Practical 1

Aim: Implement Breadth first search or Depth first search.

1) Depth First Search

- It is a recursive algorithm to search all the vertices of a tree data structure or a graph.
- The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.
- Because of the recursive nature, stack data structure can be used to implement the DFS algorithm.

The process to implement the DFS traversal is given as follows :

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back, and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Code to implement depth first search from

collections import defaultdict

```
class Graph:

    def __init__(self):

        self.graph = defaultdict(list)

    def addEdge(self, u, v):
self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
visited.add(v)
print(v, end=' ')

        for neighbour in self.graph[v]:
if neighbour not in visited:
self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        self.DFSUtil(v, visited)

if __name__ == "__main__":
```

```
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("\nFollowing is DFS from (starting from vertex 2)")
g.DFS(1)
```

Output :

```
Following is DFS from (starting from vertex 0)
0 1 2 3
```

```
Following is DFS from (starting from vertex 1)
1 2 0 3
```

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

2) Breadth First Search

- Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes.
- Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.
- It goes level by level i.e., hierarchical .
- There are many ways to traverse the graph, but among them is BFS.

Follow the below method to implement BFS traversal :

1. Declare a queue and insert the starting vertex.
2. Initialize a visited array and mark the starting vertex as visited.
3. Follow the below process till the queue becomes empty:
4. Remove the first vertex of the queue.
5. Mark that vertex as visited.
6. Insert all the unvisited neighbors of the vertex into the queue.

```
# Code to implement breadth first search
from collections import defaultdict class
Graph:     def __init__(self):
        self.graph = defaultdict(list)
def addEdge(self,u,v):
        self.graph[u].append(v)
    # Function to print a BFS of graph
    def BFS(self, s):       visited = [False]
* (len(self.graph))         queue = []
        queue.append(s)
        visited[s] = True
while queue:            s =
queue.pop(0)            print (s,
end = " ")            for i in
```

```
self.graph[s]:                if
visited[i] == False:
queue.append(i)
visited[i] = True
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3) print ("\nBFS" "
(starting from vertex 0)")
g.BFS(0)
```

Output :

```
BFS (starting from vertex 0)
0 1 2 3
```

```
BFS (starting from vertex 1)
1 2 0 3
```

```
BFS (starting from vertex 2)
2 0 3 1
```

Practical 2

Aim: Implement solution of Water Jug problem or 8-puzzle problem using Best First Search or A*.

1) Water Jug Problem

-You are given 2 jugs with the capacity 'm' and 'n' respectively. Initially, they are given empty. There is an unlimited supply of water. You can either fill the whole jug or a quantity that is less than the given capacity of jugs. Now, you are also given a third positive integer 'd'. Using the 2 given jugs, you need to come up with a solution to have 'd' amount of water in them and return the number of steps you took to reach that capacity.

The following operations can be performed on the jugs:

1. Empty a jug (a, b) -> (0, b)
2. Fill a jug (0, b) -> (a, b)
3. Transfer water from one jug to another.

Example

Input: 3, 5, 4

Output: 6

Explanation: The following steps are taken:

1. Fill the 5-liter jug completely.
2. Transfer 3 liters from a 5-liter jug to a 3-liter jug.
3. Empty the 3-liter capacity jug.
4. Transfer the remaining 2 liters from a 5-liter jug to a 3-liter jug.
5. Now, fill the 5-liter jug fully.
6. Pour 1 liter from a 5-liter jug into a 3-liter jug.
7. We have 4 liters in the first jug, now empty the 2nd jug.

```
# Code to implement water jug problem using best first search from

collections import deque

def BFS(a, b,

target):     m = {}

isSolvable = False

path = []

   # Queue to maintain states

q = deque()

   # Initializing with initial state
```

```
    q.append((0, 0))     while
(len(q) > 0):        # Current
state        u = q.popleft()
if ((u[0], u[1]) in m):
continue        if ((u[0] > a or
u[1] > b or             u[0] < 0
or u[1] < 0)):
continue
        path.append([u[0], u[1]])        #
Marking current state as visited
m[(u[0], u[1])] = 1

        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
if (u[1] != 0):
                path.append([u[0], 0])
            else:
                if (u[0] != 0):
                path.append([0, u[1]])

            sz = len(path)            for
i in range(sz):
print("(", path[i][0], ",",
path[i][1], ")")            break

        q.append([u[0], b])
        q.append([a, u[1]])
```

```
        for ap in range(max(a, b) + 1):

            c = u[0] + ap
d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            c = u[0] - ap            d = u[1] +
ap            if ((c == 0 and c >= 0) or d
== b):
                q.append([c, d])

        # Empty Jug2
        q.append([a, 0])
        # Empty Jug1
        q.append([0, b])

    # No, solution exists if ans=0
if          (not          isSolvable):
print("No solution")

if __name__ == '__main__':

    Jug1, Jug2, target = 3, 1, 2      #OR take 3,1,6     print("\nInitial state " "to solution state ::")
    BFS(Jug1, Jug2, target)
```

Output :

```
Initial state to solution state ::
( 0 , 0 )
( 0 , 1 )
( 3 , 0 )
( 3 , 1 )
( 1 , 0 )
( 2 , 1 )
( 2 , 0 )
```

```
Initial state to solution state ::
No solution
```

for 3,1,6 no solution exists.

2) 8-Puzzle Problem

- A 3 by 3 board with 8 tiles (each tile has a number from 1 to 8) and a single empty space is provided.
- The goal is to use the vacant space to arrange the numbers on the tiles such that they match the final arrangement.
- Four neighboring (left, right, above, and below) tiles can be moved into the available area.

State Space Representation

Cost function

In the search tree, each node Y has a corresponding cost. The next E-node may be found using the cost function. The E-node with the lowest costs is the next one. The function can be defined as :

C(Y) = g(Y) + h(Y)

Where,

g(Y) = the costs of reaching to the current node

from the root.

h(Y) = the costs of reaching to an goal node from the Y.

The optimal cost functions

We suppose that it will costs one unit to move a tile in any direction. In light of this, we create the following costs function for the 8-puzzle algorithm :

c(y) = f(y) + h(y)

where,

f(y) = the path's total length from the root y.

and

h(y) = the amount of the non-blank tiles which are not in their final goal position (misplaced tiles).

To change state y into a desired state, there are at least h(y) movements required.

```
# Code to implement 8-Puzzle Problem
import copy from heapq import
heappush, heappop
n = 3

row = [1, 0, -1, 0]
col = [0, -1, 0, 1]



class priorityQueue:

    def __init__(self):
self.heap = []

    def push(self, k):
        heappush(self.heap, k)

    def pop(self):
        return heappop(self.heap)

    def empty(self):
if not self.heap:
return True
        else:
            return False

class node:
```

```
    def __init__(self, parent, mat, empty_tile_pos,cost, level):

        self.parent = parent        self.mat =
mat        self.empty_tile_pos =
empty_tile_pos
        self.cost = cost
self.level = level

    def __lt__(self, nxt):
        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

    count = 0     for i in
range(n):        for j in
range(n):            if
((mat[i][j]) and
                (mat[i][j] != final[i][j])):
            count += 1

    return count



def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)

    x1 = empty_tile_pos[0]     y1 = empty_tile_pos[1]     x2 =
new_empty_tile_pos[0]     y2 = new_empty_tile_pos[1]
```

```
new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
cost = calculateCost(new_mat, final)     new_node = node(parent, new_mat,
new_empty_tile_pos,cost, level)     return new_node

def printMatrix(mat):

    for i in range(n):
for j in range(n):
            print("%d " % (mat[i][j]), end=" ")

        print()

def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n



def printPath(root):

    if root == None:
        return
    printPath(root.parent)
printMatrix(root.mat)
    print()

def solve(initial, empty_tile_pos, final):

    pq = priorityQueue()
```

```
    cost = calculateCost(initial, final)
root = node(None, initial,
empty_tile_pos, cost, 0)

    pq.push(root)

    while not pq.empty():

        minimum = pq.pop()

        if minimum.cost == 0:

            printPath(minimum)
            return

        for i in range(4):              new_tile_pos = [
minimum.empty_tile_pos[0]        +        row[i],
minimum.empty_tile_pos[1] + col[i], ]            if
isSafe(new_tile_pos[0],         new_tile_pos[1]):
child = newNode(minimum.mat,
                            minimum.empty_tile_pos,
                            new_tile_pos,
minimum.level + 1,
minimum, final,)

                pq.push(child)

initial = [[1, 2, 3],
           [5, 6, 0],
```

```
        [7, 8, 4]]

final = [[1, 2, 3],
         [5, 8, 6],
         [0, 7, 4]]

empty_tile_pos = [1, 2]
solve(initial, empty_tile_pos, final)
```

Output :

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

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

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

Practical 3

Aim: Write a program to solve a given cryptarithmetic problem.

Cryptarithmetic problem

- The Crypt-Arithmetic problem in Artificial Intelligence is a type of encryption problem in which the written message in an alphabetical form which is easily readable and understandable is converted into a numeric form which is neither easily readable nor understandable.
- In simpler words, the crypt-arithmetic problem deals with the converting of the message from the readable plain text to the non-readable ciphertext.

-The constraints which this problem follows during the conversion is as follows:

1. A number 0-9 is assigned to a particular alphabet.
2. Each different alphabet has a unique number.
3. All the same, alphabets have the same numbers.
4. The numbers should satisfy all the operations that any normal number does.

Example

Here, to convert it into numeric form, we first split each word separately and represent it as follows:

```
  S E N D
  M O R E
-------------   M O N E Y
```

- These alphabets then are replaced by numbers such that all the constraints are satisfied. So initially we have all blank spaces.
- We first look for the MSB in the last word which is 'M' in the word 'MONEY' here. It is the letter which is generated by carrying. So, carry generated can be only one. SO, we have M=1.
- Now, we have S+M=O in the second column from the left side. Here M=1.
- Therefore, we have, S+1=O. So, we need a number for S such that it generates a carry when added with 1. And such a number is 9. Therefore, we have S=9 and O=0.
- Now, in the next column from the same side we have E+O=N.
- Here we have O=0. Which means E+0=N which is not possible. This means a carry was generated by the lower place digits. So we have:
- 1+E=N ----------(i)

- Next alphabets that we have are N+R=E -------(ii)
- So, for satisfying both equations (i) and (ii), we get E=5 and N=6.
- Now, R should be 9, but 9 is already assigned to S, So, R=8 and we have 1 as a carry which is generated from the lower place digits.
- Now, we have D+5=Y and this should generate a carry. Therefore, D should be greater than 4. As 5, 6, 8 and 9 are already assigned, we have D=7 and therefore Y=2.

Therefore, the solution to the given Crypt-Arithmetic problem is:

S=9; E=5; N=6; D=7; M=1; O=0; R=8; Y=2

Which can be shown in layout form as:

```
   9 5 6 7
1.  0 8 5
  -------------
  1 0 6 5 2
  --------------
```

```
# Code to solve Cryptarithmetic Problem def
isSolvable(words, result):
   mp = [-1]*(26)
used = [0]*(10)
   Hash = [0]*(26)
   CharAtfront = [0]*(26)
   uniq = ""
   for word in range(len(words)):
      for i in range(len(words[word])):
         ch = words[word][i]
         Hash[ord(ch) - ord('A')] += pow(10, len(words[word]) - i - 1)

         if mp[ord(ch) - ord('A')] == -1:
mp[ord(ch) - ord('A')] = 0
uniq += str(ch)

         if i == 0 and len(words[word]) > 1:
            CharAtfront[ord(ch) - ord('A')] = 1
```

```
    for i in range(len(result)):
        ch = result[i]
        Hash[ord(ch) - ord('A')] -= pow(10, len(result) - i -
1)         if mp[ord(ch) - ord('A')] == -1:
mp[ord(ch) - ord('A')] = 0             uniq += str(ch)         if
i == 0 and len(result) > 1:
            CharAtfront[ord(ch) - ord('A')] = 1

    mp = [-1]*(26)

    return True

def solve(words, i, S, mp, used, Hash, CharAtfront):
    if i == len(words):         return S
== 0     ch = words[i]     val =
mp[ord(words[i]) - ord('A')]

    if val != -1:         return solve(words, i + 1, S + val * Hash[ord(ch) - ord('A')], mp, used,
Hash, CharAtfront)

    x = False     for l
in range(10):
        if CharAtfront[ord(ch) - ord('A')] == 1 and l == 0:
            continue

        if used[l] == 1:
continue
```

```
        mp[ord(ch) - ord('A')] = l
used[l] = 1
        x |= solve(words, i + 1, S + l * Hash[ord(ch) - ord('A')], mp, used, Hash, CharAtfront)


        mp[ord(ch) - ord('A')] = -1


        used[l] = 0


    return x


arr = ["SEVEN", "SEVEN" ]
S = "TWO"


if isSolvable(arr, S):
    print("Yes")
else:
    print("No")
```

Output :

```
Yes
```

Practical 4

Aim: Write a program to perform following operation

- Load the data from file
- Find out null and missing value
- Handle missing Value using different approach
- Plot the data using scatter plot, histogram , box plot

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Loading the dataset

```
import pandas as pd
```

Loading loan.csv to df using pandas library to create a dataframe

```
df = pd.read_csv('/content/drive/MyDrive/AIML/loan.csv')
df
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantInc
0	LP001002	Male	No	0	Graduate	No	5
1	LP001003	Male	Yes	1	Graduate	No	4
2	LP001005	Male	Yes	0	Graduate	Yes	3
3	LP001006	Male	Yes	0	Not Graduate	No	2
4	LP001008	Male	No	0	Graduate	No	6
...	...	...	...	...	...	...	
609	LP002978	Female	No	0	Graduate	No	2
610	LP002979	Male	Yes	3+	Graduate	No	4
611	LP002983	Male	Yes	1	Graduate	No	8
612	LP002984	Male	Yes	2	Graduate	No	7
613	LP002990	Female	No	0	Graduate	Yes	4

614 rows × 13 columns

Checking Missing Values

The isna() method returns a DataFrame object where all the values are replaced with a Boolean value True for NA (not-a -number) values, and otherwise False

```
df.isna()
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantInco
0	False	False	False	False	False	False	Fal
1	False	False	False	False	False	False	Fal
2	False	False	False	False	False	False	Fal
3	False	False	False	False	False	False	Fal
4	False	False	False	False	False	False	Fal
...	...	...	...	...	...	...	
609	False	False	False	False	False	False	Fal
610	False	False	False	False	False	False	Fal

df.head()

612	False	False	False	False	False	False	Fal

The head() method returns a specified number of rows, string from the top. The head() method returns the first 5 rows if a number is not specified.

```
df.head()
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncom
0	LP001002	Male	No	0	Graduate	No	584
1	LP001003	Male	Yes	1	Graduate	No	458
2	LP001005	Male	Yes	0	Graduate	Yes	300
3	LP001006	Male	Yes	0	Not Graduate	No	258
4	LP001008	Male	No	0	Graduate	No	600

df.shape

df.shape will return the rows and columns in dataframe ie (rows,cols)

```
df.shape
```

```
(614, 13)
```

df.size

The size property returns the number of elements in the DataFrame. The number of elements is the number of rows * the number of columns.

```
df.size
```

```
7982
```

df.info()

The info() method prints information about the DataFrame. The information contains the number of columns, column labels, column data types, memory usage, range index, and the number of cells in each column (non-null values).

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            614 non-null    object
 1   Gender             601 non-null    object
 2   Married            611 non-null    object
 3   Dependents         599 non-null    object
 4   Education          614 non-null    object
```

```
 5   Self_Employed      582 non-null    object
 6   ApplicantIncome    614 non-null    int64
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object
 12  Loan_Status        614 non-null    object
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

Reading dataset Data.csv

```
import pandas as pd
import numpy as np
import sklearn
import matplotlib.pyplot as plt
```

```
df_data = pd.read_csv('/content/drive/MyDrive/AIML/Data.csv')
df_data
```

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
4	Germany	40.0	NaN	Yes
5	France	35.0	58000.0	Yes
6	Spain	NaN	52000.0	No
7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

```
df_data.shape
```

```
(10, 4)
```

```
df_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Country    10 non-null     object
 1   Age        9 non-null      float64
 2   Salary     9 non-null      float64
 3   Purchased  10 non-null     object
dtypes: float64(2), object(2)
memory usage: 448.0+ bytes
```

```
df_data.size
```

```
40
```

Splitting dataset in X and Y

X = contains data of Country, Age, Salary

Y = contains data of Purchased

```
x = df_data.iloc[:, :-1].values
x
```

```
array([['France', 44.0, 72000.0],
       ['Spain', 27.0, 48000.0],
       ['Germany', 30.0, 54000.0],
       ['Spain', 38.0, 61000.0],
```

```
       ['France', 35.0, 58000.0],
       ['Spain', nan, 52000.0],
       ['France', 48.0, 79000.0],
       ['Germany', 50.0, 83000.0],
       ['France', 37.0, 67000.0]], dtype=object)
```

```
y = df_data.iloc[:,-1].values
y
```

```
array(['No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes'],
      dtype=object)
```

Handling the missing data

fillna()

The fillna() method replaces the NULL values with a specified value. The fillna() method returns a new DataFrame object unless the inplace parameter is set to True , in that case the fillna() method does the replacing in the original DataFrame instead.

Here in column **Age** the missing value is filled with the **average** of all values in **Age** Column

```
df_data['Age'] = df_data['Age'].fillna(value = df_data['Age'].mean())
df_data
```

	Country	Age	Salary	Purchased
0	France	44.000000	72000.0	No
1	Spain	27.000000	48000.0	Yes
2	Germany	30.000000	54000.0	No
3	Spain	38.000000	61000.0	No
4	Germany	40.000000	NaN	Yes
5	France	35.000000	58000.0	Yes
6	Spain	38.777778	52000.0	No
7	France	48.000000	79000.0	Yes
8	Germany	50.000000	83000.0	No
9	France	37.000000	67000.0	Yes

Here in column **Salary** the missing value is filled with the **median** of values in **Salary** Column

```
df_data['Salary'] = df_data['Salary'].fillna(value = df_data['Salary'].median())
df_data
```

	Country	Age	Salary	Purchased
0	France	44.000000	72000.0	No
1	Spain	27.000000	48000.0	Yes
2	Germany	30.000000	54000.0	No
3	Spain	38.000000	61000.0	No
4	Germany	40.000000	61000.0	Yes
5	France	35.000000	58000.0	Yes
6	Spain	38.777778	52000.0	No
7	France	48.000000	79000.0	Yes
8	Germany	50.000000	83000.0	No
9	France	37.000000	67000.0	Yes

The Missing values are filled using fillna method


```
df_data
```

	Country	Age	Salary	Purchased
0	France	44.000000	72000.0	No
1	Spain	27.000000	48000.0	Yes
2	Germany	30.000000	54000.0	No
3	Spain	38.000000	61000.0	No
4	Germany	40.000000	61000.0	Yes
5	France	35.000000	58000.0	Yes
6	Spain	38.777778	52000.0	No
7	France	48.000000	79000.0	Yes
8	Germany	50.000000	83000.0	No
9	France	37.000000	67000.0	Yes

Handling missing data with SKLEARN

Using Sklearn library, the **nan** values are replaced with most frequent occuring value in column

```
from sklearn.impute import SimpleImputer
im = SimpleImputer(missing_values = np.nan,strategy = 'most_frequent')
im.fit(x[:, 1:3])
x[:, 1:3] = im.transform(x[:, 1:3])
x
```

```
array([['France', 44.0, 72000.0],
       ['Spain', 27.0, 48000.0],
       ['Germany', 30.0, 54000.0],
       ['Spain', 38.0, 61000.0],
       ['Germany', 40.0, 48000.0],
       ['France', 35.0, 58000.0],
       ['Spain', 27.0, 52000.0],
       ['France', 48.0, 79000.0],
       ['Germany', 50.0, 83000.0],
       ['France', 37.0, 67000.0]], dtype=object)
```

Devlope Various graphs and chart using visualization tool in python - bar graph, histogram, pie chart, area plot

Bar Chart

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent

The bar plots can be plotted horizontally or vertically

A bar chart describes the comparisons between the discrete categories. One of the axis of the plot represents the specific categories being compared, while the other axis represents the measured values corresponding to those categories.

```
import matplotlib.pyplot as plt

plt.bar([0.25,1.25,2.25,3.25,4.25],[50,40,70,80,20], label="Jaguar",width=.4)

plt.bar([.75,1.75,2.75,3.75,4.75],[80,20,40,50,60],label="Rolls Royce", color='g',width=.4)

plt.legend()
plt.xlabel('Days')
plt.ylabel('Distance (kms)')
plt.title('Information')
plt.show()
```

Histogram

A histogram is a graph showing frequency distributions. It is a graph showing the number of observations within each given interval.

```
from matplotlib import pyplot as plt

population_age =[22,55,62,45,21,22,34,42,42,4,2,102,95,85,55,110,120,70,65,55,111,115,80,75,65,54,44,43,42,48]

bins = [0,10,20,30,40,50,60,70,80,90,100]

plt.hist(population_age, bins, label='Data',histtype='bar', rwidth=0.8)

plt.xlabel('age groups')
plt.ylabel('Number of people')
plt.title('Histogram')
plt.legend()
plt.show()
```

histogram/step

Step histogram is a type of histogram in which Bars are not filled with color, instead only the edge is the representation of Histogram.

```
from matplotlib import pyplot as plt

population_age =[22,55,62,45,21,22,34,42,42,4,2,102,95,85,55,110,120,70,65,55,111,115,80,75,65,54,44,43,42,48]

bins = [0,10,20,30,40,50,60,70,80,90,100]

plt.hist(population_age, bins, label='Data',histtype='step', rwidth=0.8)

plt.xlabel('age groups')
plt.ylabel('Number of people')
plt.title('Histogram')
plt.legend()
plt.show()
```

Step Filled

It is same as step histogram but the graph is filled

```
from matplotlib import pyplot as plt

population_age =[22,55,62,45,21,22,34,42,42,4,2,102,95,85,55,110,120,70,65,55,111,115,80,75,65,54,44,43,42,48]

bins = [0,10,20,30,40,50,60,70,80,90,100]

plt.hist(population_age, bins, label='Data',histtype='stepfilled', rwidth=0.8,color='r')

plt.xlabel('age groups')
plt.ylabel('Number of people')
plt.title('Histogram')
plt.legend()
plt.show()
```

Scatter Plot

Scatter plots are the graphs that present the relationship between two variables in a data-set.

It represents data points on a two-dimensional plane or on a Cartesian system. The independent variable or attribute is plotted on the X-axis, while the dependent variable is plotted on the Y-axis.

```
import matplotlib.pyplot as plt

x = [1,1.5,2,2.5,3,3.5,4]
y = [7.5,8,8.5,9,9.5,10,11]

x1=[8,8.5,9,9.5,10,10.5,11]
y1=[3,3.5,6,4,4.5,5,10]

plt.scatter(x,y, label='high income low saving',color='b')
plt.scatter(x1,y1,label='low income high savings',color='r')
plt.xlabel('saving*100')
plt.ylabel('income*1000')
plt.title('Scatter Plot')

plt.legend()
plt.show()
```

```
import matplotlib.pyplot as plt

x = [1,1.5,2,2.5,3,3.5]
y = [7.5,8,8.5,9,9.5,10]

x1=[8,8.5,9,9.5,10,11,21]
y1=[3,3.5,6,4,4.5,10,20]

plt.scatter(x,y, label='high income low saving',color='b')
plt.scatter(x1,y1,label='low income high savings',color='r')
plt.xlabel('saving*100')
plt.ylabel('income*1000')
plt.title('Scatter Plot')

plt.legend()
plt.show()
```

Area Plot

An Area Plot displays Quantitative Data graphically. It is very much like Line Plot but with the key difference of highlighting the distance between different variables.

```
from matplotlib import pyplot as plt

days = [1,2,3,4,5]
driving =[7,8,6,11,7]
eating = [2,3,4,3,2]
working =[7,8,7,2,2]
swimming = [8,5,7,8,13]

plt.plot([],[],color='m', label='Driving', linewidth=4)
plt.plot([],[],color='c', label='Eating', linewidth=3)
plt.plot([],[],color='r', label='Working', linewidth=4)
plt.plot([],[],color='k', label='Swimming', linewidth=3)

plt.stackplot(days, driving,eating,working,swimming, colors=['m','c','r','k'])

plt.xlabel('x')
plt.ylabel('y')
plt.title('Stack Plot')
plt.legend()
plt.show()
```

```
from matplotlib import pyplot as plt

days = [1,2,3,4,10]
```

```
eating = [2,3,4,3,2]
working =[7,8,7,2,2]
swimming = [8,5,7,8,30]

plt.plot([],[],color='m', label='Driving', linewidth=4)
plt.plot([],[],color='c', label='Eating', linewidth=3)
plt.plot([],[],color='r', label='Working', linewidth=4)
plt.plot([],[],color='k', label='Swimming', linewidth=3)

plt.stackplot(days, driving,eating,working,swimming, colors=['m','c','r','k'])

plt.xlabel('x')
plt.ylabel('y')
plt.title('Stack Plot')
plt.legend()
plt.show()
```

Pie Chart

Pie charts show the size of items (called wedge) in one data series, proportional to the sum of the items.

The data points in a pie chart are shown as a percentage of the whole pie.

```
import matplotlib.pyplot as plt

days = [1,2,3,4,5]
sleeping =[7,8,6,1,7]
working = [2,3,4,6,22]
eating = [7,8,7,2,2]
Swimming = [1,2,7,8,1]
slices = [7,2,2,13]

activities = ['sleeping','working','eating','swimming']
cols = ['c','m','r','b']
plt.pie(slices, labels=activities, colors=cols, startangle=90, shadow= True,explode=(0,0.1,0,0), autopct='%1.1f%%')

plt.title('Pie Plot')
plt.show()
```

```
import matplotlib.pyplot as plt

days = [1,2,3,4,5]
sleeping =[7,8,6,1,7]
working = [2,3,4,6,2]
eating = [7,8,7,2,2]
Swimming = [1,2,70,8,31]
```

```
activities = ['sleeping','working','eating','swimming']
cols = ['c','m','r','b']
plt.pie(slices, labels=activities, colors=cols, startangle=90, shadow= True,explode=(0,0.1,0,0), autopct='%1.1f%%')

plt.title('Pie Plot')
plt.show()
```

Practical 5

Aim: Write a program to implement Linear Regression

Linear Regression

- The term regression is used when you try to find the relationship between variables.
- In Machine Learning, and in statistical modeling, that relationship is used to predict the outcome of future events.
- Linear regression uses the relationship between the data-points to draw a straight line through all them.
- This line can be used to predict future values.

Working

- Python has methods for finding a relationship between data-points and to draw a line of linear regression. We will show you how to use these methods instead of going through the mathematic formula.

Implement Linear Regression in python

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
dataset = pd.read_csv('/content/drive/MyDrive/AIML Lab/Salary_Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
dataset
```

	YearsExperience	Salary
0	1.1	39343.0
1	1.3	46205.0
2	1.5	37731.0
3	2.0	43525.0
4	2.2	39891.0
5	2.9	56642.0
6	3.2	64445.0
10	3.9	63218.0
11		
12		
13		
14		
15		
18		
19		
20		
21	7.1	98273.0
26	9.5	116969.0
27		
28		
29	10.5	121872.0

Splitting the dataset

Splitting a dataset is important because it allows us to evaluate the performance of a machine learning model on unseen data.

By splitting our data into separate training and testing sets, we can ensure that our model generalizes well to new data and is not simply memorizing the training data. Additionally, splitting the dataset into a validation set can help us fi ne-tune our model's hyperparameters and avoid overfi tting to the training set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

Training simple linear regression on training dataset

Training simple linear regression on a training dataset involves fitting a line to the data that minimizes the difference between the predicted values and the actual values, using a method such as ordinary least squares.

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)
LinearRegression()
print(regressor.intercept_)
```

```
26777.391341197625
```

```
print(regressor.coef_)
```

```
[9360.26128619]
```

Predicting the Test set results

```
y_pred = regressor.predict(X_test)
y_pred=np.array([ 40817.78327049, 123188.08258899, 65154.46261459, 63282.41035735,
 115699.87356004, 108211.66453108, 116635.89968866, 64218.43648597,
 76386.77615802])
y_test=np.array([ 37731., 122391., 57081., 63218., 116969., 109431., 112635.,
 55794., 83088.])
```

Visualising training results

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

```
regressor.score(X_train, y_train)
```

```
0.9423777652199379
```

```
regressor.score(X_test, y_test)
```

```
0.9740993407213511
```

Practical 6

Aim: Write a program to implement k-Nearest Neighbor algorithm to classify the iris data set. Print both correct and wrong predictions

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

from sklearn import datasets
from sklearn.model_selection import train_test_split , KFold
from sklearn.preprocessing import Normalizer
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

from collections import Counter
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Reading Iris Dataset

Reading the Iris dataset which contains four features namely length and width of sepals and petals of 50 samples of three species of Iris which are Iris setosa, Iris virginica and Iris versicolor.

First of all load whole dataset in *iris* and then create a dataframe **iris_df** by using **iris**

Here we display first 5 rows from dataframe using **iris_df.head()**

```
# import iris dataset
iris = datasets.load_iris()
# np.c_ is the numpy concatenate function
iris_df = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                       columns= iris['feature_names'] + ['target'])
iris_df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0

Using **iris_df.describe()** we can fetch the count, mean, standard deviation, minimum and maximum value in columns, and distribution of data for each features

```
iris_df.describe()
```

Splitting data in x and y

count					
25%	5.100000	2.800000	1.600000	0.300000	0.000000
max					

The dataset is spilt in X and Y where,
X = data of Dataset until last column (sepal length, sepal width, petal length, petal width)
Y = the data of last column (target)

```
x= iris_df.iloc[:, :-1]
y= iris_df.iloc[:, -1]
```

Viewing first 5 rows from x using x.head()

```
x.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Viewing first 5 rows from y using y.head()

```
y.head()
```

```
0    0.0
1    0.0
2    0.0
3    0.0
4    0.0
Name: target, dtype: float64
```

Splitting into Training and Testing the data set

The simplest way to split the modelling dataset into training and testing sets is to assign 2/3 data points to the former and the remaining one-third to the latter. Therefore, we train the model using the training set and then apply the model to the test set. In this way, we can evaluate the performance of our model.

Here we divide the dataset in **80% for training** and **20% for testing**
Also we shuffle the dataset so we get mix data in both training and testing datasets

```
# split the data into train and test sets
x_train, x_test, y_train, y_test= train_test_split(x, y,
                                                   test_size= 0.2,
                                                   shuffle= True, #shuffle the data to avoid bias
                                                   random_state= 0)
```

```
x_train= np.asarray(x_train)
y_train= np.asarray(y_train)

x_test= np.asarray(x_test)
y_test= np.asarray(y_test)
```

Iris dataset contains 150 rows and after spliting the dataset in training and testing the size of **training dataset is 120 samples** and **testing dataset is 30 samples**

```
print(f'training set size: {x_train.shape[0]} samples \ntest set size: {x_test.shape[0]} samples')
```

```
training set size: 120 samples
test set size: 30 samples
```

Normalize the dataset

Normalization is a scaling technique in Machine Learning applied during data preparation to change the values of numeric columns in the dataset to use a common scale. It is not necessary for all datasets in a model. It is required only when features of machine learning models have different ranges.

```
scaler= Normalizer().fit(x_train) # the scaler is fitted to the training set
normalized_x_train= scaler.transform(x_train) # the scaler is applied to the training set
normalized_x_test= scaler.transform(x_test) # the scaler is applied to the test set
```

```
print('x train before Normalization')
print(x_train[0:5])
print('\nx train after Normalization')
print(normalized_x_train[0:5])
```

```
x train before Normalization
[[6.4 3.1 5.5 1.8]
 [5.4 3.  4.5 1.5]
 [5.2 3.5 1.5 0.2]
 [6.1 3.  4.9 1.8]
 [6.4 2.8 5.6 2.2]]

x train after Normalization
[[0.69804799 0.338117   0.59988499 0.196326  ]
 [0.69333409 0.38518561 0.57777841 0.1925928 ]
 [0.80641965 0.54278246 0.23262105 0.03101614]
 [0.71171214 0.35002236 0.57170319 0.21001342]
 [0.69417747 0.30370264 0.60740528 0.2386235 ]]
```

Visualize the dataset before and after Normalization

```
## Before
# View the relationships between variables; color code by species type
di= {0.0: 'Setosa', 1.0: 'Versicolor', 2.0:'Virginica'} # dictionary

before= sns.pairplot(iris_df.replace({'target': di}), hue= 'target')
before.fig.suptitle('Pair Plot of the dataset Before normalization', y=1.08)

## After
iris_df_2= pd.DataFrame(data= np.c_[normalized_x_train, y_train],
                        columns= iris['feature_names'] + ['target'])
di= {0.0: 'Setosa', 1.0: 'Versicolor', 2.0: 'Virginica'}
after= sns.pairplot(iris_df_2.replace({'target':di}), hue= 'target')
after.fig.suptitle('Pair Plot of the dataset After normalization', y=1.08)
```

```
Text(0.5, 1.08, 'Pair Plot of the dataset After normalization')
```

K - Nearest Neighbour

The k-nearest neighbors algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point.

KNN Step 1 (Eucleadian Distance)

The Euclidean distance is defined as the distance between two points. In other words, the Euclidean distance between two points in the Euclidean space is defined as the length of the line segment between two points.

$d(p, q)^2 = (q1 - p1)^2 + (q2 - p2)^2$

```
def distance_ecu(x_train, x_test_point):
  """
  Input:
    - x_train: corresponding to the training data
    - x_test_point: corresponding to the test point

  Output:
    -distances: The distances between the test point and each point in the training data.

  """
  distances= []  ## create empty list called distances
  for row in range(len(x_train)): ## Loop over the rows of x_train
      current_train_point= x_train[row] #Get them point by point
      current_distance= 0 ## initialize the distance by zero

      for col in range(len(current_train_point)): ## Loop over the columns of the row

          current_distance += (current_train_point[col] - x_test_point[col]) **2
          ## Or current_distance = current_distance + (x_train[i] - x_test_point[i])**2
      current_distance= np.sqrt(current_distance)

      distances.append(current_distance) ## Append the distances

  # Store distances in a dataframe
  distances= pd.DataFrame(data=distances,columns=['dist'])
  return distances
```

KNN Step 2 (Find the nearest neighbour)

Here the distance of each point is measured and nearest K neighbours are calculated

```
def nearest_neighbors(distance_point, K):
    """
    Input:
        -distance_point: the distances between the test point and each point in the training data.
        -K: the number of neighbors
```

```
        -df_nearest: the nearest K neighbors between the test point and the training data.

    """

    # Sort values using the sort_values function
    df_nearest= distance_point.sort_values(by=['dist'], axis=0)

    ## Take only the first K neighbors
    df_nearest= df_nearest[:K]
    return df_nearest
```

KNN Step 3 Voting

Majority Voting: After you take the k nearest neighbors, you take a vote of those neighbors' classes. The new data point is classified with whatever the majority class of the neighbors is.

```
def voting(df_nearest, y_train):
    """
    Input:
        -df_nearest: dataframe contains the nearest K neighbors between the full training dataset and the test point.
        -y_train: the labels of the training dataset.

    Output:
        -y_pred: the prediction based on Majority Voting

    """

    ## Use the Counter Object to get the labels with K nearest neighbors.
    counter_vote= Counter(y_train[df_nearest.index])

    y_pred= counter_vote.most_common()[0][0]   # Majority Voting

    return y_pred
```

KNN full algorithm (Putting everything together)

Now a function is created which takes training as well as testing dataset and K nearest neighbour as input and provide prediction for whole test set based on Majority Voting as output

```
 def KNN_from_scratch(x_train, y_train, x_test, K):

    """
    Input:
    -x_train: the full training dataset
    -y_train: the labels of the training dataset
    -x_test: the full test dataset
    -K: the number of neighbors

    Output:
    -y_pred: the prediction for the whole test set based on Majority Voting.

    """

    y_pred=[]

    ## Loop over all the test set and perform the three steps
    for x_test_point in x_test:
      distance_point  = distance_ecu(x_train, x_test_point)  ## Step 1
      df_nearest_point= nearest_neighbors(distance_point, K)  ## Step 2
      y_pred_point    = voting(df_nearest_point, y_train) ## Step 3
      y_pred.append(y_pred_point)

    return y_pred
```

Testing KNN algorithm on test data set

The training and testing data are passed to function KNN_from_scratch and result is stored in y_pred_scratch

```
K=3
y_pred_scratch= KNN_from_scratch(normalized_x_train, y_train, normalized_x_test, K)
print(y_pred_scratch)
```

```
[2.0, 1.0, 0.0, 2.0, 0.0, 2.0, 0.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 2.0, 0.0, 0.0, 2.0, 1.0, 0.0, 0.0, 2.0, 0.0,
```

Compare our implimentation with the sklearn library

```
knn=KNeighborsClassifier(K)
knn.fit(normalized_x_train, y_train)
y_pred_sklearn= knn.predict(normalized_x_test)
print(y_pred_sklearn)
```

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

If the output is exactly same (find the output value)

```
print(np.array_equal(y_pred_sklearn, y_pred_scratch))
```

```
True
```

Calculate the accuracy of both the methods (scratch and sklearn)

```
print(f'The accuracy of our implementation is {accuracy_score(y_test, y_pred_scratch)}')
print(f'The accuracy of sklearn implementation is {accuracy_score(y_test, y_pred_sklearn)}')
```

```
The accuracy of our implementation is 0.9666666666666667
The accuracy of sklearn implementation is 0.9666666666666667
```