Week-01 : NumPy

1. Numpy

This is a quick overview of arrays in NumPy. It demonstrates how n-dimensional (n>=2) arrays are represented and can be manipulated. In particular, if you don't know how to apply common functions to n-dimensional arrays (without using for-loops), or if you want to understand axis and shape properties for n-dimensional arrays.

Learning Objectives

You should be able to:

- Understand the difference between one-, two- and n-dimensional arrays in NumPy;
- Understand how to apply some linear algebra operations to n-dimensional arrays without using for-loops;
- Understand axis and shape properties for n-dimensional arrays.

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called axes.

For example, the array for the coordinates of a point in 3D space, [1, 2, 1], has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[1., 0., 0.],
 [0., 1., 2.]]
```

NumPy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an ndarray object are:

ndarray.ndim

The number of axes (dimensions) of the array.

ndarray.shape

The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.

ndarray.size

The total number of elements of the array. This is equal to the product of the elements of shape.

ndarray.dtype

An object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize

The size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

ndarray.data

The buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

Example

```
>>>import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
```

```
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

Array Creation

There are several ways to create arrays.

For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>>import numpy as np
>>> a = np.array([2, 3, 4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>>b = np.array([(1.5, 2, 3), (4, 5, 6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>>c = np.array([[1, 2], [3, 4]], dtype=complex)
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64, but it can be specified via the key word argument dtype.

```
>>>np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones((2, 3, 4), dtype=np.int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty((2, 3))
array([[3.73603959e-262, 6.02658058e-154, 6.55490914e-260],  # may vary
          [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

To create sequences of numbers, NumPy provides the arange function which is analogous to the Python built-in range, but returns an array.

```
>>>np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 2, 0.3)  # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

```
>>>from numpy import pi
>>> np.linspace(0, 2, 9)                 # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace(0, 2 * pi, 100) # useful to evaluate
                                    #function at lots of points
>>> f = np.sin(x)
```

Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
>>>a = np.arange(6)                    # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>> b = np.arange(12).reshape(4, 3)     # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> c = np.arange(24).reshape(2, 3, 4)  # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>>print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100, 100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using set_printoptions.

```
>>> np.set_printoptions(threshold=sys.maxsize)  # sys module should be imported
```

Basic Operations

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> c = a - b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10 * np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a < 35
```

```
array([ True,  True, False, False])
```

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

```
>>> A = np.array([[1, 1],
...               [0, 1]])
>>> B = np.array([[2, 0],
...               [3, 4]])
>>> A * B     # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B     # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)  # another matrix product
array([[5, 4],
       [3, 4]])
```

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

```
>>> rg = np.random.default_rng(1)  # create instance of default random number generator
>>> a = np.ones((2, 3), dtype=int)
>>> b = rg.random((2, 3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b  # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int64') with casting rule 'same_kind'
```

When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0, pi, 3)
>>> b.dtype.name
'float64'
>>> c = a + b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c * 1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
```

'complex128'

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

```
>>> a = rg.random((2, 3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
       [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.1057109529998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

Questions

1. Convert a 1-D array into a 2-D array with 3 rows.

Start with:
Assign-1 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
Desired output:
[[0, 1, 2]
[3, 4, 5]
[6, 7, 8]]

2. Replace all odd numbers in the given array with -1

Start with:
Assign-2 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Desired output:
[0, -1, 2, -1, 4, -1, 6, -1, 8, -1]

3. Find the positions of:

elements in x where its value is more than its corresponding element in y, and elements in x where its value is equals to its corresponding element in y.

Start with these:
x = np.array([21, 64, 86, 22, 74, 55, 81, 79, 90, 89])
y = np.array([21, 7, 3, 45, 10, 29, 55, 4, 37, 18])

Desired output:
(array([1, 2, 4, 5, 6, 7, 8, 9]),) and (array([0]),)

4. **Extract the first four columns of this 2-D array.**
Start with this:
Assign-4= np.arange(100).reshape(5,-1)

Desired output:
[[0 1 2 3]
[20 21 22 23]
[40 41 42 43]
[60 61 62 63]
[80 81 82 83]]

Additional questions

1. **Generate a 1-D array of 10 random integers. Each integer should be a number between 30 and 40 (inclusive).**
Sample of desired output:
[36, 30, 36, 38, 31, 35, 36, 30, 32, 34]

2. Consider the following matrices :
A= ((1, 2, 3), (4, 5, 6), (7, 8, 10)) and B = ((7, 8, 10) ,(4, 5, 6), (1, 2, 3))
Write a python program to perform the following using Numeric Python (numpy).
i) Add and Subtract of the Matrix A and B, print the resultant matrix C for add and E for subtract.
ii) Compute the sum of all elements of Matrix A, sum of each column of Matrix B and sum of each row of Matrix C
iii) Product of two matrices A and B, and print the resultant matrix D
iv) Sort the elements of resultant matrix C and print the resultant Matrix E.
v) Transpose the Matrix E and print the result

WEEK-02: DATA ANALYSIS_AND_VISUALISATION_WITH_PYTHON

Matplotlib

Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002. One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals. Matplotlib consists of several plots like line, bar, scatter, histogram etc.

Basic plots in Matplotlib :

Matplotlib comes with a wide variety of plots. Plots helps to understand trends, patterns, and to make correlations. They're typically instruments for reasoning about quantitative information. Some of the sample plots are covered here.

Line plot :

Example 1

```
# importing matplotlib module
from matplotlib import pyplot as plt
# x-axis values
x = [5, 2, 9, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot
plt.plot(x,y)
# function to show the plot
plt.show()
```

Bar plot:

Example 2

```
# importing matplotlib module
from matplotlib import pyplot as plt
# x-axis values
x = [5, 2, 9, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot the bar
plt.bar(x,y)
# function to show the plot
plt.show()
```

Hist plot:

A histogram is basically used to represent data provided in a form of some groups.It is accurate method for the graphical representation of numerical data distribution.It is a type of bar plot where X-axis represents the bin ranges while Y-axis gives information about frequency.

Creating a Histogram

To create a histogram the first step is to create bin of the ranges, then distribute the whole range of the values into a series of intervals, and count the values which fall into each of the intervals.Bins are clearly identified as consecutive, non-overlapping intervals of variables.The matplotlib.pyplot.hist() function is used to compute and create histogram of x.

In Matplotlib, we use the hist() function to create histograms. The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument. For simplicity we use NumPy to

randomly generate an array with 250 values, where the values will concentrate around 170, and the standard deviation is 10.
You can read from the histogram that there are approximately:
2 people from 140 to 145cm
5 people from 145 to 150cm
15 people from 151 to 156cm
31 people from 157 to 162cm
46 people from 163 to 168cm
53 people from 168 to 173cm
45 people from 173 to 178cm
28 people from 179 to 184cm
21 people from 185 to 190cm
4 people from 190 to 195cm

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:

The hist() function will read the array and produce a histogram:

Simple histogram:

Example 3

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

Ref: https://www.w3schools.com/python/matplotlib_histograms.asp

Example 4

```
# importing matplotlib module
from matplotlib import pyplot as plt
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot histogram
plt.hist(y)
# Function to show the plot
plt.show()
```

Ref: https://www.geeksforgeeks.org/plotting-histogram-in-python-using-matplotlib/

Scatter plot:

Example 5

```
# importing matplotlib module
from matplotlib import pyplot as plt
# x-axis values
x = [5, 2, 9, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot scatter
plt.scatter(x, y)
# function to show the plot
plt.show()
```

SciPy

What is SciPy?

- SciPy is a scientific computation library that uses NumPy underneath.
- SciPy stands for Scientific Python.
- It provides more utility functions for optimization, stats and signal processing.
- Like NumPy, SciPy is open source so we can use it freely.
- SciPy was created by NumPy's creator Travis Olliphant.

Why Use SciPy?

- If SciPy uses NumPy underneath, why can we not just use NumPy?
- SciPy has optimized and added functions that are frequently used in NumPy and Data Science.

Which Language is SciPy Written in?

- SciPy is predominantly written in Python, but a few segments are written in C.

Import SciPy

- Once SciPy is installed, import the SciPy module(s) you want to use in your applications by adding the from scipy import module statement:

  from scipy import constants

constants: SciPy offers a set of mathematical constants, one of them is liter which returns 1 liter as cubic meters.

Unit Categories

- The units are placed under these categories:
- Metric
- Binary
- Mass
- Angle
- Time
- Length
- Pressure
- Volume
- Speed
- Temperature
- Energy
- Power
- Force

Example 6

```
from scipy import constants
print(constants.peta)    #1000000000000000.0
print(constants.tera)    #1000000000000.0
print(constants.giga)    #1000000000.0
print(constants.mega)    #1000000.0
print(constants.kilo)    #1000.0
print(constants.hecto)   #100.0
print(constants.deka)    #10.0
print(constants.deci)    #0.1
print(constants.centi)   #0.01
print(constants.milli)   #0.001
print(constants.micro)   #1e-06
print(constants.nano)    #1e-09
print(constants.pico)    #1e-12
```

Sparse Data

- Sparse data is data that has mostly unused elements (elements that don't carry any information).
- It can be an array like this one: [1, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0, 0]
- **Sparse Data:** is a data set where most of the item values are zero.
- **Dense Array:** is the opposite of a sparse array: most of the values are *not* zero.
- In scientific computing, when we are dealing with partial derivatives in linear algebra we will come across sparse data.

Work With Sparse Data

SciPy has a module, scipy.sparse that provides functions to deal with sparse data.

- There are primarily two types of sparse matrices that we use:
- CSC - Compressed Sparse Column. For efficient arithmetic, fast column slicing.
- CSR - Compressed Sparse Row. For fast row slicing, faster matrix vector products We will use the CSR matrix in this tutorial.

CSR Matrix

We can create CSR matrix by passing an arrray into function scipy.sparse.csr_matrix().

Example 7

Create a CSR matrix from an array:

```
import numpy as np
from scipy.sparse import csr_matrix
arr = np.array([0, 0, 0, 0, 0, 1, 1, 0, 2])
print(csr_matrix(arr))
```

The example above returns:

```
  (0, 5) 1
  (0, 6) 1
  (0, 8) 2
```

From the result we can see that there are 3 items with value.
The 1. item is in row 0 position 5 and has the value 1.
The 2. item is in row 0 position 6 and has the value 1.
The 3. item is in row 0 position 8 and has the value 2.

Viewing stored data (not the zero items) with the data property:

Example 8

```
import numpy as np
from scipy.sparse import csr_matrix
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])
print(csr_matrix(arr).data)
```

Converting from csr to csc with the tocsc() method:

Example 9

```
import numpy as np
from scipy.sparse import csr_matrix
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])
newarr = csr_matrix(arr).tocsc()
print(newarr)
```

Pandas

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data. The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Use of Pandas

Pandas allows us to analyze big data and make conclusions based on statistical theories. Pandas can clean messy data sets, and make them readable and relevant. Relevant data is very important in data science.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.

Once Pandas is installed, import it in your applications by adding the import keyword:

import pandas

Example 10

Get your own Python Server

```
import pandas
mydataset = {   'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2] }
myvar = pandas.DataFrame(mydataset)
print(myvar)
```

Create an alias with the as keyword while importing:

import pandas as pd

Now the Pandas package can be referred to as pd instead of pandas.

Example 11

```
import pandas as pd
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
myvar = pd.DataFrame(mydataset)
print(myvar)
```

Indices in a pandas series:

- A pandas series is similar to a list, but differs in the fact that a series associates a label with each element. This makes it look like a dictionary.
- If an index is not explicitly provided by the user, pandas creates a RangeIndex ranging from 0 to N-1.
- Each series object also has a data type.

Example 12

```
import panadad as pd
new_series= pd.series([5,6,7,8,9,10])
print(new_series)
```

- As you may suspect by this point, a series has ways to extract all of the values in the series, as well as individual elements by index.

Example 13

```
import panadad as pd
new_series= pd.series([5,6,7,8,9,10])
print(new_series.values)
print('_____')
print(new_series[4])
```

- You can also provide an index manually.

Example 14

```
import panadad as pd
new_series= pd.series([5,6,7,8,9,10], indec=['a','b','c','d','e','f')
print(new_series.values)
print('_____________')
print(new_series['f'])
```

- It is easy to retrieve several elements of a series by their indices or make group assignments.

Example 15

```
import panadad as pd
new_series= pd.series([5,6,7,8,9,10], indec=['a','b','c','d','e','f')
print(new_series)
print('_____________')
print(new_series['a','b','f'])=0
print(new_series)
```

- Filtering and maths operations are easy with Pandas as well.

Example 16

```
import panadad as pd
new_series= pd.series([5,6,7,8,9,10], indec=['a','b','c','d','e','f')
new_series2= new_series[new_series>0]  # check with 7 instead of 0
print(new_series2)
print('_____________')
new_series2= new_series[new_series>0]  * 2
print(new_series2)
```

DataFrame:

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.
- Simplistically, a data frame is a table, with rows and columns.
- Each column in a data frame is a series object.
- Rows consist of elements inside series.

Case ID	*Variable one*	*Variable two*	*Variable 3*
1	*123*	*ABC*	*10*
2	*456*	*DEF*	*20*
3	*789*	*XYZ*	*30*

- Pandas data frames can be constructed using Python dictionaries.

Example 17

Get your own Python Server

Create a simple Pandas DataFrame:

```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
```

Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns. Pandas use the loc attribute to return one or more specified row(s)

Example 18

Return row 0:

```
#refer to the row index:
print(df.loc[0])
```

Named Indexes

With the index argument, you can name your own indexes.

Example 19

Add a list of names to give each row a name:

```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

- You can also create a data frame from a list.

Example 20

```
import pandas as pd
list2=[[0,1,2],[3,4,5],[6,7,8]]
df= pd.DataFrame(list2)
print(df)
df.columns=['V1','V2','V3']
print(df)
```

Example 21

```
import pandas as pd
df=pd.DataFrame({
    'Country': ['Kazakhstan','Russia','Belarus','Ukraine'],
    'Population': [17.04,143.5,9.5,45.5]
    'Square':[2724902, 17125191,207600,603628]
   })
print(df)
```

- You can ascertain the type of a column with the type() function.

```
print(type(df['Country']))
```

- A Pandas data frame object as two indices; a column index and row index.
- Again, if you do not provide one, Pandas will create a RangeIndex from 0 to N-1.

Example 22

```
import pandas as pd
df=pd.DataFrame({
    'Country': ['Kazakhstan','Russia','Belarus','Ukraine'],
    'Population': [17.04,143.5,9.5,45.5]
    'Square':[2724902, 17125191,207600,603628]
   })
print(df.columns)
print('_____')
print(df.index)
```

- There are numerous ways to provide row indices explicitly.
- For example, you could provide an index when creating a data frame:

Example 23

```
import pandas as pd
df=pd.DataFrame({
    'Country': ['Kazakhstan','Russia','Belarus','Ukraine'],
    'Population': [17.04,143.5,9.5,45.5]
    'Square':[2724902, 17125191,207600,603628]
   }, index = ['KZ','RU','BY','UA'])
print(df)
```

- or do it during runtime.
- Here, I also named the index ‘country code’.

Example 24

```
import pandas as pd
df=pd.DataFrame({
    ‘Country’: [‘Kazakhstan’,’Russia’,’Belarus’,’Ukraine’],
    ‘Population’: [17.04,143.5,9.5,45.5]
    ‘Square’:[2724902, 17125191,207600,603628]
   })
print(df)
print(‘_____’)
df.index = =[‘KZ’,’RU’,’BY’,’UA’]
df.index.name = ‘Country Code’
print(df)
```

- Row access using index can be performed in several ways.
- First, you could use .loc() and provide an index label.

Example 25

```
print(df.loc[‘KZ’]
```

- Second, you could use .iloc() and provide an index number

```
print(df.iloc[0]
```

- A selection of particular rows and columns can be selected this way.

```
print(df.loc[[‘KZ’,’RU’],’population’])
```

- You can feed .loc() two arguments, index list and column list, slicing operation is supported as well:

```
print(df.loc[[‘KZ’,’BY’, :])
```

Filtering

- Filtering is performed using so-called Boolean arrays.

Example 26

```
print(df[df.population > 10][[‘Country’,’Square’]])
```

Deleting columns

You can delete a column using the drop() function.

```
print(df)
df = df.drop([‘Population’], axis = ‘columns’)
print(df)
```

Load Files Into a DataFrame

If your data sets are stored in a file, Pandas can load them into a DataFrame.

Example 27

Load a comma separated file (CSV file) into a DataFrame:

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

Read CSV Files

A simple way to store big data sets is to use CSV files (comma separated files). CSV files contains plain text and is a well know format that can be read by everyone including Pandas. In our examples we will be using a CSV file called 'data.csv'.

Example 28

Get your own Python Server

Load the CSV into a DataFrame:

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.to_string())  # use to_string() to print the entire DataFrame.
print(df)   #Print the DataFrame without the to_string() method
```

Data Cleaning

Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

Plotting

Pandas uses the plot() method to create diagrams. We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.

Example 29

Get your own Python Server

Import pyplot from Matplotlib and visualize our DataFrame:

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

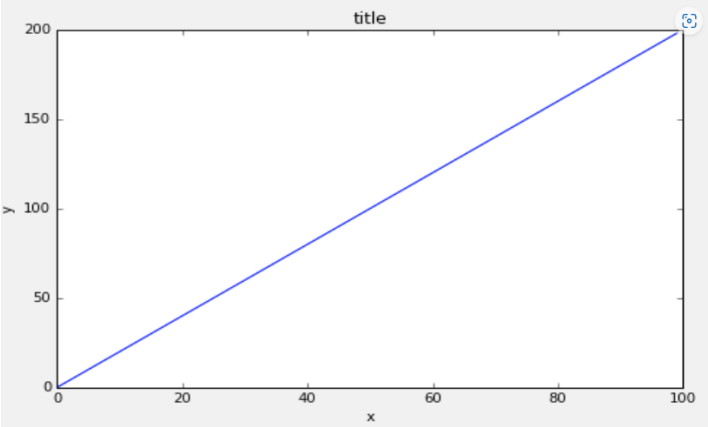
Questions

1. Follow along with these steps:

a) Create a figure object called fig using plt.figure()

b) Use add_axes to add an axis to the figure canvas at [0,0,1,1]. Call this new axis ax.

c) Plot (x,y) on that axes and set the labels and titles to match the plot below:



2. Create a figure object and put two axes on it, ax1 and ax2. Located at [0,0,1,1] and [0.2,0.5,.2,.2] respectively. Now plot (x,y) on both axes. And call your figure object to show it.

3. Use the company sales dataset csv file, read Total profit of all months and show it using a line plot Total profit data provided for each month. Generated line plot must include the following properties: –

a. X label name = Month Number
b. Y label name = Total profit

4. Use the company sales dataset csv file, get total profit of all months and show line plot with the following Style properties. Generated line plot must include following Style properties: –

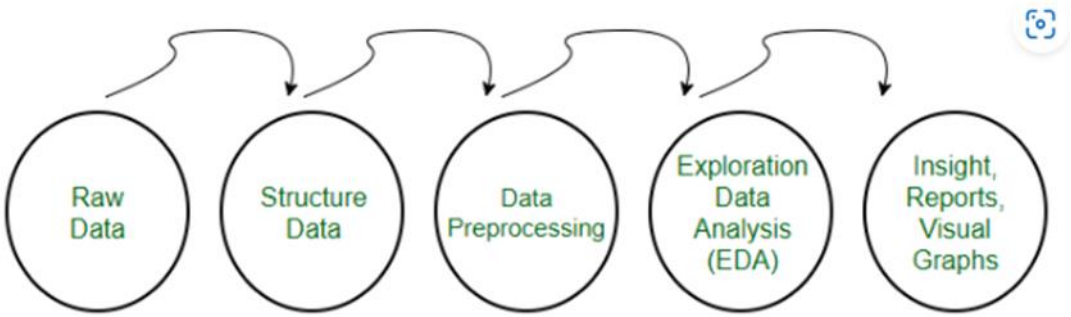
a. Line Style dotted and Line-color should be red
b. Show legend at the lower right location.
c. X label name = Month Number
d. Y label name = Sold units number
e. Add a circle marker.
f. Line marker color as read
g. Line width should be 3

Additional Questions

1. Use the company sales dataset csv file, read all product sales data and show it using a multiline plot. Display the number of units sold per month or each product using multiline plots. (i.e., Separate Plotline for each product).
2. Use the company sales dataset csv file, calculate total sale data for last year for each product and show it using a Pie chart.
   Note: In Pie chart display Number of units sold per year for each product in percentage.

WEEK -03 : DATA PREPROCESSING AND REGRESSION

Data pre-processing is a basic requirement of any good machine learning model. Pre-processing the data implies using the data which is easily readable by the machine learning model. In this week, we are going to discuss the basics of data pre-processing and how to make the data suitable for machine learning models.



Data Preprocessing

What is data pre-processing?

Data pre-processing is the process of preparing the raw data and making it suitable for machine learning models. Data pre-processing includes data cleaning for making the data ready to be given to machine learning model.

After data cleaning, data pre-processing requires the data to be transformed into a format that is understandable to the machine learning model.

Why is data pre-processing required?

Data pre-processing is mainly required for the following:

- **Accurate data:**For making the data readable for machine learning model, it needs to be accurate with no missing value, redundant or duplicate values.
- **Trusted data:**The updated data should be as accurate or trusted as possible.
- **Understandable data:**The data updated needs to be interpreted correctly.

All in all, data pre-processing is important for the machine learning model to learn from such data which is correct in order to lead the model to the right predictions/outcomes.

Examples of data preprocessing for different data set types with Python

Since data comes in various formats, let us discuss how different data types can be converted into a format that the ML model can read accurately. Let us see how to feed correct features from datasets with:

- Missing values
- Outliers
- Overfitting
- Data with no numerical values
- Different date formats

▪ *Missing values*

Missing values are a common problem while dealing with data! The values can be missed because of various reasons such as human errors, mechanical errors, etc.

Data cleansing is an important step before you even begin the algorithmic trading process, which begins with historical data analysis for making the prediction model as accurate as possible.

Based on this prediction model you create the trading strategy. Hence, leaving missed values in the data set can wreak havoc by giving faulty predictive results that can lead to erroneous strategy creation and further the results can not be great to state the obvious.

There are three techniques to solve the missing values' problem in order to find out the most accurate features, and they are:

Dropping
Numerical imputation
Categorical imputation

Dropping

Dropping is the most common method to take care of the missed values. Those rows in the data set or the entire columns with missed values are dropped in order to avoid errors to occur in data nalysis.

There are some machines that are programmed to automatically drop the rows or columns that include missed values resulting in a reduced training size. Hence, the dropping can lead to a decrease in the model performance.

A simple solution for the problem of a decreased training size due to the dropping of values is to use imputation. We will discuss the interesting imputation methods further. In case of dropping, you can define a threshold to the machine.

For instance, the threshold can be anything. It can be 50%, 60% or 70% of the data. Let us take 60% in our example, which means that 60% of data with missing out values will be accepted by the model/algorithm as the training data set, but the features with more than 60% missing values will be dropped.

For dropping the values, following Python codes are used:

```
#Dropping columns in the data higher than 60% threshold
data = data[data.columns[data.isnull().mean() < threshold]]
#Dropping rows in the data higher than 60% threshold
data = data.loc[data.isnull().mean(axis=1) < threshold]
```

By using the above Python codes, the missed values will be dropped and the machine learning model will learn on the rest of the data.

Numerical imputation

The word imputation implies replacing the missing values with such a value that makes sense. And, numerical imputation is done in the data with numbers.

For instance, if there is a tabular data set with the number of stocks, commodities and derivatives traded in a month as the columns, it is better to replace the missed value with a "0" than leaving them as it is.

With numerical imputation, the data size is preserved and hence, predictive models like linear regression can work better to predict in the most accurate manner.

A linear regression model can not work with missing values in the data set since it is biased toward the missed values and considers them "good estimates". Also, the missed values can be replaced with the median of the columns since median values are not sensitive to outliers unlike averages of columns.

Let us see the Python codes for numerical imputation, which are as follows:

```
#For filling all the missed values as 0
data = data.fillna(0)
#For replacing missed values with median of columns
data = data.fillna(data.median())
```

Categorical imputation

This technique of imputation is nothing but replacing the missed values in the data with the one which occurs the maximum number of times in the column. But, in case there is no such value that occurs frequently or dominates the other values, then it is best to fill the same as "NAN".

The following Python code can be used here:

```
#Categorical imputation
data['column_name'].fillna(data['column_name'].value_counts().idxmax(), inplace=True)
```

- *Outliers*

An outlier differs significantly from other values and is too distanced from the mean of the values. Such values that are considered outliers are usually due to some systematic errors or flaws.

Let us see the following Python codes for identifying and removing outliers with standard deviation:

```
#For identifying the outliers with the standard deviation method
outliers = [x for x in data if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))
#Remove outliers
outliers_removed = [x for x in data if x >= lower and x <= upper]
print('Non-outlier observations: %d' % len(outliers_removed))
```

In the codes above, “lower” and “upper” signify the upper and lower limit in the data set.

- *Overfitting*

In both machine learning and statistics, overfitting occurs when the model fits the data too well or simply put when the model is too complex.

Overfitting model learns the detail and noise in the training data to such an extent that it negatively impacts the performance of the model on new data/test data.

The overfitting problem can be solved by decreasing the number of features/inputs or by increasing the number of training examples to make the machine learning algorithms more generalised.

The most common solution is regularisation in an overfitting case. Binning is the technique that helps with the regularisation of the data which also makes you lose some data every time you regularise it.

For instance, in the case of numerical binning, the data can be as follows:

Stock value	Bin
100-250	Lowest
251-400	Mid
401-500	High

Here is the Python code for binning:

```
data['bin'] = pd.cut(data['value'], bins=[100,250,400,500], labels=["Lowest", "Mid", "High"])
```

Your output should look something like this:

	Value	Bin
0	102	Low
1	300	Mid
2	107	Low
3	470	High

- *Data with no numerical values*

In the case of the data set with no numerical values, it becomes impossible for the machine learning model to learn the information.

The machine learning model can only handle numerical values and thus, it is best to spread the values in the columns with assigned binary numbers “0” or “1”. This technique is known as one-hot encoding.

In this type of technique, the grouped columns already exist. For instance, below I have mentioned a grouped column:

Infected	Covid variants
2	Delta
4	Lambda
5	Omicron
6	Lambda
4	Delta
3	Omicron
5	Omicron
4	Lambda
2	Delta

Now, the above-grouped data can be encoded with the binary numbers ”0” and “1” with one hot encoding technique. This technique subtly converts the categorical data into a numerical format in the following manner:

Infected	Delta	Lambda	Omicron
2	1	0	0
4	0	1	0
5	0	0	1
6	0	1	0
4	1	0	0
3	0	0	1
5	0	0	1
4	0	1	0
2	1	0	0

Hence, it results in better handling of grouped data by converting the same into encoded data for the machine learning model to grasp the encoded (which is numerical) information quickly.

Problem with the approach

Going further, in case there are more than three categories in a data set that is to be used for feeding the machine learning model, the one-hot encoding technique will create as many columns. Let us say, there are 2000 categories, then this technique will create 2000 columns and it will be a lot of information to feed to the model.

Solution:
To solve this problem, while using this technique, we can apply the target encoding technique which implies calculating the "mean" of each predictor category and using the same mean for all other rows with the same category under the predictor column. This will convert the categorical column into the numeric column and that is our main aim.
Let us understand this with the same example as above but this time we will use the "mean" of the values under the same category in all the rows. Let us see how.
In Python, we can use the following code:

```
#Convert data into numerical values with mean
Infected = [2, 4, 5, 6, 4, 3]
Predictor = ['Delta', 'Lambda', 'Omicron', 'Lambda', 'Delta', 'Omicron']
Infected_df = pd.DataFrame(data={'Infected':Infected, 'Predictor':Predictor})
means = Infected_df.groupby('Predictor')['Infected'].mean()
Infected_df['Predictor_encoded'] = Infected_df['predictor'].map(means)
Infected_df
```

Output:

Infected	Predictor	Predictor_encoded
2	Delta	3
4	Lambda	5
5	Omicron	4
6	Lambda	5
4	Delta	3
3	Omicron	4

In the output above, the Predictor column depicts the Covid variants and the Predictor_encoded column depicts the "mean" of the same category of Covid variants which makes 2+4/2 = 3 as the mean value for Delta, 4+6/2 = 5 as the mean value for Lambda and so on.
Hence, the machine learning model will be able to feed the main feature (converted to a number) for each predictor category for the future.

- *Different date formats*

With the different date formats such as "25-12-2021", "25th December 2021" etc. the machine learning model needs to be equipped with each of them. Or else, it is difficult for the machine learning model to understand all the formats.

With such a data set, you can preprocess or decompose the data by mentioning three different columns for the parts of the date, such as Year, Month and Day.
In Python, the preprocessing of the data with different columns for the date will look like this:

```
#Convert to datetime object
df['Date'] = pd.to_datetime(df['Date'])
#Decomposition
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
df[['Year','Month','Day']].head()
```

Output:

Year	Month	Day
2019	1	5
2019	3	8
2019	3	3
2019	1	27
2019	2	8

In the output above, the data set is in date format which is numerical. And because of decomposing the date in different parts such as Year, Month and Day, the machine learning model will be able to learn the date format.

REGRESSION

Regression is a supervised learning technique that supports finding the correlation among variables.
A regression problem is when the output variable is a real or continuous value.
What is a Regression?

In Regression, we plot a graph between the variables which best fit the given data points. The machine learning model can deliver predictions regarding the data. In naïve words, **"Regression shows a line or curve that passes through all the data points on a target-predictor graph in such a way that the vertical distance between the data points and the regression line is minimum."** *It is used principally for prediction, forecasting, time series modeling, and determining the causal-effect relationship between variables.*

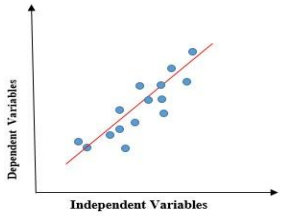
TYPES OF REGRESSION MODELS

1. Linear Regression
2. Polynomial Regression
3. Logistics Regression

1. Linear Regression:

Linear regression is a quiet and simple statistical regression method used for predictive analysis and shows the relationship between the continuous variables. Linear regression shows the linear relationship between the independent variable (X-axis) and the dependent variable (Y-axis), consequently called linear regression. If there is a single input variable (x), such linear regression is called **simple linear regression**. And if there

is more than one input variable, such linear regression is called **multiple linear regression.** The linear regression model gives a sloped straight line describing the relationship within the variables.



The above graph presents the linear relationship between the dependent variable and independent variables. When the value of x (**independent variable**) increases, the value of y (**dependent variable**) is likewise increasing. The red line is referred to as the best fit straight line. Based on the given data points, we try to plot a line that models the points the best.

To calculate best-fit line linear regression uses a traditional slope intercept form.

$$y = mx+b \quad \Longrightarrow \quad y = a_0+a_1x$$

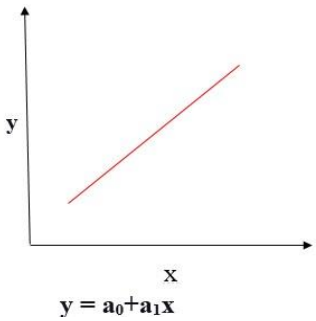
y= Dependent Variable.
x= Independent Variable.
a0= intercept of the line.
a1 = Linear regression coefficient.
Need of a Linear regression:

Linear regression estimates the relationship between a dependent variable and an independent variable. Let's say we want to estimate the salary of an employee based on year of experience. You have the recent company data, which indicates that the relationship between experience and salary. Here year of experience is an independent variable, and the salary of an employee is a dependent variable, as the salary of an employee is dependent on the experience of an employee. Using this insight, we can predict the future salary of the employee based on current & past information.

A regression line can be a Positive Linear Relationship or a Negative Linear Relationship.

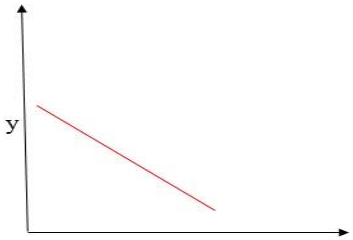
Positive Linear Relationship:

If the dependent variable expands on the Y-axis and the independent variable progress on X-axis, then such a relationship is termed a Positive linear relationship.



Negative Linear Relationship:

If the dependent variable decreases on the Y-axis and the independent variable increases on the X-axis, such a relationship is called a negative linear relationship.



$y = -a_0 + a_1 x$

The goal of the linear regression algorithm is to get the best values for a0 and a1 to find the best fit line. The best fit line should have the least error means the error between predicted values and actual values should be minimized.

Cost function

The cost function helps to figure out the best possible values for a0 and a1, which provides the best fit line for the data points.

Cost function optimizes the regression coefficients or weights and measures how a linear regression model is performing. The cost function is used to find the accuracy of the **mapping function** that maps the input variable to the output variable. This mapping function is also known as **the Hypothesis function**.

In Linear Regression, **Mean Squared Error (MSE)** cost function is used, which is the average of squared error that occurred between the predicted values and actual values.

By simple linear equation y=mx+b we can calculate MSE as:

Let's y = actual values, y_i = predicted values

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

Using the MSE function, we will change the values of a0 and a1 such that the MSE value settles at the minima. Model parameters ***xi, b (a_0,a_1)*** can be manipulated to minimize the cost function. These parameters can be determined using the gradient descent method so that the cost function value is minimum.

SCIKIT LEARN

- It is mainly used in machine learning
- It has lot of statistics related tools
- It is open source.
- By using the Scikit library the efficiency will improve tremendously as it is quite accurate.
- It is very useful in algorithms which are very famous in machine learning like K-mean, K-nearest, clustering etc.
- It is available to everybody so any programmer if he or she feels like utilizing it then can use it.
- Scikit requires Numpy

Features of Scikit learn are as follows:

- Clustering: Scikit can be applied in clustering algorithm, in clustering the grouping is done on the basis of similarities like eg: age, color etc.
- Cross validation
    - *Feature selection*

Example:

```
from sklearn.datasets import load_iris
iris = load_iris()
A= iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names
print("Feature names:", feature_names)
```

```
print("Target names:", target_names)
print("\nFirst 10 rows of A:\n", A[:10])
```

How to split a Dataset into Train and Test Sets using Python

Scikit-learn alias **sklearn** is the most useful and robust library for machine learning in Python. The **scikit-learn library** provides us with the model_selection module in which we have the splitter function train_test_split().

Syntax:

train_test_split(*arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)

Parameters:

1. *arrays: inputs such as lists, arrays, data frames, or matrices
2. test_size: this is a float value whose value ranges between 0.0 and 1.0. it represents the proportion of our test size. its default value is none.
3. train_size: this is a float value whose value ranges between 0.0 and 1.0. it represents the proportion of our train size. its default value is none.
4. random_state: this parameter is used to control the shuffling applied to the data before applying the split. it acts as a seed.
5. shuffle: This parameter is used to shuffle the data before splitting. Its default value is true.
6. stratify: This parameter is used to split the data in a stratified fashion.

Example

```
# read the dataset
# import modules
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
df = pd.read_csv('/home/student/Downloads/Real-estate.csv')
 # get the locations
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
# split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05, random_state=0)
```

In the above example, **We import the pandas package and sklearn package**. after that to import the CSV file we use the read_csv() method. The variable df now contains the data frame. in the example “house price” is the column we’ve to predict so we take that column as y and the rest of the columns as our X variable. test_size = 0.05 specifies only 5% of the whole data is taken as our test set, and 95% as our train set. The random state helps us get the same random split each time.

Simple Linear Regression With scikit-learn

You’ll start with the simplest case, which is simple linear regression. There are five basic steps when you’re implementing linear regression:

1. Import the packages and classes that you need.
2. Provide data to work with, and eventually do appropriate transformations.
3. Create a regression model and fit it with existing data.
4. Check the results of model fitting to know whether the model is satisfactory.
5. Apply the model for predictions.

These steps are more or less general for most of the regression approaches and implementations. Throughout the rest of the tutorial, you'll learn how to do these steps for several different scenarios.

Step 1: Import packages and classes

The first step is to import the package numpy and the class LinearRegression from sklearn.linear_model:

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
```

Now, you have all the functionalities that you need to implement linear regression.

The fundamental data type of NumPy is the array type called numpy.ndarray. The rest of this tutorial uses the term array to refer to instances of the type numpy.ndarray.

You'll use the class sklearn.linear_model.LinearRegression to perform linear and polynomial regression and make predictions accordingly.

Step 2: Provide data

The second step is defining data to work with. The inputs (regressors, x) and output (response, y) should be arrays or similar objects. This is the simplest way of providing data for regression:

:

```
>>> x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))
>>> y = np.array([5, 20, 14, 32, 22, 38])
```

Now, you have two arrays: the input, x, and the output, y. You should call .reshape() on x because this array must be **two-dimensional**, or more precisely, it must have **one column** and **as many rows as necessary**. That's exactly what the argument (-1, 1) of .reshape() specifies.

This is how x and y look now:

```
>>> x
array([[ 5],
       [15],
       [25],
       [35],
       [45],
       [55]])

>>> y
array([ 5, 20, 14, 32, 22, 38])
```

As you can see, x has two dimensions, and x.shape is (6, 1), while y has a single dimension, and y.shape is (6,).

Step 3: Create a model and fit it

The next step is to create a linear regression model and fit it using the existing data.

Create an instance of the class LinearRegression, which will represent the regression model:

```
>>> model = LinearRegression()he
```

This statement creates the variable model as an instance of LinearRegression. You can provide several optional parameters to LinearRegression:

- fit_intercept is a Boolean that, if True, decides to calculate the intercept b_0 or, if False, considers it equal to zero. It defaults to True.
- normalize is a Boolean that, if True, decides to normalize the input variables. It defaults to False, in which case it doesn't normalize the input variables.
- copy_X is a Boolean that decides whether to copy (True) or overwrite the input variables (False). It's True by default.
- n_jobs is either an integer or None. It represents the number of jobs used in parallel computation. It defaults to None, which usually means one job. -1 means to use all available processors.

Your model as defined above uses the default values of all parameters.

It's time to start using the model. First, you need to call .fit() on model:

```
>>> model.fit(x, y)
LinearRegression()
```

With .fit(), you calculate the optimal values of the weights b_0 and b_1, using the existing input and output, x and y, as the arguments. In other words, .fit() **fits the model**. It returns self, which is the variable model itself. That's why you can replace the last two statements with this one:

```
>>> model = LinearRegression().fit(x, y)
```

This statement does the same thing as the previous two. It's just shorter.

Step 4: Get results

Once you have your model fitted, you can get the results to check whether the model works satisfactorily and to interpret it.

You can obtain the coefficient of determination, R^2, with .score() called on model:

```
>>> r_sq = model.score(x, y)
>>> print(f"coefficient of determination: {r_sq}")
coefficient of determination: 0.7158756137479542
```

When you're applying .score(), the arguments are also the predictor x and response y, and the return value is R^2.

The attributes of model are .intercept_, which represents the coefficient b_0, and .coef_, which represents b_1:

```
>>> print(f"intercept: {model.intercept_}")
intercept: 5.633333333333329

>>> print(f"slope: {model.coef_}")
slope: [0.54]
```

The code above illustrates how to get b_0 and b_1. You can notice that .intercept_ is a scalar, while .coef_ is an array.

Note: In scikit-learn, by convention, a trailing underscore indicates that an attribute is estimated. In this example, .intercept_ and .coef_ are estimated values.

The value of b_0 is approximately 5.63. This illustrates that your model predicts the response 5.63 when x is zero. The value $b_1 = 0.54$ means that the predicted response rises by 0.54 when x is increased by one.

You'll notice that you can provide y as a two-dimensional array as well. In this case, you'll get a similar result. This is how it might look:

```
>>> new_model = LinearRegression().fit(x, y.reshape((-1, 1)))
>>> print(f"intercept: {new_model.intercept_}")
intercept: [5.63333333]

>>> print(f"slope: {new_model.coef_}")
slope: [[0.54]]
```

As you can see, this example is very similar to the previous one, but in this case, .intercept_ is a one-dimensional array with the single element b_0, and .coef_ is a two-dimensional array with the single element b_1.

Step 5: Predict response

Once you have a satisfactory model, then you can use it for predictions with either existing or new data. To obtain the predicted response, use .predict():

```
>>> y_pred = model.predict(x)
>>> print(f"predicted response:\n{y_pred}")
predicted response:
[ 8.33333333 13.73333333 19.13333333 24.53333333 29.93333333 35.33333333]
```

When applying .predict(), you pass the regressor as the argument and get the corresponding predicted response. This is a nearly identical way to predict the response:

```
>>> y_pred = model.intercept_ + model.coef_ * x
>>> print(f"predicted response:\n{y_pred}")
predicted response:
[[ 8.33333333]
 [13.73333333]
 [19.13333333]
 [24.53333333]
 [29.93333333]
 [35.33333333]]
```

In this case, you multiply each element of x with model.coef_ and add model.intercept_ to the product.

The output here differs from the previous example only in dimensions. The predicted response is now a two-dimensional array, while in the previous case, it had one dimension.

If you reduce the number of dimensions of x to one, then these two approaches will yield the same result. You can do this by replacing x with x.reshape(-1), x.flatten(), or x.ravel() when multiplying it with model.coef_.

In practice, regression models are often applied for forecasts. This means that you can use fitted models to calculate the outputs based on new inputs:

```
>>> x_new = np.arange(5).reshape((-1, 1))
>>> x_new
array([[0],
       [1],
       [2],
       [3],
```

[4]])

```
>>> y_new = model.predict(x_new)
>>> y_new
array([5.63333333, 6.17333333, 6.71333333, 7.25333333, 7.79333333])
```

Here .predict() is applied to the new regressor x_new and yields the response y_new. This example conveniently uses arange() from numpy to generate an array with the elements from 0, inclusive, up to but excluding 5—that is, 0, 1, 2, 3, and 4.

Ref: https://realpython.com/linear-regression-in-python/

Question

1. Consider the hepatitis/ pima-indians-diabetes csv file, perform the following date pre-processing.
    1. Load data in Pandas.
    2. Drop columns that aren't useful.
    3. Drop rows with missing values.
    4. Create dummy variables.
    5. Take care of missing data.
    6. Convert the data frame to NumPy.
    7. Divide the data set into training data and test data.

2. a. Construct a CSV file with the following attributes:
Study time in hours of ML lab course (x)
Score out of 10 (y)
The dataset should contain 10 rows.
b. Create a regression model and display the following:
Coefficients: B0 (intercept) and B1 (slope)
RMSE (Root Mean Square Error)
Predicted responses
c. Create a scatter plot of the data points in red color and plot the graph of x vs. predicted y in blue color.
d. Implement the model using two methods:
Pedhazur formula (intuitive)
Calculus method (partial derivatives, refer to class notes)
e. Compare the coefficients obtained using both methods and compare them with the analytical solution.
f. Test your model to predict the score obtained when the study time of a student is 10 hours.
Note: Do not use scikit-learn.

Additional Question

1. a. Consider the hepatitis/diabetes CSV file. Create a regression model and display the following:

- Coefficients: B0 (intercept) and B1 (slope)
- RMSE (Root Mean Square Error)
- Predicted responses

b. Create a scatter plot of the data points in red color and plot the graph of x vs. predicted y in blue color.
c. Implement the model using two methods:
    1. Pedhazur formula (intuitive)
    2. Calculus method (partial derivatives, refer to class notes)

d. Compare the coefficients obtained using both methods. For a given data point, check the predicted y value.
Note: Do not use scikit-learn.

WEEK-04: Polynomial and Multiple Regression

Machine Learning Model evaluation metrics

The various ways to check the performance of our machine learning or deep learning model and why to use one in place of the other. We will discuss terms like:

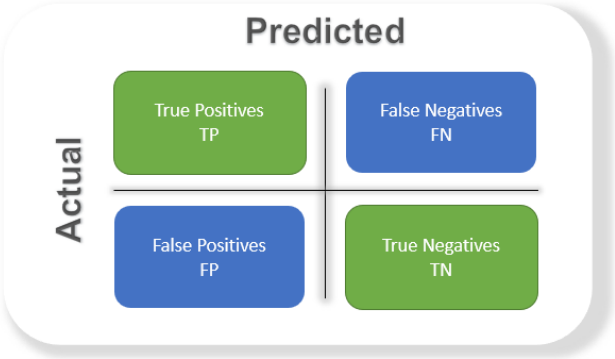
- Confusion matrix
- Accuracy
- Precision
- Recall
- Specificity
- F1 score
- Precision-Recall or PR curve
- **ROC** (**R**eceiver **O**perating **C**haracteristics) curve
- PR vs ROC curve.

For simplicity, we will mostly discuss things in terms of a binary classification problem where let's say we'll have to find if an image is of a cat or a dog. Or a patient is having cancer (positive) or is found healthy (negative). Some common terms to be clear with are:

True positives (TP): Predicted positive and are actually positive.
False positives (FP): Predicted positive and are actually negative.
True negatives (TN): Predicted negative and are actually negative.
False negatives (FN): Predicted negative and are actually positive.

Confusion matrix

It's just a representation of the above parameters in a matrix format. Better visualization is always good :)



Accuracy

The most commonly used metric to judge a model and is actually not a clear indicator of the performance. The worse happens when classes are imbalanced.

$$\frac{TP + TN}{TP + FP + TN + FN}$$

Take for example a cancer detection model. The chances of actually having cancer are very low. Let's say out of 100, 90 of the patients don't have cancer and the remaining 10 actually have it. We don't want to miss on a patient who is having cancer but goes undetected (false negative). Detecting everyone as not having cancer gives an accuracy of 90% straight. The model did nothing here but just gave cancer free for all the 100 predictions.

We surely need better alternatives.

Precision

Percentage of positive instances out of the ***total predicted positive*** instances. Here denominator is the model prediction done as positive from the whole given dataset. Take it as to find out '*how much the model is right when it says it is right'.*

$$\frac{TP}{TP + FP}$$

Recall/Sensitivity/True Positive Rate

Percentage of positive instances out of the ***total actual positive*** instances. Therefore denominator (*TP + FN)* here is the *actual* number of positive instances present in the dataset. Take it as to find out '*how much extra right ones, the model missed when it showed the right ones'.*

$$\frac{TP}{TP + FN}$$

Specificity

Percentage of negative instances out of the ***total actual negative*** instances. Therefore denominator (*TN + FP)* here is the *actual* number of negative instances present in the dataset. It is similar to recall but the shift is on the negative instances. *Like finding out how many healthy patients were not having cancer and were told they don't have cancer*. Kind of a measure to see how separate the classes are.

$$\frac{TN}{TN + FP}$$

F1 score

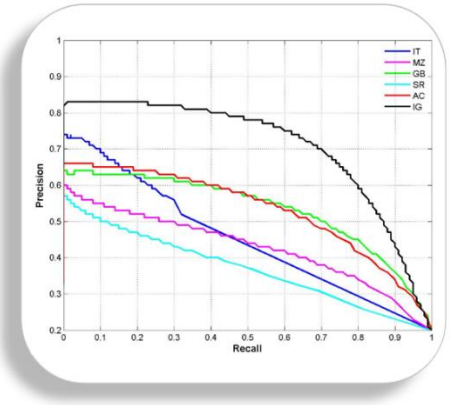
It is the harmonic mean of precision and recall. This takes the contribution of both, so higher the F1 score, the better. See that due to the product in the numerator if one goes low, the final F1 score goes down significantly. So a model does well in F1 score if the positive predicted are actually positives (precision) and doesn't miss out on positives and predicts them negative (recall).

$$\frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2 * precision * recall}{precision + recall}$$

One drawback is that both precision and recall are given equal importance due to which according to our application we may need one higher than the other and F1 score may not be the exact metric for it. Therefore either weighted-F1 score or seeing the PR or ROC curve can help.

PR curve

It is the curve between precision and recall for various threshold values. In the figure below we have 6 predictors showing their respective precision-recall curve for various threshold values. The top right part of the graph is the ideal space where we get high precision and recall. Based on our application we can choose the predictor and the threshold value. PR AUC is just the area under the curve. The higher its numerical value the better.

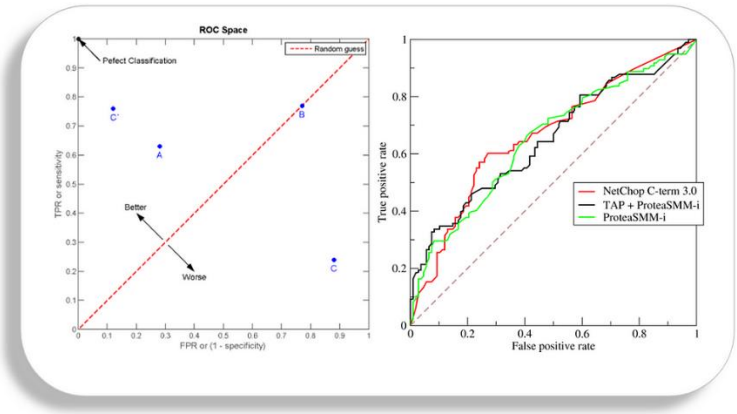


ROC curve

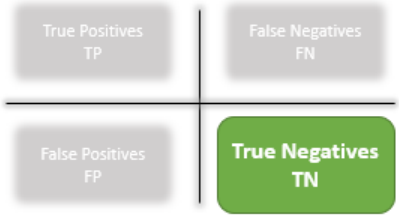
ROC stands for receiver operating characteristic and the graph is plotted against TPR and FPR for various threshold values. As TPR increases FPR also increases. As you can see in the first figure, we have four categories and we want the threshold value that leads us closer to the top left corner. Comparing different predictors (here 3) on a given dataset also becomes easy as you can see in figure 2, one can choose the threshold according to the application at hand. ROC AUC is just the area under the curve, the higher its numerical value the better.

$$True\ Positive\ Rate\ (TPR) = RECALL = \frac{TP}{TP+FN}$$

$$False\ Positive\ Rate\ (FPR) = 1 - Specificity = \frac{FP}{TN+FP}$$



PR vs ROC curve
Both the metrics are widely used to judge a models performance.
Which one to use PR or ROC?



The answer lies in TRUE NEGATIVES.
Due to the absence of TN in the precision-recall equation, they are useful in imbalanced classes. In the case of class imbalance when there is a majority of the negative class. The metric doesn't take much into consideration the high number of TRUE NEGATIVES of the negative class which is in majority, giving better resistance to the imbalance. This is important when the detection of the positive class is very important.
Like to detect cancer patients, which has a high class imbalance because very few have it out of all the diagnosed. We certainly don't want to miss on a person having cancer and going undetected (recall) and be sure the detected one is having it (precision).
Due to the consideration of TN or the negative class in the ROC equation, it is useful when both the classes are important to us. Like the detection of cats and dog. The importance of true negatives makes sure that both the classes are given importance, like the output of a ML/DL model in determining the image is of a cat or a dog.

Classification
Here the goal is to learn a mapping from inputs x to outputs y, where $y \in \{1,...,C\}$, with C being the number of classes. If C = 2, this is called binary classification (in which case we often assume $y \in \{0, 1\}$); if C > 2, this is called multiclass classification. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), we call it multi-label classification, but this is best viewed as predicting multiple related binary class labels (a so-called multiple output model). When we use the term "classification", we will mean multiclass classification with a single output, unless we state otherwise.
One way to formalize the problem is as function approximation. We assume y = f(x) for some unknown function f, and the goal of learning is to estimate the function f given a labeled training set, and then to make predictions using $\hat{y} = \hat{f}(\mathbf{x})$. (We use the hat symbol to denote an estimate.) Our main goal is to make predictions on novel inputs, meaning ones that we have not seen before (this is called generalization), since predicting the response on the training set is easy.

LINEAR REGRESSION

Regression analysis may broadly be defined as the analysis of relationships among variables. This relationship is given as an equation that helps to predict the dependent variable Y through one or more independent variables. In regression analysis, the variable whose values vary with the variations in the values of the other variable(s) is called the **dependent variable** or **response variable**. The other variables which are independent in nature and influence the response variable are called **independent variables**, **predictor variables** or **regressor variables**.

Example: Suppose a statistician employed by a cold drink bottler is analysing the product delivery and service operation for vending machines. He would like to find how the delivery time taken by the delivery man to load and service a machine is related to the volume of delivery cases. The statistician visits 50 randomly chosen retailer shops having vending machines and observes the delivery time (in minutes) and the volume of delivery cases for each shop. He plots those 50 observations on a graph, which shows that an approximate linear relationship exists between the delivery time and delivery volume. If Y represents the delivery time and X, the delivery volume, the equation of a straight line relating these two variables may be given as

$$\mathbf{Y = a + bX \ldots (1)}$$

where a is the intercept and b, the slope.

In such cases, we draw a straight line in the form of equation (1) so that the data points generally fall near the straight line. Now, suppose the points do not fall exactly on the straight line. Then we should modify equation (1) to minimise the difference between the observed value of Y and that given by the straight line (a + b X). This is known as **error**.

The error e, which is the difference between the observed value and the predicted value of the variable of interest Y, may be conveniently assumed as a statistical error. This error term accounts for the variability in Y that cannot be explained by the linear relationship between X and Y. It may arise due to the effects of other factors. Thus, a more plausible model for the variable of interest (Y) may be given as

$$\mathbf{Y = a + b\,X + e \ldots (2)}$$

where the intercept a and the slope b are unknown constants and e is a random error component.

Equation (2) is called a **linear regression model**.

Fitting of regression line

Let the given data of n pairs of observations on X and Y be as follows:

X: $X_1\ X_2\ X_3$. … … Xi ….. … Xn

Y: $Y_1\ Y_2\ Y_3$. … … Yi …..… Yn

where Y is the dependent variable and X, the independent variable.

Suppose, we wish to fit the following simple regression equation to the data: Y = a + bX

where a is the intercept and b is the slope of the equation.

For fitting equation to the data on (X, Y), we follow the steps given below:

Step 1: We draw a scatter diagram by plotting the (X, Y) points given in data.

Step 2: We construct a table as given below and take the sum of the values of X_i, Y_i, X_iY_i, and X_i^2. We write the values of $\sum X$, $\sum Y$, $\sum XY$ and $\sum X^2$ in the last row.

Step 3: We express of aˆ given in equation (1) as follows:

$$\hat{a} = \bar{Y} - b\bar{X} = \frac{1}{n}\left[\sum Y - b\sum X\right]$$

Where

$$\hat{b} = \frac{n\sum XY - \sum X \sum Y}{n\sum X^2 - \left(\sum X\right)^2}$$

substitute above values in the regression equation and get

$$\hat{Y} = \hat{a} + \hat{b}X$$

POLYNOMIAL REGRESSION

Some engineering data is poorly represented by a straight line. For these cases, a curve would be better suited to it these data. The method is to it polynomials to the data using polynomial regression.

Polynomial Regression is a form of regression analysis in which the relationship between the independent variables and dependent variables are modeled in the nth degree polynomial. Polynomial Regression models are usually fit with the method of least squares. The **least square method** minimizes the variance of the coefficients, under the Gauss Markov Theorem. Polynomial Regression is a special case of Linear Regression where we fit the polynomial equation on the data with a curvilinear relationship between the dependent and independent variables. A Quadratic Equation is a Polynomial Equation of 2nd Degree. However, this degree can increase to nth values.

The least-squares procedure can be readily extended to it the data to a higher-order polynomial. For example, suppose that we it a second-order polynomial or quadratic:

$$y = a_0 + a_1 x + a_2 x^2 + e$$

Where x-independent variable, y-dependent variable, a0,a1 and a2 are coefficients, and e –error term.

For this case the sum of the squares of the residuals is

$$S_r = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i - a_2 x_i^2)^2$$

we take the derivative with respect to each of the unknown coefficients of the polynomial, as in

$$\frac{\partial S_r}{\partial a_0} = -2 \sum (y_i - a_0 - a_1 x_i - a_2 x_i^2)$$

$$\frac{\partial S_r}{\partial a_1} = -2 \sum x_i (y_i - a_0 - a_1 x_i - a_2 x_i^2)$$

$$\frac{\partial S_r}{\partial a_2} = -2 \sum x_i^2 (y_i - a_0 - a_1 x_i - a_2 x_i^2)$$

These equations can be set equal to zero and rearranged to develop the following set of normal equations:

$$(n) a_0 + \left(\sum x_i\right) a_1 + \left(\sum x_i^2\right) a_2 = \sum y_i$$

$$\left(\sum x_i\right) a_0 + \left(\sum x_i^2\right) a_1 + \left(\sum x_i^3\right) a_2 = \sum x_i y_i$$

$$\left(\sum x_i^2\right) a_0 + \left(\sum x_i^3\right) a_1 + \left(\sum x_i^4\right) a_2 = \sum x_i^2 y_i$$

where all summations are from i = 1 through n. Note that the above three equations are linear and have three unknowns: a0 , a1 , and a2 . The coefficients of the unknowns can be calculated directly from the observed data. For this case, we see that the problem of determining a least-squares second-order polynomial is equivalent to solving a system of three simultaneous linear equations.

These equations can be rewritten in matrix form as follows:

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} x_i^2 y_i \end{bmatrix}$$

Where:

- *n* is the number of data points.
- *xi* and *yi* are the values of the independent and dependent variables for the *i*-th data point.

The most common way to solve a system of linear equations is by using matrix algebra and methods like Gaussian elimination or matrix inversion. Here are the general steps to solve a system of linear equations:

Step 1: Write Down the System of Equations

Write the system of linear equations in standard form, where each equation has the form:

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = b$$

Where:

- $a1,a2,\ldots,an$ are the coefficients of the variables $x1,x2,\ldots,xn$.
- b is the constant on the right-hand side of the equation.

You should have as many equations as there are variables.

Step 2: Create a Coefficient Matrix

Construct a coefficient matrix A by extracting the coefficients of the variables from the left side of each equation. This matrix will have dimensions $n \times n$ if you have n variables.

Step 3: Create a Right-Hand Side Vector

Create a right-hand side vector B by extracting the constants from the right side of each equation. This vector will have dimensions $n \times 1$.

Step 4: Solve the System

There are several methods to solve the system:

There are several methods to solve the system:

- **Matrix Inversion**: If A is invertible (non-singular), you can solve the system using the formula $X=A^{-1}B$, where X is the vector of solutions.
- **Gaussian Elimination**: Use row operations to transform the augmented matrix $[A|B]$ into row-echelon or reduced row-echelon form. Then, back-substitute to find the values of the variables.
- **Matrix Factorization**: Factorize A into LU or QR form and use the factorization to solve the system more efficiently.
- **Numerical Methods**: For large or ill-conditioned systems, numerical methods like the Gauss-Seidel method or the Conjugate Gradient method are used.

Here's a simple example using matrix inversion in Python:

```
import numpy as np
# Define the coefficient matrix A and the right-hand side vector B
A = np.array([[2, 1], [1, 3]])
B = np.array([4, 7])
# Solve for the variables X using matrix inversion
X = np.linalg.inv(A).dot(B)
print("Solution:")
print(X)
```

The above code solves the system $2x_1+x_2=4$ and $x_1+3x_2=7$ and prints the values of x_1 and x_2 as the solution.

Solution:

[1. 2.]

Polynomial regression is a type of regression analysis in which the relationship between the independent variable (X) and the dependent variable (Y) is modeled as an nth-degree polynomial. In Python, you can perform polynomial regression using libraries like NumPy and scikit-learn. Here's a basic example of polynomial regression using scikit-learn:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
import matplotlib.pyplot as plt

# Generate sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
Y = 4 + 3 * X + np.random.randn(100, 1)

# Fit a polynomial regression model
degree = 2  # You can change the degree as needed
poly_features = PolynomialFeatures(degree=degree)
X_poly = poly_features.fit_transform(X)
```

```
model = LinearRegression()
model.fit(X_poly, Y)

# Make predictions
X_new = np.linspace(0, 2, 100).reshape(-1, 1)
X_new_poly = poly_features.transform(X_new)
Y_new = model.predict(X_new_poly)

# Plot the original data and the polynomial regression curve
plt.scatter(X, Y, label='Original Data')
plt.plot(X_new, Y_new, 'r-', label='Polynomial Regression')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
# The coefficients of the multivariate polynomial regression model
coefficients = model.coef_
intercept = model.intercept_
print("Coefficients:")
print(coefficients)
print("Intercept:")
print(intercept)
```

In this example:

1. We generate some sample data points with random noise.
2. We use **PolynomialFeatures** from scikit-learn to transform our input features **X** into polynomial features up to a specified degree.
3. We then fit a linear regression model to the polynomial features.
4. Finally, we make predictions using the model and plot the original data along with the polynomial regression curve.

You can adjust the **degree** variable to control the degree of the polynomial you want to fit to your data. Higher-degree polynomials can capture more complex relationships but may also lead to overfitting, so be cautious when choosing the degree.

MULTIPLE LINEAR REGRESSION

Multiple linear regression is a method we can use to quantify the relationship between two or more predictor variables and a response variable.

The Regression Line: With one independent variable, we may write the regression equation as:

$$Y = a + bX + e$$

Where Y is an observed score on the dependent variable, *a* is the intercept, *b* is the slope, X is the observed score on the independent variable, and *e* is an error or residual.

We can extend this to any number of independent variables:

$$Y = a + b_1X_1 + b_2X_2 + \ldots + b_kX_k + e \quad (3.1)$$

Note that we have *k* independent variables and a slope for each. We still have one error and one intercept. Again we want to choose the estimates of *a* and *b* so as to minimize the sum of squared errors of prediction. The prediction equation is:

$$Y' = a + b_1X_1 + b_2X_2 + \ldots + b_kX_k \quad (3.2)$$

Finding the values of *b* (the slopes) is tricky for k>2 independent variables, and you really need matrix algebra to see the computations. It's simpler for k=2 IVs, which we will discuss here. But the basic ideas are the same no matter how many independent variables you have. If you understand the meaning of the slopes with two independent variables, you will likely be good no matter how many you have.

For the one variable case, the calculation of *b* and *a* was:

$$b = \frac{\sum xy}{\sum x^2}$$

$$a = \overline{Y} - b\overline{X}$$

For the two variable case:

$$b_1 = \frac{(\sum x_2^2)(\sum x_1 y) - (\sum x_1 x_2)(\sum x_2 y)}{(\sum x_1^2)(\sum x_2^2) - (\sum x_1 x_2)^2}$$

and

$$b_2 = \frac{(\sum x_1^2)(\sum x_2 y) - (\sum x_1 x_2)(\sum x_1 y)}{(\sum x_1^2)(\sum x_2^2) - (\sum x_1 x_2)^2}$$

where

- $\Sigma x_1^2 = \Sigma X_1^2 - ((\Sigma X_1)^2 / n)$
- $\Sigma x_2^2 = \Sigma X_2^2 - ((\Sigma X_2)^2 / n)$
- $\Sigma x_1 y = \Sigma X_1 y - ((\Sigma X_1 \Sigma y) / n)$
- $\Sigma x_2 y = \Sigma X_2 y - ((\Sigma X_2 \Sigma y) / n)$
- $\Sigma x_1 x_2 = \Sigma X_1 X_2 - ((\Sigma X_1 \Sigma X_2) / n)$

At this point, you should notice that all the terms from the one variable case appear in the two variable case. In the two variable case, the other X variable also appears in the equation. For example, X_2 appears in the equation for b_1. Note that terms corresponding to the variance of both X variables occur in the slopes. Also note that a term corresponding to the covariance of X1 and X2 (sum of deviation cross-products) also appears in the formula for the slope.
The equation for *a* with two independent variables is:

$$a = \overline{Y} - b_1\overline{X}_1 - b_2\overline{X}_2$$

This equation is a straight-forward generalization of the case for one independent variable.

MLR MATRIX FORMULATION

Multiple linear regression is a generalized form of simple linear regression, in which the data contains multiple explanatory variables.

	SLR		MLR				
	x	**y**	$\mathbf{x}_1$	$\mathbf{x}_2$	...	$\mathbf{x}_p$	**y**
case 1:	x_1	y_1	x_{11}	x_{12}	...	x_{1p}	y_1
case 2:	x_2	y_2	x_{21}	x_{22}	...	x_{2p}	y_2
	⋮	⋮	⋮	⋮	⋱	⋮	⋮
case *n*:	x_n	y_n	x_{n1}	x_{n2}	...	x_{np}	y_n

- For SLR, we observe pairs of variables.
- For MLR, we observe rows of variables.
- Each row (or pair) is called a case, a record, or a data point
- yi is the response (or dependent variable) of the ith observation
- There are p explanatory variables (or covariates, predictors, independent variables), and xik is the value of the explanatory variable xk of the ith case

$$y_i = \beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip} + \varepsilon_i \quad \text{where } \varepsilon_i\text{'s are i.i.d. } N(0, \sigma^2)$$

In the model above,

- ε_i's (errors, or noise) are i.i.d. $N(0, \sigma^2)$
- Parameters include:

$\beta_0 =$ intercept;

$\beta_k =$ regression coefficients (slope) for the kth explanatory variable, $k = 1, \ldots, p$

$\sigma^2 = \mathrm{Var}(\varepsilon_i)$ is the variance of errors

Refer further https://online.stat.psu.edu/stat462/node/132/

REGRESSION METRICS

Residual

The difference between the fitted value $\hat{Y}_i$ and Yi is known as the residual and is denoted by

$r_i = Y_i - \hat{Y}_i, \qquad i = 1, 2, \ldots, n$

The role of the residuals and its analysis is very important in regression modelling.

Mean Squared Error (MSE)

Mean squared error (MSE) measures the amount of error in statistical models. It assesses the average squared difference between the observed and predicted values. When a model has no error, the MSE equals zero. As model error increases, its value increases. The mean squared error is also known as the ***mean squared deviation (MSD)***.

For example, in regression, the mean squared error represents the average squared residual/error.

$$MSE = \frac{\sum(y_i - \hat{y}_i)^2}{n}$$

Where:
yi is the ith observed value.
ŷi is the corresponding predicted value.
n = the number of observations.

Squaring the error gives higher weight to the outliers, which results in a smooth gradient for small errors. Optimization algorithms benefit from this penalization for large errors as it is helpful in finding the optimum values for parameters. MSE will never be negative since the errors are squared. The value of the error ranges from zero to infinity. MSE increases exponentially with an increase in error. A good model will have an MSE value closer to zero.

Root Mean Square Error (RMSE)

Root Mean Squared Error (RMSE) is a popular metric used in machine learning and statistics to measure the accuracy of a predictive model. It quantifies the differences between predicted values and actual values, squaring the errors, taking the mean, and then finding the square root. RMSE provides a clear understanding of the model's performance, with lower values indicating better predictive accuracy.

RMSE is computed by taking the square root of MSE. RMSE is also called the Root Mean Square Deviation. It measures the average magnitude of the errors and is concerned with the deviations from the actual value. RMSE value with zero indicates that the model has a perfect fit. The lower the RMSE, the better the model and its predictions. A higher RMSE indicates that there is a large deviation from the residual to the ground truth. RMSE can be used with different features as it helps in figuring out if the feature is improving the model's prediction or not.

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

Ref: https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics

Questions:

1. The data set of size n = 15 (Yield data) contains measurements of yield from an experiment done at five different temperature levels. The variables are y = yield and x = temperature in degrees Fahrenheit. The table below gives the data used for this analysis.

i	Temp.	Yield
1	50	3.3
2	50	2.8
3	50	2.9
4	70	2.3
5	70	2.6
6	70	2.1
7	80	2.5
8	80	2.9
9	80	2.4
10	90	3.0
11	90	3.1
12	90	2.8
13	100	3.3
14	100	3.5
15	100	3.0

a. Create a CSV file with sample data.
b. Write a Python function program to:
Find the fitted simple linear and polynomial regression equations for the given data.
c. Compare the coefficients obtained from manually intuitive and matrix formulation methods with your program.
d. Plot the scatterplot of the raw data and then another scatterplot with lines pertaining to a linear fit and a quadratic fit overlayed.
e. Compute the error, MSE, and RMSE.
Note: Do not use scikit-learn.

2. When heart muscle is deprived of oxygen, the tissue dies and leads to a heart attack ("myocardial infarction"). Apparently, cooling the heart reduces the size of the heart attack. It is not known, however, whether cooling is only effective if it takes place before the blood, flow to the heart becomes restricted. Some researchers (Hale, et al, 1997) hypothesized that cooling the heart would be effective in reducing the size of the heart attack even if it takes place after the blood flow becomes restricted.
To investigate their hypothesis, the researchers conducted an experiment on 32 anesthetized rabbits that were subjected to a heart attack. The researchers established three experimental groups:

- *Rabbits whose hearts were cooled to 6º C within 5 minutes of the blocked artery ("**early cooling**")*
- *Rabbits whose hearts were cooled to 6º C within 25 minutes of the blocked artery ("**late cooling**")*
- *Rabbits whose hearts were not cooled at all ("**no cooling**")*

*At the end of the experiment, the researchers measured the size of the **infarcted** (i.e., damaged) **area** (in grams) in each of the 32 rabbits. But, as you can imagine, there is great variability in the size of hearts. The size of a rabbit's infarcted area may be large only because it has a larger heart. Therefore, in order to adjust for differences in heart sizes, the researchers also measured the size of the **region at risk** for infarction (in grams) in each of the 32 rabbits.*

Infarc	***Area***	***Group***	***X2***	***X3***
0.119	*0.34*	*3*	*0*	*0*
0.19	*0.64*	*3*	*0*	*0*
0.395	*0.76*	*3*	*0*	*0*
0.469	*0.83*	*3*	*0*	*0*
0.13	*0.73*	*3*	*0*	*0*
0.311	*0.82*	*3*	*0*	*0*
0.418	*0.95*	*3*	*0*	*0*

0.48	*1.06*	*3*	*0*	*0*
0.687	*1.2*	*3*	*0*	*0*
0.847	*1.47*	*3*	*0*	*0*
0.062	*0.44*	*1*	*1*	*0*
0.122	*0.77*	*1*	*1*	*0*
0.033	*0.9*	*1*	*1*	*0*
0.102	*1.07*	*1*	*1*	*0*
0.206	*1.01*	*1*	*1*	*0*
0.249	*1.03*	*1*	*1*	*0*
0.22	*1.16*	*1*	*1*	*0*
0.299	*1.21*	*1*	*1*	*0*
0.35	*1.2*	*1*	*1*	*0*
0.35	*1.22*	*1*	*1*	*0*
0.588	*0.99*	*1*	*1*	*0*
0.379	*0.77*	*2*	*0*	*1*
0.149	*1.05*	*2*	*0*	*1*
0.316	*1.06*	*2*	*0*	*1*
0.39	*1.02*	*2*	*0*	*1*
0.429	*0.99*	*2*	*0*	*1*
0.477	*0.97*	*2*	*0*	*1*
0.439	*1.12*	*2*	*0*	*1*
0.446	*1.23*	*2*	*0*	*1*
0.538	*1.19*	*2*	*0*	*1*
0.625	*1.22*	*2*	*0*	*1*
0.974	*1.4*	*2*	*0*	*1*

With their measurements in hand, the researchers' primary research question was: Does the mean size of the infarcted area differ among the three treatment groups — no cooling, early cooling, and late cooling — when controlling for the size of the region at risk for infarction?
A regression model that the researchers might use in answering their research question is:

$$y_i=(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2}+\beta_3 x_{i3})+\epsilon_i$$

where:

- y_i *is the size of the infarcted area (in grams) of rabbit i*
- x_{i1} *is the size of the region at risk (in grams) of rabbit i*
- x_{i2} *= 1 if early cooling of rabbit i, 0 if not*
- x_{i3} *= 1 if late cooling of rabbit i, 0 if not*

and the independent error terms ε_i *follow a normal distribution with mean 0 and equal variance* σ^2*. Illustrates the need for being able to "translate" a research question into a statistical procedure. Often, the procedure involves four steps, namely:*

- *formulating a multiple regression model*
- *determining how the model helps answer the research question*
- *checking the model*
- *and performing a hypothesis test (or calculating a confidence interval)*

a. Create a CSV file with sample data.
b. Write a Python function program to:
c. Find the fitted multiple linear regression equation for the given data.
d. Compare the coefficients obtained manually using intuitive and matrix formulation methods with your program.
e. Plot the data adorned with the estimated regression equation.
f. Compute the error, MSE, and RMSE.
Note: Do not use scikit-learn.

Additional questions

1. Consider the Table Contains the Average Annual Gold Rate from 1965 – 2022. Gold prices fluctuated throughout the year 2020 because of the COVID-19 epidemic. With gold functioning as a safe haven for investors, demand for the precious metal grew, and its price followed suit. During the epidemic, the stock market weakened, but it began to recover by the end of 2020 when the price of gold fell slightly.

It's crucial to remember that gold prices fluctuate during the year, and the figure below represents the average price for that year.

With the exception of a few lows shared across a few years, The table shows that the gold price trend has always been upward, supporting the claim that gold is a secure investment over extended periods of time.

Create CSV file and Write a python program to find the fitted simple linear regression equation for the given data. Compare the coefficients obtained from sklearn model with your program. Compute the error, MSE and RMSE. Predict the gold price with the year 2025 for 1 gram.

This Table Contains the Average Annual Gold Rate from 1965 - 2022			
Year	**Price (24 karat per 10 grams)**	**Year**	**Price (24 karat per 10 grams)**
2022	₹ 52,950	1993	₹ 4,140
2021	₹ 50,045	1992	₹ 4,334
2020	₹ 48,651	1991	₹ 3,466
2019	₹ 35,220	1990	₹ 3,200
2018	₹ 31,438	1989	₹ 3,140
2017	₹ 29,667	1988	₹ 3,130
2016	₹ 28,623	1987	₹ 2,570
2015	₹ 26,343	1986	₹ 2,140
2014	₹ 28,006	1985	₹ 2,130
2013	₹ 29,600	1984	₹ 1,970
2012	₹ 31,050	1983	₹ 1,800
2011	₹ 26,400	1982	₹ 1,645
2010	₹ 18,500	1981	₹ 1,800
2009	₹ 14,500	1980	₹ 1,330
2008	₹ 12,500	1979	₹ 937
2007	₹ 10,800	1978	₹ 685
2006	₹ 8,400	1977	₹ 486

2005	₹ 7,000	1976	₹ 432
2004	₹ 5,850	1975	₹ 540
2003	₹ 5,600	1974	₹ 506
2002	₹ 4,990	1973	₹ 279
2001	₹ 4,300	1972	₹ 202
2000	₹ 4,400	1971	₹ 193
1999	₹ 4,234	1970	₹ 184
1998	₹ 4,045	1969	₹ 176
1997	₹ 4,725	1968	₹ 162
1996	₹ 5,160	1967	₹ 103
1995	₹ 4,680	1966	₹ 84
1994	₹ 4,598	1965	₹ 72

2. Consider the Question no 1 gold price with following year-wise silver price. Create a CSV file and Write a python program to find the fitted multiple linear regression equation for the given data. Compare the coefficients obtained from sklearn model with your program. Compute the error, MSE and RMSE. Predict the gold and silver price with the year 2024 for 1 gram.

Year	**Silver Rates in Rs./Kg.**	**Year**	**Silver Rates in Rs./Kg.**
1981	Rs.2715	2002	Rs.7875
1982	Rs.2720	2003	Rs.7695
1983	Rs.3105	2004	Rs.11770
1984	Rs.3570	2005	Rs.10675
1985	Rs.3955	2006	Rs.17405
1986	Rs.4015	2007	Rs.19520
1987	Rs.4794	2008	Rs.23625
1988	Rs.6066	2009	Rs.22165
1989	Rs.6755	2010	Rs.27255
1990	Rs.6463	2011	Rs.56900
1991	Rs.6646	2012	Rs.56290
1992	Rs.8040	2013	Rs.54030
1993	Rs.5489	2014	Rs.43070
1994	Rs.7124	2015	Rs.37825
1995	Rs.6335	2016	Rs.36990
1996	Rs.7346	2017	Rs.37825
1997	Rs.7345	2018	Rs.41400
1998	Rs.8560	2019	Rs.40600
1999	Rs.7615	2020	Rs.63435
2000	Rs.7900	2021	Rs.62572
2001	Rs.7215	2022	Rs.55100

3. Suppose you have a gold/silver price dataset with a single independent variable (X) and a dependent variable (Y). You want to fit a polynomial regression model to this data. Implement the process of selecting the appropriate degree for the polynomial (e.g., linear, quadratic, cubic) based on the dataset using Python.

4. Imagine you have a gold and silver price dataset with two independent variables (X1 and X2) and a dependent variable (Y). Implement in python, how you can perform multivariate polynomial regression to model the relationship between the independent variables and the dependent variable.

WEEK -05: LOGISTIC REGRESSION AND STOCHASTIC GRADIENT DESCENT (SGD)

Logistic regression is a supervised machine learning algorithm mainly used for classification tasks where the goal is to predict the probability that an instance of belonging to a given class or not. It is a kind of statistical algorithm, which analyze the relationship between a set of independent variables and the dependent binary variables. It is a powerful tool for decision-making. For example email spam or not.

It is used for classification algorithms its name is logistic regression. it's referred to as regression because it takes the output of the linear regression function as input and uses a sigmoid function to estimate the probability for the given class. The difference between linear regression and logistic regression is that linear regression output is the continuous value that can be anything while logistic regression predicts the probability that an instance belongs to a given class or not.

It is used for predicting the categorical dependent variable using a given set of independent variables.

- Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value.
- It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.
- Logistic Regression is much similar to the Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas Logistic regression is used for solving the classification problems.
- In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
- The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
- Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.
- Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification.

Logistic Function (Sigmoid Function):

- The sigmoid function is a mathematical function used to map the predicted values to probabilities.
- It maps any real value into another value within a range of 0 and 1. o The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form.
- The S-form curve is called the Sigmoid function or the logistic function.
- In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.

Type of Logistic Regression:

On the basis of the categories, Logistic Regression can be classified into three types:

1. **Binomial:** In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
2. **Multinomial:** In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
3. **Ordinal:** In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

Terminologies involved in Logistic Regression:

Here are some common terms involved in logistic regression:

- **Independent variables:** The input characteristics or predictor factors applied to the dependent variable's predictions.
- **Dependent variable:** The target variable in a logistic regression model, which we are trying to predict.
- **Logistic function:** The formula used to represent how the independent and dependent variables relate to one another. The logistic function transforms the input variables into a probability value between 0 and 1, which represents the likelihood of the dependent variable being 1 or 0.
- **Odds:** It is the ratio of something occurring to something not occurring. it is different from probability as the probability is the ratio of something occurring to everything that could possibly occur.
- **Log-odds:** The log-odds, also known as the logit function, is the natural logarithm of the odds. In logistic regression, the log odds of the dependent variable are modeled as a linear combination of the independent variables and the intercept.

- **Coefficient:** The logistic regression model's estimated parameters, show how the independent and dependent variables relate to one another.
- **Intercept:** A constant term in the logistic regression model, which represents the log odds when all independent variables are equal to zero.
- **Maximum likelihood estimation:** The method used to estimate the coefficients of the logistic regression model, which maximizes the likelihood of observing the data given the model.

How does Logistic Regression work?

The logistic regression model transforms the linear regression function continuous value output into categorical value output using a sigmoid function, which maps any real-valued set of independent variables input into a value between 0 and 1. This function is known as the logistic function.

Let the independent input features be

$$X = \begin{bmatrix} x_{11} & \dots & x_{1m} \\ x_{21} & \dots & x_{2m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{bmatrix}$$

and the dependent variable is Y having only binary value i.e. 0 or 1.

$$Y = \begin{cases} 0 & \text{if } Class\ 1 \\ 1 & \text{if } Class\ 2 \end{cases}$$

then apply the multi-linear function to the input variables X

$$z = (\sum_{i=1}^{n} w_i x_i) + b$$

x_i $w_i = [w_1, w_2, w_3, \cdots, w_m]$

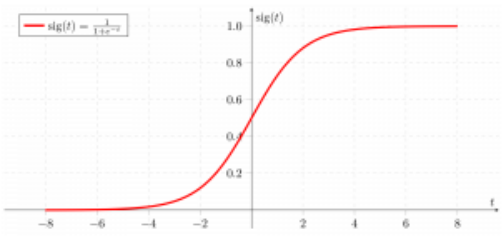
Here is the ith observation of X, is the weights or Coefficient, and b is the bias term also known as intercept. simply this can be represented as the dot product of weight and bias.

$$z = w \cdot X + b$$

Sigmoid Function

Now we use the sigmoid function where the input will be z and we find the probability between 0 and 1. i.e predicted y.

$$\sigma(z) = \frac{1}{1-e^{z}}$$



Sigmoid function

As shown above, the figure sigmoid function converts the continuous variable data into the probability i.e. between 0 and 1.

- $\sigma(z)$ tends towards 1 as $z \to \infty$
- $\sigma(z)$ tends towards 0 as $z \to -\infty$
- $\sigma(z)$ is always bounded between 0 and 1

where the probability of being a class can be measured as:

$$P(y=1) = \sigma(z)$$
$$P(y=0) = 1 - \sigma(z)$$

Logistic Regression Equation

The odd is the ratio of something occurring to something not occurring. it is different from probability as the probability is the ratio of something occurring to everything that could possibly occur. so odd will be

$$\frac{p(x)}{1-p(x)} = e^z$$

Applying natural log on odd. then log odd will be

$$\log\left[\frac{p(x)}{1-p(x)}\right] = z$$
$$\log\left[\frac{p(x)}{1-p(x)}\right] = w \cdot X + b$$

then the final logistic regression equation will be:

$$p(X; b, w) = \frac{e^{w \cdot X + b}}{1 + e^{w \cdot X + b}} = \frac{1}{1 + e^{-w \cdot X + b}}$$

Likelihood function for Logistic Regression

The predicted probabilities will p(X;b,w) = p(x) for y=1 and for y = 0 predicted probabilities will 1-p(X;b,w) = 1-p(x)

$$L(b, w) = \prod_{i=1} np(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$$

Taking natural logs on both sides

$$\begin{aligned} l(b, w) = \log(L(b, w)) &= \sum_{i=1}^{n} y_i \log p(x_i) \; + \; (1 - y_i) \log(1 - p(x_i)) \\ &= \sum_{i=1}^{n} y_i \log p(x_i) + \log(1 - p(x_i)) - y_i \log(1 - p(x_i)) \\ &= \sum_{i=1}^{n} \log(1 - p(x_i)) + \sum_{i=1}^{n} y_i \log \frac{p(x_i)}{1 - p(x_i} \\ &= \sum_{i=1}^{n} - \log 1 - e^{-(w \cdot x_i + b)} + \sum_{i=1}^{n} y_i (w \cdot x_i + b) \\ &= \sum_{i=1}^{n} - \log 1 + e^{w \cdot x_i + b} + \sum_{i=1}^{n} y_i (w \cdot x_i + b) \end{aligned}$$

Gradient of the log-likelihood function

To find the maximum likelihood estimates, we differentiate w.r.t w,

$$\begin{aligned}\frac{\partial J(l(b,w)}{\partial w_j} &= -\sum_{i=n}^{n} \frac{1}{1+e^{w\cdot x_i+b}} e^{w\cdot x_i+b} x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \\ &= -\sum_{i=n}^{n} p(x_i;b,w) x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \\ &= \sum_{i=n}^{n} (y_i - p(x_i;b,w)) x_{ij}\end{aligned}$$

Assumptions for Logistic Regression

The assumptions for Logistic regression are as follows:

- **Independent observations:** Each observation is independent of the other. meaning there is no correlation between any input variables.
- **Binary dependent variables:** It takes the assumption that the dependent variable must be binary or dichotomous, meaning it can take only two values. For more than two categories softmax functions are used.
- **Linearity relationship between independent variables and log odds:** The relationship between the independent variables and the log odds of the dependent variable should be linear.
- **No outliers:** There should be no outliers in the dataset.
- **Large sample size:** The sample size is sufficiently large

Logistic regression is a commonly used algorithm for binary classification problems. Here's a Python program for logistic regression with an example using the popular Iris dataset for classification:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data[:, :2]  # We'll use only the first two features for simplicity
y = (iris.target != 0) * 1  # Convert target labels to binary (0 or 1)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the feature data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create a logistic regression model
model = LogisticRegression(solver='liblinear')

# Train the model
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
```

```
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Plot the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.RdBu, alpha=0.8)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu)
plt.xlabel('Sepal Length (standardized)')
plt.ylabel('Sepal Width (standardized)')
plt.title('Logistic Regression Decision Boundary')
plt.show()
```

In this program:

1. We load the Iris dataset and select only the first two features for binary classification.
2. We split the data into training and testing sets using **train_test_split**.
3. We standardize the feature data using **StandardScaler** to have zero mean and unit variance.
4. We create a logistic regression model using **LogisticRegression** from scikit-learn.
5. We train the model on the training data.
6. We make predictions on the test data and evaluate the model's performance using a confusion matrix and a classification report.
7. Finally, we plot the decision boundary of the logistic regression model to visualize how it separates the two classes.

You can adjust the dataset, features, and model parameters as needed for your specific classification problem.

Stochastic gradient descent (SGD)

Ref: https://machinelearningmastery.com/linear-regression-tutorial-using-gradient-descent-for-machine-learning/

Questions

Note: "Refer to the table that contains the average annual gold rate from 1965 to 2022 and the year-wise silver prices available at Week-4 ML lab manual."

1. Let's say we have a fictional dataset of pairs of variables, a mother and her daughter's heights:

mother height	daughter height
58	60
62	60
60	58
64	60
67	70
70	72

height of mother(x)/daughter (y) pairs

Create a CSV file for the above training data and write a Python function program to find the fitted linear regression with gradient descent technique. Compare the coefficients obtained from the sklearn model with your program. Compute the error, MSE and RMSE. Plot the graph Daughter height (Y-axis) vs Mother height (X-axis) with blue colour. Also, plot the line of best fit with red colour. Predict her daughter's height with given a new mother height as 63. Plot the graph of error in y-axis and iteration in x-axis with 4 epochs (6x4=24 iterations).

2.

Hours of Study (X)	Pass (Y)
1	0
2	0
3	0
4	0
5	1
6	1
7	1
8	1

Here, X is the number of hours of study, and Y is the outcome (0 for fail, 1 for pass).

Create a CSV file for the above training data and write a Python function program to find the fitted logistic regression with gradient descent technique. Compare the coefficients obtained from the sklearn model with your program. Compute the predicted y and assign the class label (prediction = 0 IF p(fail) < 0.5 and prediction = 1 IF p(pass) >= 0.5) and compute the accuracy. Find the error for each iteration and predict the probability that a student will pass the exam if they study for a) 3.5 hours b) 7.5 hours. Plot the graph of error in y-axis and iteration in x-axis with 3 epochs (8x3=24 iterations).

3.

	x_1	x_2	y
1)	4	1	2
2)	2	8	-14
3)	1	0	1
4)	3	2	-1
5)	1	4	-7
6)	6	7	-8

Consider the above dataset with two independent variables (X1 and X2) and a dependent variable (Y). Implement in python, how you can perform the logistic regression to model the relationship between the independent variables and the dependent variable.

Additional Questions

1. Write a python program for SGD by considering the year wise gold and silver price data. Compare the coefficients obtained from sklearn model with your program. Compute the error, MSE and RMSE. Predict the gold price with the year 2025 for 1 gram and, gold and silver price with the year 2024 for 1 gram.

2. Suppose you have a gold/silver price dataset with a single independent variable (X) and a dependent variable (Y). You want to fit a logistic regression model to this data. Develop an example code snippet in Python.

3. Consider the gold and/or silver price dataset and to evaluate a logistic regression model using ROC and AUC:

1. Calculate predicted probabilities for each instance in the test set.
2. Plot the ROC curve using the True Positive Rate (Sensitivity) and False Positive Rate.
3. Calculate the AUC to summarize the model's performance.