['abc ', ' ', ' ', 'sdfds ', ' ', ' ', 'sdfds ', 'huy']
Remove additional spaces from the said list:
['abc', '', '', 'sdfds', '', '', 'sdfds', 'huy']

9. Write a Python program to multiply all the items in a dictionary

10. Write a Python program to print all unique values in a dictionary.
Sample Data : [{"V":"S001"}, {"V": "S002"}, {"VI": "S001"}, {"VI": "S005"}, {"VII":"S005"}, {"V":"S009"},{"VIII":"S007"}]
Expected Output : Unique Values: {'S005', 'S002', 'S007', 'S001', 'S009'}

11. Write a Python program to create a dictionary of keys x, y, and z where each key has as value a list from 11-20, 21-30, and 31-40 respectively. Access the fifth value of each key from the dictionary.
{'x': [11, 12, 13, 14, 15, 16, 17, 18, 19],
'y': [21, 22, 23, 24, 25, 26, 27, 28, 29],
'z': [31, 32, 33, 34, 35, 36, 37, 38, 39]}
15
25
35
x has value [11, 12, 13, 14, 15, 16, 17, 18, 19]
y has value [21, 22, 23, 24, 25, 26, 27, 28, 29]
z has value [31, 32, 33, 34, 35, 36, 37, 38, 39]

12. Write a Python program to print a tuple with string formatting.

Sample tuple : (100, 200, 300)

Output : This is a tuple (100, 200, 300)

13. Write a Python program to replace last value of tuples in a list.

Sample list: [(10, 20, 40), (40, 50, 60), (70, 80, 90)]

Expected Output: [(10, 20, 100), (40, 50, 100), (70, 80, 100)]

14. Write a Python program to find the elements in a given set that are not in another set.

15. Write a Python program to check a given set has no elements in common with other given set.

16. Write a Python program that accept name of given subject and marks. Input number of subjects in first line and subject name,marks separated by a space in next line. Print subject name and marks in order of its first occurrence.

Sample Output:
Powered by
Number of subjects: 3
Input Subject name and marks: Urdu 58
Input Subject name and marks: English 62
Input Subject name and marks: Math 68
Urdu 58
English 62
Math 68

LAB # 3 Control Flow Statements and Functions in Python

Controls the order in which the code is executed.

Python if ... else Statement

The **if…elif…else** statement is used in Python for decision making.

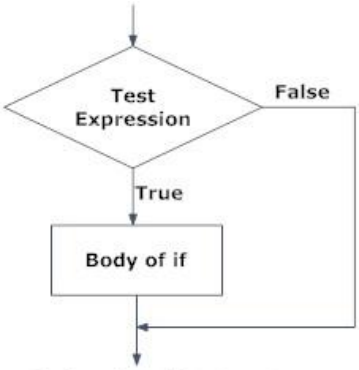


Fig: Operation of if statement

if statement syntax

if test expression:

statement(s)

The program evaluates the test expression and will execute statement(s) only if the text expression is True.

If the text expression is False, the statement(s) is not executed. Python interprets non-zero values as True. None and 0 are interpreted as False.

```
# if Example
num = 1
# try 0, -1 and None
if num>0:
    print("Number is positive")
print("This will print always")      #This print statement always print
```

```
In [1]: runfile('C:/Users/ALI/untitled0.py', wdir='C:/Users/ALI')
Number is positive
This will print always
```

if ... else Statement

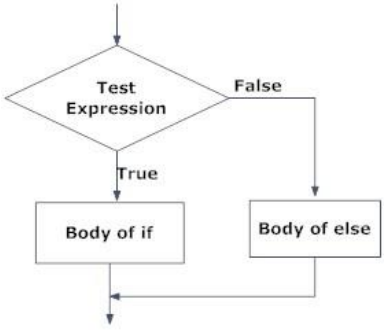


Fig: Operation of if...else statement

Syntax:

```
if test expression:
   Body of if
else:
   Body of else
```

```
# if else Exampple
num = -1
if num > 0:
    print("Positive number")
else:
    print("Negative Number")
print('statements out of the body')
```

```
Negative Number
statemnets out of the body
```

if...elif...else Statement

Syntax:

```
if test expression:
   Body of if
elif test expression:
   Body of elif
else:
   Body of else
```

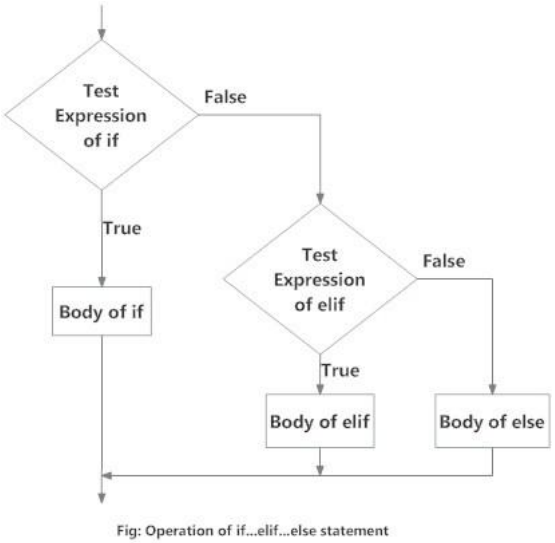


Fig: Operation of if...elif...else statement

```
# if elif statements
num = 0
if num > 0:
    print("Positive number")
elif num == 0:
    print("ZERO")
else:
    print("Negative Number")
```

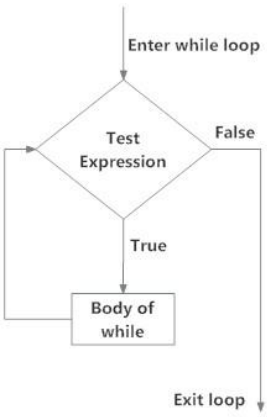
Nested if Statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming

```
# nested example
num = -12
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative Number")
```

Python while Loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.



Syntax:

```
while test_expression:
   Body of while
```

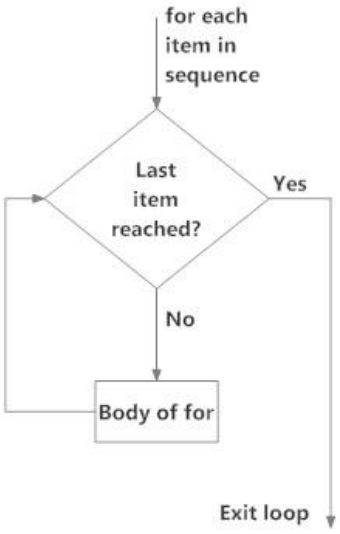
The body of the loop is entered only if the test_expression evaluates to True.

After one iteration, the test expression is checked again.

This process continues until the test_expression evaluates to False.

```
# while loop example (product of numbers in a list)
lst = [10, 20, 30, 40, 50]
product = 1
i=0
while i < len(lst):
    product=lst[i]*product # can use product *= lst[i]
    i=i+1 # can use += 1
print("Product is: {}".format(product))
```

Python for Loop



The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.

Iterating over a sequence is called traversal.

Syntax:

for element in sequence :

Body of for

Here, element is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence.

```
#for loop example
lst = [10, 20, 30, 40, 50,60]
product = 1
#iterating over the list
for num in lst:
    print(type(num))
    product *= num
print("Product is: {}".format(product))
```

range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

```
#for in range example
for i in range(0,10,1):
    print(i)
```

Python break and continue Statements

In Python, break and continue statements can alter the flow of a normal loop. Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the while loop without checking test expression. The break and continue statements are used in these cases.

```
# break example
numbers = [1, 2, 3, 4, 5,6]
for num in numbers:          #iterating over list
    if num == 4:
        break
    print(num)
else:
    print("in the else-block")
print("Outside of for loop")

# Continue example
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        continue
    print(num)
else:
    print("else-block")
```

Python Functions

Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. It avoids repetition and makes code reusable.

Syntax :

def function_name(parameters):

"""

Doc String

"""

Statement(s)

keyword "def" marks the start of function header

Parameters (arguments) through which we pass values to a function. These are optional

A colon(:) to mark the end of funciton header

Doc string describe what the function does. This is optional

"return" statement to return a value from the function. This is optional

Example:

```
def print_name(name):
    """
    This function prints the name
    """
    print("Hello " + str(name))
```

Function Call

Once we have defined a function, we can call it from anywhere

```
print_name('ALI')
```

Doc String

The first string after the function header is called the docstring and is short for documentation string. Although optional, documentation is a good programming practice, always document your code. Doc string will be written in triple quotes so that docstring can extend up to multiple lines

```
print(print_name.__doc__) # print doc string of the function
```

return Statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax:

return [expression]

-> return statement can contain an expression which gets evaluated and the value is returned.

-> if there is no expression in the statement or the return statement itself is not present inside a function,

then the function will return None Object

```
def get_sum(lst):
    """
    This function returns the sum of all the elements in a list
    """
    #initialize sum
    _sum = 0

    #iterating over the list
    for num in lst:
        _sum += num
    return _sum

s = get_sum([1, 2, 3, 4])
print(s)

#print doc string
print(get_sum.__doc__)
```

Scope and Life Time of Variables

-> Scope of a variable is the portion of a program where the variable is recognized

-> variables defined inside a function is not visible from outside. Hence, they have a local scope.

-> Lifetime of a variable is the period throughout which the variable exits in the memory.

-> The lifetime of variables inside a function is as long as the function executes.

-> Variables are destroyed once we return from the function.

Example:

```
global_var = "global variable"

def test_life_time():
    """
    This function test the life time of a variables
    """
    local_var = "local variable"
    print(local_var)       #print local variable local_var

    print(global_var)      #print global variable global_var


#calling function
test_life_time()

#print global variable global_var
print(global_var)

#print local variable local_var
print(local_var)
```

Python program to print Highest Common Factor (HCF) of two numbers

```
def computeHCF(a, b):
    """
    Computing HCF of two numbers
    """
    smaller = b if a > b else a  #consice way of writing if else
statement

    hcf = 1
    for i in range(1, smaller+1):
        if (a % i == 0) and (b % i == 0):
            hcf = i
    return hcf

num1 = 6
num2 = 36

print("H.C.F of {0} and {1} is: {2}".format(num1, num2,
computeHCF(num1, num2)))
```

Types Of Functions

Built-in Functions

User-defined Functions

Built-in Functions

1. abs()
2. all()
3. dir()
4. divmod exp: print(divmod(9, 2)) #print quotient and remainder as a tuple
5. enumerate()

User-defined Functions

Functions that we define ourselves to do certain specific task are referred as user-defined functions. If we use functions written by others in the form of library, it can be termed as library functions.

Advantages

User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.

If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.

Programmars working on large project can divide the workload by making different functions.

Example:

Python program to make a simple calculator that can add, subtract, multiply and division

```
def add(a, b):
    """
    This function adds two numbers
    """
    return a + b

def multiply(a, b):
    """
    This function multiply two numbers
    """
    return a * b

def subtract(a, b):
    """
    This function subtract two numbers
    """
    return a - b

def division(a, b):
```

```
    """
    This function divides two numbers
    """
    return a / b

print("Select Option")
print("1. Addition")
print ("2. Subtraction")
print ("3. Multiplication")
print ("4. Division")

#take input from user
choice = int(input("Enter choice 1/2/3/4"))

num1 = float(input("Enter first number:"))
num2 = float(input("Enter second number:"))
if choice == 1:
    print("Addition of {0} and {1} is {2}".format(num1, num2, add(num1,
num2)))
elif choice == 2:
    print("Subtraction of {0} and {1} is {2}".format(num1, num2,
subtract(num1, num2)))
elif choice == 3:
    print("Multiplication of {0} and {1} is {2}".format(num1, num2,
multiply(num1, num2)))
elif choice == 4:
    print("Division of {0} and {1} is {2}".format(num1, num2,
division(num1, num2)))
else:
    print("Invalid Choice")
```

Function Arguments

```
def greet(name, msg):
    """
    This function greets to person with the provided message
    """
    print("Hello {0} , {1}".format(name, msg))
```

#call the function with arguments

```
greet("ALI", "Good Morning")
```

#suppose if we pass one argument

```
greet("ALI") #will get an error
```

Different Forms of Arguments

1. Default Arguments

We can provide a default value to an argument by using the assignment operator (=).

```
def greet(name, msg="Good Morning"):
    """
    This function greets to person with the provided message
    if message is not provided, it defaults to "Good Morning"
    """
    print("Hello {0} , {1}".format(name, msg))

greet("ALI", "Good Night")
```

#with out msg argument

```
greet("Ali")
```

Once we have a default argument, all the arguments to its right must also have default values.

```
def greet(msg="Good Morning", name)
```

will get a SyntaxError : non-default argument follows default argument

2. Keyword Arguments

kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function

Example:¶

```
def greet(**kwargs):
    """
    This function greets to person with the provided message
    """
    if kwargs:
        print("Hello {0} , {1}".format(kwargs['name'], kwargs['msg']))

greet(name="Ali", msg="Good Morning")
```

3. Arbitary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

Example:

```
def greet(*names):
    """
    This function greets all persons in the names tuple
    """
```

```
    print(names)

    for name in names:
        print("Hello,  {0} ".format(name))

greet("Ali", "Saad", "Adnan", "Zubair")
```

Recurison

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Example:

```
#python program to print factorial of a number using recurion
def factorial(num):
    """
    This is a recursive function to find the factorial of a given
number
    """
    return 1 if num == 1 else (num * factorial(num-1))

num = 5
print ("Factorial of {0} is {1}".format(num, factorial(num)))
```

Factorial of 5 is 120

Advantages

1.Recursive functions make the code look clean and elegant.

2.A complex task can be broken down into simpler sub-problems using recursion.

3.Sequence generation is easier with recursion than using some nested iteration.

Disadvantages

1.Sometimes the logic behind recursion is hard to follow through.

2.Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

3.Recursive functions are hard to debug.

Modules

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: abc.py, is called a module and its module name would be "abc".

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

How to import a module?

We use the import keyword to do this.

```
import example #imported example module
```

Using the module name we can access the function using dot (.) operation.

```
example.add(10, 20)
```

Python has a lot of standard modules available.

https://docs.python.org/3/py-modindex.html

Examples:

```
import math
print(math.pi)

import datetime
datetime.datetime.now()
```

import with renaming

```
import math as m
print(m.pi)
```

from...import statement

We can import specific names form a module without importing the module as a whole.

```
from datetime import datetime
datetime.now()
```

import all names

```
from math import *
print("Value of PI is " + str(pi))
```

dir() built in function

We can use the dir() function to find out names that are defined inside a module.

```
Import math
dir(math)
```

Lab Tasks:

1. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included)
2. Write a Python program to convert temperatures to and from Celsius, Fahrenheit.
   [Formula: c/5 = f-32/9 [where c = temperature in Celsius and f = temperature in Fahrenheit]

Expected Output :

60°C is 140 in Fahrenheit

45°F is 7 in Celsius

3. Write a Python program to construct the following pattern, using a nested for loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

4. Write a Python program to count the number of even and odd numbers from a series of numbers.

5. Write a Python program that prints each item and its corresponding type from the following list.

Sample List : datalist = [1452, 11.23, 1+2j, True, 'w3resource', (0, -1), [5, 12], {"class":'V', "section":'A'}]

6. Write a Python program to get the Fibonacci series between 0 to 50.

7. Write a Python program to check the validity of password input by users.
Validation :

a. At least 1 letter between [a-z] and 1 letter between [A-Z].

b. At least 1 number between [0-9].

c. At least 1 character from [$#@].

d. Minimum length 6 characters.

e. Maximum length 16 characters.

8. Write a Python program to check a string represent an integer or not.
Expected Output:

```
Input a string: Python
The string is not an integer.
```

9. Write a Python function to find the Max of three numbers.

10. Write a Python program to reverse a string.

Sample String : "1234abcd"

Expected Output : "dcba4321"

11. Write a Python function to check whether a number is in a given range.

12. Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.

Sample String : 'The quick Brow Fox'

Expected Output :

No. of Upper case characters : 3

No. of Lower case Characters : 12

13. Write a Python function to check whether a number is perfect or not.

According to Wikipedia : In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum). Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).

Example : The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and 1 + 2 + 3 = 6. Equivalently, the number 6 is equal to half the sum of all its positive divisors: (1 + 2 + 3 + 6) / 2 = 6. The next perfect number is 28 = 1 + 2 + 4 + 7 + 14. This is followed by the perfect numbers 496 and 8128.

14. Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is a word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

15. Write a Python program to access a function inside a function.

16. Write a recursive function to calculate the sum of numbers from 0 to 10.

LAB # 04 Introduction to IRIS data set and scatter plots.

Plotting for Exploratory data analysis (EDA)

Basic Terminology

- What is EDA?
- Data-point/vector/Observation
- Data-set.
- Feature/Variable/Input-variable/Independent-varibale
- Label/depdendent-variable/Output-varible/Class/Class-label/Response label
- Vector: 2-D, 3-D, 4-D,.... n-D

Q. What is a 1-D vector: Scalar

Iris Flower dataset

Toy Dataset: Iris Dataset: [https://en.wikipedia.org/wiki/Iris_flower_data_set]

- A simple dataset to learn the basics.
- 3 flowers of Iris species. [see images on wikipedia link above]
- 1936 by Ronald Fisher.
- Petal and Sepal: http://terpconnect.umd.edu/~petersd/666/html/iris_with_labels.jpg
- **Objective:** Classify a new flower as belonging to one of the 3 classes given the 4 features.
- Importance of domain knowledge.
- Why use petal and sepal dimensions as features?
- Why do we not use 'color' as a feature?

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

Download iris.csv from https://raw.githubusercontent.com/uiuc-cse/data-fa14/gh-pages/data/iris.csv

Load Iris.csv into a pandas data Frame.

```
iris = pd.read_csv("iris.csv")
```

how many data-points and features?

```
print (iris.shape)
(150, 5)
```

What are the column names in our dataset?

```
print (iris.columns)
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
       'species'],
      dtype='object')
```

How many data points for each class are present?

How many flowers for each species are present?

```
iris["species"].value_counts()
# balanced-dataset vs imbalanced datasets
#Iris is a balanced dataset as the number of data points for every class is 50.
```

```
virginica     50
versicolor    50
setosa        50
Name: species, dtype: int64
```

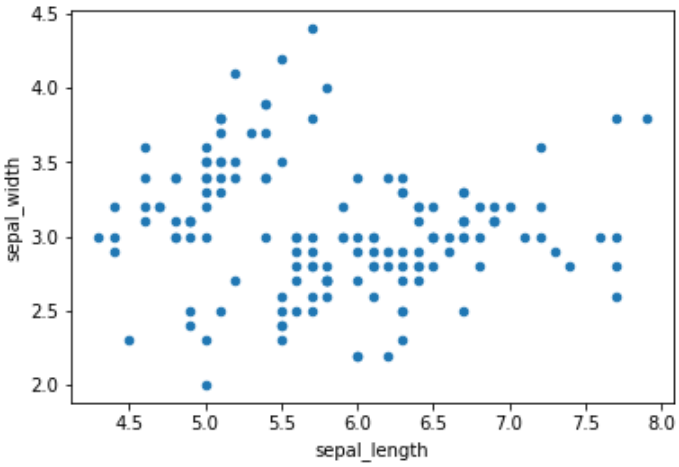
2-D Scatter Plot

ALWAYS understand the axis: labels and scale.

```
iris.plot(kind='scatter', x='sepal_length', y='sepal_width') ;
plt.show()
```

cannot make much sense out it.

What if we colour the points by their class-label/flower-type.



2-D Scatter plot with color-coding for each flower type/class.

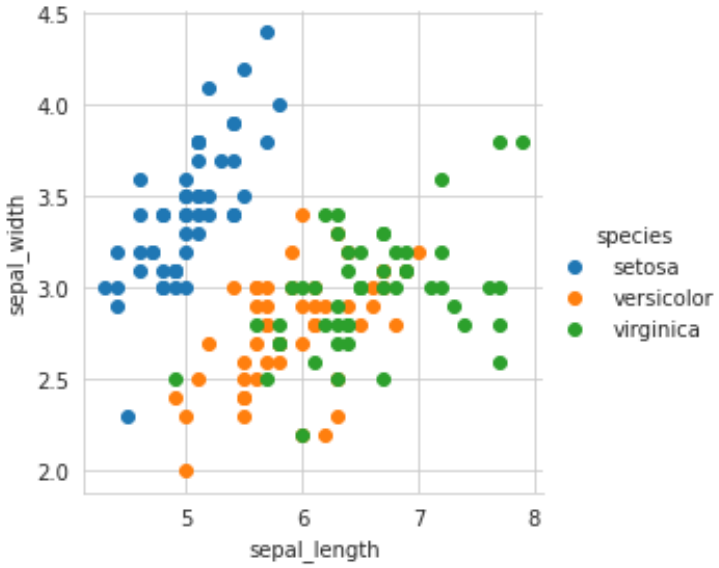
Here 'sns' corresponds to seaborn.

```
sns.set_style("whitegrid");
sns.FacetGrid(iris, hue="species", size=4) \
   .map(plt.scatter, "sepal_length", "sepal_width") \
   .add_legend();
plt.show();
```

Notice that the blue points can be easily separated from red and green by drawing a line. But red and green data points cannot be easily separated.

Can we draw multiple 2-D scatter plots for each combination of features?

How many combinations exist? 4C2 = 6.



Observation(s):

Using sepal_length and sepal_width features, we can distinguish Setosa flowers from others.

Seperating Versicolor from Viginica is much harder as they have considerable overlap.

3D Scatter plot

https://plot.ly/pandas/3d-scatter-plots/
Needs a lot to mouse interaction to interpret data.
What about 4-D, 5-D or n-D scatter plot?

Pair-plot

Pairwise scatter plot
Dis-advantages:
Can be used when number of features are high.
Cannot visualize higher dimensional patterns in 3-D and 4-D.
Only possible to view 2D patterns.

```
plt.close();
sns.set_style("whitegrid");
sns.pairplot(iris, hue="species", size=3);
plt.show()
```

Observations

petal_length and petal_width are the most useful features to identify various flower types.

While Setosa can be easily identified (linearly seperable), Virnica and Versicolor have some overlap (almost linearly seperable).

We can find "lines" and "if-else" conditions to build a simple model to classify the flower types.

Histogram, PDF, CDF

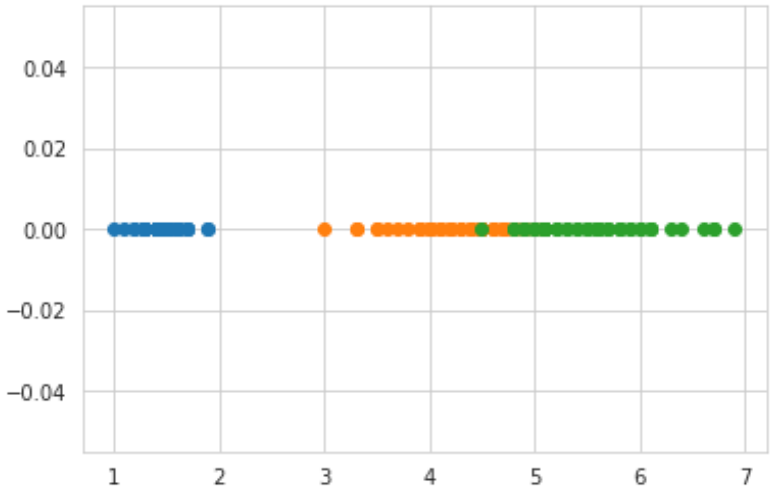
What about 1-D scatter plot using just one feature?

1-D scatter plot of petal-length

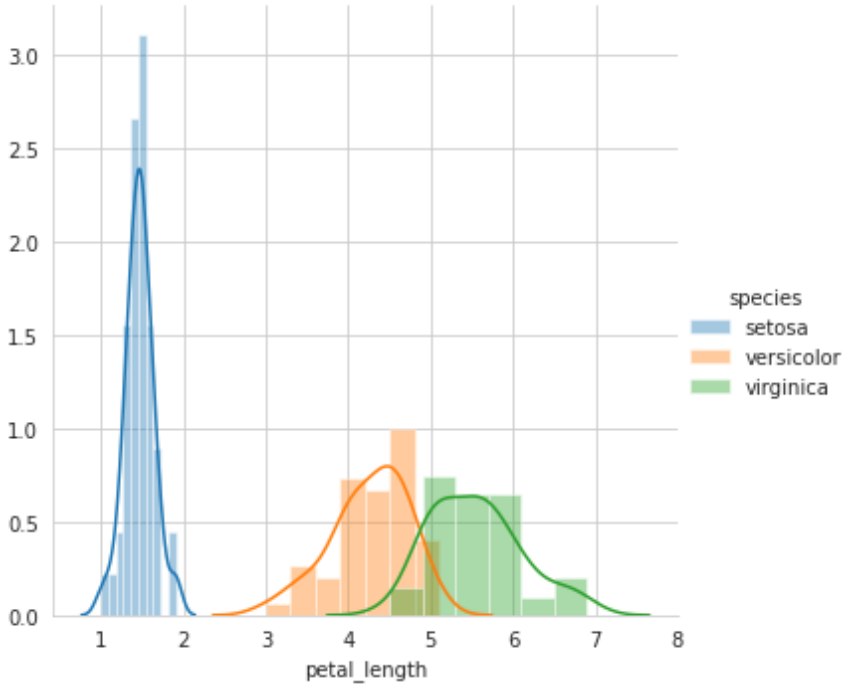
```
import numpy as np
iris_setosa = iris.loc[iris["species"] == "setosa"];
iris_virginica = iris.loc[iris["species"] == "virginica"];
iris_versicolor = iris.loc[iris["species"] == "versicolor"];
#print(iris_setosa["petal_length"])
plt.plot(iris_setosa["petal_length"], np.zeros_like(iris_setosa['petal_length']), 'o')
plt.plot(iris_versicolor["petal_length"],
np.zeros_like(iris_versicolor['petal_length']), 'o')
plt.plot(iris_virginica["petal_length"],
np.zeros_like(iris_virginica['petal_length']), 'o')

plt.show()
```

Disadvantages of 1-D scatter plot: Very hard to make sense as points are overlapping a lot.

Are there better ways of visualizing 1-D scatter plots?




```
sns.FacetGrid(iris, hue="species", size=5) \
   .map(sns.distplot, "petal_length") \
   .add_legend();
plt.show();
```




```
sns.FacetGrid(iris, hue="species", size=5) \
   .map(sns.distplot, "petal_width") \
   .add_legend();
plt.show();

sns.FacetGrid(iris, hue="species", size=5) \
   .map(sns.distplot, "sepal_length") \
   .add_legend();
plt.show();


sns.FacetGrid(iris, hue="species", size=5) \
   .map(sns.distplot, "sepal_width") \
   .add_legend();
plt.show();
```

Histograms and Probability Density Functions (PDF)

How to compute PDFs using counts/frequencies of data points in each window.

How window width effects the PDF plot.

Interpreting a PDF:

why is it called a density plot?

Why is it called a probability plot?

for each value of petal_length, what does the value on y-axis mean?

Notice that we can write a simple if..else condition as if(petal_length) < 2.5 then flower type is setosa.

Using just one feature, we can build a simple "model" suing if..else... statements.

Disadv of PDF: Can we say what percentage of versicolor points have a petal_length of less than 5?

Do some of these plots look like a bell-curve you studied in under-grad?

Gaussian/Normal distribution.

What is "normal" about normal distribution?

e.g: Hieghts of male students in a class.

One of the most frequent distributions in nature.

Need for Cumulative Distribution Function (CDF)

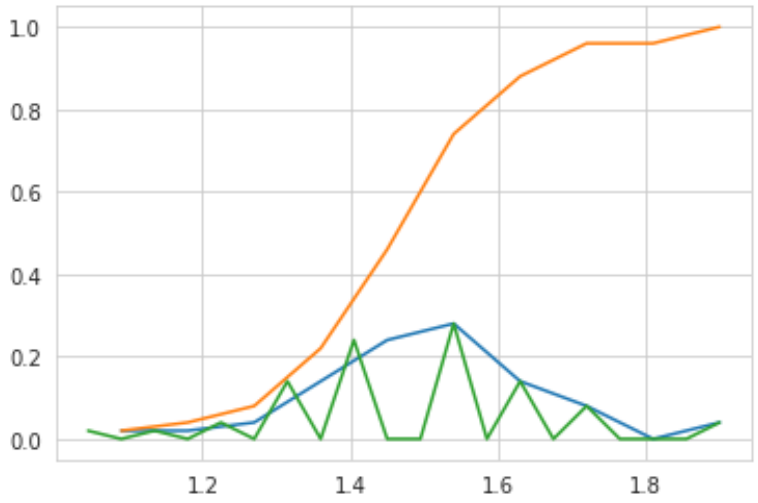
We can visually see what percentage of versicolor flowers have a petal_length of less than 5?

How to construct a CDF?

How to read a CDF?

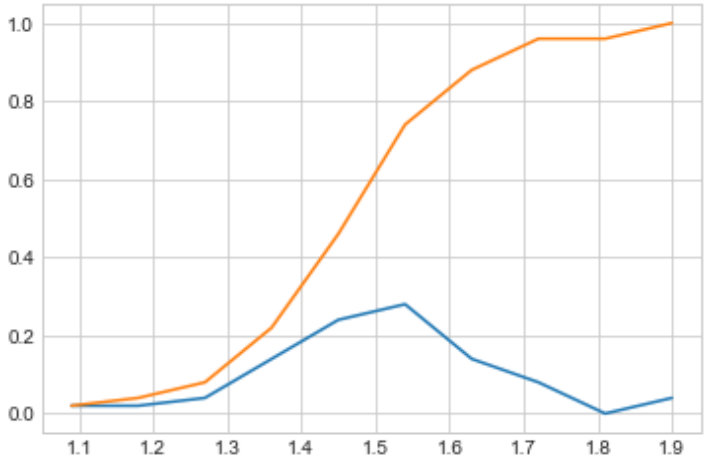
Plot CDF of petal_length

```
counts, bin_edges = np.histogram(iris_setosa['petal_length'], bins=10,
                                 density = True)
pdf = counts/(sum(counts))
print(pdf);
print(bin_edges);
cdf = np.cumsum(pdf)
plt.plot(bin_edges[1:],pdf);
plt.plot(bin_edges[1:], cdf)


counts, bin_edges = np.histogram(iris_setosa['petal_length'], bins=20,
                                 density = True)
pdf = counts/(sum(counts))
plt.plot(bin_edges[1:],pdf);

plt.show();
```



Need for Cumulative Distribution Function (CDF)
We can visually see what percentage of versicolor flowers have a
petal_length of less than 1.6?
How to construct a CDF?
How to read a CDF?
Plot CDF of petal_length

```
counts, bin_edges = np.histogram(iris_setosa['petal_length'], bins=10,
                                 density = True)
pdf = counts/(sum(counts))
print(pdf);
print(bin_edges)
#compute CDF
cdf = np.cumsum(pdf)
plt.plot(bin_edges[1:],pdf)
plt.plot(bin_edges[1:], cdf)
plt.show();
```

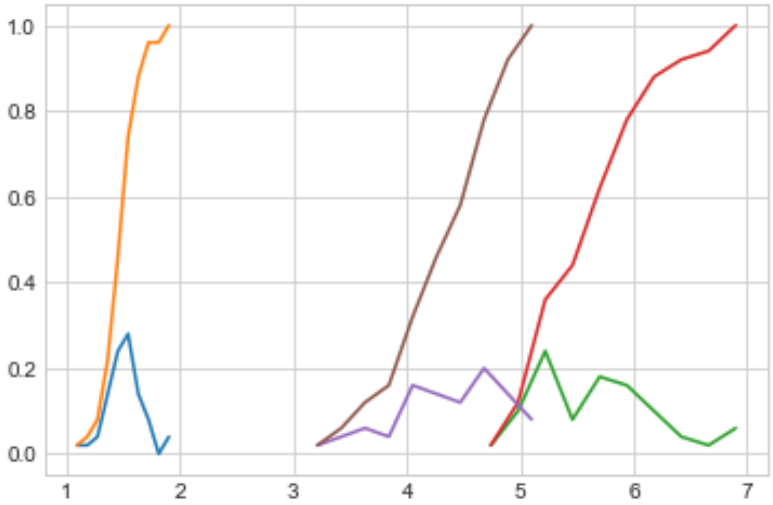


Plots of CDF of petal_length for various types of flowers.

Misclassification error if you use petal_length only.

```
counts, bin_edges = np.histogram(iris_setosa['petal_length'], bins=10,density = True)
pdf = counts/(sum(counts))
print(pdf);
print(bin_edges)
cdf = np.cumsum(pdf)
plt.plot(bin_edges[1:],pdf)
plt.plot(bin_edges[1:], cdf)

# virginica
counts, bin_edges = np.histogram(iris_virginica['petal_length'], bins=10, density =
True)
pdf = counts/(sum(counts))
print(pdf);
print(bin_edges)
cdf = np.cumsum(pdf)
plt.plot(bin_edges[1:],pdf)
plt.plot(bin_edges[1:], cdf)

#versicolor
```

```
counts, bin_edges = np.histogram(iris_versicolor['petal_length'], bins=10, density =
True)
pdf = counts/(sum(counts))
print(pdf);
print(bin_edges)
cdf = np.cumsum(pdf)
plt.plot(bin_edges[1:],pdf)
plt.plot(bin_edges[1:], cdf)


plt.show();
```



Mean, Variance and Std-dev

Mean, Variance, Std-deviation,

```
print("Means:")
print(np.mean(iris_setosa["petal_length"]))
```

Mean with an outlier.

```
print(np.mean(np.append(iris_setosa["petal_length"],50)));
print(np.mean(iris_virginica["petal_length"]))
print(np.mean(iris_versicolor["petal_length"]))

print("\nStd-dev:");
print(np.std(iris_setosa["petal_length"]))
print(np.std(iris_virginica["petal_length"]))
print(np.std(iris_versicolor["petal_length"]))
```

Median, Percentile, Quantile, IQR, MAD

```
#Median, Quantiles, Percentiles, IQR.
print("\nMedians:")
print(np.median(iris_setosa["petal_length"]))

#Median with an outlier
print(np.median(np.append(iris_setosa["petal_length"],50)));
print(np.median(iris_virginica["petal_length"]))
print(np.median(iris_versicolor["petal_length"]))
```

```
print("\nQuantiles:")
print(np.percentile(iris_setosa["petal_length"],np.arange(0, 100, 25)))
print(np.percentile(iris_virginica["petal_length"],np.arange(0, 100, 25)))
print(np.percentile(iris_versicolor["petal_length"], np.arange(0, 100, 25)))

print("\n90th Percentiles:")
print(np.percentile(iris_setosa["petal_length"],90))
print(np.percentile(iris_virginica["petal_length"],90))
print(np.percentile(iris_versicolor["petal_length"], 90))

from statsmodels import robust

print ("\nMedian Absolute Deviation")
print(robust.mad(iris_setosa["petal_length"]))
print(robust.mad(iris_virginica["petal_length"]))
print(robust.mad(iris_versicolor["petal_length"]))
```

Box plot and Whiskers

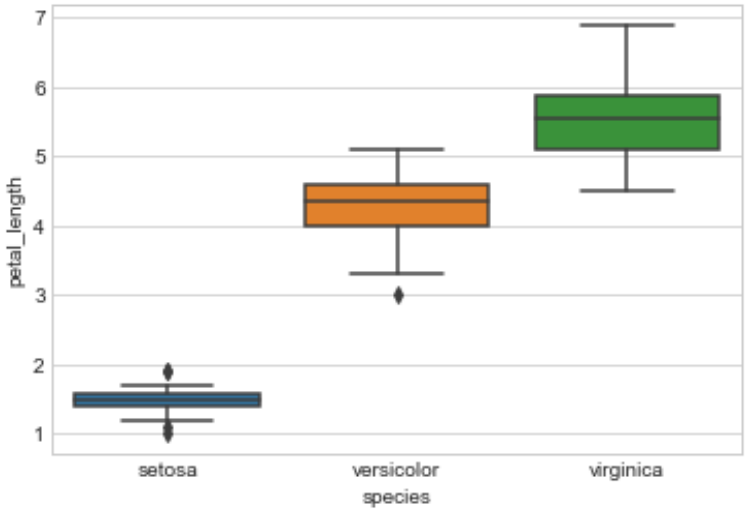
- Box-plot with whiskers: another method of visualizing the 1-D scatter plot more intuitivey.
- The Concept of median, percentile, quantile.
- How to draw whiskers: [no standard way] Could use min and max or use other complex statistical techniques.
- IQR like idea.

NOTE: IN the plot below, a technique call inter-quartile range is used in plotting the whiskers.

Whiskers in the plot below donot correposnd to the min and max values.

Box-plot can be visualized as a PDF on the side-ways.

```
sns.boxplot(x='species',y='petal_length', data=iris)
plt.show()
```

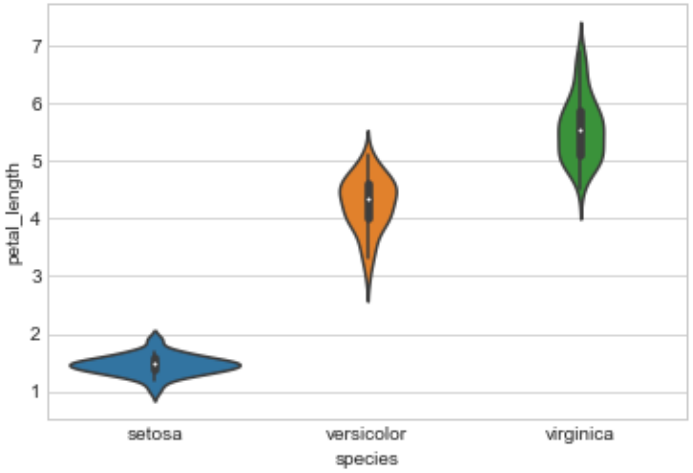


Violin plots

A violin plot combines the benefits of the previous two plots and simplifies them

Denser regions of the data are fatter, and sparser ones thinner in a violin plot

```
sns.violinplot(x="species", y="petal_length", data=iris, size=8)
plt.show()
```



Summarizing plots in english

Exaplain your findings/conclusions in plain english

Never forget your objective (the probelm you are solving).

Perform all of your EDA aligned with your objectives.

Univariate, bivariate and multivariate analysis.

```
Def: Univariate, Bivariate and Multivariate analysis.
  File "<ipython-input-20-f25211abae88>", line 3
    Def: Univariate, Bivariate and Multivariate analysis.
                   ^
SyntaxError: invalid syntax
```

Multivariate probability density, contour plot.

```
#2D Density plot, contors-plot
sns.jointplot(x="petal_length", y="petal_width", data=iris_setosa, kind="kde");
plt.show();
```

Lab Tasks:

Download Haberman Cancer Survival dataset from Kaggle. You may have to create a Kaggle account to download data. (https://www.kaggle.com/gilsousa/habermans-survival-data-set)

Perform a similar analaysis as above on this dataset with the following sections:

High level statistics of the dataset: number of points, number of features, number of classes, data-points per class.

Explain our objective.

Perform Univaraite analysis(PDF, CDF, Boxplot, Voilin plots) to understand which features are useful towards classification.

Perform Bi-variate analysis (scatter plots, pair-plots) to see if combinations of features are useful in classfication.

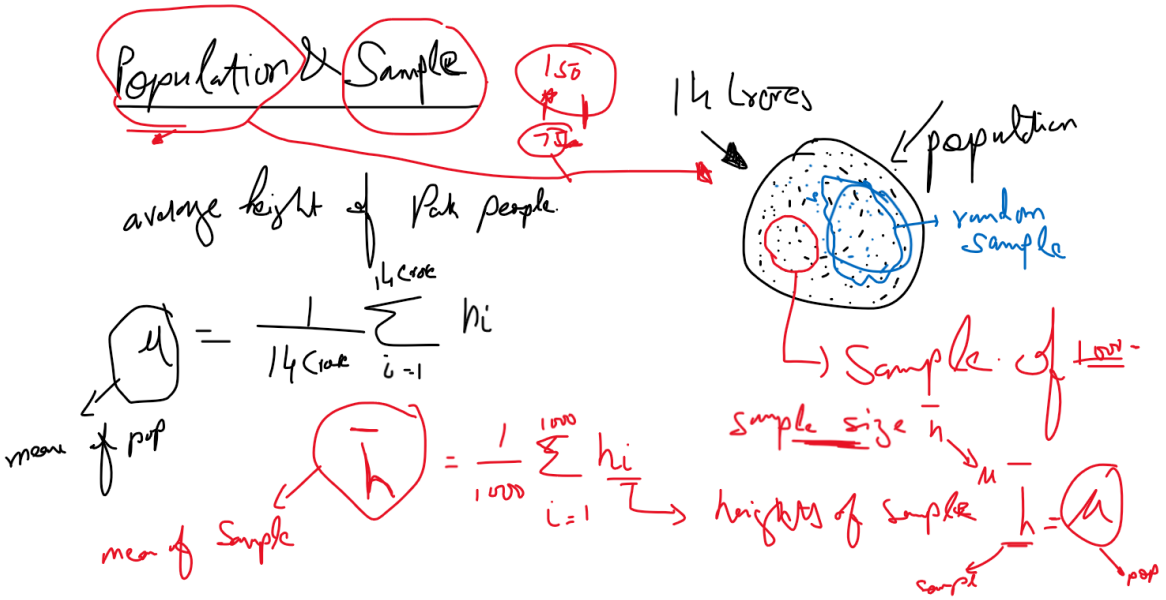
Write your observations in english as crisply and unambigously as possible. Always quantify your results.

```
iris_virginica_SW = iris_virginica.iloc[:,1]
iris_versicolor_SW = iris_versicolor.iloc[:,1]

from scipy import stats
stats.ks_2samp(iris_virginica_SW, iris_versicolor_SW)

x = stats.norm.rvs(loc=0.2, size=10)
stats.kstest(x,'norm')

x = stats.norm.rvs(loc=0.2, size=100)
stats.kstest(x,'norm')

x = stats.norm.rvs(loc=0.2, size=1000)
stats.kstest(x,'norm')
```

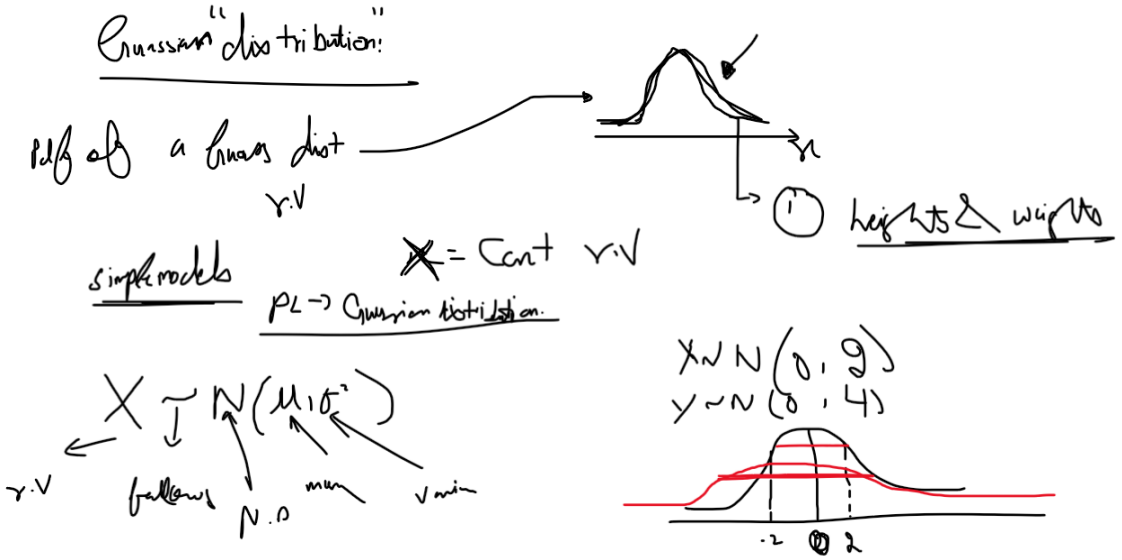
LAB # 05: Probability and Statistics Operation in Python.

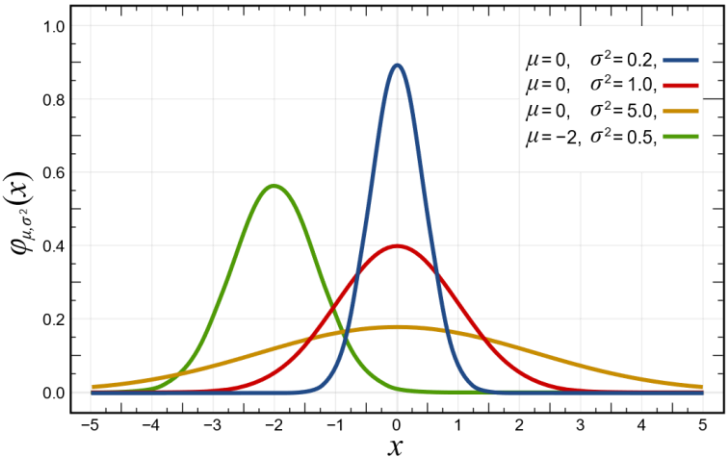
Probability and Statistics

- Fundamental Area
- Super Important in Machine Learning.
- Histogram, PDF, CDF, mean, var, Std-dev.
- Random Variable.
  - Can take any value from the result of a random experiment.
  - Discrete Random Variable
  - Continuous Random Variable.
- Outlier.
- Population and Sample.

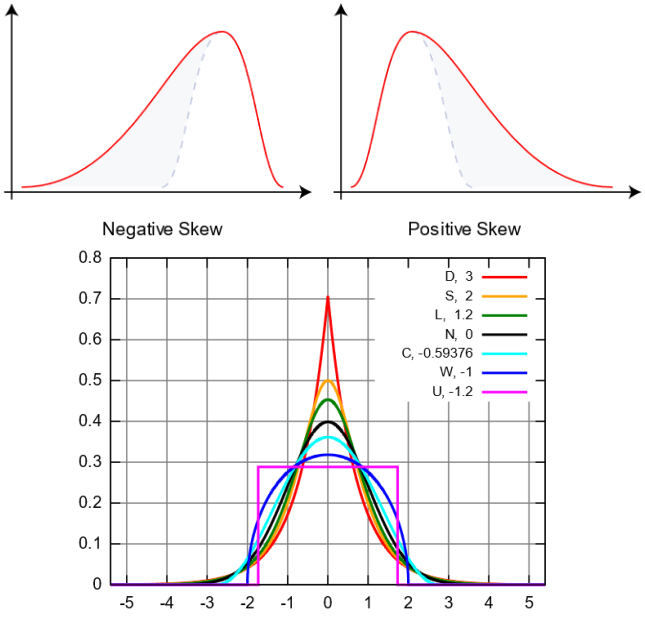


- Gaussian/Normal Distribution.

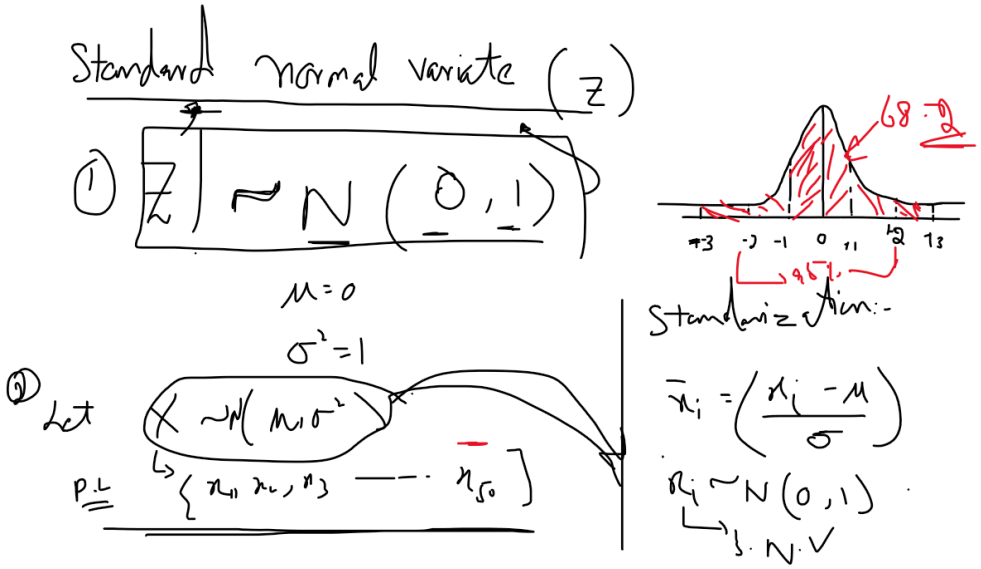




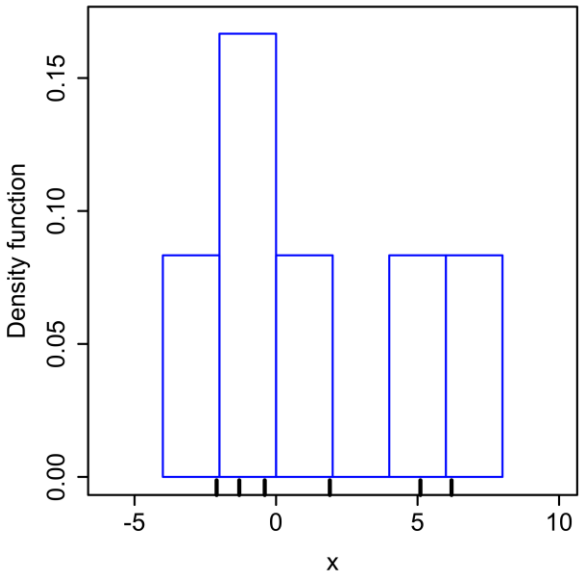
- Skewness and kurtosis.

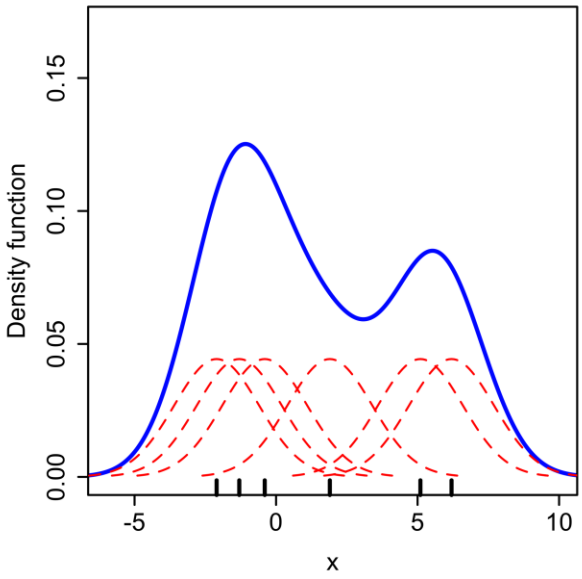


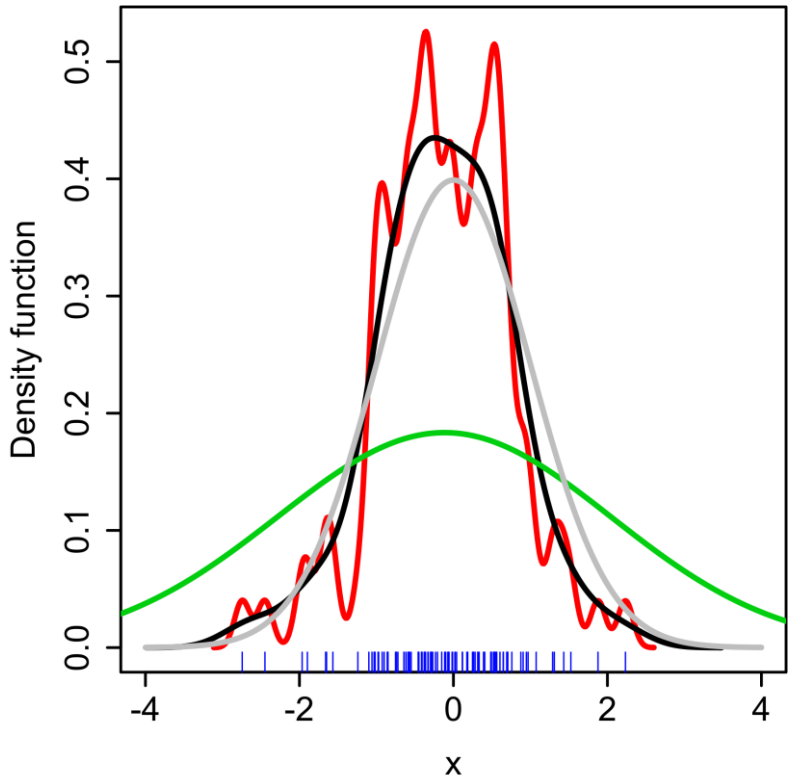
- Standard Normal Variate.



- Kernel Density Estimate.

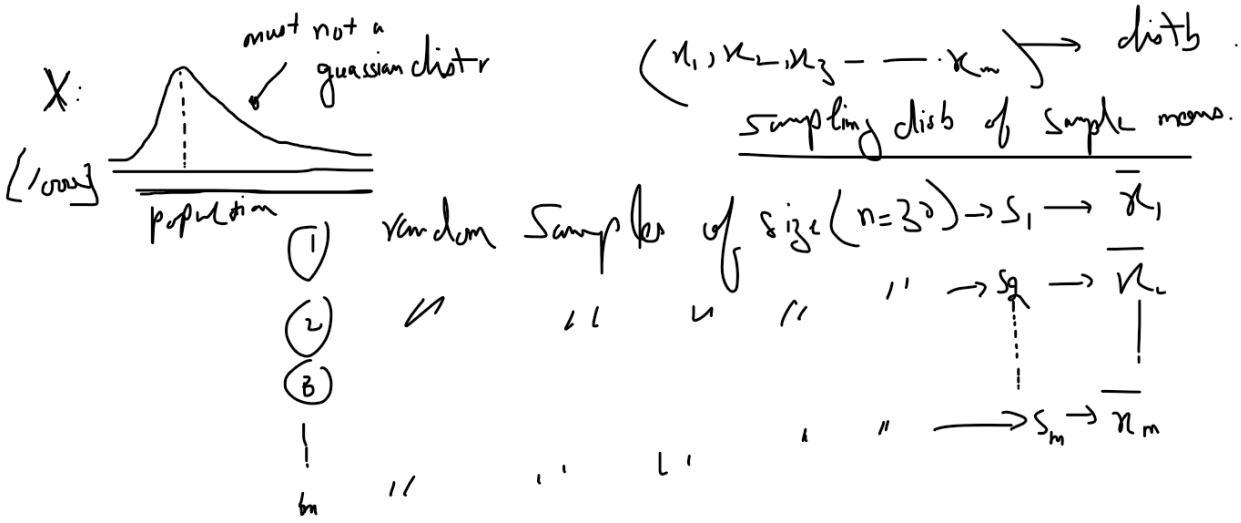


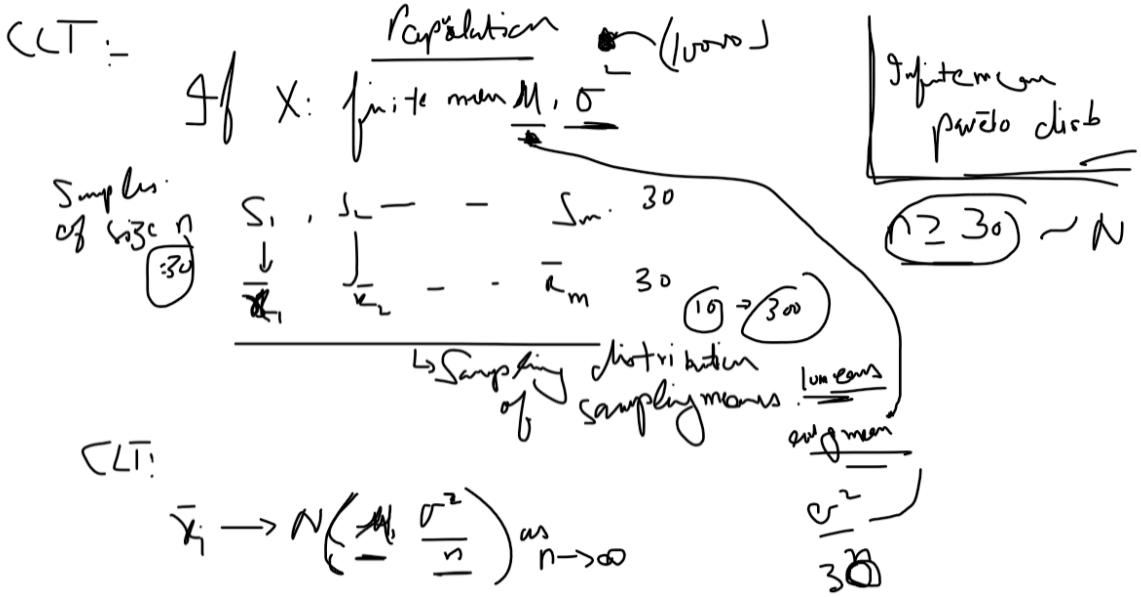




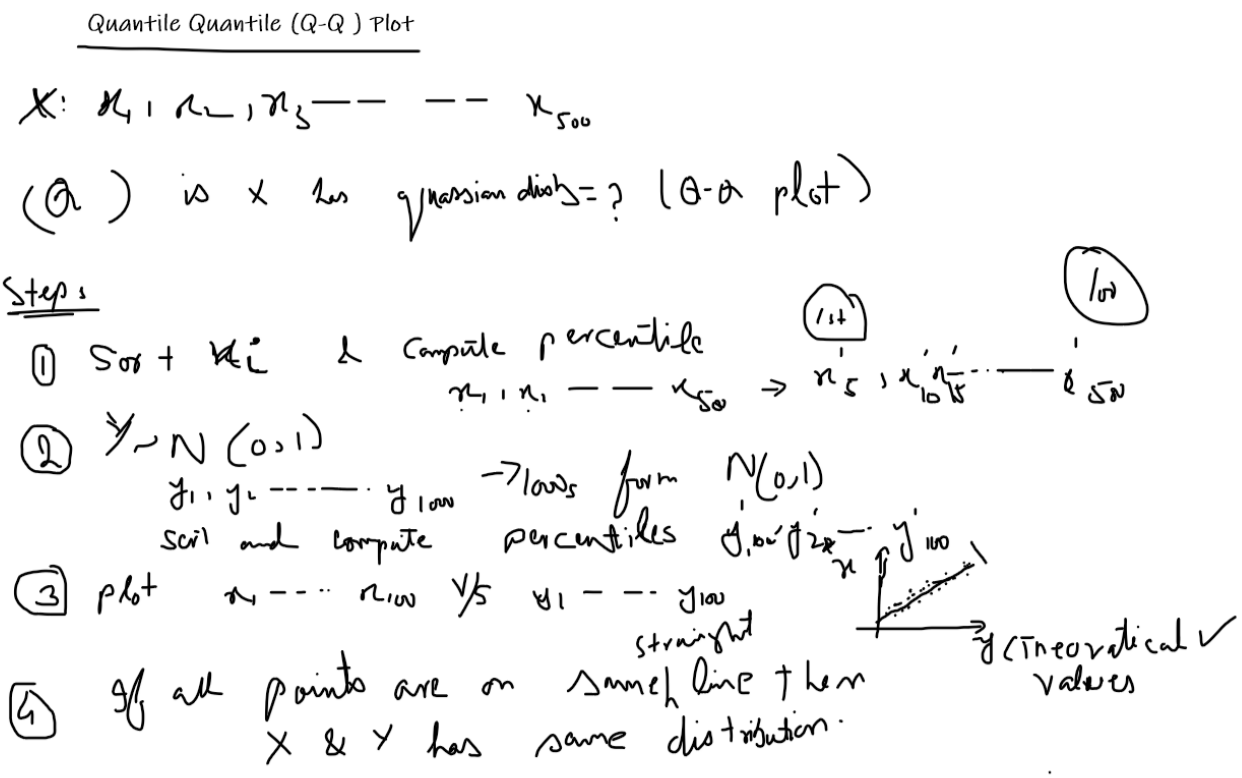
- Sampling Distribution and Central Limit Theorem

Sampling distribution & CLT Central limit Theorem





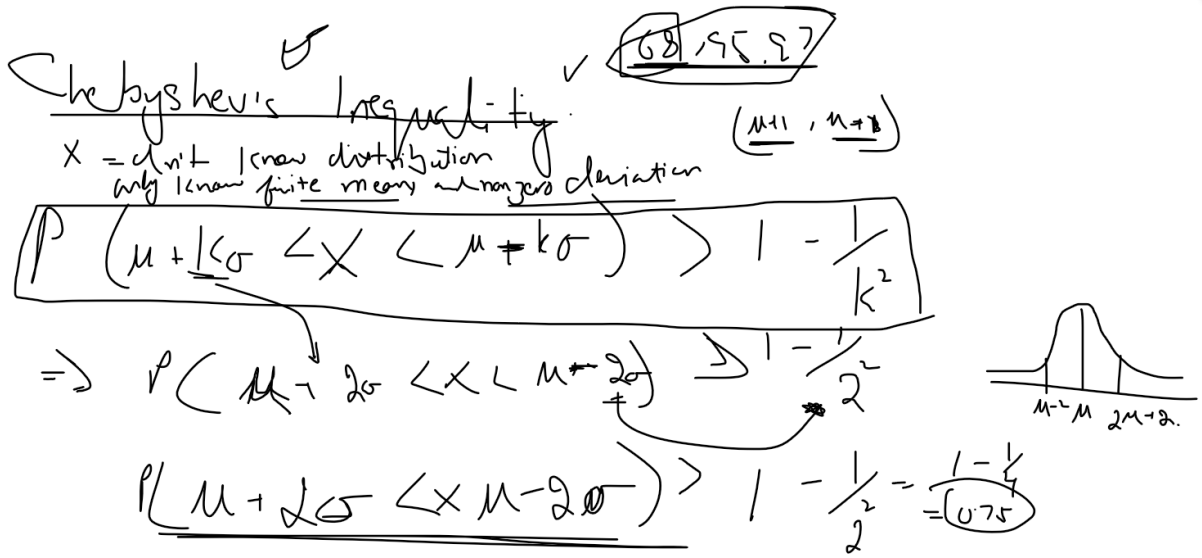
Quantile-Quantile (Q-Q) Plot:



Code :-

```
import numpy as np
import pylab
import scipy.stats as stats

# N(0,1)
std_normal = np.random.normal(loc = 0, scale = 1, size=100)
# 0 to 100th percentiles of std-normal
for i in range(0,101):
    print(i, np.percentile(std_normal,i))

#generate 100 sanples from N(20,5)
measurements = np.random.normal(loc = 20, scale = 5, size=1000000)
#try size=1000
stats.probplot(measurements, dist="norm", plot=pylab)
pylab.show()
# generate 100 sanples from N(20,5)
measurements = np.random.uniform(low=-1, high=1, size=10000)
#try size=10000
stats.probplot(measurements, dist="norm", plot=pylab)
pylab.show()
```

- **Chebyshev's Inequality.**

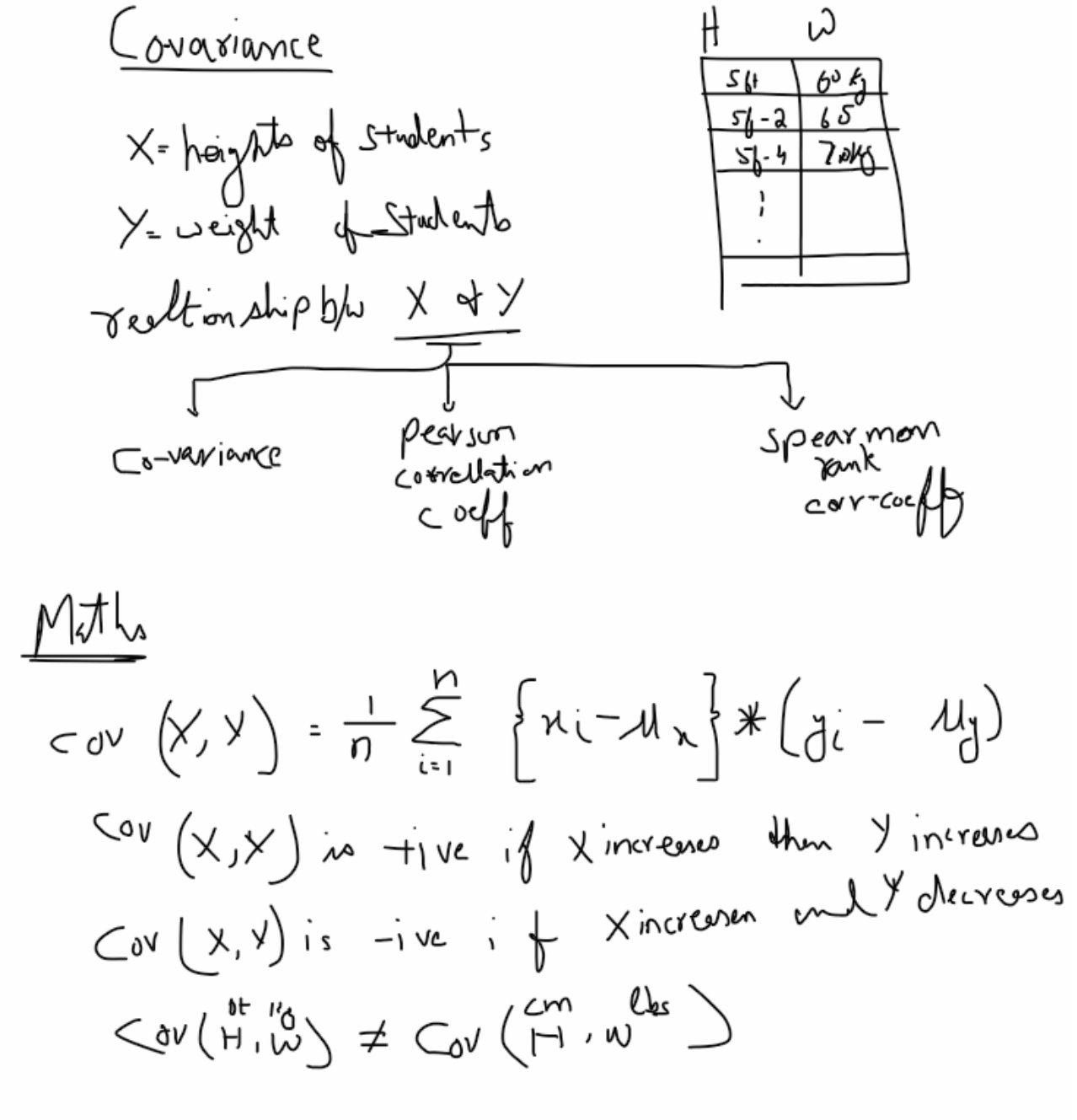


- **Uniform Distribution.**

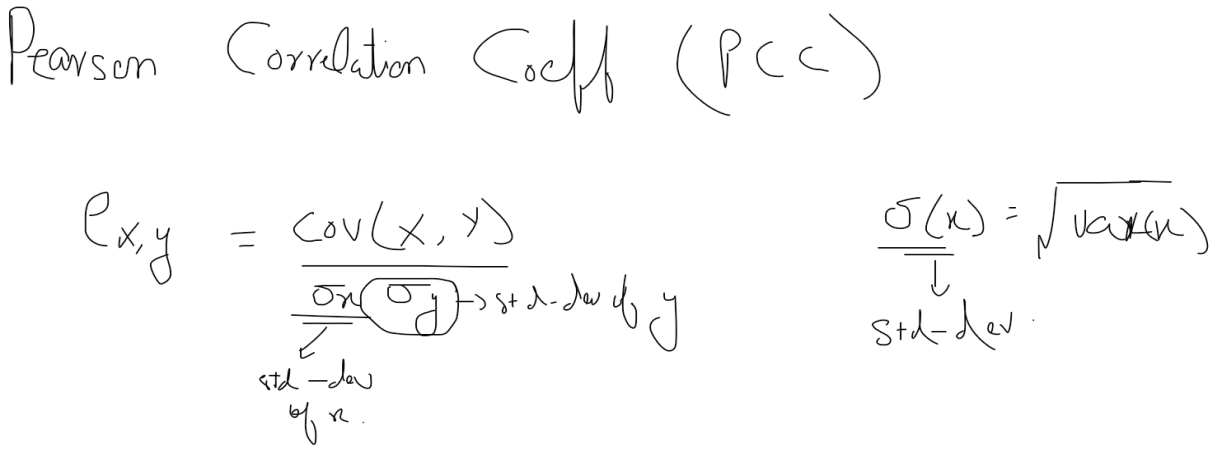
- **Sampled data in Uniform Distribution**

```
import random
print(random.random())
# load iris data set
from sklearn import datasets
iris=datasets.load_iris()
d=iris.data
d.shape
# Sample 30 points randomly from the 150 point dataset
n=150
m=30
p=m/n
sampled_data=[];
for i in range(0,n):
    if random.random() <= p:
        sampled_data.append(d[i,:])
len(sampled_data)
print(sampled_data)

m=30
n=150 #len(iris)
sampled=[]
for i in range(0,n) :
      while i<=m: #This ensures sample size always = 30
        if random.random() < 0.01 and len(sampled)<=30:
            sampled.append(i)
        i+=1
len(sampled)
print(sampled)
```

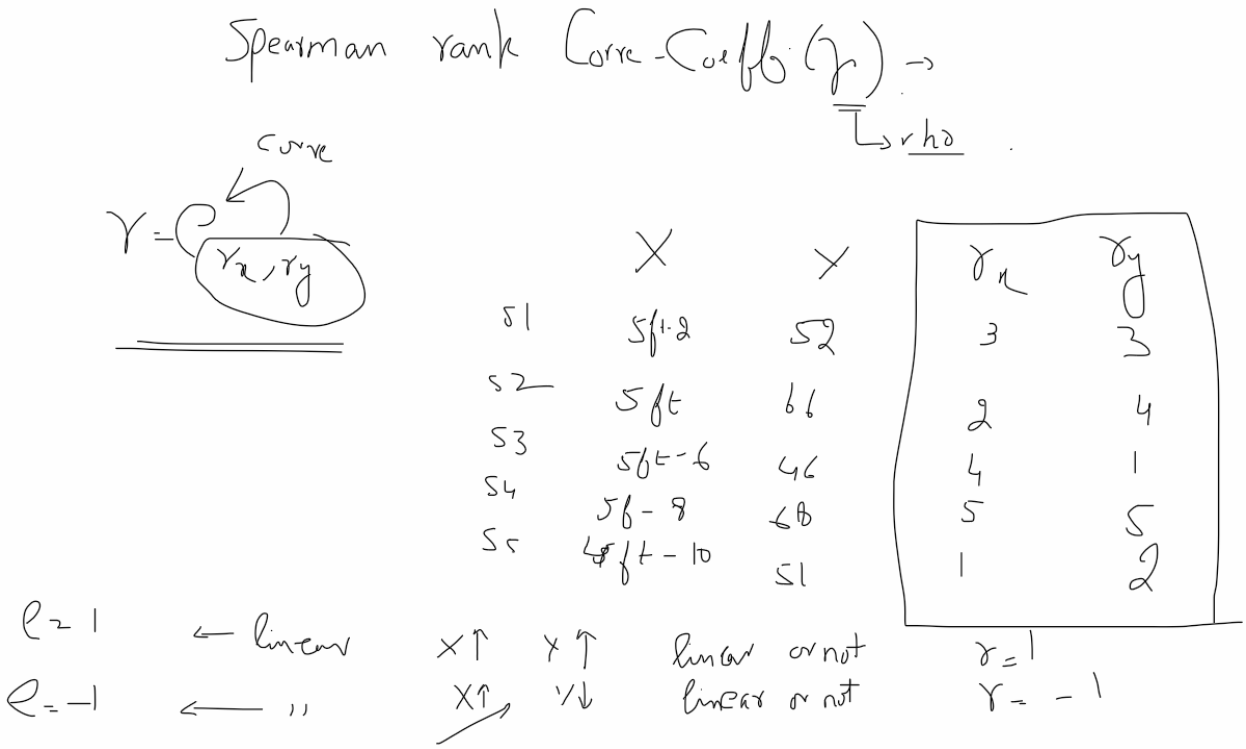
- **Bernoulli Distribution and Binomial distribution**
- **Log Normal Distribution and Power law**
- **Co-Variance:**



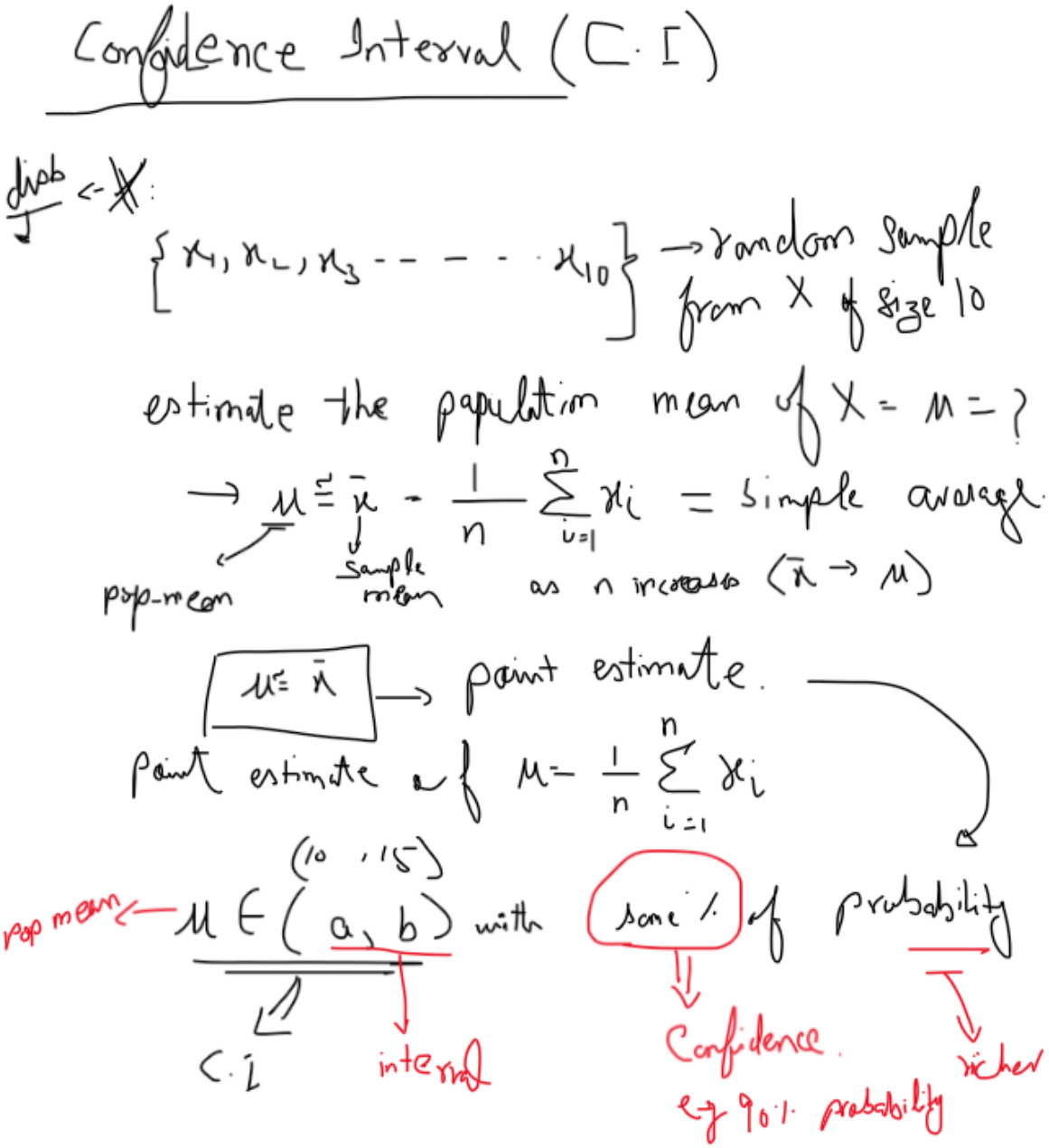
- **Pearson correlation coefficient**



- **Spearman rank corr-coeff:**



- **Confidence Interval:**

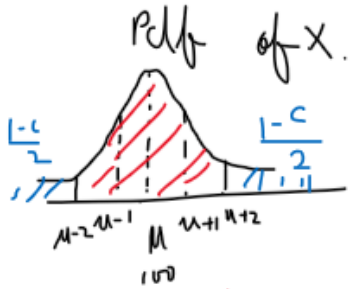


Computing C.I given the underlying distribution

$X \sim N(\mu, \sigma)$

Let $\mu = 100$

$\sigma = 1$



[98, 102] 95% of info

C.I = $[\mu - 2\sigma, \mu + 2\sigma]$

95% of info is in between of $\mu-2$ & $\mu+2$

confidence C

with 95% probability.

C.I for mean (μ) of a R.V

$X \sim F$ with pop-mean μ & std dev of σ

$\{x_1, x_2, x_3, \dots, x_{10}\} \rightarrow$ sample of size $10 \Rightarrow n = 10$

95% C.I of μ = ?

Case-1

σ-given = 2

$\bar{x} = 10$

C.L.T

$$\bar{x} \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

$\bar{x}$ ↓ sample mean

$$10 - \frac{2 \times 2}{\sqrt{10}} = 8.4$$

$$\mu \in \left[\bar{x} - \frac{2\sigma}{\sqrt{n}}, \bar{x} + \frac{2\sigma}{\sqrt{n}}\right] \text{ with 95\% confidence.}$$

Case 2

if we dn't knw σ

t-disb: $\bar{x} \sim t(n-1)$

↳ degrees of freedom

$\bar{x} \sim t(n-1)$

C.I using empirical bootstrap

→ Computational method.

→ C. I for median, var, std dev, 90th percentile.

$X \sim F$ C.I for median of X with 95% = ?

$S = \{x_1, x_2, \ldots, x_{10}\}$

$S: x_1, x_2, \ldots, x_m$ (1) (2) (10)

$m \le n$

median m_1

Bootstrap Samples

S_2

S_3

$S_k: x_1^{(k)} \ldots x_m^{(k)}$

random Sample with repeat of size m from n.

median m_k of sample S_k.

$\{m_1, m_2, \ldots, m_k\}$

→ medians of Bootstrap Samples.

Sort medians.

$m_1' < m_2' < m_3' \ldots m_k'$

Let k = 1000.

25 values $m_{25}' \ldots m_{975}'$ ... 25 values.

95% of C.I of medians of X

$[m_{25}' \text{ & } m_{975}']$

non-parametric technique. {no assumptions of distributions data}

Code :

```
import numpy
from pandas import read_csv
from sklearn.utils import resample
from sklearn.metrics import accuracy_score
from matplotlib import pyplot

# load dataset
x =
numpy.array([180,162,158,172,168,150,171,183,165,176,456,512,163,2
10])

# configure bootstrap
n_iterations = 1000
n_size = 12

# run bootstrap
medians = list()
for i in range(n_iterations):
    # prepare train and test sets
    s = resample(x, n_samples=n_size);
    m = numpy.percentile(s,90);
    #print(m)
    medians.append(m)

# plot scores
pyplot.hist(medians)
pyplot.show()

# confidence intervals
alpha = 0.95
p = ((1.0-alpha)/2.0) * 100
lower =  numpy.percentile(medians, p)

p = (alpha+((1.0-alpha)/2.0)) * 100
upper =  numpy.percentile(medians, p)
print('%.1f confidence interval %.1f and %.1f' % (alpha*100,
lower, upper))
```

- **Kolmogorov–Smirnov test**

The Kolmogorov–Smirnov test may also be used to test whether two underlying one-dimensional probability distributions differ. In this case, the Kolmogorov–Smirnov statistic is

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{2,m}(x)|,$$

where $F_{1,n}$ and $F_{2,m}$ are the empirical distribution functions of the first and the second sample respectively, and $\sup$ is the supremum function.

For large samples, the null hypothesis is rejected at level α if

$$D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{n \cdot m}}.$$

Where n and m are the sizes of first and second sample respectively. The value of $c(\alpha)$ is given in the table below for the most common levels of α

α	0.20	0.15	0.10	0.05	0.025	0.01	0.005	0.001
$c(\alpha)$	1.073	1.138	1.224	1.358	1.48	1.628	1.731	1.949

and in general[18] by

$$c(\alpha) = \sqrt{-\ln\left(\frac{\alpha}{2}\right) \cdot \frac{1}{2}},$$

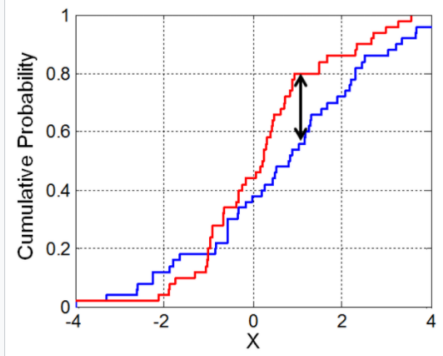


Illustration of the two-sample Kolmogorov–Smirnov statistic. Red and blue lines each correspond to an empirical distribution function, and the black arrow is the two-sample KS statistic.

Code :

```
import numpy as np
import seaborn as sns
from scipy import stats
import matplotlib.pyplot as plt

#generate a gaussian r.v X
x = stats.norm.rvs(size=1000);
sns.set_style('whitegrid')
sns.kdeplot(np.array(x), bw=0.5)
plt.show()


stats.kstest(x, 'norm')

# Y ~ Continous Uniform Distribution(0,1)
y = np.random.uniform(0,1,10000);
sns.kdeplot(np.array(y), bw=0.1)
plt.show()

stats.kstest(y, 'norm')
```

Lab Tasks:

1. What is PDF?
2. What is CDF?
3. explain about 1-std-dev, 2-std-dev, 3-std-dev range?
4. What is Symmetric distribution, Skewness and Kurtosis?
5. How to do Standard normal variate (z) and standardization?
6. What is Kernel density estimation?
7. Importance of Sampling distribution & Central Limit theorem.
8. Importance of Q-Q Plot: Is a given random variable Gaussian distributed?
9. What is Uniform Distribution and random number generators
10. What Discrete and Continuous Uniform distributions?
11. How to randomly sample data points?
12. Explain about Bernoulli and Binomial distribution?
13. What is Log-normal and power law distribution?
14. What is Power-law & Pareto distributions: PDF, examples
15. Explain about Box-Cox/Power transform?
16. What is Co-variance?
17. Importance of Pearson Correlation Coefficient?
18. Importance Spearman Rank Correlation Coefficient?
19. Correlation vs Causation?
20. What is Confidence Intervals?
21. Confidence Interval vs Point estimate?
22. Explain about Hypothesis testing?
23. Define Hypothesis Testing methodology, Null-hypothesis, test-statistic, p-value?
24. How to do K-S Test for similarity of two distributions?

LAB # 06 Principal Component Analysis

Simplest Dimensionality reduction technique used in machine learning.

Principal Component Analysis (PCA)

Simplest Dimensionality Reduction technique.

d-dimension data points → d-dim

$x_i \in R^d$

d-dim → d'-dimen

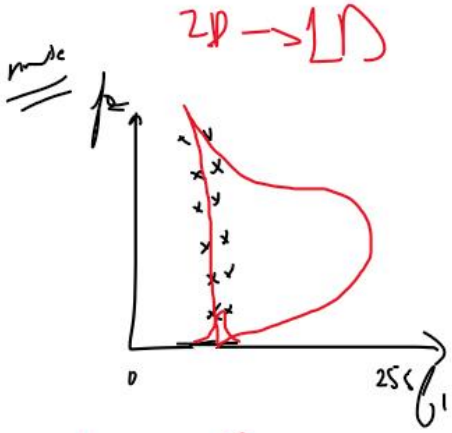
$d' < d$

MNIST data set

784-dimensions

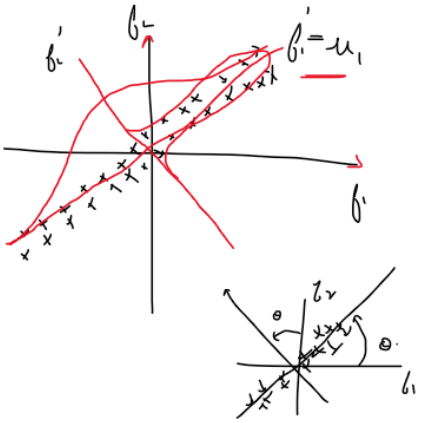
↓ PCA

2-dim (To visualize)



f_2 = ✓

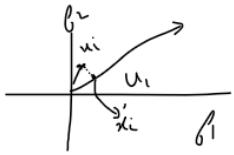
Variance Maximization



u_1 = unit Vector (direction)

$\|u_1\| = 1$

we are interested in direction of u_1



x_i' = projection of x_i

$\Rightarrow \bar{x}_i = \text{Proj}_{u_1} = \frac{u_1 \cdot x_i}{\|u_1\|^2} = u_1^T x_i$

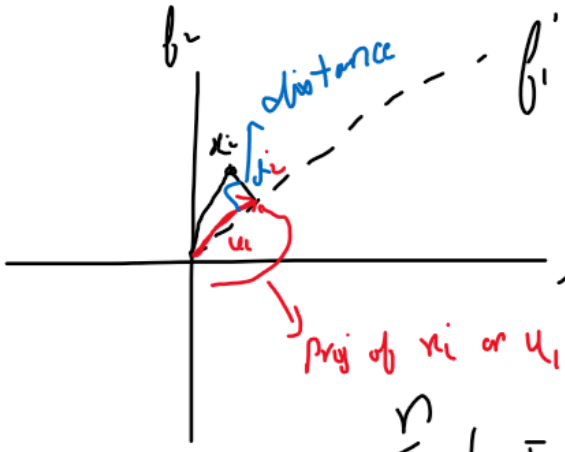
$$\text{Var}\left\{u_1^T x_i\right]_{i=1}^{n} = \frac{1}{n}\sum_{i=1}^{n}\left(u_1^T x_i - u_1^T \bar{x}_i\right)^2$$

$$\text{Var}\left[x_i'\right]_{i=1}^{n} = \frac{1}{n}\sum_{i=1}^{n}\left(u_1^T x_i\right)^2$$

↳ Data matrix

find u_1 to maximize $\text{Var}\left[x_i'\right]^n$

Alternative Formulation of PCA : Distance minimization



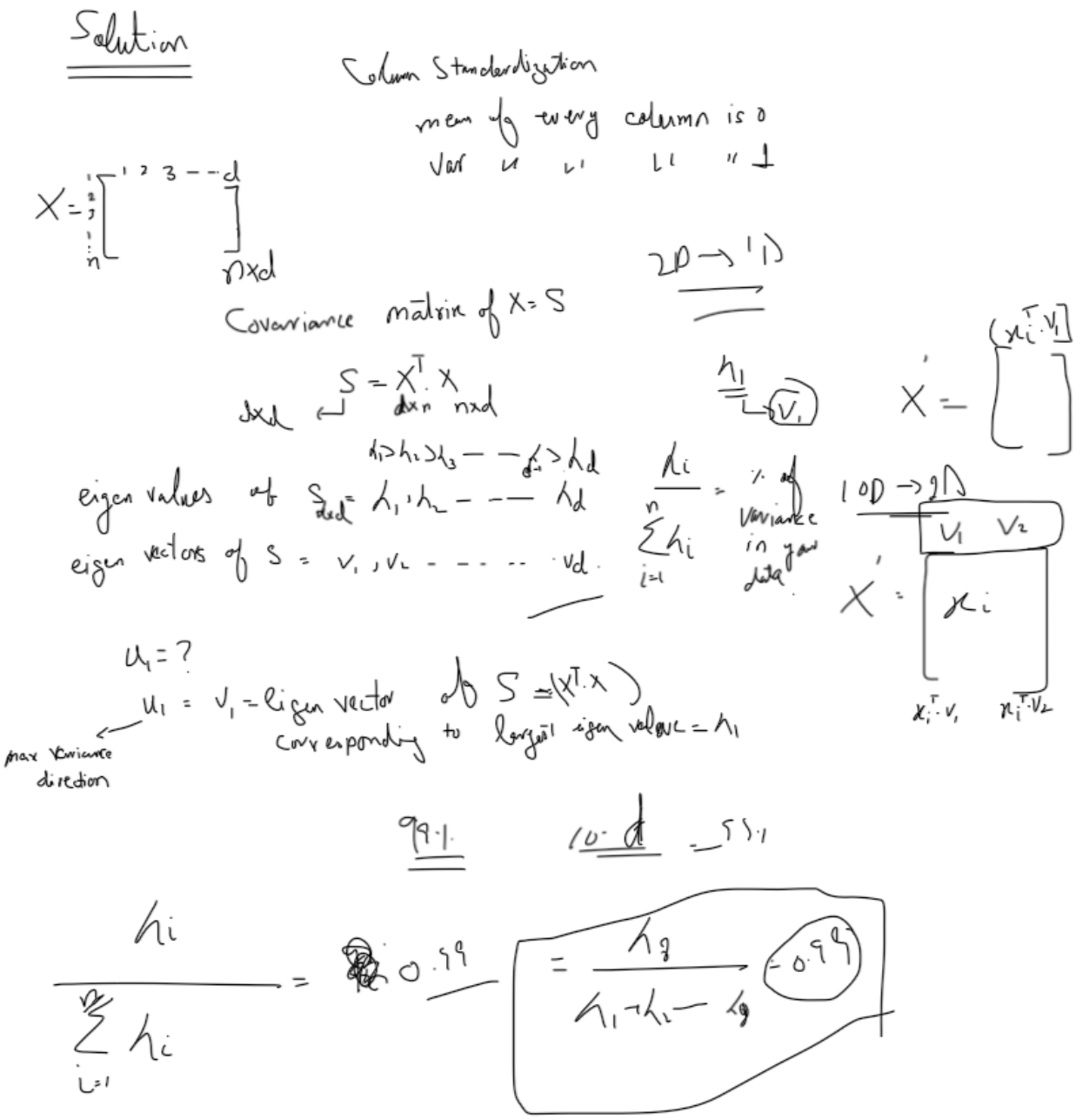
$$d_i^2 = \|x_i\|^2 - \|u_1^T x_i\|^2$$

$$\min_{u_1} \sum_{i=1}^{n} \left(x_i^T x_i - u_1^T x_i\right)^2$$

↳ data matrix

d_i^2

Constraint $u_1 u_1^T = 1$



Code

```
# MNIST dataset downloaded from Kaggle :
#https://www.kaggle.com/c/digit-recognizer/data

# Functions to read and show images.

import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt

d0 = pd.read_csv('mnist_train.csv')

print(d0.head(5)) # print first five rows of d0.

# save the labels into a variable l.
l = d0['label']

# Drop the label feature and store the pixel data in d.
d = d0.drop("label",axis=1)

print(d.shape)
print(l.shape)

# display or plot a number.
plt.figure(figsize=(7,7))
idx = 1

grid_data = d.iloc[idx].to_numpy().reshape(28,28)  # reshape from 1d to 2d
pixel array
plt.imshow(grid_data, interpolation = "none", cmap = "gray")
plt.show()

print(l[idx])


# Pick first 15K data-points to work on for time-effeciency.
#Excercise: Perform the same analysis on all of 42K data-points.

labels = l.head(15000)
data = d.head(15000)

print("the shape of sample data = ", data.shape)

# Data-preprocessing: Standardizing the data

from sklearn.preprocessing import StandardScaler
standardized_data = StandardScaler().fit_transform(data)
print(standardized_data.shape)

#find the co-variance matrix which is : A^T * A
sample_data = standardized_data

# matrix multiplication using numpy
covar_matrix = np.matmul(sample_data.T , sample_data)

print ( "The shape of variance matrix = ", covar_matrix.shape)

# finding the top two eigen-values and corresponding eigen-vectors
# for projecting onto a 2-Dim space.

from scipy.linalg import eigh
```

```
# the parameter 'eigvals' is defined (low value to heigh value)
# eigh function will return the eigen values in asending order
# this code generates only the top 2 (782 and 783) eigenvalues.
values, vectors = eigh(covar_matrix, eigvals=(782,783))

print("Shape of eigen vectors = ",vectors.shape)
# converting the eigen vectors into (2,d) shape for easyness of further
computations
vectors = vectors.T

print("Updated shape of eigen vectors = ",vectors.shape)
# here the vectors[1] represent the eigen vector corresponding 1st principal
eigen vector
# here the vectors[0] represent the eigen vector corresponding 2nd principal
eigen vector

# projecting the original data sample on the plane
#formed by two principal eigen vectors by vector-vector multiplication.

import matplotlib.pyplot as plt
new_coordinates = np.matmul(vectors, sample_data.T)

print (" resultanat new data points' shape ", vectors.shape, "X",
sample_data.T.shape," = ", new_coordinates.shape)

import pandas as pd

# appending label to the 2d projected data
new_coordinates = np.vstack((new_coordinates, labels)).T

# creating a new data frame for ploting the labeled points.
dataframe = pd.DataFrame(data=new_coordinates, columns=("1st_principal",
"2nd_principal", "label"))
print(dataframe.head())

import pandas as pd
df=pd.DataFrame()
df['1st']=[-5.558661,-5.043558,6.193635 ,19.305278]
df['2nd']=[-1.558661,-2.043558,2.193635 ,9.305278]
df['label']=[1,2,3,4]

import seaborn as sn
import matplotlib.pyplot as plt
sn.FacetGrid(df, hue="label", size=6).map(plt.scatter, '1st',
'2nd').add_legend()
plt.show()

# ploting the 2d data points with seaborn
import seaborn as sn
sn.FacetGrid(dataframe, hue="label", size=6).map(plt.scatter, '1st_principal',
'2nd_principal').add_legend()
plt.show()
```

```
# initializing the pca
from sklearn import decomposition
pca = decomposition.PCA()

# configuring the parameteres
# the number of components = 2
pca.n_components = 2
pca_data = pca.fit_transform(sample_data)

# pca_reduced will contain the 2-d projects of simple data
print("shape of pca_reduced.shape = ", pca_data.shape)


# attaching the label for each 2-d data point
pca_data = np.vstack((pca_data.T, labels)).T

# creating a new data fram which help us in ploting the result data
pca_df = pd.DataFrame(data=pca_data, columns=("1st_principal",
"2nd_principal", "label"))
sn.FacetGrid(pca_df, hue="label", size=6).map(plt.scatter, '1st_principal',
'2nd_principal').add_legend()
plt.show()

# PCA for dimensionality redcution (non-visualization)

pca.n_components = 784
pca_data = pca.fit_transform(sample_data)

percentage_var_explained = pca.explained_variance_ /
np.sum(pca.explained_variance_);

cum_var_explained = np.cumsum(percentage_var_explained)

# Plot the PCA spectrum
plt.figure(1, figsize=(6, 4))

plt.clf()
plt.plot(cum_var_explained, linewidth=2)
plt.axis('tight')
plt.grid()
plt.xlabel('n_components')
plt.ylabel('Cumulative_explained_variance')
plt.show()
```

Lab Tasks:

Run the same analysis using 42K points with various values of perplexity and iterations.

If you use all of the points, you can expect plots like this blog below:
http://colah.github.io/posts/2014-10-Visualizing-MNIST/

LAB # 07 Implementation of Linear Regression and Gradient Descent in Python.

Linear Regression is one of the easiest algorithms in machine learning. In this manual we will explore this algorithm and we will implement it using Python from scratch. As the name suggests this algorithm is applicable for Regression problems. Linear Regression is a Linear Model. Which means, we will establish a linear relationship between the input variables(X) and single output variable(Y). When the input(X) is a single variable this model is called Simple Linear Regression and when there are multiple input variables(X), it is called Multiple Linear Regression.

Simple Linear Regression

We discussed that Linear Regression is a simple model. Simple Linear Regression is the simplest model in machine learning.

Model Representation

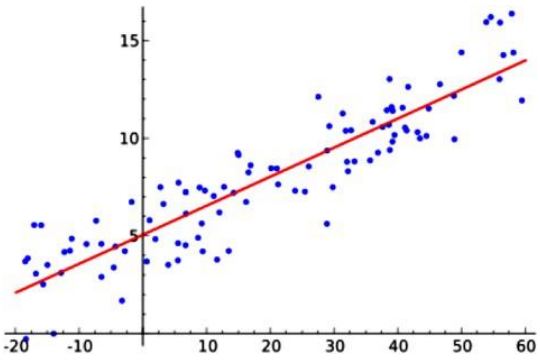
In this problem we have an input variable - X and one output variable - Y. And we want to build linear relationship between these variables. Here the input variable is called Independent Variable and the output variable is called Dependent Variable. We can define this linear relationship as follows:

$$Y = \beta 0 + \beta 1 X$$

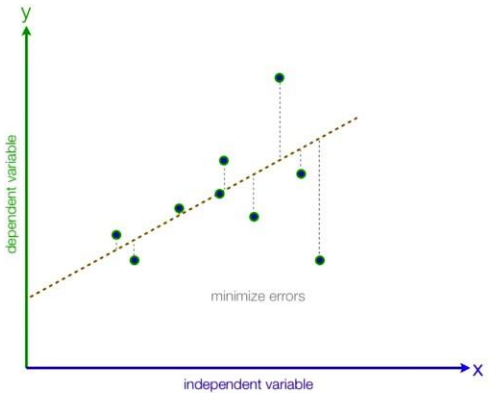
The β1 is called a scale factor or coefficient and β0 is called bias coefficient. The bias coefficient gives an extra degree of freedom to this model. This equation is similar to the line equation y = mx + b with m = β1 (Slope) and b = β0 (Intercept). So in this Simple Linear Regression model we want to draw a line between X and Y which estimates the relationship between X and Y. But how do we find these coefficients? That's the learning procedure. We can find these using different approaches. One is called Ordinary Least Square Method and other one is called Gradient Descent Approach. We will use Ordinary Least Square Method in Simple Linear Regression and Gradient Descent Approach in Multiple Linear Regression.

Ordinary Least Square Method

Earlier in this manual we discussed that we are going to approximate the relationship between X and Y to a line. Let's say we have few inputs and outputs. And we plot these scatter points in 2D space, we will get something like the following image.



And you can see a line in the image. That's what we are going to accomplish. And we want to minimize the error of our model. A good model will always have least error. We can find this line by reducing the error. The error of each point is the distance between line and that point. This is illustrated as follows.



And total error of this model is the sum of all errors of each point. i-e:

$$D = \sum_{i=1}^{m} d_i^2$$

d_i -Distance between line and i^{th} point.

m- Total number of points

You might have noticed that we are squaring each of the distances. This is because, some points will be above the line and some points will be below the line. We can minimize the error in the model by minimizing D. And after the mathematics of minimizing D, we will get;

$$\beta_1 = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m}(x_i - \bar{x})^2}$$

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

In these equations $\overline{x}$ is the mean value of input variable **x** and $\overline{y}$ is the mean value of output variable **y**. Now we have the model. This method is called Ordinary Least Square Method. Now we will implement this model in Python.

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

$$\beta_1 = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m}(x_i - \bar{x})^2}$$

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

Implementation

We are going to use a dataset containing head size and brain weight of different people. This data set has other features. But, we will not use them in this model. This dataset is available in this Github Repo (https://github.com/mubaris/potential-enigma). Let's start off by importing the data.

```
# Importing Necessary Libraries

%matplotlib  inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (20.0, 10.0)
# Reading Data

data = pd.read_csv('headbrain.csv')
print(data.shape)

data.head()
```

```
(237, 4)
```

	Gender	Age Range	Head Size(cm^3)	Brain Weight(grams)
0	1	1	4512	1530
1	1	1	3738	1297
2	1	1	4261	1335
3	1	1	3777	1282
4	1	1	4177	1590

As you can see there are 237 values in the training set. We will find a linear relationship between Head Size and Brain Weights. So, now we will get these variables.

```
# Collecting X and Y
X = data['Head Size(cm^3)'].values

Y = data['Brain Weight(grams)'].values
```

To find the values β1 and β0, we will need mean of **X** and **Y**. We will find these and the coefficients.

```
# Mean X and Y
mean_x = np.mean(X)
mean_y = np.mean(Y)
# Total number of values

m = len(X)
# Using the formula to calculate b1 and b2

numer = 0
denom = 0

for i in range(m):

numer += (X[i] - mean_x) * (Y[i] - mean_y)
denom += (X[i] - mean_x) ** 2
```

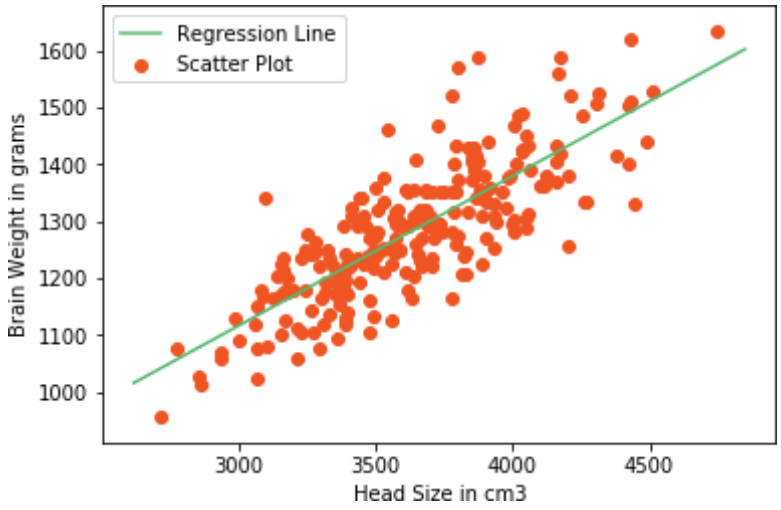
```
b1 = numer / denom

b0 = mean_y - (b1 * mean_x)
# Print coefficients
print(b1, b0)
```

There we have our coefficients.

$$\text{Brain Weight} = 325.573421049 + 0.263429339489 * \text{Head Size}$$

That is our linear model. Now we will see this graphically.

```
# Plotting Values and Regression Line

max_x = np.max(X) + 100
min_x = np.min(X) - 100
# Calculating line values x and y

x = np.linspace(min_x, max_x, 1000)
y = b0 + b1 * x
# Ploting Line

plt.plot(x, y, color='#58b970', label='Regression Line')
# Ploting Scatter Points
plt.scatter(X, Y, c='#ef5423', label='Scatter Plot')
plt.xlabel('Head Size in cm3')

plt.ylabel('Brain Weight in grams')
plt.legend()

plt.show()
```



This model is not so bad. But we need to find how good our model is. There are many methods to evaluate models. We will use **Root Mean Squared Error** and **Coefficient of Determination (R^2 Score).** Root Mean Squared Error is the square root of sum of all errors divided by number of values, or Mathematically,

$$RMSE = \sqrt{\sum_{i=1}^{m} \frac{1}{m}(\hat{y}_i - y_i)^2}$$

Here $\hat{y}$ is the i[th] predicted output values. Now we will find RMSE.

```
# Calculating Root Mean Squares Error

rmse = 0
for i in range(m):
y_pred = b0 + b1 * X[i]

rmse += (Y[i] - y_pred) ** 2
rmse = np.sqrt(rmse/m)
print(rmse)
```

Now we will find $\mathbf{R^2}$ score. $\mathbf{R^2}$ is defined as follows,

$$SS_t = \sum_{i=1}^{m}(y_i - \bar{y})^2$$

$$SS_r = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

$$R^2 \equiv 1 - \frac{SS_r}{SS_t}$$

$\mathbf{SS_t}$ is the total sum of squares and $\mathbf{SS_r}$ is the total sum of squares of residuals.

$\mathbf{R^2}$ Score usually range from 0 to 1. It will also become negative if the model is completely wrong. Now we will find $\mathbf{R^2}$ Score

```
ss_t = 0

ss_r = 0

for i in range(m):
y_pred = b0 + b1 * X[i]

ss_t += (Y[i] - mean_y) ** 2
ss_r += (Y[i] - y_pred) ** 2
r2 = 1 - (ss_r/ss_t)
print(r2)
```

0.63 is not so bad. Now we have implemented Simple Linear Regression Model using Ordinary Least Square Method. Now we will see how to implement the same model using a Machine Learning Library called scikit-learn (http://scikit-learn.org/)

The scikit-learn approach

scikit-learn (http://scikit-learn.org/) is simple machine learning library in Python. Building Machine

Learning models are very easy using scikit-learn. Let's see how we can build this Simple Linear

Regression Model using scikit-learn.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
# Cannot use Rank 1 matrix in scikit learn

X = X.reshape((m, 1))
# Creating Model

reg = LinearRegression()
# Fitting training data
```

```
reg = reg.fit(X, Y)
# Y Prediction
Y_pred = reg.predict(X)
# Calculating RMSE and R2 Score

mse = mean_squared_error(Y, Y_pred)
rmse = np.sqrt(mse)

r2_score = reg.score(X, Y)
print(np.sqrt(mse))
print(r2_score)

72.1206213784
```

You can see that this exactly equal to model we built from scratch, but simpler and less code. Now we will move on to Multiple Linear Regression.

Multiple Linear Regression

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

Model Representation

Similar to Simple Linear Regression, we have input variable(X) and output variable(Y). But the input variable has n features. Therefore, we can represent this linear model as follows;

$$Y = \beta_0 + \beta_1 x_1 + \beta_1 x_2 + \ldots + \beta_n x_n$$

x_i is the i^{th} feature in input variable. By introducing $x_0 = 1$, we can rewrite this equation.

$$Y = \beta_0 x_0 + \beta_1 x_1 + \beta_1 x_2 + \ldots + \beta_n x_n$$

$$x_0 = 1$$

Now we can convert this equation to matrix form.

$$Y = \beta^T X$$

Where,

$$\beta = [\beta_0 \quad \beta_1 \quad \beta_2 \quad .. \quad \beta_n]^T$$

And

$$X = [x_0 \quad x_1 \quad x_2 \quad .. \quad x_n]^T$$

We have to define the cost of the model. Cost basically gives the error in our model. Y in above equation is our hypothesis (approximation). We are going to define it as our hypothesis function.

$$h_\beta(x) = \beta^T x$$

And the cost is,

$$J(\beta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\beta(x^{(i)}) - y^{(i)})^2$$

By minimizing this cost function, we can get find β. We use Gradient Descent for this.

Gradient Descent

Gradient Descent is an optimization algorithm. We will optimize our cost function using Gradient Descent Algorithm.

Step 1

Initialize β0, β1,….. βn with some value. In this case we will initialize with 0.

Step 2

Iteratively update,

$$\beta_j := \beta_j - \alpha\frac{\partial}{\partial\beta_j}J(\beta)$$

Until it converges.

This is the procedure. Here α is the learning rate. This $\frac{\partial}{\partial\beta_j}J(\beta)$ operation means we are finding partial derivate of cost with respect to each βj. This is called Gradient.

In step 2 we are changing the values of βj in a direction in which it reduces our cost function. And Gradient gives the direction in which we want to move. Finally we will reach the minima of our cost function. But we don't want to change values of βj drastically, because we might miss the minima. That's why we need learning rate.

But we still didn't find $\frac{\partial}{\partial\beta_j}J(\beta)$ the value of . After we applying the mathematics. The step 2 becomes.

$$\beta_j := \beta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\beta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

We iteratively change values βj of according to above equation. This particular method is called Batch Gradient Descent.

Implementation

Let's try to implement this in Python. This looks like a long procedure. But the implementation is comparatively easy since we will vectorize all the equations. If you are unfamiliar with vectorization, read this post (https://www.datascience.com/blog/straightening-loops-how-to-vectorize- dataaggregation-with-pandas-and-numpy/) we will be using a student score dataset. In this particular dataset, we have math, reading and writing exam scores of 1000 students. We will try to find a predict score of writing exam from math and reading scores. You can get this dataset from this Github Repo (https://github.com/mubaris/potentialenigma). Here we have 2 features (input variables). Let's start by importing our dataset.

```
%matplotlib  inline
import  numpy  as  np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (20.0, 10.0)
from mpl_toolkits.mplot3d import Axes3D

data = pd.read_csv('student.csv')
print(data.shape)

data.head()
```

```
(1000, 3)
```

	Math	Reading	Writing
0	48	68	63
1	62	81	72
2	79	80	78
3	76	83	79
4	59	64	62

We will get scores to an array.

```
math = data['Math'].values
read = data['Reading'].values
write =data['Writing'].values
# Ploting the scores as scatter plot
fig = plt.figure()
ax = Axes3D(fig)

ax.scatter(math, read, write, color='#ef1234')
plt.show()
```

Now we will generate our X, Y and β.

```
m = len(math)
x0 =np.ones(m)
X = np.array([x0, math, read]).T
# Initial Coefficients
B = np.array([0, 0,0])
Y = np.array(write)
alpha = 0.0001
```

We'll define our cost function.

```
def cost_function(X, Y, B):
m = len(Y)

J = np.sum((X.dot(B) - Y) ** 2)/(2 * m)
return J

inital_cost = cost_function(X, Y, B)
print(inital_cost)

2470.11
```

As you can see our initial cost is huge. Now we'll reduce our cost periodically using Gradient Descent.

Hypothesis: $h_\beta(x) = \beta^T x$

Loss: $(h_\beta(x) - y)$

Gradient: $(h_\beta(x) - y)x_j$

Gradient Descent Updation: $\beta_j := \beta_j - \alpha(h_\beta(x) - y)x_j)$

```
def gradient_descent(X, Y, B, alpha, iterations):
cost_history = [0] * iterations
m = len(Y)

for iteration in range(iterations):
# Hypothesis Values

h = X.dot(B)
# Difference b/w Hypothesis and Actual Y
loss = h - Y
# Gradient Calculation

gradient = X.T.dot(loss) / m
# Changing Values of B using Gradient

B = B - alpha * gradient
# New Cost Value
cost = cost_function(X, Y, B)
cost_history[iteration] = cost
return B, cost_history
```

Now we will compute final value of β

```
# 100000 Iterations
newB, cost_history = gradient_descent(X, Y, B, alpha, 100000)
# New Values of B
print(newB)
# Final Cost of new B
print(cost_history[-1])
[-0.47889172 0.09137252 0.90144884]
```

We can say that in this model,

$$S_{writing} = -0.47889172 + 0.09137252 * S_{math} + 0.90144884 * S_{reading}$$

There we have final hypothesis function of our model. Let's calculate RMSE and R2 Score of our model to evaluate.

```
Model Evaluation - RMSE
def rmse(Y, Y_pred):
rmse = np.sqrt(sum((Y - Y_pred) ** 2) / len(Y))
return rmse
```

```
# Model Evaluation - R2 Score
def r2_score(Y, Y_pred):
mean_y = np.mean(Y)
ss_tot = sum((Y - mean_y) ** 2)
ss_res = sum((Y - Y_pred) ** 2)
r2 = 1 - (ss_res / ss_tot)
return r2
Y_pred = X.dot(newB)
print(rmse(Y, Y_pred))
print(r2_score(Y, Y_pred))
4.57714397273
```

We have very low value of RMSE score and a good R^2 score. I guess our model was pretty good. Now we will implement this model using scikit-learn.

The scikit-learn Approach

scikit-learn approach is very similar to Simple Linear Regression Model and simple too. Let's implement this.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
# X and Y Values

X = np.array([math, read]).T
Y = np.array(write)
# Model Intialization

reg = LinearRegression()
# Data Fitting

reg = reg.fit(X, Y)
# Y Prediction

Y_pred = reg.predict(X)
# Model Evaluation
rmse = np.sqrt(mean_squared_error(Y, Y_pred))
r2 = reg.score(X, Y)

print(rmse)
print(r2)
4.57288705184

0.909890172672
```

You can see that this model is better than one which we have built from scratch by a small margin. That's it for Linear Regression. I assume, so far you have understood Linear Regression, Ordinary Least Square Method and Gradient Descent.

Lab Tasks

1. Apply Simple Linear Regression model to another data set and evaluate the model with **RMSE** and **R^2** Methods.
2. Apply Multiple Linear Regression model to another data set and evaluate the model with **RMSE** and **R^2** Methods.

LAB # 08 Python Machine Learning with Perceptron.

What is a Perceptron?

A perceptron uses the basic ideas of machine learning and neural networks. The idea is that you feed a program a bunch of inputs, and it learns how to process those inputs into an output. It does that by assigning each input a weight. Each input is multiplied by that weight, and summed together. Lastly, we need to turn that sum into a value: 1 or -1. When training a perceptron, we evaluate the output that our program generates, and adjust our weights based on what the inputs were.

The perceptron can be used for supervised learning. It can solve binary linear classification problems. A comprehensive description of the functionality of a perceptron is out of scope here. To follow this lab manual you already should know what a perceptron is and understand the basics of its functionality. Additionally a fundamental understanding of stochastic gradient descent is also needed. To better understand the internal processes of a perceptron in practice, we will step by step develop a perceptron from scratch now.

Our Ingredients

First we will import numpy to easily manage linear algebra and calculus operations in python. To plot the learning progress later on, we will use matplotlib.

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

Stochastic Gradient Descent

We will implement the perceptron algorithm in python 3 and numpy. The perceptron will learn using the stochastic gradient descent algorithm (SGD). Gradient Descent minimizes a function by following the gradients of the cost function.

Calculating the Error

To calculate the error of a prediction we first need to define the objective function of the perceptron.

Hinge Loss Function

To do this, we need to define the loss function, to calculate the prediction error. We will use hinge loss for our perceptron:

$$c(x, y, f(x)) = (1 - y * f(x))_+$$

c is the loss function, x the sample, y is the true label, f(x) the predicted label. This means the following:

$$c(x, y, f(x)) = \begin{cases} 0, & \text{if } y * f(x) \geq 1 \\ 1 - y * f(x), & \text{else} \end{cases}$$

So consider, if y and f(x) are signed values

- The loss is 0, if y*f(x) are positive, respective both values have the same sign.
- Loss is if is negative –y*f(x) if y*f(x) is negative.

Objective Function

As we defined the loss function, we can now define the objective function for the perceptron:

$$l_i(w) = \big(- y_i \langle x_i, w \rangle \big)_+$$

We can write this without the dot product with a sum sign:

$$l_i(w) = (-y_i \sum_{i=1}^{n} x_i w)_+$$

So the sample xi is misclassified, if yi ⟨xi,w⟩ ≤ 0 . The general goal is, to find the global minima of this function, respectively find a parameter, where the error is zero.

Derive the Objective Function

To do this we need the gradients of the objective function. The gradient of a function is the vector of its partial derivatives. The gradient can be calculated by the partially derivative of the objective function.

$$\nabla l_i(w) = -y_i x_i$$

This means, if we have a misclassified xi sample, respectively yi ⟨xi,w⟩ ≤ 0 update the weight vector w by moving it in the direction of the misclassified sample.

$$w = w + y_i x_i$$

With this update rule in mind, we can start writing our perceptron algorithm in python.

Our Data Set

First we need to define a labeled data set.

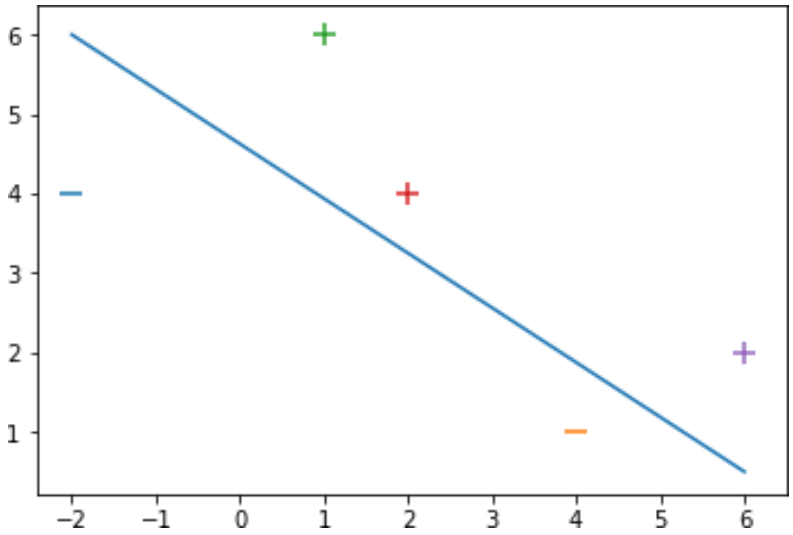
```
X = np.array([
[-2, 4],
[4, 1],
[1, 6],
[2, 4],
[6, 2]  ])
```

Next we fold a bias term -1 into the data set. This is needed for the SGD to work.

```
X = np.array([
    [-2,4,-1],
    [4,1,-1],
    [1, 6, -1],
    [2, 4, -1],
    [6, 2, -1], ])
y = np.array([-1,-1,1,1,1])
```

This small toy data set contains two samples labeled with-1 and three samples labeled with +1. This means we have a binary classification problem, as the data set contains two sample classes. Let's plot the dataset to see, that is linearly separable:

```
for d, sample in enumerate(X):
  # Plot the negative samples if
  d < 2:
    plt.scatter(sample[0], sample[1], s=120, marker='_', linewidths=2)
  # Plot the positive samples
  else:
    plt.scatter(sample[0], sample[1], s=120, marker='+', linewidths=2)

# Print a possible hyperplane, that is separating the two classes.
plt.plot([-2,6],[6,0.5])
```



Let's Start implementing Stochastic Gradient Descent

Finally, we can code our SGD algorithm using our update rule. To keep it simple, we will linearly loop over the sample set. For larger data sets it makes sense, to randomly pick a sample during each iteration in the for-loop.

```
def perceptron_sgd(X,Y):
   w = np.zeros(len(X[0]))
  eta = 1
   epochs = 20
   for t in range(epochs):
```

```
        for i, x in enumerate(X):
      if (np.dot(X[i], w)*Y[i]) <= 0: w
        = w + eta*X[i]*Y[i]

return w
w = perceptron_sgd(X,y)
```

Code Description Line by Line

line 2: Initialize the weight vector for the perceptron with zeros

line 3: Set the learning rate to 1

line 4: Set the number of epochs

line 6: Iterate n times over the whole data set.

line 7: Iterate over each sample in the data set

line 8: Misclassification condition yi $\langle xi,w \rangle \leq 0$

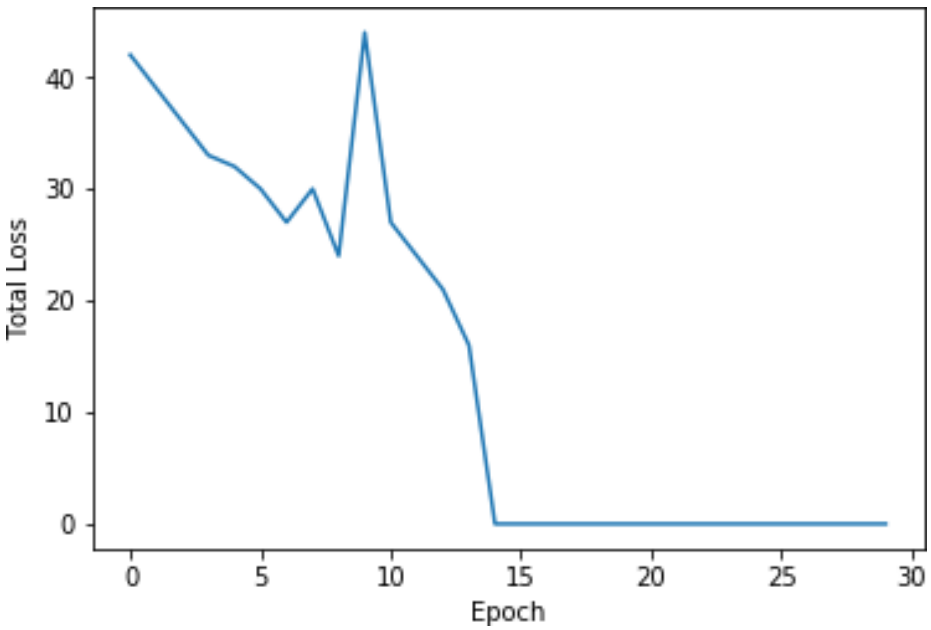
line 9: Update rule for the weights w = w + yi $*$ xi including the learning rate

Let the Perceptron learn!

Next we can execute our code and check, how many iterations are needed, until all samples are classified right. To see the learning progress of the perceptron, we add a plotting feature to our algorithm, counting the total error in each epoch.

```
def perceptron_sgd_plot(X, Y):
   w = np.zeros(len(X[0]))

   eta = 1
   n = 30
   errors = []
   for t in range(n):
    total_error = 0
    for i, x in enumerate(X):
      if (np.dot(X[i], w)*Y[i]) <= 0:
        total_error += (np.dot(X[i],w)*Y[i])
        w = w + eta*X[i]*Y[i]
    errors.append(total_error*-1)
  plt.figure()
  plt.plot(errors)
  plt.xlabel('Epoch')
  plt.ylabel('Total Loss')
```

```
    return w

print(perceptron_sgd_plot(X,y))
```



This means, that the perceptron needed 14 epochs to classify all samples right (total error is zero). In other words, the algorithm needed to see the data set 14 times, to learn its structure. The weight vector including the bias term is (2, 3, 13).

We can extract the following prediction function now:

$$f(x) = \langle x, (2,3) \rangle - 13$$

Evaluation

Let's classify the samples in our data set by hand now, to check if the perceptron learned properly:

First sample $(-2, 4)$, supposed to be negative:

$$-2 * 2 + 4 * 3 - 13 = sign(-5) = -1$$

Second sample $(4, 1)$, supposed to be negative:

$$4 * 2 + 1 * 3 - 13 = sign(-2) = -1$$

Third sample $(1, 6)$, supposed to be positive:

$$1 * 2 + 6 * 3 - 13 = sign(7) = +1$$

Fourth sample $(2, 4)$, supposed to be positive:

$$2 * 2 + 4 * 3 - 13 = sign(3) = +1$$

Fifth sample $(6, 2)$, supposed to be positive:

$$6 * 2 + 2 * 3 - 13 = sign(5) = +1$$

Lets define two test samples now, to check how well our perceptron generalizes to unseen data:

First test sample $(2, 2)$, supposed to be negative:

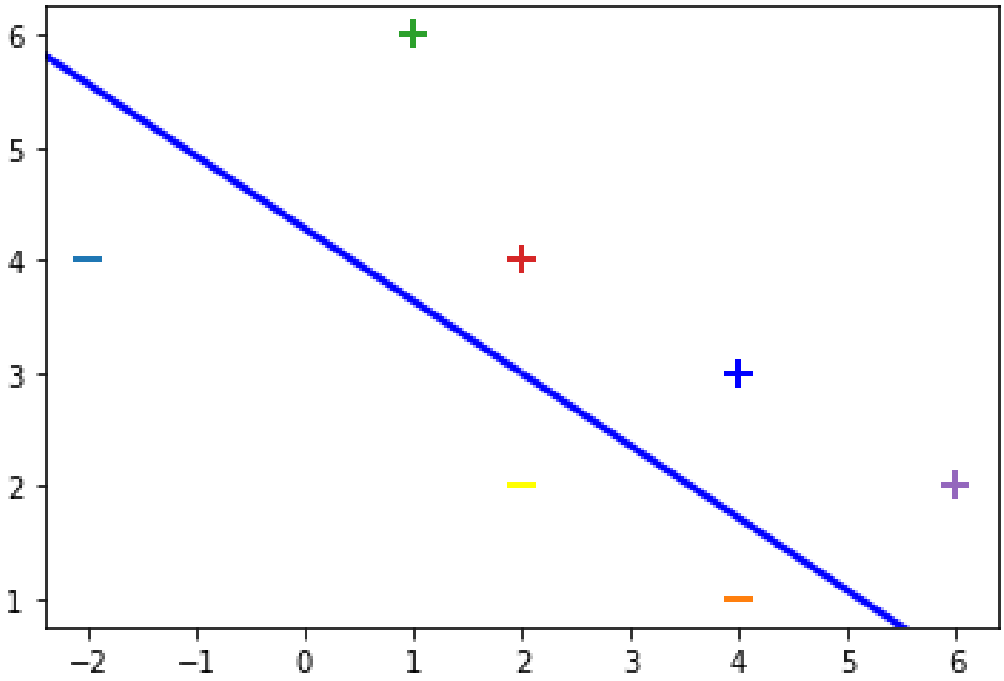
$$2 * 2 + 2 * 3 - 13 = sign(-3) = -1$$

Second test sample $(4, 3)$, supposed to be positive:

$$4 * 2 + 3 * 3 - 13 = sign(4) = +1$$

Both samples are classified right. To check this geometrically, lets plot the samples including test samples and the hyperplane.

```
plt.figure()

for d, sample in enumerate(X):

  # Plot the negative samples if
  d < 2:

    plt.scatter(sample[0], sample[1], s=120, marker='_', linewidths=2)

  # Plot the positive samples
  else:

    plt.scatter(sample[0], sample[1], s=120, marker='+', linewidths=2)

# Add our test samples

plt.scatter(2,2, s=120, marker='_', linewidths=2, color='yellow')

plt.scatter(4,3, s=120, marker='+', linewidths=2, color='blue')

# Print the hyperplane calculated by
svm_sgd() x2=[w[0],w[1],-w[1],w[0]]
x3=[w[0],w[1],w[1],-w[0]]

x2x3 =np.array([x2,x3])
X,Y,U,V = zip(*x2x3) ax =
plt.gca()
ax.quiver(X,Y,U,V,scale=1, color='blue')
```



That's all about it. If you got so far, keep in mind, that the basic structure is the SGD applied to the objective function of the perceptron.

Lab Tasks:

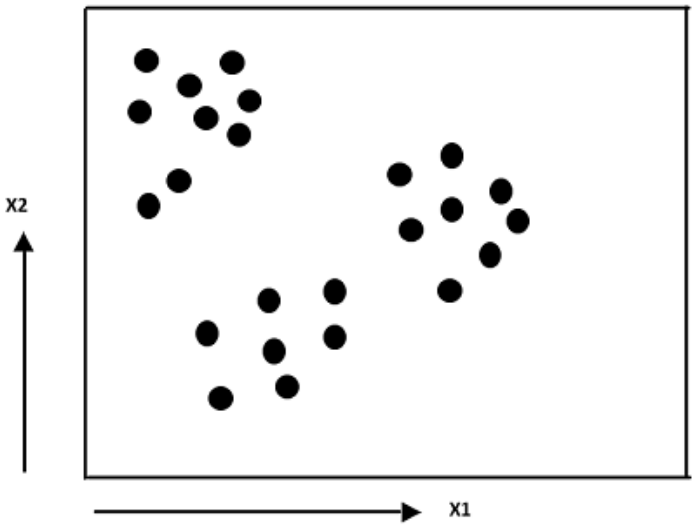
1. Apply the perceptron Algorithm to another two data sets of your choice and verify them mathematically and show their graphs in Python.

LAB # 09 K-means Algorithm with Python.

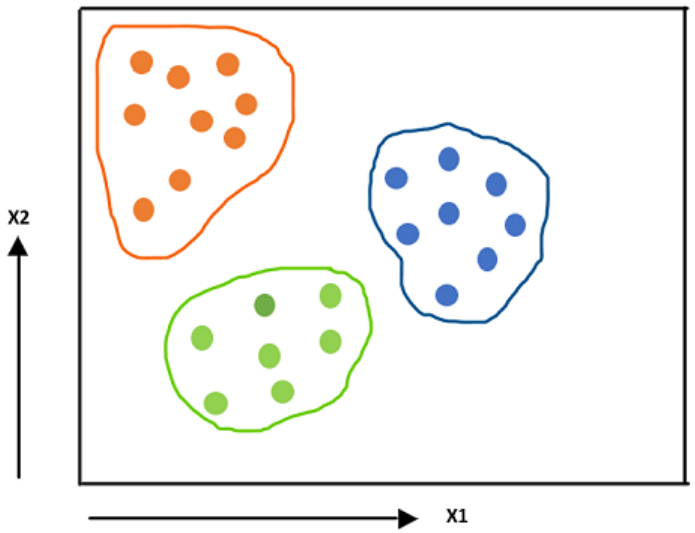
In this Lab, we shall be covering the role of unsupervised learning algorithms, their applications, and K-means clustering approach. On a brief note, Machine learning algorithms can be classified as Supervised and unsupervised learning. In the supervised learning, there will be the data set with input features and the target variable. The aim of the algorithm is to learn the dataset, find the hidden patterns in it and predict the target variable. The target variable can be continuous as in the case of Regression or discrete as in the case of Classification. Examples of Regression problems include housing price prediction, Stock market prediction, Air humidity, and temperature prediction. Examples of classification problems include Cancer prediction (either benign or malignant), email spam classification etc. This article demonstrates the next category of machine learning which is unsupervised learning.

So now one of the other areas of machine learning is unsupervised learning, where we will have the data, but we don't have any target variable as in the case of supervised learning. So the goal here is to observe the hidden patterns among the data and group them into clusters i.e the data points which have some shared properties will fall into one cluster or one alike group. So where is clustering used? Ever observed the google news where each category like sports, politics, movies, science have 1000's of news articles? This is clustering. The algorithm groups the news articles which have common features in them into separate groups to form a cluster. Other examples of clustering include social network analysis, market segmentation, recommendation systems etc.

One of the basic clustering algorithms is K-means clustering algorithm which we are going to discuss and implement from scratch in this Lab. Let's look at the final aim of the clustering from the two sample images and a practical example. The first image is the plot of the data set with features x1 and x2. You can see that data is unclusttered, so we can't conclude anything just by looking into the plot. The second image is obtained after performing Clustering, we can observe that data points which are close to each other are grouped as a cluster. In practical situations, this is can be treated as a case where we are dealing with customer data and are asked to segment the market based on customer's income and the number of previous transactions. Suppose here x1 feature is the annual income and x2 feature is the number of transactions, based on these features we can cluster the data and segment them into three categories like customers with low annual income but a high number of transactions (like orange cluster), customers with medium income and a medium number of transactions (like green cluster), customers with high income but low number of transactions (like blue cluster). Based on these segments, the marketing team of the company can redefine their marketing strategies to get more transactions by recommending the products, which each cluster's customer might buy.



Before Clustering



After Clustering

K-Means Clustering Intuition:

So far we have discussed the goal of clustering and a practical application, now it's time to dive into K-means clustering implementation and algorithm. As the name itself suggests, this algorithm will regroup **n** data points into **K** number of clusters. So given a large amount of data, we need to cluster this data into K clusters. But the problem is how to choose the number of clusters? Most of the times we will know what type of clusters we need to use, for example in the case of Google news, we already knew that say 4 types of clusters (sports, politics, science, movies) exist and news articles should be clustered into any of them. But in the case of market segmentation problem, as the goal is to cluster the type of customers, we don't know how many types of customers are present in the dataset (like rich, poor, who loves shopping, who doesn't love shopping etc.) i.e. we don't know exactly how many number of clusters that need to choose and we can't randomly assume the number of clusters based on some ground rules (like people with annual income greater than 'x' amount should be one cluster and lower than 'x' should fall into another). We shall discuss the solution to this problem, later in the article but now let's assume that we are going to segment our data into 3 clusters. Some of the mathematical terms involved in K-means clustering are centroids, Euclidian distance. On a quick note centroid of a data is the average or mean of the data and Euclidian distance is the distance between two points in the coordinate plane. Given two points A(x1,y1) and B(x2,y2), the Euclidian distance between these two points is :

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

While implementing, we won't consider the square root as we can compare the square of the distances and arrive at conclusions.

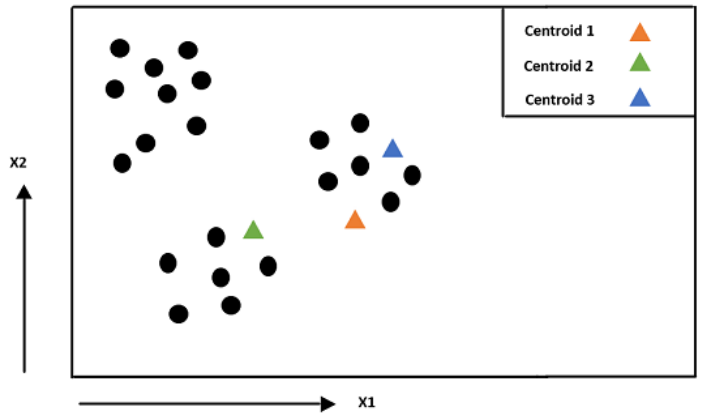
Algorithm:

Now let's start talking about the implementation steps. We shall be using either cluster centers or centroids words to describe the cluster centers.

Step 1:

Randomly initialize the cluster centers of each cluster from the data points. In fact, random initialization is not an efficient way to start with, as sometimes it leads to increased numbers of required clustering iterations to reach convergence, a greater overall runtime, and a less-efficient algorithm overall. So there are many

techniques to solve this problem like K-means++ etc. Let's also discuss one of the approaches to solve the problem of random initialization later in the article. So let's assume K=3, so we choose randomly 3 data points and assume them as centroids.



Random Initialization of Centroids

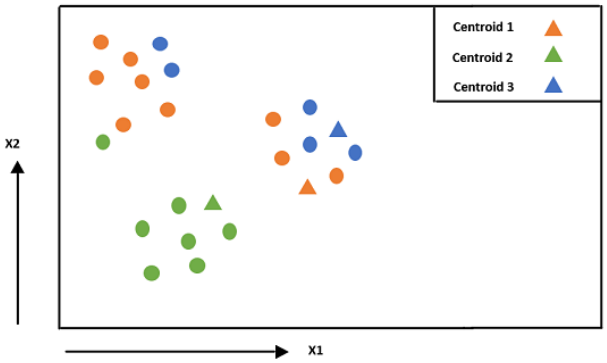
Here three cluster centers or centroids with the green, orange, and blue triangle markers are chosen randomly. Again this is not an efficient method to choose the initial cluster centers.

Step 2:

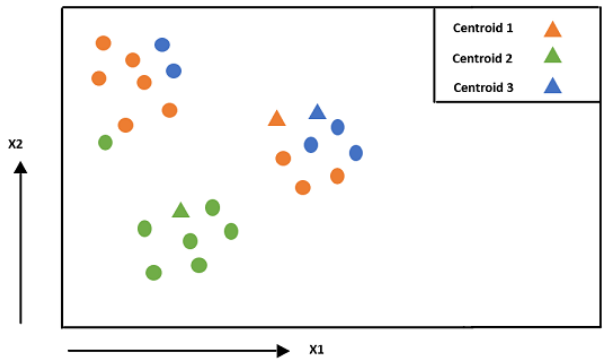
2a. For each data point, compute the Euclidian distance from all the centroids (3 in this case) and assign the cluster based on the minimal distance to all the centroids. In our example, we need to take each black dot, compute its Euclidian distance from all the centroids (green, orange and blue), and finally color the black dot to the color of the closest centroid.

2b. Adjust the centroid of each cluster by taking the average of all the data points which belong to that cluster on the basis of the computations performed in step 2a. In our example, as we have assigned all the data points to one of the clusters, we need to calculate the mean of all the individual clusters and move the centroid to calculated mean.

Repeat this process till clusters are well separated or convergence is achieved.

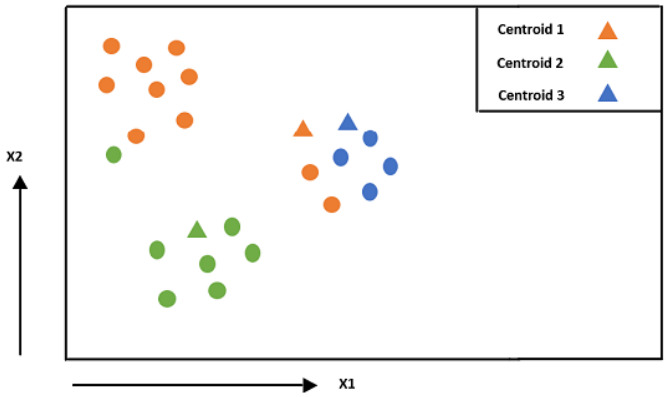


2.a Assign the centroids to each data point based on computed Euclidian distances

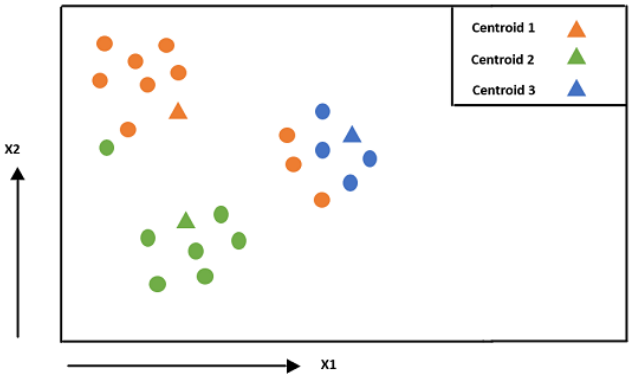


2b. Adjust the centroids by taking the average of all data points which belong to that Cluster

Repeat these steps until convergence is achieved:



Step 2.a



Step 2.b

Implementation from scratch:
Now as we are familiar with intuition, let's implement the algorithm in python from scratch. We need numpy, pandas and matplotlib libraries to improve the computational complexity and visualization of the results. The dataset which we are going to use is 'Mall_Customers.csv' and the link to the dataset can be found in the GitHub page. (Note: The dataset has been downloaded from SuperDataScience website).

Let's read the dataset and get the data examples

```
#import libraries
import numpy as np
```

```
import matplotlib.pyplot as plt
import pandas as pd
import random as rd

dataset=pd.read_csv('Mall_Customers.csv')
dataset.describe()
```

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

For visualization convenience, we are going to take Annual Income and Spending score as our data.

```
X = dataset.iloc[:, [3, 4]].values
```

Now X is two matrix of shape (200,2).
Next step is to choose the number of iterations which might guarantee convergence. We need to try many possibilities to find optimum number of iterations required for convergence. There is no need to choose a very large number because say at 100th iteration, if the centroids arrived to their true location or best possible location, even after performing 1000 extra iterations, the algorithm will give same results. So for convenience let's start by choosing number of iterations as 100.

```
m=X.shape[0] #number of training examples
n=X.shape[1] #number of features. Here n=2
n_iter=100
```

Next step is to choose number of clusters K. Let's take 5 as K and as it has been mentioned earlier we are going to see a method later in the article, which will find us the optimum number of clusters K. We are ready to implement our K-means Clustering steps. Let's proceed:

Step 1: Initialize the centroids randomly from the data points:

```
Centroids=np.array([]).reshape(n,0)
```

Centroids is a n x K dimentional matrix, where each column will be a centroid for one cluster.

```
for i in range(K):

    rand=rd.randint(0,m-1)
```

```
    Centroids=np.c_[Centroids,X[rand]]
```

Step 2.a: For each training example compute the Euclidian distance from the centroid and assign the cluster based on the minimal distance

The output of our algorithm should be a dictionary with cluster number as Keys and the data points which belong to that cluster as values. So let's initialize the dictionary.

```
Output={}
```

We find the euclidian distance from each point to all the centroids and store in a m X K matrix. So every row in EuclidianDistance matrix will have distances of that particular data point from all the centroids. Next, we shall find the minimum distance and store the index of the column in a vector C.

```
EuclidianDistance=np.array([]).reshape(m,0)

for k in range(K):
       tempDist=np.sum((X-Centroids[:,k])**2,axis=1)
        EuclidianDistance=np.c_[EuclidianDistance,tempDist]
C=np.argmin(EuclidianDistance,axis=1)+1
```

Step 2.b: We need to regroup the data points based on the cluster index C and store in the Output dictionary and also compute the mean of separated clusters and assign it as new centroids. Y is a temporary dictionary which stores the solution for one particular iteration.

```
Y={}
for k in range(K):
    Y[k+1]=np.array([]).reshape(2,0)
for i in range(m):
    Y[C[i]]=np.c_[Y[C[i]],X[i]]
for k in range(K):
    Y[k+1]=Y[k+1].T
for k in range(K):
    Centroids[:,k]=np.mean(Y[k+1],axis=0)
```

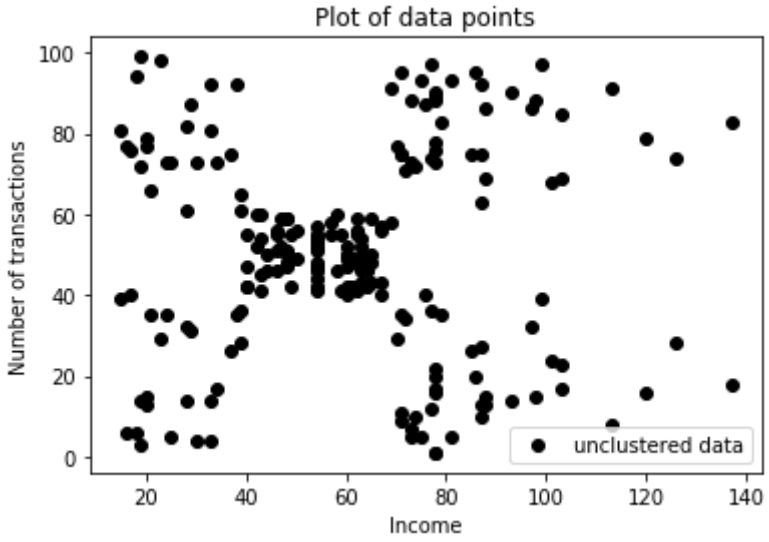
Now we need to repeat step 2 till convergence is achieved. In other words, we loop over n_iter and repeat the step 2.a and 2.b as shown:

```
for i in range(n_iter):
     #step 2.a
      EuclidianDistance=np.array([]).reshape(m,0)
      for k in range(K):
          tempDist=np.sum((X-Centroids[:,k])**2,axis=1)
          EuclidianDistance=np.c_[EuclidianDistance,tempDist]
      C=np.argmin(EuclidianDistance,axis=1)+1
     #step 2.b
      Y={}
      for k in range(K):
          Y[k+1]=np.array([]).reshape(2,0)
```

```
    for i in range(m):
        Y[C[i]]=np.c_[Y[C[i]],X[i]]

    for k in range(K):
        Y[k+1]=Y[k+1].T

    for k in range(K):
        Centroids[:,k]=np.mean(Y[k+1],axis=0)
    Output=Y
```

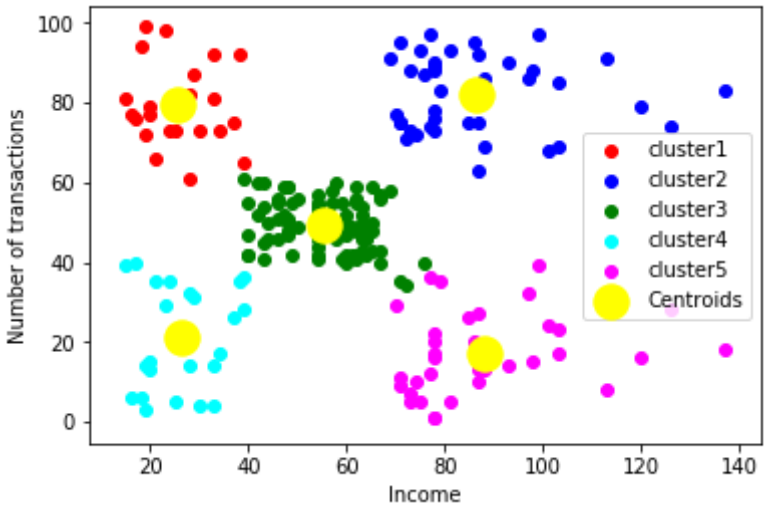
Now it's time to visualize the algorithm and notice how the original data is clustered. To start with, let's scatter the original unclustered data first.

```
plt.scatter(X[:,0],X[:,1],c='black',label='unclustered data')
plt.xlabel('Income')
plt.ylabel('Number of transactions')
plt.legend()
plt.title('Plot of data points')
plt.show()
```



Now let's plot the clustered data:

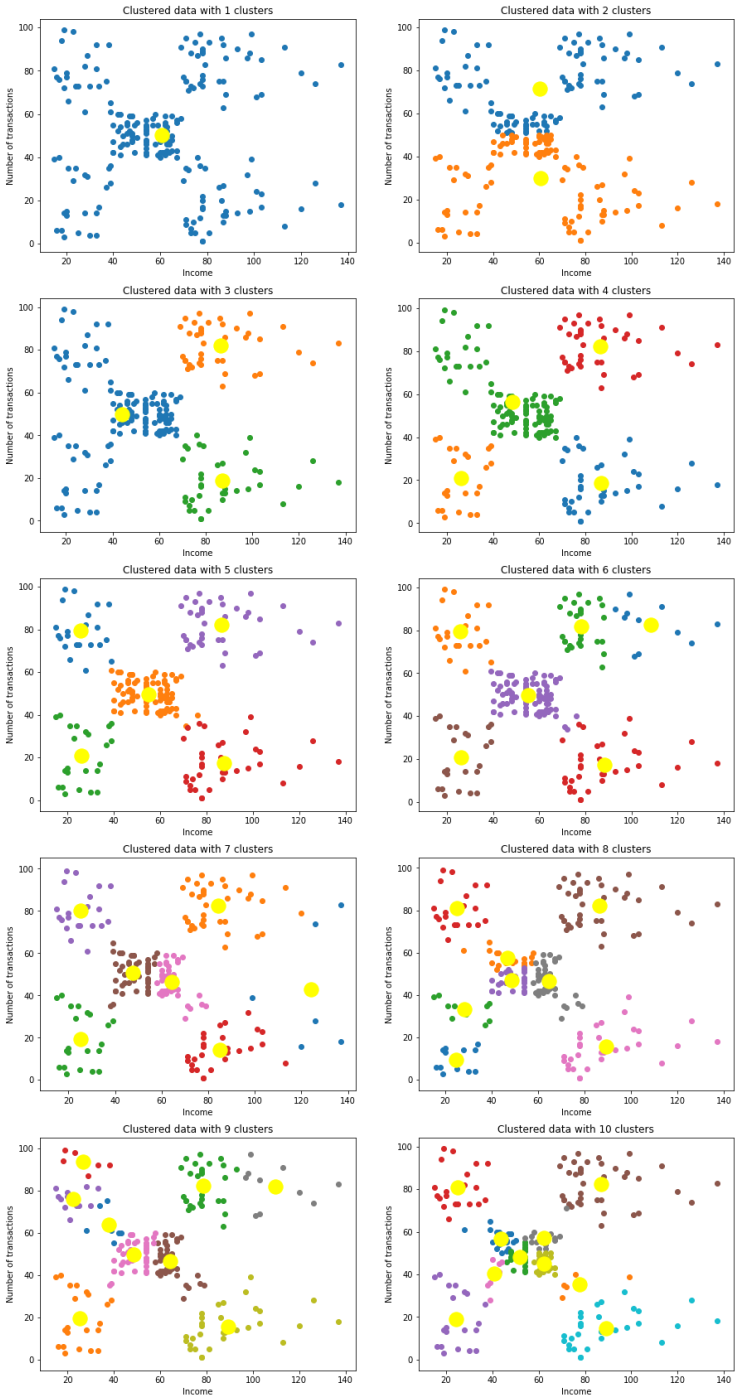
```
color=['red','blue','green','cyan','magenta']
labels=['cluster1','cluster2','cluster3','cluster4','cluster5']
for k in range(K):
    plt.scatter(Output[k+1][:,0],Output[k+1][:,1],c=color[k],label=labels[k])
plt.scatter(Centroids[0,:],Centroids[1,:],s=300,c='yellow',label='Centroids')
plt.xlabel('Income')
plt.ylabel('Number of transactions')
plt.legend()
plt.show()
```



Wow, it's so beautiful and informative too! Our data has become a clustered data from unclustered raw data. We can observe that there are five categories of clusters which are :

1. Customers with low income but a High number of transactions (For these type may be the company can recommend products with low price) — Red cluster
2. Customers with low income and a low number of transactions (Maybe these type of customers are too busy saving their money) — Cyan Cluster
3. Customers with medium income and a medium number of transactions — Green Cluster
4. Customers with High income and a low number of transactions — Magenta Cluster
5. Customers with High income and a High number of transactions — Blue cluster.

So the company can divide their customers into 5 classes and design different strategies for a different type of customers to increase their sales. So far so good right, but how do we arrive at the conclusion that there should be 5 number of clusters? To find out how, let's visualize the clustered data with different clusters starting from 1 to 10.



Now we see a lot of plots showing the clustered data with a different number of clusters. So the question is what is the best value for K? Suppose we have **n** data points, and we choose **n** clusters i.e every data point is a cluster, is it a good model? No, obviously not and the converse is also not a good model i.e one cluster for **n** data points. So how do we find the appropriate K? The answer lies in the fact that for every data point, it's cluster center should be the nearest or in other words, "**Sum of squares of distances of every data point from its corresponding cluster centroid should be as minimum as possible".** This statement will again contradict with the fact that if all data points are treated as individual clusters, then the sum of squares of distances will be 0. So to counter this fact, we use a method called ELBOW method to find the appropriate number of clusters. The parameter which will be taken into consideration is **Sum of squares of distances of every data point from its corresponding cluster centroid** which is called **WCSS** (**Within-Cluster Sums of Squares).**

Steps involved in ELBOW method are:

1. Perform K means clustering on different values of K ranging from 1 to any upper limit. Here we are taking the upper limit as 10.
2. For each K, calculate WCSS
3. Plot the value for WCSS with the number of clusters K.
4. The location of a bend (knee) in the plot is generally considered as an indicator of the appropriate number of clusters. i.e the point after which WCSS doesn't decrease more rapidly is the appropriate value of K.

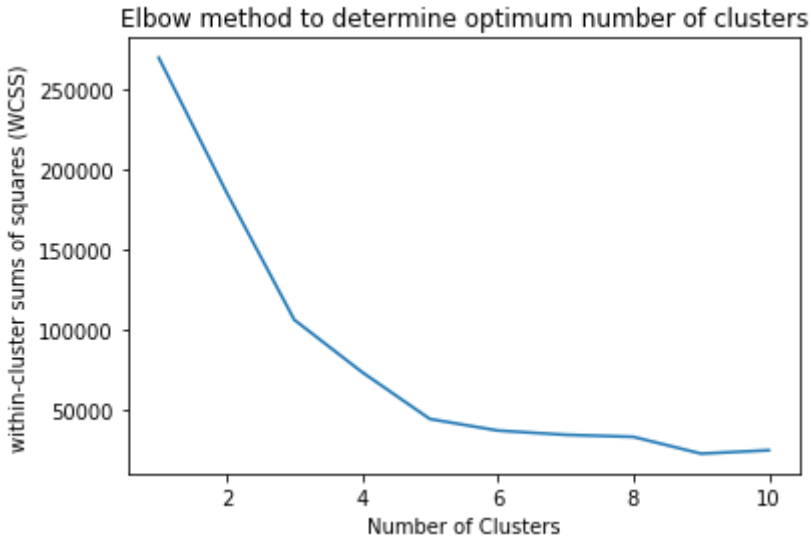
So let's implement this and see how K=5 is the best.

Note: I have converted the algorithm into an object-oriented manner. Kmeans is the name of the class, fit method will perform the Kmeans Clustering, and predict will return the Output dictionary and Centroid matrix.

```
# Using the elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
```

Now let's plot the WCSS_array with clusters from 1 to 10.

```
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



Now if we observe the point after which there isn't a sudden change in WCSS is K=5. So we choose K=5 as an appropriate number of clusters. So for any given dataset, we need to find the appropriate number of clusters first and start making predictions and alterations to the conclusions.

So are we done yet? No. we have one more problem left which is random initialization. This is certainly a problem because if two initial centroids are very near, it would take a lot of iterations for the algorithm to converge and so something need to be done in order to make sure that initial centroids are far apart from each other. This is Described as KMeans++ algorithm, where only the initialization of the centroids will change, rest everything is similar to conventional KMeans.

The objective of the KMeans++ initialization is that chosen centroids should be far from one another. The first cluster center is chosen uniformly at random from the data points that are being clustered, after which each subsequent cluster center is chosen from the remaining data points with probability proportional to its squared distance from the point's closest existing cluster center. This will be understood well if we represent it as steps.

Steps involved in Kmeans++ initialization are:

1. Randomly select the first cluster center from the data points and append it to the centroid matrix.
2. Loop over the number of Centroids that need to be chosen (K):
3. For each data point calculate the euclidian distance square from already chosen centroids and append the minimum distance to a Distance array.
4. Calculate the probabilities of choosing the particular data point as the next centroid by dividing the Distance array elements with the sum of Distance array. Let's call this probability distribution as PD.
5. Calculate the cumulative probability distribution from this PD distribution. We knew that the cumulative probability distribution ranges from 0 to 1.
6. Select a random number between 0 to 1, get the index (i) of the cumulative probability distribution which is just greater than the chosen random number and assign the data point corresponding to the selected index (i).
7. Repeat the process until we have K number of cluster centers.

Here is a one-dimensional example. Our observations are [0, 1, 2, 3, 4]. Let the first center, c1 , be 0. The probability that the next cluster center, c2, is x is proportional to ||c1-x||^2. So, P(c2 = 1) = 1a, P(c2 = 2) = 4a, P(c2 = 3) = 9a, P(c2 = 4) = 16a, where a = 1/(1+4+9+16).

Suppose c2=4. Then, P(c3 = 1) = 1a, P(c3 = 2) = 4a, P(c3 = 3) = 1a, where a = 1/(1+4+1).

Example probs=[0.1,0.2,0.3,0.4], cumprobs=[0.1,0.3,0.6,1.0]. Let's choose a random number r between 0 and 1. if chosen r<cumprobs[0], then this event has probability 0.1, else if chosen r<cumprobs[1], then this event has probability 0.3 and so the data point corresponding to the index 1 is chosen as next cluster center.

More about Kmeans++ can be found at Wikipedia-link and the above example has been referenced from StackOverFlow-question.

Now let's implement this initialization and compare the results with random initialization.

1. Randomly select the first cluster center from the data points and append it to the centroid matrix.

Randomly select the first cluster center from the data points and append it to the centroid matrix.

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

Centroid=np.array([X[i]])
```

2. Let's put everything into for loop later.

3. For each data point calculate the euclidian distance square from already chosen centroids and append the minimum distance to a Distance array.

```
D=np.array([])

for x in X:

    D=np.append(D,np.min(np.sum((x-Centroid)**2)))
```

4. Calculate the probabilities of choosing the particular data point as the next centroid by dividing the Distance array elements with the sum of Distance array. Let's call this probability distribution as P.

```
prob=D/np.sum(D)
```

We can't just select the data point with the highest probability as the cluster center because there might be a chance that the selected cluster will coincide with the previous cluster centers.

5. Calculate the cumulative probability distribution from this PD distribution. We knew that the cumulative probability distribution ranges from 0 to 1.

```
cummulative_prob=np.cumsum(prob)
```

6. Select a random number between 0 to 1, get the index (i) of the cumulative probability distribution which is just greater than the chosen random number and assign the data point corresponding to the selected index (i).

```
r=rd.random()

i=0

for j,p in enumerate(cummulative_prob):

    if r<p:

        i=j
```

```
        break
Centroid=np.append(Centroid,[X[i]],axis=0)
```

Putting together in a for loop for K clusters:

```
i=rd.randint(0,X.shape[0])
Centroid=np.array([X[i]])
K=5
for k in range(1,K):
    D=np.array([])
    for x in X:
        D=np.append(D,np.min(np.sum((x-Centroid)**2)))
    prob=D/np.sum(D)
    cummulative_prob=np.cumsum(prob)
    r=rd.random()
    i=0
    for j,p in enumerate(cummulative_prob):
        if r<p:
            i=j
            break
    Centroid=np.append(Centroid,[X[i]],axis=0)
```

The final part of the lab is to visualize the results and see how Kmeans++ solves the problem of random initialization.

Centroid_rand is the randomly chosen cluster centers and Centroid is the ones obtained from Kmeans++.

```
for i in range(K):
    rand=rd.randint(0,m-1)
    Centroids_rand=np.c_[Centroids_rand,X[rand]]
plt.scatter(X[:,0],X[:,1])
```

```
plt.scatter(Centroid_temp[:,0],Centroid_temp[:,1],s=200,color='yellow')

plt.scatter(Centroids_rand[0,:],Centroids_rand[1,:],s=300,color='black')
```

Yellow dots are the cluster centers chosen using Kmeans++ algorithm and Black dots are cluster centers chosen using Random initialization.

It is clear that yellow dots are spread wide over the plot and black dots are much closer. This will definitely show an impact in improving the computational complexity of the Kmeans clustering algorithm.

Finally, let's implement all these steps using the sklearn library so that we can compare the results:

```
#lets implement the same algorithm using sklearn libraries

# Using the elbow method to find the optimal number of clusters

from sklearn.cluster import KMeans

wcss = []

for i in range(1, 11):

    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)

    kmeans.fit(X)

    wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)

plt.title('The Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()
```

We can observe that the curve is much smoother than our implemented curve.

Centroid Initialization in sklearn happens using k-means++ parameter as shown.

```
# Fitting K-Means to the dataset

kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)

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 = 'Cluster 1')

plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue',
label = 'Cluster 2')

plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c =
'green', label = 'Cluster 3')

plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan',
label = 'Cluster 4')

plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c =
'magenta', label = 'Cluster 5')

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
s = 300, c = 'yellow', label = 'Centroids')

plt.title('Clusters of customers')

plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.legend()

plt.show()
```

As a conclusion, in this Lab,

We have learned K-means Clustering from scratch and implemented the algorithm in python.

Solved the problem of choosing the number of clusters based on the Elbow method.

Solved the problem of random initialization using KMeans++ algorithm.

Lab Task :

1. Implement K means with IRIS Data Set. (Both with manual calculation code and then use sklearn and compare the results of both)

LAB # 10 Implementation of a Feed Forward Neural Network and Back Propagation Algorithm.

The backpropagation algorithm is the classical feed-forward artificial neural network. It is the technique still used to train large deep learning networks. In this lab, you will discover how to implement the backpropagation algorithm with Python.

After completing this Lab, you will know:

How to forward-propagate an input to calculate an output.

How to back-propagate error and train a network.

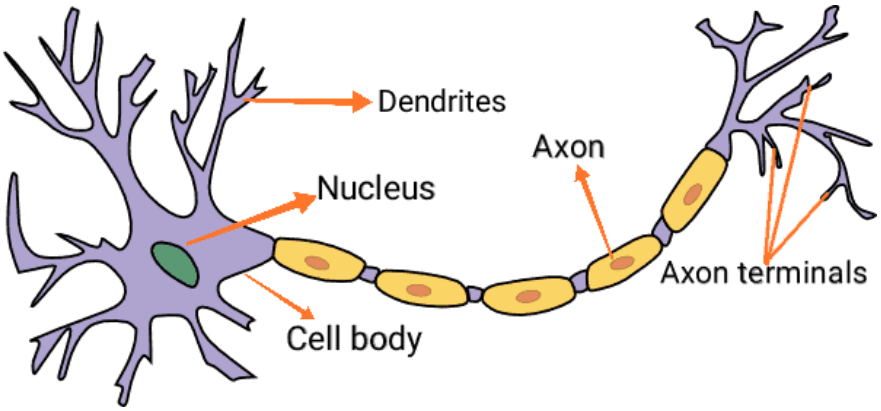
How to apply the backpropagation algorithm to a real-world predictive modeling problem.

Description

This section provides a brief introduction to the Backpropagation Algorithm and the Wheat Seeds dataset That we will be using in this Lab.

Backpropagation Algorithm

The Backpropagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks. Feed-forward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.



The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state. Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more

layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer. Backpropagation can be used for both classification and regression problems, but we will focus on classification. In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

Wheat Seeds Dataset

The seeds dataset involves the prediction of species given measurements seeds from different varieties of wheat. There are 201 records and 7 numerical input variables. It is a classification problem with 3 output classes. The scale for each numeric input value vary, so some data normalization may be required for use with algorithms that weight inputs like the backpropagation algorithm. Below is a sample of the first 5 rows of the dataset.

Using the Zero Rule algorithm that predicts the most common class value, the baseline accuracy for the problem

```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
```

is 28.095%. You can learn more and download the seeds dataset from the UCI Machine Learning Repository. Download the seeds dataset and place it into your current working directory with the filename seeds_dataset.csv. The dataset is in tab-separated format, so you must convert it to CSV using a text editor or a spreadsheet program.

Implementation

This Implementation is broken down into 6 parts:

1. Initialize Network.
2. Forward Propagate.
3. Back Propagate Error.
4. Train Network.
5. Predict.
6. Seeds Dataset Case Study.

These steps will provide the foundation that you need to implement the backpropagation algorithm from scratch and apply it to your own predictive modeling problems.

Initialize Network

Let's start with something easy, the creation of a new network ready for training. Each neuron has a set of weights that need to be maintained. One weight for each input connection and an additional weight for the bias. We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent

each neuron and store properties by names such as 'weights' for the weights. A network is organized into layers. The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value. We will organize layers as arrays of dictionaries and treat the whole network as an array of layers. It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1.

Below is a function named initialize_network() that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs. You can see that for the hidden layer we create n_hidden neurons and each neuron in the hidden layer has n_inputs + 1 weights, one for each input column in a dataset and an additional one for the bias. You can also see that the output layer that connects to the hidden layer has n_outputs neurons, each with n_hidden + 1 weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

Below is a complete example that creates a small network.

```
from random import seed
from random import random
from math import exp
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for
i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]}
     for i in range(n_outputs)]
    network.append(output_layer)
    return network

seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
    print(layer)
```

Running the example, you can see that the code prints out each layer one by one. You can see the hidden layer has one neuron with 2 input weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights':
[0.4494910647887381, 0.651592972722763]}]
```

Now that we know how to create and initialized a network, let's see how we can use it to calculate an output.

Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We call this forward-propagation. It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data. We can break forward propagation down into three parts:

1. Neuron Activation.
2. Neuron Transfer.
3. Forward Propagation.

Neuron Activation

The first step is to calculate the activation of one neuron given an input. The input could be a row from our training dataset, as in the case of the hidden layer. It may also be the outputs from each neuron in the hidden layer, in the case of the output layer. Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression. Below is an implementation of this in a function named activate(). You can see that the function assumes that the bias is the last weight in the list of weights. This helps here and later to make the code easier to read.

```
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
```

Now, let's see how to use the neuron activation.

Neuron Transfer

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used. It is traditional to use the sigmoid activation function, but you can also use the tanh (hyperbolic tangent) function to transfer outputs. More recently, the rectifier transfer function has been

popular with large deep learning networks. The sigmoid activation function looks like an S shape, it's also called the logistic function. It can take any input value and produce a number between 0 and 1 on an S-curve. It is also a function of which we can easily calculate the derivative (slope) that we will need later when backpropagating error. We can transfer an activation function using the sigmoid function as follows:

```
output = 1 / (1 + e^(-activation))
```

Where e is the base of the natural logarithms (Euler's number). Below is a function named transfer() that implements the sigmoid equation.

```
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
```

Now that we have the pieces, let's see how they are used.

Forward Propagation

Forward propagating an input is straightforward. We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. Below is a function named forward_propagate() that implements the forward propagation for a row of data from our dataset with our neural network. You can see that a neuron's output value is stored in the neuron with the name 'output'. You can also see that we collect the outputs for a layer in an array named new_inputs that becomes the array inputs and is used as inputs for the following layer. The function returns the outputs from the last layer also called the output layer.

```
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
```

Let's put all of these pieces together and test out the forward propagation of our network. We define our network inline with one hidden neuron that expects 2 input values and an output layer with two neurons Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer

has two neurons, we get a list of two numbers as output. The actual output values are just nonsense for now, but next, we will start to learn how to make the weights in the neurons more useful.

```
row = [1, 0, None]
output = forward_propagate(network, row)
print(output)
```

```
[0.6629970129852887, 0.7253160725279748]
```

Back Propagate Error

The backpropagation algorithm is named for the way in which weights are trained. Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go. The math for backpropagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form. This part is broken down into two sections.

1. Transfer Derivative.
2. Error Backpropagation.

Transfer Derivative

Given an output value from a neuron, we need to calculate it's slope. We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

```
# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)
```

Now, let's see how this can be used.

Error Backpropagation

The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network. The error for a given neuron can be calculated as follows:

```
error = (expected - output) * transfer_derivative(output)
```

Where expected is the expected output value for the neuron, output is the output value for the neuron and transfer_derivative() calculates the slope of the neuron's output value, as shown above. This error calculation is used for neurons in the output layer. The expected value is the class value itself. In the hidden layer, things are a little more complicated. The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the

neurons in the hidden layer. The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

```
error = (weight_k * error_j) * transfer_derivative(output)
```

Where error_j is the error signal from the jth neuron in the output layer, weight_k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron. Below is a function named backward_propagate_error() that implements this procedure. You can see that the error signal calculated for each neuron is stored with the name 'delta'. You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have 'delta' values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name 'delta' to reflect the change the error implies on the neuron (e.g. the weight delta). You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number j is also the index of the neuron's weight in the output layer neuron['weights'][j].

```
# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron['delta'] = errors[j] *
transfer_derivative(neuron['output'])
```

Let's put all of the pieces together and see how it works. We define a fixed neural network with output values and backpropagate an expected output pattern.

```
# test backpropagation of error
expected=[0,1]
backward_propagate_error(network, expected)
for layer in network:
    print(layer)
```

Running the example prints the network after the backpropagation of error is complete. You can see that error values are calculated and stored in the neurons for the output layer and the hidden layer.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614],
'output': 0.7105668883115941}]
[{'weights': [0.2550690257394217, 0.49543508709194095], 'output':
0.6629970129852887, 'delta': -0.14813473120687762}, {'weights':
[0.4494910647887381, 0.651592972722763], 'output': 0.7253160725279748,
'delta': 0.05472601157879688}]
```

LAB # 11 Back Propagation Algorithm.

The backpropagation algorithm is the classical feed-forward artificial neural network. It is the technique still used to train large deep learning networks. In this lab, you will discover how to implement the backpropagation algorithm with Python.

After completing this Lab, you will know:

How to forward-propagate an input to calculate an output.

How to back-propagate error and train a network.

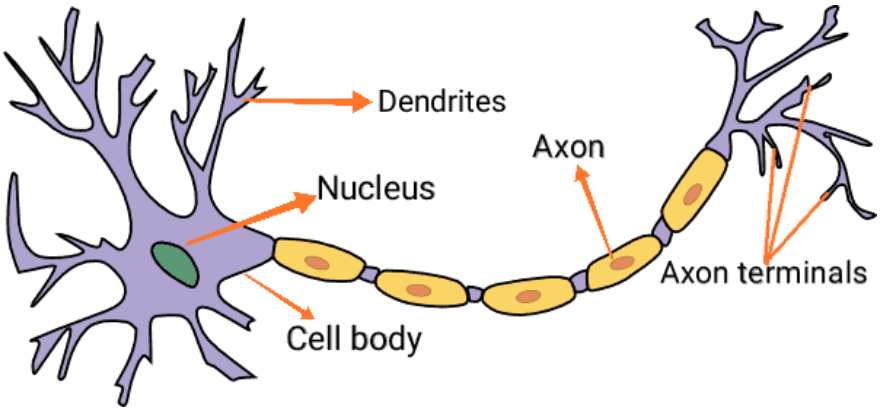
How to apply the backpropagation algorithm to a real-world predictive modeling problem.

Description

This section provides a brief introduction to the Backpropagation Algorithm and the Wheat Seeds dataset That we will be using in this Lab.

Backpropagation Algorithm

The Backpropagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks. Feed-forward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.



The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state. Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one

hidden layer, and one output layer. Backpropagation can be used for both classification and regression problems, but we will focus on classification. In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

Wheat Seeds Dataset

The seeds dataset involves the prediction of species given measurements seeds from different varieties of wheat. There are 201 records and 7 numerical input variables. It is a classification problem with 3 output classes. The scale for each numeric input value vary, so some data normalization may be required for use with algorithms that weight inputs like the backpropagation algorithm. Below is a sample of the first 5 rows of the dataset.

Using the Zero Rule algorithm that predicts the most common class value, the baseline accuracy for the problem

```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
```

is 28.095%. You can learn more and download the seeds dataset from the UCI Machine Learning Repository. Download the seeds dataset and place it into your current working directory with the filename seeds_dataset.csv. The dataset is in tab-separated format, so you must convert it to CSV using a text editor or a spreadsheet program.

Implementation

This Implementation is broken down into 6 parts:

1. Initialize Network.
2. Forward Propagate.
3. Back Propagate Error.
4. Train Network.
5. Predict.
6. Seeds Dataset Case Study.

These steps will provide the foundation that you need to implement the backpropagation algorithm from scratch and apply it to your own predictive modeling problems.

Initialize Network

Let's start with something easy, the creation of a new network ready for training. Each neuron has a set of weights that need to be maintained. One weight for each input connection and an additional weight for the bias. We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent each neuron and store properties by names such as 'weights' for the weights. A network is organized into layers.

The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value. We will organize layers as arrays of dictionaries and treat the whole network as an array of layers. It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1.

Below is a function named initialize_network() that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs. You can see that for the hidden layer we create n_hidden neurons and each neuron in the hidden layer has n_inputs + 1 weights, one for each input column in a dataset and an additional one for the bias. You can also see that the output layer that connects to the hidden layer has n_outputs neurons, each with n_hidden + 1 weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

Below is a complete example that creates a small network.

```
from random import seed
from random import random
from math import exp
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for
i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]}
     for i in range(n_outputs)]
    network.append(output_layer)
    return network

seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
    print(layer)
```

Running the example, you can see that the code prints out each layer one by one. You can see the hidden layer has one neuron with 2 input weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights':
[0.4494910647887381, 0.651592972722763]}]
```

Now that we know how to create and initialized a network, let's see how we can use it to calculate an output.

Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We call this forward-propagation. It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data. We can break forward propagation down into three parts:

1. Neuron Activation.
2. Neuron Transfer.
3. Forward Propagation.

Neuron Activation

The first step is to calculate the activation of one neuron given an input. The input could be a row from our training dataset, as in the case of the hidden layer. It may also be the outputs from each neuron in the hidden layer, in the case of the output layer. Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression. Below is an implementation of this in a function named activate(). You can see that the function assumes that the bias is the last weight in the list of weights. This helps here and later to make the code easier to read.

```
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
```

Now, let's see how to use the neuron activation.

Neuron Transfer

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used. It is traditional to use the sigmoid activation function, but you can also use the tanh (hyperbolic tangent) function to transfer outputs. More recently, the rectifier transfer function has been popular with large deep learning networks. The sigmoid activation function looks like an S shape, it's also called the logistic function. It can take any input value and produce a number between 0 and 1 on an S-curve. It is also

a function of which we can easily calculate the derivative (slope) that we will need later when backpropagating error. We can transfer an activation function using the sigmoid function as follows:

```
output = 1 / (1 + e^(-activation))
```

Where e is the base of the natural logarithms (Euler's number). Below is a function named transfer() that implements the sigmoid equation.

```
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
```

Now that we have the pieces, let's see how they are used.

Forward Propagation

Forward propagating an input is straightforward. We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. Below is a function named forward_propagate() that implements the forward propagation for a row of data from our dataset with our neural network. You can see that a neuron's output value is stored in the neuron with the name 'output'. You can also see that we collect the outputs for a layer in an array named new_inputs that becomes the array inputs and is used as inputs for the following layer. The function returns the outputs from the last layer also called the output layer.

```
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
```

Let's put all of these pieces together and test out the forward propagation of our network. We define our network inline with one hidden neuron that expects 2 input values and an output layer with two neurons Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer has two neurons, we get a list of two numbers as output. The actual output values are just nonsense for now, but next, we will start to learn how to make the weights in the neurons more useful.

```
row = [1, 0, None]
```

```
output = forward_propagate(network, row)
print(output)
```

```
[0.6629970129852887, 0.7253160725279748]
```

Back Propagate Error

The backpropagation algorithm is named for the way in which weights are trained. Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go. The math for backpropagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form. This part is broken down into two sections.

1. Transfer Derivative.
2. Error Backpropagation.

Transfer Derivative

Given an output value from a neuron, we need to calculate it's slope. We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

```
# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)
```

Now, let's see how this can be used.

Error Backpropagation

The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network. The error for a given neuron can be calculated as follows:

```
error = (expected - output) * transfer_derivative(output)
```

Where expected is the expected output value for the neuron, output is the output value for the neuron and transfer_derivative() calculates the slope of the neuron's output value, as shown above. This error calculation is used for neurons in the output layer. The expected value is the class value itself. In the hidden layer, things are a little more complicated. The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer. The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

```
error = (weight_k * error_j) * transfer_derivative(output)
```

Where error_j is the error signal from the jth neuron in the output layer, weight_k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron. Below is a function named backward_propagate_error() that implements this procedure. You can see that the error signal calculated for each neuron is stored with the name 'delta'. You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have 'delta' values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name 'delta' to reflect the change the error implies on the neuron (e.g. the weight delta). You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number j is also the index of the neuron's weight in the output layer neuron['weights'][j].

```
# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron['delta'] = errors[j] *
transfer_derivative(neuron['output'])
```

Let's put all of the pieces together and see how it works. We define a fixed neural network with output values and backpropagate an expected output pattern.

```
# test backpropagation of error
```

```
expected=[0,1]
backward_propagate_error(network, expected)
for layer in network:
    print(layer)
```

Running the example prints the network after the backpropagation of error is complete. You can see that error values are calculated and stored in the neurons for the output layer and the hidden layer.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614],
'output': 0.7105668883115941}]
[{'weights': [0.2550690257394217, 0.49543508709194095], 'output':
0.6629970129852887, 'delta': -0.14813473120687762}, {'weights':
[0.4494910647887381, 0.651592972722763], 'output': 0.7253160725279748,
'delta': 0.05472601157879688}]
```

Train Network

The network is trained using stochastic gradient descent. This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, backpropagating the error and updating the network weights. This part is broken down into two sections:

1. Update Weights.
2. Train Network.

1. Update Weights

Once errors are calculated for each neuron in the network via the back propagation method above, they can be used to update weights. Network weights are updated as follows:

weight = weight + learning_rate * error * input

Where weight is a given weight, learning_rate is a parameter that you must specify, error is the error calculated by the backpropagation procedure for the neuron and input is the input value that caused the error. The same procedure can be used for updating the bias weight, except there is no input term, or input is the fixed value of 1.0. Learning rate controls how much to change the weight to correct for the error. For example, a value of 0.1 will update the weight 10% of the amount that it possibly could be updated. Small learning rates are preferred that cause slower learning over a large number of training iterations. This increases the likelihood of the network finding a good set of weights across all layers rather than the fastest set of weights that minimize error (called premature convergence). Below is a function named update_weights() that updates the weights for a network given an input row of data, a learning rate and

assume that a forward and backward propagation have already been performed. Remember that the input for the output layer is a collection of outputs from the hidden layer.

```
# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']
```

Now we know how to update network weights, let's see how we can do it repeatedly.

2. Train Network

As mentioned, the network is updated using stochastic gradient descent. This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset. Because updates are made for each training pattern, this type of learning is called online learning. If errors were accumulated across an epoch before updating the weights, this is called batch learning or batch gradient descent. Below is a function that implements the training of an already initialized neural network with a given training dataset, learning rate, fixed number of epochs and an expected number of output values. The expected number of output values is used to transform class values in the training data into a one hot encoding. That is a binary vector with one column for each class value to match the output of the network. This is required to calculate the error for the output layer. You can also see that the sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch.

```
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
```

```
            sum_error += sum([(expected[i]-outputs[i])**2 for i in
range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate,
sum_error))
```

We now have all of the pieces to train the network. We can put together an example that includes everything we've seen so far including network initialization and train a network on a small dataset. Below is a small contrived dataset that we can use to test out training our neural network.

X1	X2	Y
2.7810836	2.550537003	0
1.465489372	2.362125076	0
3.396561688	4.400293529	0
1.38807019	1.850220317	0
3.06407232	3.005305973	0
7.627531214	2.759262235	1
5.332441248	2.088626775	1
6.922596716	1.77106367	1
8.675418651	-0.242068655	1
7.673756466	3.508563011	1

Below is the complete example. We will use 2 neurons in the hidden layer. It is a binary classification problem (2 classes) so there will be two neurons in the output layer. The network will be trained for 20 epochs with a learning rate of 0.5, which is high because we are training for so few iterations.

```
seed(1)
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
```

```
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
    print(layer)
```

```
epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delta': 0.0026279652850863837}]
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta': 0.03803132596437354}]
```

Running the example first prints the sum squared error each training epoch. We can see a trend of this error decreasing with each epoch. Once trained, the network is printed, showing the learned weights. Also

still in the network are output and delta values that can be ignored. We could update our training function to delete these data if we wanted.

Once a network is trained, we need to use it to make predictions.

Predict

Making predictions with a trained neural network is easy enough. We have already seen how to forward-propagate an input pattern to get an output. This is all we need to do to make a prediction. We can use the output values themselves directly as the probability of a pattern belonging to each output class. It may be more useful to turn this output back into a crisp class prediction. We can do this by selecting the class value with the larger probability. This is also called the arg max function. Below is a function named predict() that implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

```
def predict(network, row):
	outputs = forward_propagate(network, row)
	return outputs.index(max(outputs))
```

We can put this together with our code above for forward propagating input and with our small contrived dataset to test making predictions with an already-trained network. The example hardcodes a network trained from the previous step

```
dataset = [[2.7810836,2.550537003,0],
	[1.465489372,2.362125076,0],
	[3.396561688,4.400293529,0],
	[1.38807019,1.850220317,0],
	[3.06407232,3.005305973,0],
	[7.627531214,2.759262235,1],
	[5.332441248,2.088626775,1],
	[6.922596716,1.77106367,1],
	[8.675418651,-0.242068655,1],
	[7.673756466,3.508563011,1]]
network = [[{'weights': [-1.482313569067226, 1.8308790073202204, 1.078381922048799]}, {'weights': [0.23244990332399884, 0.3621998343835864, 0.40289821191094327]}],
	[{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757805829]}, {'weights': [-2.429350576245497, 0.8357651039198697, 1.0699217181280656]}]]
for row in dataset:
	prediction = predict(network, row)
	print('Expected=%d, Got=%d' % (row[-1], prediction))
```

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
```

```
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

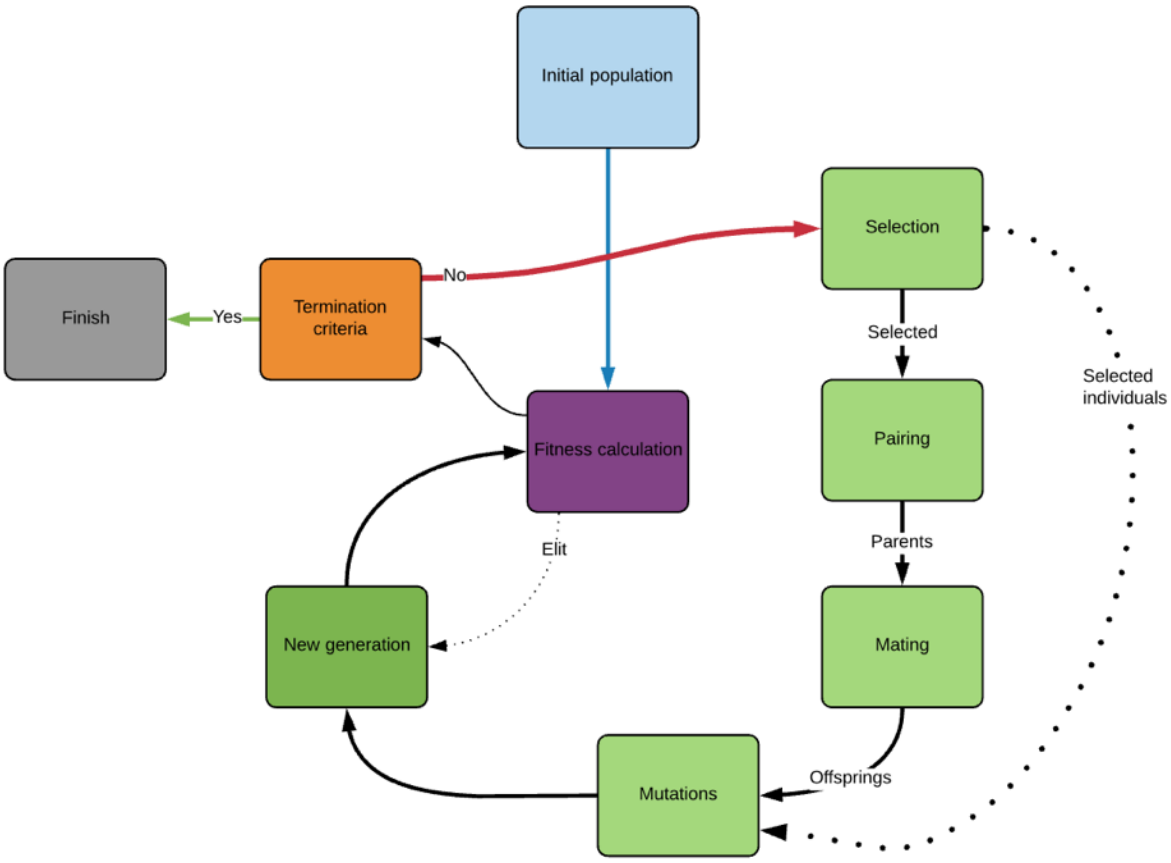
Running the example prints the expected output for each record in the training dataset, followed by the crisp prediction made by the network. It shows that the network achieves 100% accuracy on this small dataset. Now we are ready to apply our backpropagation algorithm to a real world dataset.

Lab Tasks:

1. Apply the Backpropagation algorithm to the wheat seeds dataset provided with the manual. A network with 5 neurons in the hidden layer and 3 neurons in the output layer will be constructed. The network will be trained for 500 epochs with a learning rate of 0.3. These parameters will find with a little trial and error, but you may be able to do much better.

LAB # 12: Genetic Algorithm with Python.

Genetic algorithm is a powerful optimization technique that was inspired by nature. Genetic algorithms mimic evolution to find the best solution. Unlike most optimization algorithms, genetic algorithms do not use derivatives to find the minima. One of the most significant advantages of genetic algorithms is their ability to find a global minimum without getting stuck in local minima. Randomness plays a substantial role in the structure of genetic algorithms, and it is the main reason genetic algorithms keep searching the search space. The continuous in the title means the genetic algorithm we are going to create will be using floating numbers or integers as optimization parameters instead of binary numbers.



Genetic algorithms create an initial population of randomly generated candidate solutions, these candidate solutions are evaluated, and their fitness value is calculated. The fitness value of a solution is the numeric value that determines how good a solution is, higher the fitness value better the solution. The figure below shows an example generation with 8 individuals. Each individual is made up of 4 genes, which represent the optimization parameters, and each individual has a fitness value, which in this case is the sum of the values of the genes.

Generation	Gene 1	Gene 2	Gene 3	Gene 4	Fitness
Individual 1	10	1	9	9	29
Individual 2	8	8	5	7	28
Individual 3	10	5	6	6	27
Individual 4	8	5	7	4	24
Individual 5	4	4	6	3	17
Individual 6	2	5	5	4	16
Individual 7	8	0	6	0	14
Individual 8	4	3	2	1	10

An Example of Generation

If the initial population does not meet the requirements of the termination criteria, genetic algorithm creates the next generation. The first genetic operation is Selection; in this operation, the individuals that are going to be moving on to the next generation are selected. After the selection process, pairing operation commences. Pairing operation pairs the selected individuals two by two for the Mating operation. The Mating operation takes the paired parent individuals and creates off springs, which will be replacing the individuals that were not selected in the Selection operation, so the next generation has the same number of individuals as the previous generation. This process is repeated until the termination criteria is met.

In this Lab, the genetic algorithm code was created from scratch using the Python standard library and Numpy. Each of the genetic operations discussed before are created as functions. Before we begin with the genetic algorithm code we need to import some libraries as;

```
import numpy as np
from numpy.random import randint
from random import random as rnd
from random import gauss, randrange
```

Initial Population

Genetic algorithms begin the optimization process by creating an initial population of candidate solutions whose genes are randomly generated. To create the initial population, a function which creates individuals must be created;

```
def individual(number_of_genes, upper_limit, lower_limit):
    individual=[round(rnd()*(upper_limit-lower_limit)
                +lower_limit,1) for x in range(number_of_genes)]
    return individual
```

The function takes number of genes, upper and lower limits for the genes as inputs and creates the individual. After the function to create individuals is created, another function is needed to create the population. The function to create a population can be written as;

```
def population(number_of_individuals,
               number_of_genes, upper_limit, lower_limit):
    return [individual(number_of_genes, upper_limit, lower_limit)
```

```
for x in range(number_of_individuals)]
```

Using these two functions, the initial population can be created. After the genetic algorithm creates the first generation, the fitness values of the individuals are calculated.

Fitness Calculation

Fitness calculation function determines the fitness value of an individual, how to calculate the fitness value depends on the optimization problem. If the problem is to optimize the parameters of a function, that function should be implemented to the fitness calculation function. The optimization problem can be very complex, and using specific software may be needed to solve the problem; in that case, the fitness calculation function should run simulations and collect the results from the software that is being used. For simplicity, we will go over the generation example given at the beginning of the Lab.

```
def fitness_calculation(individual):
    fitness_value = sum(individual)
    return fitness_value
```

This is a very simple fitness function with only one parameter. Fitness function can be calculated for multiple parameters. For multiple parameters, normalizing the different parameters is very important, the difference in magnitude between different parameters may cause one of the parameters to become obsolete for the fitness function value. Parameters can be optimized with different methods, one of the normalization methods is rescaling. Rescaling can be shown as;

$$m_s = \frac{m_o - min(m_o)}{max(m_o) - min(m_o)}$$

Function for normalizing parameters

Where the m_s is the scaled value of the parameter, m_o is the actual value of the parameter. In this function, maximum and minimum value of the parameter should be determined according to the nature of the problem.

After the parameters are normalized, the importance of the parameters are determined by the biases given to each parameter in the fitness function. Sum of the biases given to the parameters should be 1. For multiple parameters, the fitness function can be written as;

$$J = b_1p_1 + b_2p_2 \dots\dots\dots + b_np_n$$

Multi-parameter fitness function

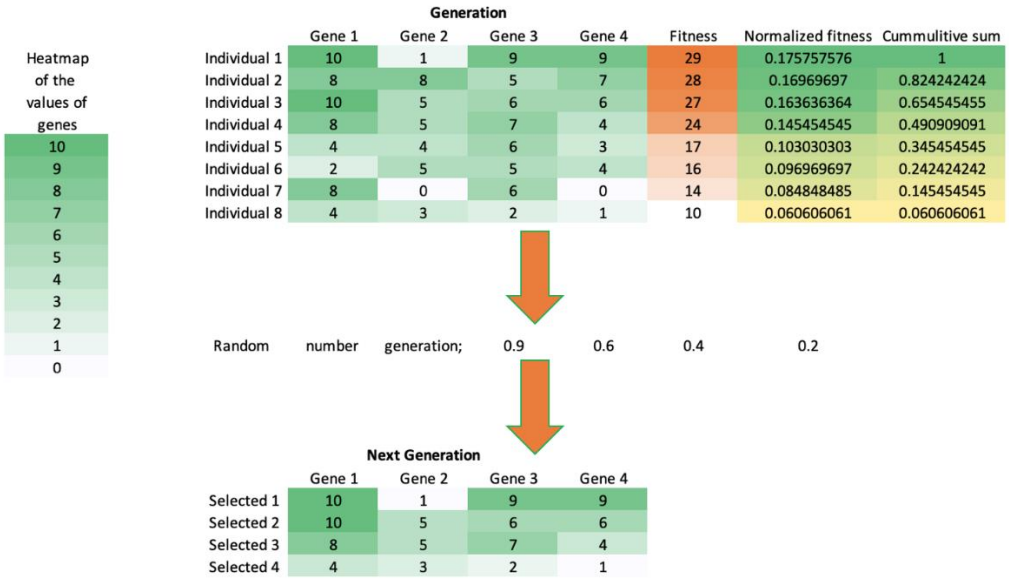
Where b represents the biases of the fitness function and p represents the normalized parameters.

Selection

The Selection function takes the population of candidate solutions and their fitness values (a generation) and outputs the individuals that are going to be moving on to the next generation. Elitism can be

introduced to the genetic algorithm, which will automatically select the best individual in a generation, so we do not lose the best solution. There are a few selection methods that can be used. Selection methods given in this Lab are;

- **Roulette wheel selection**: In roulette wheel selection, each individual has a chance to be selected. The chance of an individual to be selected is based on the fitness value of the individual. Fitter individuals have a higher chance to be selected.

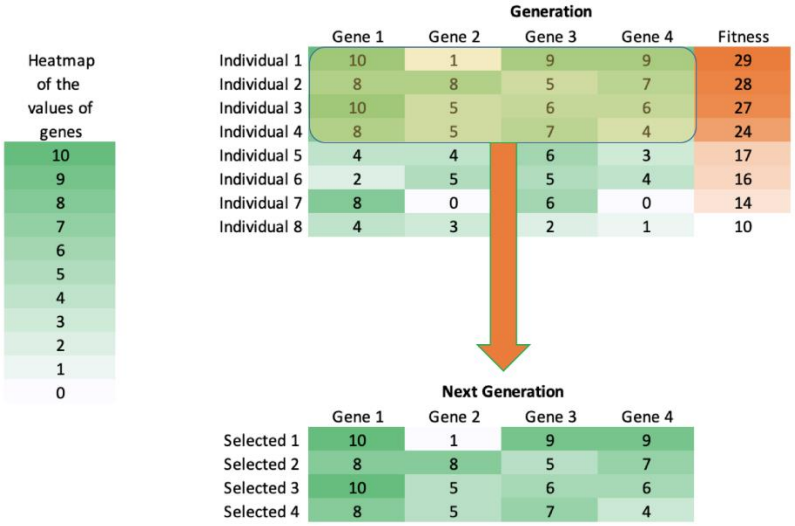


Roulette wheel selection figure

The function for roulette wheel selection takes the cumulative sums and the randomly generated value for the selection process and returns the number of the selected individual. By calculating the cumulative sums, each individual have a unique value between 0 and 1. To select individuals, a number between 0 and 1 is randomly generated and the individual that is closes to the randomly generated number is selected. The roulette function can be written as;

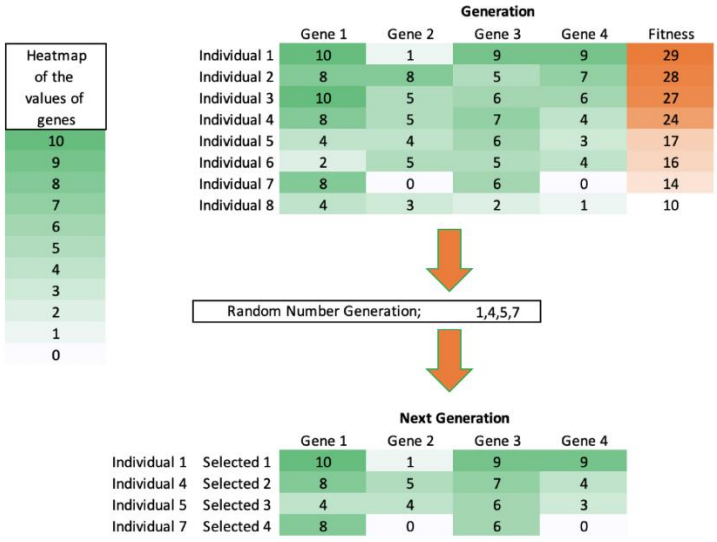
```
def roulette(cum_sum, chance):
    veriable = list(cum_sum.copy())
    veriable.append(chance)
    veriable = sorted(veriable)
    return veriable.index(chance)
```

- **Fittest half selection**: In this selection method, fittest half of the candidate solutions are selected to move to the next generation.



Fittest half selection figure

- **Random Selection**: In this method, individuals selected randomly.



Random selection figure

Selection function can be written as;

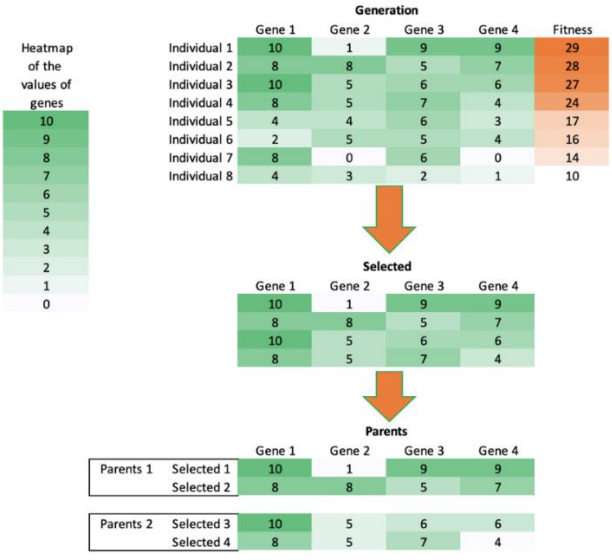
```
def selection(generation, method='Fittest Half'):
    generation['Normalized Fitness'] = \
        sorted([generation['Fitness'][x]/sum(generation['Fitness'])
        for x in range(len(generation['Fitness']))], reverse = True)
    generation['Cumulative Sum'] = np.array(
        generation['Normalized Fitness']).cumsum()
    if method == 'Roulette Wheel':
        selected = []
        for x in range(len(generation['Individuals'])//2):
            selected.append(roulette(generation
                ['Cumulative Sum'], rnd()))
            while len(set(selected)) != len(selected):
                selected[x] = \
                    (roulette(generation['Cumulative Sum'], rnd()))
        selected = {'Individuals':
```

```
            [generation['Individuals'][int(selected[x])]
                for x in range(len(generation['Individuals'])//2)]
                ,'Fitness': [generation['Fitness'][int(selected[x])]
                for x in range(
                    len(generation['Individuals'])//2)]}
    elif method == 'Fittest Half':
        selected_individuals = [generation['Individuals'][-x-1]
            for x in range(int(len(generation['Individuals'])//2))]
        selected_fitnesses = [generation['Fitness'][-x-1]
            for x in range(int(len(generation['Individuals'])//2))]
        selected = {'Individuals': selected_individuals,
                    'Fitness': selected_fitnesses}
    elif method == 'Random':
        selected_individuals = \
            [generation['Individuals']
                [randint(1,len(generation['Fitness']))]
            for x in range(int(len(generation['Individuals'])//2))]
        selected_fitnesses = [generation['Fitness'][-x-1]
            for x in range(int(len(generation['Individuals'])//2))]
        selected = {'Individuals': selected_individuals,
                    'Fitness': selected_fitnesses}
    return selected
```

Pairing

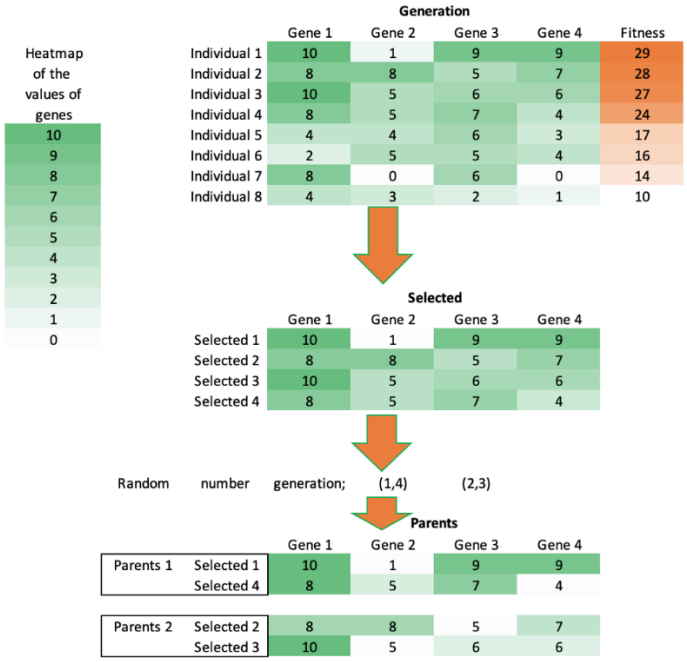
Pairing and mating are used as a single operation in most genetic algorithm applications, but for creating simpler functions and to be able to used different mating and paring algorithms easily, the two genetic operations are separated in this application. If there is elitism in the genetic algorithm, the elit must be an input to the function as well as the selected individuals. We are going to discuss three different pairing methods;

- **Fittest**: In this method, individuals are paired two by two, starting from the fittest individual. By doing so, fitter individuals are paired together, but less fit individuals are paired together as well.



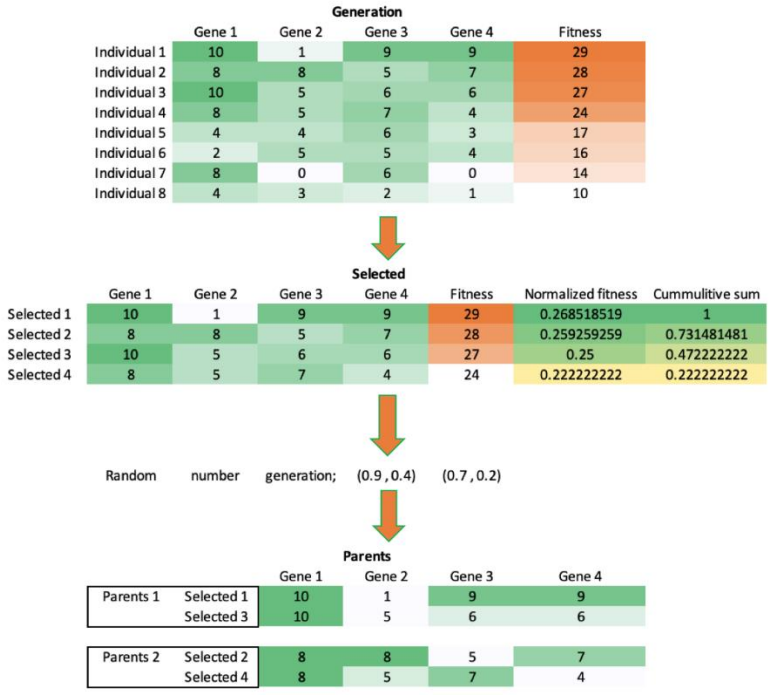
Fittest pairing figure

- **Random**: In this method, individuals are paired two by two randomly.



Random pairing figure

- **Weighted random**: In this method, individuals are paired randomly two by two, but fitter individuals have a higher chance to be selected for pairing.



Weighted random pairing

Pairing function can be written as;

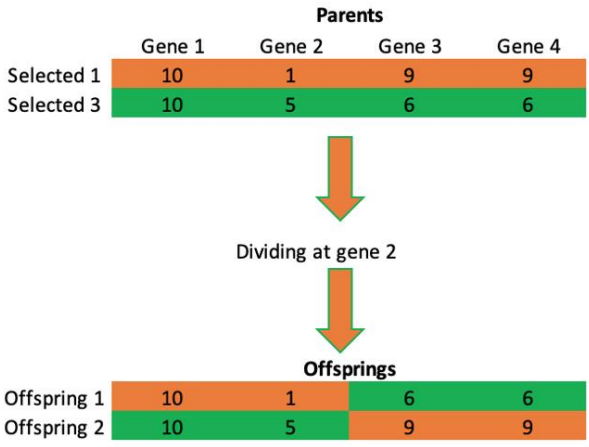
```
def pairing(elit, selected, method = 'Fittest'):
    individuals = [elit['Individuals']]+selected['Individuals']
    fitness = [elit['Fitness']]+selected['Fitness']
    if method == 'Fittest':
```

```
        parents = [[individuals[x],individuals[x+1]]
                   for x in range(len(individuals)//2)]
    if method == 'Random':
        parents = []
        for x in range(len(individuals)//2):
            parents.append(
                [individuals[randint(0,(len(individuals)-1))],
                 individuals[randint(0,(len(individuals)-1))]])
            while parents[x][0] == parents[x][1]:
                parents[x][1] = individuals[
                    randint(0,(len(individuals)-1))]
    if method == 'Weighted Random':
        normalized_fitness = sorted(
            [fitness[x] /sum(fitness)
             for x in range(len(individuals)//2)], reverse = True)
        cummulitive_sum = np.array(normalized_fitness).cumsum()
        parents = []
        for x in range(len(individuals)//2):
            parents.append(
                [individuals[roulette(cummulitive_sum,rnd())],
                 individuals[roulette(cummulitive_sum,rnd())]])
            while parents[x][0] == parents[x][1]:
                parents[x][1] = individuals[
                    roulette(cummulitive_sum,rnd())]
    return parents
```

Mating

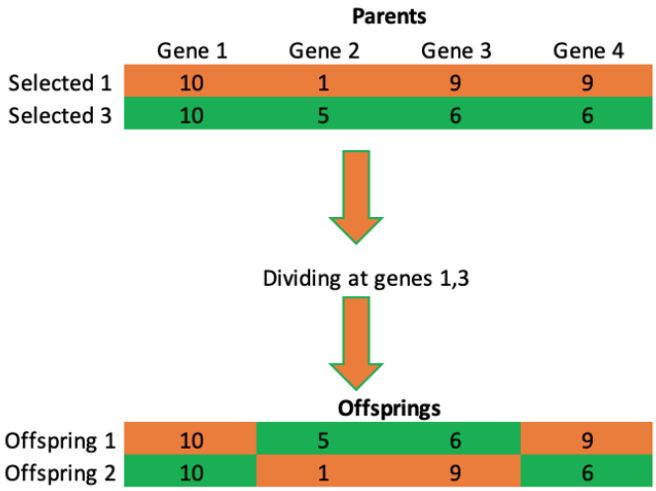
We will discuss two different mating methods. In the Python code given below, two selected parent individuals create two offsprings. There are two mating methods we are going to discuss.

Single point: In this method, genes after a single point are replaced with the genes of the other parent to create two offsprings.



Single point mating

- **Two points**: In this method, genes between two points are replaced with the genes of the other parent to create two offsprings.



Two points mating

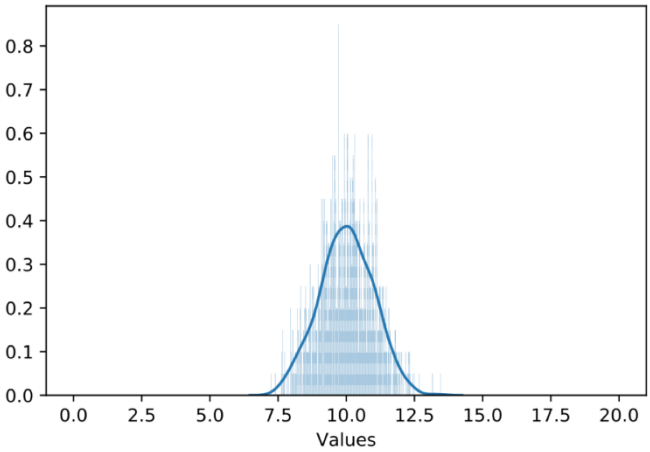
Mating function can be coded as;

```
def mating(parents, method='Single Point'):
    if method == 'Single Point':
        pivot_point = randint(1, len(parents[0]))
        offsprings = [parents[0] \
            [0:pivot_point]+parents[1][pivot_point:]]
        offsprings.append(parents[1]
            [0:pivot_point]+parents[0][pivot_point:])
    if method == 'Two Pionts':
        pivot_point_1 = randint(1, len(parents[0]-1))
        pivot_point_2 = randint(1, len(parents[0]))
        while pivot_point_2<pivot_point_1:
            pivot_point_2 = randint(1, len(parents[0]))
        offsprings = [parents[0][0:pivot_point_1]+
            parents[1][pivot_point_1:pivot_point_2]+
            [parents[0][pivot_point_2:]]]
        offsprings.append([parents[1][0:pivot_point_1]+
            parents[0][pivot_point_1:pivot_point_2]+
            [parents[1][pivot_point_2:]]])
    return offsprings
```

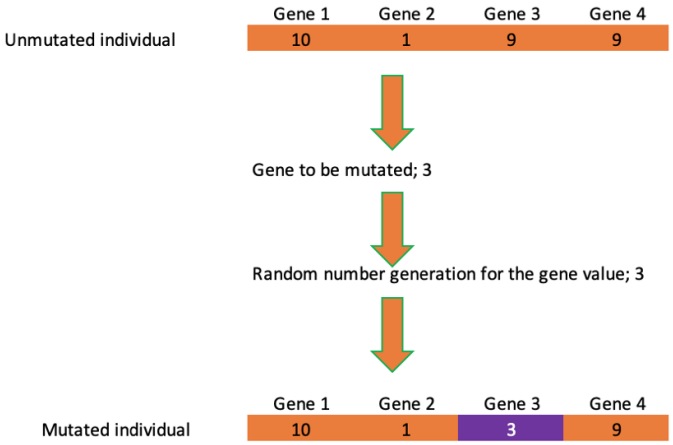
Mutations

The final genetic operation is random mutations. Random mutations occur in the selected individuals and their offsprings to improve variety of the next generation. If there is elitism in the genetic algorithm, elit individual does not go through random mutations so we do not lose the best solution. We are going to discuss two different mutation methods.

- **Gauss:** In this method, the gene that goes through mutation is replaced with a number that is generated according to gauss distribution around the original gene.



- **Reset**: In this method, the original gene is replaced with a randomly generated gene.



The mutation function can be written as;

```
def mutation(individual, upper_limit, lower_limit, muatation_rate=2,
    method='Reset', standard_deviation = 0.001):
    gene = [randint(0, 7)]
    for x in range(muatation_rate-1):
        gene.append(randint(0, 7))
        while len(set(gene)) < len(gene):
            gene[x] = randint(0, 7)
    mutated_individual = individual.copy()
    if method == 'Gauss':
        for x in range(muatation_rate):
            mutated_individual[x] = \
            round(individual[x]+gauss(0, standard_deviation), 1)
    if method == 'Reset':
        for x in range(muatation_rate):
            mutated_individual[x] = round(rnd()* \
                (upper_limit-lower_limit)+lower_limit,1)
    return mutated_individual
```

Creating the Next Generation

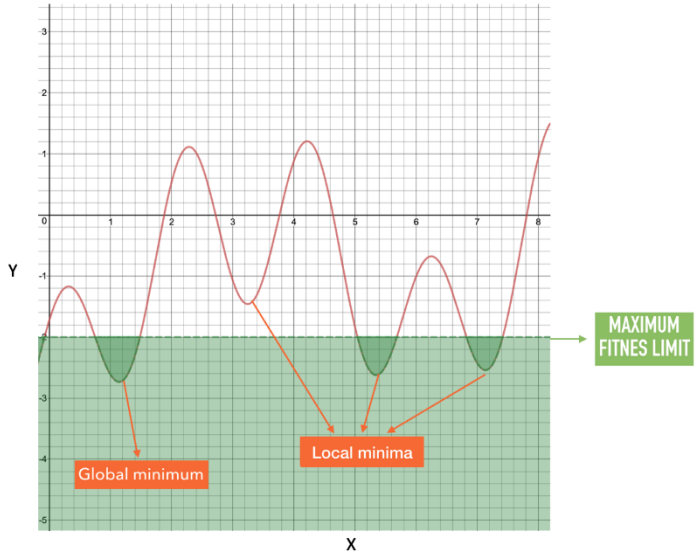
The next generation is created using the genetic operations we discussed. Elitism can be introduced to the genetic algorithm during the creating of next generation. Elitism is the python code to create the next generation can be written as;

```
def next_generation(gen, upper_limit, lower_limit):
    elit = {}
    next_gen = {}
    elit['Individuals'] = gen['Individuals'].pop(-1)
    elit['Fitness'] = gen['Fitness'].pop(-1)
    selected = selection(gen)
    parents = pairing(elit, selected)
    offsprings = [[[mating(parents[x])
                    for x in range(len(parents))]
                    [y][z] for z in range(2)]
                    for y in range(len(parents))]
    offsprings1 = [offsprings[x][0]
                   for x in range(len(parents))]
    offsprings2 = [offsprings[x][1]
                   for x in range(len(parents))]
    unmutated = selected['Individuals']+offsprings1+offsprings2
    mutated = [mutation(unmutated[x], upper_limit, lower_limit)
        for x in range(len(gen['Individuals']))]
    unsorted_individuals = mutated + [elit['Individuals']]
    unsorted_next_gen = \
        [fitness_calculation(mutated[x])
         for x in range(len(mutated))]
    unsorted_fitness = [unsorted_next_gen[x]
        for x in range(len(gen['Fitness']))] + [elit['Fitness']]
    sorted_next_gen = \
        sorted([[unsorted_individuals[x], unsorted_fitness[x]]
            for x in range(len(unsorted_individuals))],
                key=lambda x: x[1])
    next_gen['Individuals'] = [sorted_next_gen[x][0]
        for x in range(len(sorted_next_gen))]
    next_gen['Fitness'] = [sorted_next_gen[x][1]
        for x in range(len(sorted_next_gen))]
    gen['Individuals'].append(elit['Individuals'])
    gen['Fitness'].append(elit['Fitness'])
    return next_gen
```

Termination Criteria

After a generation is created, termination criteria are used to determine if the genetic algorithm should create another generation or should it stop. Different termination criteria can be used at the same time and if the genetic algorithm satisfies one of the criteria the genetic algorithm stops. We are going to discuss four termination criteria.

- **Maximum fitness**: This termination criteria checks if the fittest individual in the current generation satisfies our criteria. Using this termination method, desired results can be obtained. As seen from the figure below, maximum fitness limit can be determined to include some of the local minima.



- **Maximum average fitness**: If we are interested in a set of solutions average values of the individuals in the current generations can be checked to determine if the current generation satisfies our expectations.
- **Maximum number of generations**: We could limit the maximum number of generations created by the genetic algorithm.
- **Maximum similar fitness number**: Due to elitism best individual in a generation moves on to the next generation without mutating. This individual can be the best individual in the next generation as well. We can limit the number for the same individual to be the best individual as this can be sing that the genetic algorithm got stuck in a local minima. The function for checking if the maximum fitness value have changed can be written as;

```
def fitness_similarity_chech(max_fitness, number_of_similarity):
    result = False
    similarity = 0
    for n in range(len(max_fitness)-1):
        if max_fitness[n] == max_fitness[n+1]:
            similarity += 1
        else:
            similarity = 0
    if similarity == number_of_similarity-1:
        result = True
    return result
```

Running the Algorithm

Now that all of the function we need for the genetic algorithm is ready, we can begin the optimization process. To run the genetic algorithm with 20 individuals in each generation;

```
# Generations and fitness values will be written to this file
Result_file = 'GA_Results.txt'
# Creating the First Generation
```

```
def first_generation(pop):
    fitness = [fitness_calculation(pop[x])
        for x in range(len(pop))]
    sorted_fitness = sorted([[pop[x], fitness[x]]
        for x in range(len(pop))], key=lambda x: x[1])
    population = [sorted_fitness[x][0]
        for x in range(len(sorted_fitness))]
    fitness = [sorted_fitness[x][1]
        for x in range(len(sorted_fitness))]
    return {'Individuals': population, 'Fitness': sorted(fitness)}
pop = population(20,8,1,0)
gen = []
gen.append(first_generation(pop))
fitness_avg = np.array([sum(gen[0]['Fitness'])/
                        len(gen[0]['Fitness'])])
fitness_max = np.array([max(gen[0]['Fitness'])])
res = open(Result_file, 'a')
res.write('\n'+str(gen)+'\n')
res.close()
finish = False
while finish == False:
    if max(fitness_max) > 6:
        break
    if max(fitness_avg) > 5:
        break
    if fitness_similarity_chech(fitness_max, 50) == True:
        break
    gen.append(next_generation(gen[-1],1,0))
    fitness_avg = np.append(fitness_avg, sum(
        gen[-1]['Fitness'])/len(gen[-1]['Fitness']))
    fitness_max = np.append(fitness_max, max(gen[-1]['Fitness']))
    res = open(Result_file, 'a')
    res.write('\n'+str(gen[-1])+'\n')
    res.close()
```

Conclusion

Genetic algorithms can be used to solve multi-parameter constraint optimization problems. Like most of optimization algorithms, genetic algorithms can be implemented directly from some libraries like sklearn, but creating the algorithm from scratch gives a perspective on how it works and the algorithm can be tailored to a specific problem.

Lab Task:

Apply the above algorithm using SK learn library results should matched to this work.