Term Work - 1

* Problem defination:
Implement DFID algosithm and compare its performance with BFS and DFS algosithm:

* Objectives:
1. Understand the working of BFS, DFS, DFID algosithms and implementing them.
2. Compare performance of DFID with DFS & BFS.

* Theory:

DFS: it is the most commonly used to search for goal node in a search tree. DFS follows each path to its greatest depth before moving on to next path. If we start from left side and move towards right, then DFS involves working all the way down to the leftmost path in the tree until a leaf node is reached. If its a goal state, search is complete and success is reported, otherwise search backtracks upto next highest node with an unexplored path. This process continues until all nodes have been examined, in which case search fails, or until a goal node is found.

BFS: It is used to search a tree by breadth rather than by depth. It starts examining all nodes one level below the root node. If a goal state is reached, success is reported. Otherwise search continues by expanding paths from all nodes in the

current level down to next level. In this way search continues examining nodes in a particular level, reporting success when a goal node is found and reporting failure if all nodes are examined and no goal node has been found.

DFID: It is a combo of DFS and BFS. It involves repeatedly carrying out DFS on a tree, starting with a DFS limited to depth one, then with depth two and so on until goal is found.

* Algorithm:

1. DFS:

```
Function depth ()
{
   queue = [];
   state = root_node;
   while (true)
   {
     if is_goal (state)
        then return SUCCESS;
     else add_to_front_of_queue (successors (state));
     if queue == []
        then report FAILURE;
     state = queue [0];
     remove_first_item_from (queue);
   }
}
```

2. BFS :

```
Function breadth()
{
  queue = [];
  state = root_node;
  while(true)
  {
    if is_goal(state)
      then report success;
    else add_to_back_of_queue (successors(state));
    if queue == []
      then report FAILURE;
    state = queue[0];
    remove_first_item_from(queue);
  }
}
```

3. DFID :

The algorithm for DFID is just to use DFS function with varying depths, beginning with 1 & incrementing by 1 in successive calls till the max_depth is reached.

* Source Code :

```
From collections import defaultdict

steps = -1
found = False

class Graph :
```

```
def __init__(self):
    self.graph = defaultdict(list)

def addEdge(self, vertex, neighbour):
    self.graph[vertex].append(neighbour)

def bfs(self, node, value):
    global found
    visited = [node]
    queue = [node]
    steps = 1

    while queue:
        enode = queue.pop(0)
        if value == enode:
            found = True
            print("Found in %d steps" %steps)
            return

        steps += 1
        for neighbour in self.graph[enode]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

def dfs(self, node, visited, value, depth):
    global steps, found

    if depth < 0: return
    if node not in visited:
```

```
        steps += 1
        visited.add(node)
        if node == value:
            found = True
            print("Node found in %d steps" % steps)
            return
        for neighbour in self.graph[node]:
            self.dfs(neighbour, visited, value, depth-1)
        if node == 1:
            steps -= 1
            return
        steps += 1

    def iddfs(self, start, value, maxdepth):
        self.dfs(start, visited, value, depth)
        if found:
            return
        visited = set()

def main():
    global steps, found
    numofNodes = int(input("Enter the number of nodes in
                            graph:"))
    g = Graph()

    for i in range(1, numofNodes+1):
        neighbours = list(map(int, input("Enter the neighbours
                        of node %d:" %i).split()))
        for neighbour in neighbours:
```

```
        g.addEdge(i, neighbour)

    value = int(input("Enter the element to search:"))
    print("Using BFS:")
    g.bfs(1, value)
    if not found:
        print("Not Found")

    print("Using DFS:")
    found = False
    maxdepth = int(input("Enter the max depth:"))
    g.dfs(1, set(), value, maxdepth)
    if not found:
        print("Not found")

    print("Using DFID:")
    steps = -1
    found = False
    g.iddfs(1, value, maxdepth)
    if not found:
        print("Not found")

if __name__ == "__main__":
    main()
```

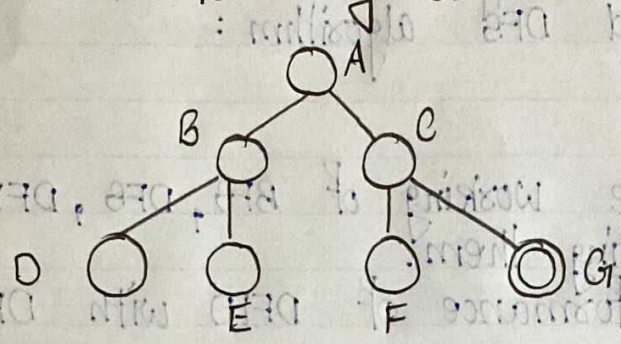
* Github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/TW1.py.

* Conclusion :
We successfully understood working of BFS, DFS and DFID algorithms and we were able to implement the same and compare the performance of DFS, BFS and DFID.

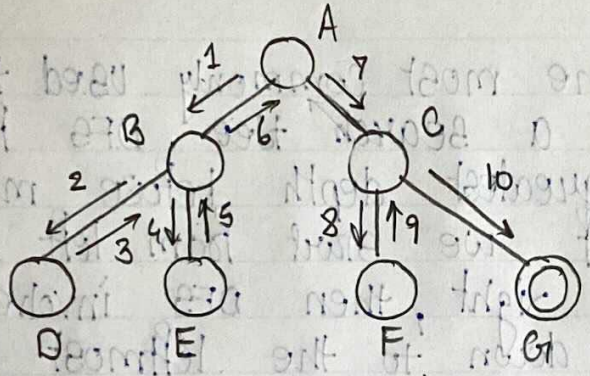
* Reference :
Ben coppin, Artificial Intelligence Illuminated, Jones and Bartlett, 2004.

Example :

Consider the following search tree

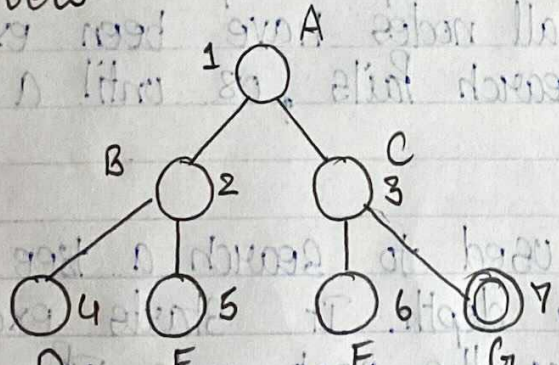


Using DFS, the number on arrows show the step number :



10 steps using DFS.

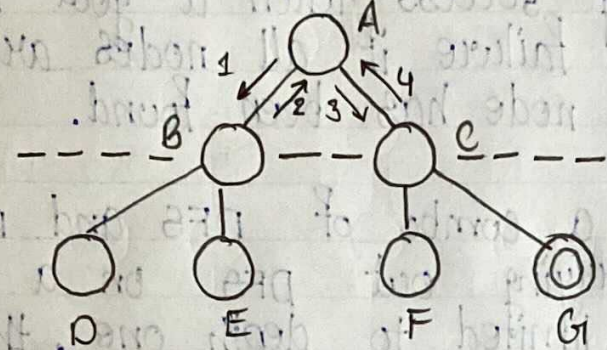
Using BFS, the numbers near the nodes show the step numbers :



7 steps using BFS.

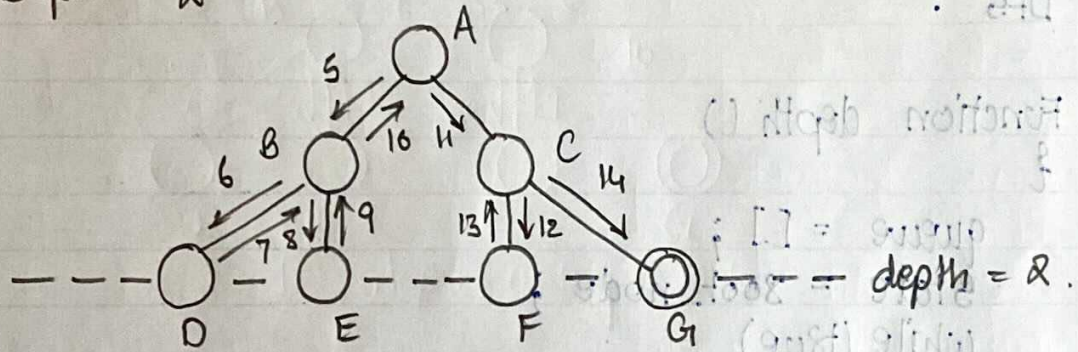
Using DFID

1. Depth = 1



Steps = 4, not found in depth = 1

2. Depth = 2



Using DFID, found in 14 steps

```
==== RESTART: D:/6th sem notes/AI-ML/termworks final/2GI18CS074_Termwork1.py ===
Enter Target Vertex>>>   5
Output of Deapth First Search >>>>>  [0, 2, 5]
Comparative Factor >>>>>6
Enter Target Vertex>>>   3
Output of Breadth First Search >>>> [0, 1, 2, 3]
Comparative Factor >>>>>8
0 [1, 2]
0 [1, 2]
0 [1, 2]
1 [0, 3]
1 [0, 3]
0 [1, 2]
2 [0, 4, 5]
2 [0, 4, 5]
2 [0, 4, 5]
0 [1, 2]
1 [0, 3]
0 [1, 2]
0 [1, 2]
1 [0, 3]
3 [1]
0 [1, 2]
2 [0, 4, 5]
0 [1, 2]
0 [1, 2]
2 [0, 4, 5]
4 [2, 6]
4 [2, 6]
Comparative Factor >>>>>4
Path is Available!!!
>>>
```

Term Work-2

* Problem defination:
Implement Best First Search algorithm:

* Objectives:
1. Understand working of best first search algorithm.
2. Implement Best First Search in Python.

* Theory:
To understand the best first search algorithms, we first need to understand hill climbing algorithm. now, hill climbing can be thought of as a variation of DFS, which uses information about how far each node is from goal node to choose which path to follow next at any given point. Now using this heuristic, hill climbing proceeds as with DFS, but at each step, the new nodes to be added to the queue are sorted into order of distance from the goal.

Now coming to best first search, it employs a heuristic similar to hill climbing but with a difference that the entire queue is sorted after new paths have been added to it, rather than adding a set of sorted paths. Hence, best first search follows the best path available from current tree, rather than always following a depth first style approach.

* Algorithm :

```
Function best()
{
   queue = [];
   state = root_node;
   while (true)
   {
      if is_goal(state)
         then return SUCCESS;
      else
         {
            add_to_front_of_queue(successors(state));
            sort(queue);
         }
         if queue == []
            then report FAILURE;
      state = queue[0];
      remove_first_item_from(queue);
   }
}
```

* Source code :

```
From collections import defaultdict

class Graph:

    def __init__(self):
        self.graph[vertex].extend(neighbours):
```

```
def bestFirstSearch(self, goal, distance):
    queue = []
    state = 'A'
    path = ['A']
    while True:
        if state == goal:
            print("SUCCESS: Path is ", ' '.join(path))
            return True

        queue = self.graph[state] + queue
        queue.sort(key = lambda x: distance[ord(x)-65])

        if queue == []:
            return False
        state = queue.pop(0)
        path.append(state)

def main()
    numOfNodes = int(input("Enter the number of nodes
                 in graph :"))
    g = Graph()

    distance = []
    goal = input("Enter the goal node :")

    for i in range(numOfNodes):
        successors = input("Enter the successors of node
        %s : " %chr(65+i)).split()
        distance.append(int(input("Enter the straight line
```

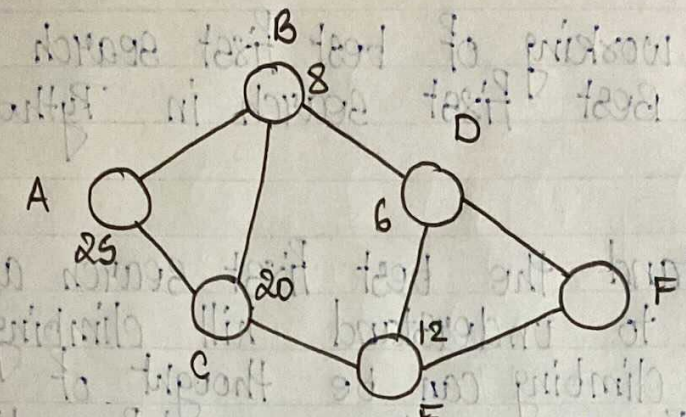
```
            distance from node %s to the goal
            node %s :" %(chr(65+i), goal))
            g.addEdge(chr(65+i), successors)

    if not g.bestFirstSearch(goal, distance):
        print("FAILURE !!!")

if __name__=="__main__":
    main()
```

* Conclusion :
We understood the implementation of best first search and implement the same.

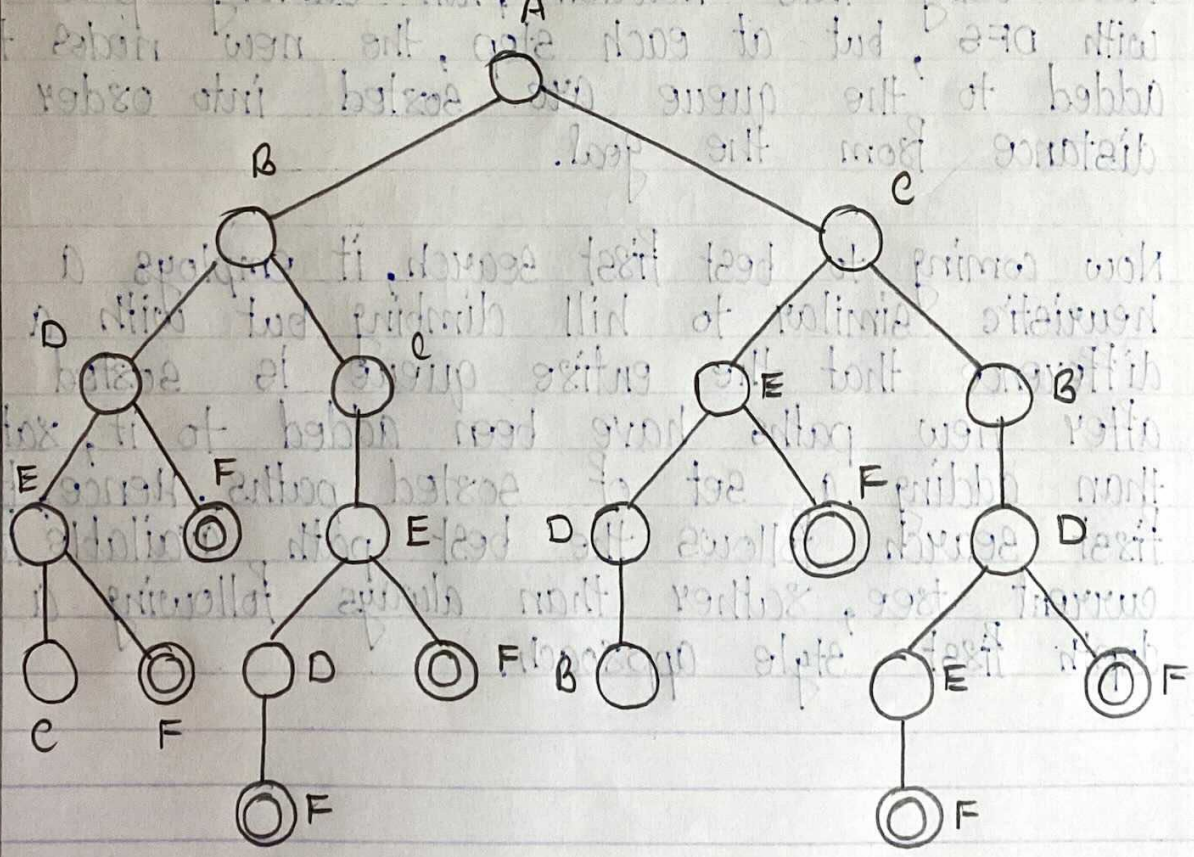
* github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/TW2.py.

* Reference :
Ben Coppin, Artificial Intelligence Illuminated, Jones and Bartlett, 2004.

Consider the following graph where the straight line distance of each city to goal city (F) is shown:



Consider the following search tree:



Analysis of best first search of the search tree:

Step	State	Queue	Notes
1	A	(empty)	Queue starts out empty & the initial state is root node ie A.
2	A	C, B	Successors of A ie B, C are placed in queue.
3	A	B, C	Queue is sorted, leaving B in front of C because B is closer to F.
4	B	C	B is chosen as next state to explore
5	B	C, D, C	Successors of B ie C & D are added to front of queue.
6	B	D, C, C	Queue is sorted leaving D at front because D is closer to F than C.
7	D	C, C	Although queue contains C twice, this is just an artifact of the way the search tree was constructed. In facts, those 2 C's are distinct & represent diff. paths in search tree.

8	D	E, F, C, C	Successors of D are added to front of queue
9	D	F, E, C, C	Queue is sorted, moving F to front.
10	F	E, C, C	SUCCESS

```
Unsorted OPEN= [['H', 7], ['B', 6], ['E', 8], ['D', 9]]
Sorted OPEN= [['B', 6], ['H', 7], ['E', 8], ['D', 9]]

<<<<<<<<<<---(4)--->>>>>>>>>>

N= ['B', 6]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6]]
CHILD= [['G', 14], ['F', 12]]
Unsorted OPEN= [['G', 14], ['F', 12], ['H', 7], ['E', 8], ['D', 9]]
Sorted OPEN= [['H', 7], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]

<<<<<<<<<<---(5)--->>>>>>>>>>

N= ['H', 7]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7]]
CHILD= [['J', 6], ['I', 5]]
Unsorted OPEN= [['J', 6], ['I', 5], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]
Sorted OPEN= [['I', 5], ['J', 6], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]

<<<<<<<<<<---(6)--->>>>>>>>>>

N= ['I', 5]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5]]
CHILD= [['M', 2], ['L', 10], ['K', 1]]
Unsorted OPEN= [['M', 2], ['L', 10], ['K', 1], ['J', 6], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]
Sorted OPEN= [['K', 1], ['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G', 14]]

<<<<<<<<<<---(7)--->>>>>>>>>>

N= ['K', 1]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1]]
Best First Search Path >>>> [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5]] <<<True>>>
```

Term Work - 3.

* Problem definition:
Implement AND/OR/NOT gates using single level perceptron.

* Objectives:
1. To understand the working of single layer perceptron.
2. To implement AND/OR/NOT gates using single layer perceptron.

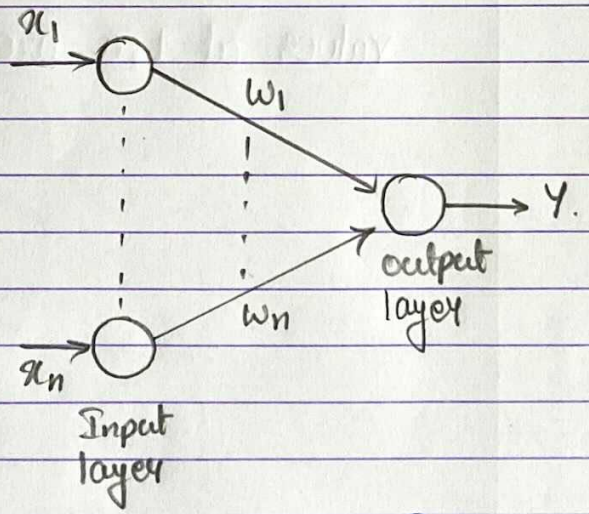
* Theory:
A perceptron is a simple neuron that is used to classify its inputs into one of the two categories. A perceptron can have any number of inputs. A perceptron uses a step function as activation function, which returns +1 if the weighted sum of inputs, x, is greater than threshold t, and -1 or 0 if x is less than or equal to t.

$$x = \sum_{i=1}^{n} w_i x_i$$

$$y = \begin{cases} +1, & \text{for } x > t \\ 0, & \text{for } x \leq t \end{cases}$$

Y is also written as step(x).
Therefore, we can write:

$$Y = \text{step}\left(\sum_{i=0}^{n} w_i x_i\right)$$



The learning process for a perceptron is as follows:
First, random weights are assigned to inputs, typically in range $[-0.5, +0.5]$.

Next, an item of training data is presented to the perceptron and its output is observed. If output is incorrect, weights are adjusted to try to more closely classify this input. The weight modification formula is as follows:

$$w_i \leftarrow w_i + (a \times x_i \times e)$$

where: e, a represent error and learning rate resply. $0 < a < 1$.

Once this modification has been applied next piece of training data is used in same way and once all training data has been used, the process restarts until all weights are correct and all errors are zero. Each iteration of this process is called an epoch.

* Algorithm:

```
Function learn_perceptron(w1, w2, x1list, x2list, expOplist, t, a)
{
    predicted Op = [];
    while (predictedOp != expOplist)
    {
        for i from 0 to 3 do:
            x = Σ (j=0 to 2) wj x jlist[i];

            Y = 0 if x <= t else 1;
            predictedOp[i] = Y;
            err = expOplist[i] - Y;
```

```
            if err != 0
                then for k from 1 to 2 do:
                    wk = wk + (a * x klist[i] * err);
        }
    }
```

* Source Code :

```
w1 = w2 = None
xlist = x2list = None
expectedOp = None
t = None
a = None

def main():
    w1, w2 = -0.2, 0.4
    xlist, x2list = [0,0,1,1], [0,1,0,1]
    expectedOp = [0,1,1,1]
    t = 0
    a = 0.2
    epochs = 0
    actualOp = []
    print("OR Grate :")
    print("Epoch X1 X2 Expected_Y Actual_Y Esssos w1 w2")
    while expectedOp != actualOp:
        epochs += 1
        actualOp = []
        for i in range(4):
            actualop.append(1 if w1*xlist[i]+w2*x2list
                                [i]>t else 0)
        err = expectedOp[i] - actualOp[i]
```

```
        if err:
            w1 += a * x1list[i] * err
            w2 += a * x2list[i] * err
        print(epochs, ' ', xlist[i], '', x2list[i], '',
            expectedOp[i], '\t ', actualOp[i], ' ',
            err, '  ', w1, w2)

print("..........")

w1, w2 = 1.2, 0.6
xlist, x2list = [0,0,1,1], [0,1,0,1]
expectedOp = [0,0,0,1]
t = 1
a = 0.5
epochs = 0
actualOp = []
print("AND Grate:")
print("Epoch X1 X2 Expected_Y Actual_Y Error w1 w2")
while expectedOp != actualOp;
    epochs += 1
    actualOp = []
    for i in range(4):
        actualOp.append(1 if w1 * xlist[i] + w2 * x2list
                        [i] > t else 0)
        err = expectedOp[i] - actualOp[i]
        if err:
            w1 += a * xlist[i] * err
            w2 += a * x2list[i] * err
        print(epochs, ' ', x1list[i], '', x2list[i], '',
```

```
                    expectedOp[i], '\t ', actualOp[i], ' ',
                    err, ' ', w1, w2)

print("..........................")

w1 = -0.04
xllist = [0,1]
expectedOp = [1,0]
t = -0.1
a = 0.5
epochs=0
actualOp = []
print("Not gate:")
print("Epoch x1 Expected_Y Actual_Y Error w1")
while expectedop != actualop:
    epochs +=1
    actualOp = []
    for i in range(2):
        actualOp.append(1 if w1*x1list[i]>t else 0)
        err = expectedop[i] - actualOp[i]
        if err:
            w1+= a*x1list[i]*err
        print(epochs, ' ', x1list[i], ' ', expectedOp[i],
              '\t ', actualOp[i], ' ', err, ' ', w1)

if __name__=="__main__":
    main()
```

* github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/Tw3.py

* Conclusion:
We understood the concept of single layer perceptron and could implement AND/OR/NOT logic gates with it.

* Reference:
Ben Coppin, Artificial Intelligence Illuminated, Jones & Bartlett 2004.

A sample run showing how the weights change for a simple perceptron when it learns to represent the logical OR function.

Epoch	x1	x2	Expected Y	Actual Y	Error	w1	w2
1	0	0	0	0	0	-0.2	0.4
1	0	1	1	1	0	-0.2	0.4
1	1	0	1	0	1	0	0.4
1	1	1	1	1	0	0	0.4
2	0	0	0	0	0	0	0.4
2	0	1	1	1	0	0	0.4
2	1	0	1	0	1	0.2	0.4
2	1	1	1	1	0	0.2	0.4
3	0	0	0	0	0	0.2	0.4
3	0	1	1	1	0	0.2	0.4
3	1	0	1	1	0	0.2	0.4
3	1	1	1	1	0	0.2	0.4

values at t,a are 0, 0.2 resply

Python 3.8.1 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: D:\6th sem notes\AI-ML\AI ML Termworks\aimltw3.py ==========
OR(0, 0) = 0
OR(0, 1) = 1
OR(1, 0) = 1
OR(1, 1) = 1
>>>
```

Term Work 4(a):

* Problem defination:
Implement NOR gate using multi layer perceptron/ Error propogation

* Objectives:
To demonstrate the ability of multilayer presentation in solving linearly inseperable pattern declassification problems and to implement NOR using MLP/EBP.

* Theory:
Basically a multilayer perceptron (MLP) is a class of feed forward artificial neural network (ANN).

It consists of 3 main layers:
a. Input layer.
b. Output layer.
c. Hidden layer.

Implementation logic gates using ANN help us to understand the mathematical computation by which a neural network processes its inputs to arrive of a certain output.

NOR gates provides output as True(1) if either of input is True & outputs as False(0) if both inputs are same.

For above XOR implementation, the ANN will consist of one i/p layer, with two nodes (x_1, x_2) one i/p

layer ; one hidden layer with two nodes ; and one o/p layer with one node (y).

* Algorithm :

1. → Initialize the weight and biasses immediately randomly.
2. → Iterate over data
   i. Compute the predicted output using sigmoid function.
   ii. Compute loss using the square error loss function.
   iii. w(new) = w(old) = $\alpha \Delta w$.
   iv. B(new) = B(old) = $\alpha \Delta B$.
3. Repeat until error is minimal.

The entire algorithm is divided into two parts :

- Forward pass and
- back propogation

* Source code :

```
import numpy as np
# np.random.seed(0)

def sigmoid(x):
    return 1/(1+ np.exp(-x))

def sigmoid_derivation(x):
    return x * (1-x)

#input datasets
```

```
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])

epochs = 10000
lr = 0.1
inputLayerNeuron, hiddenLayerNeurons, outputLayerNeurons = 2,2,1

#Random weights and bias initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons,
                                         hiddenLayerNeurons))
hidden_bias = np.random.uniform(size=(1, hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons,
                                         outputLayerNeurons))
output_bias = np.random.uniform(size=(1, outputLayerNeuron))

print("Initial hidden weights: ", end='')
print(*hidden_weights)
print("Initial hidden biases: ", end='')
print(*hidden_bias)
print("Initial output weights: ", end='')
print(*output_weights)
print("Initial output biases: ", end='')
print(*output_bias)

#Training algorithm:
for _ in range(epochs):
    #Forward Propogation
    hidden_layer_activation = np.dot(inputs, hidden_weights)
    hidden_layer_activation += hidden_bias
```

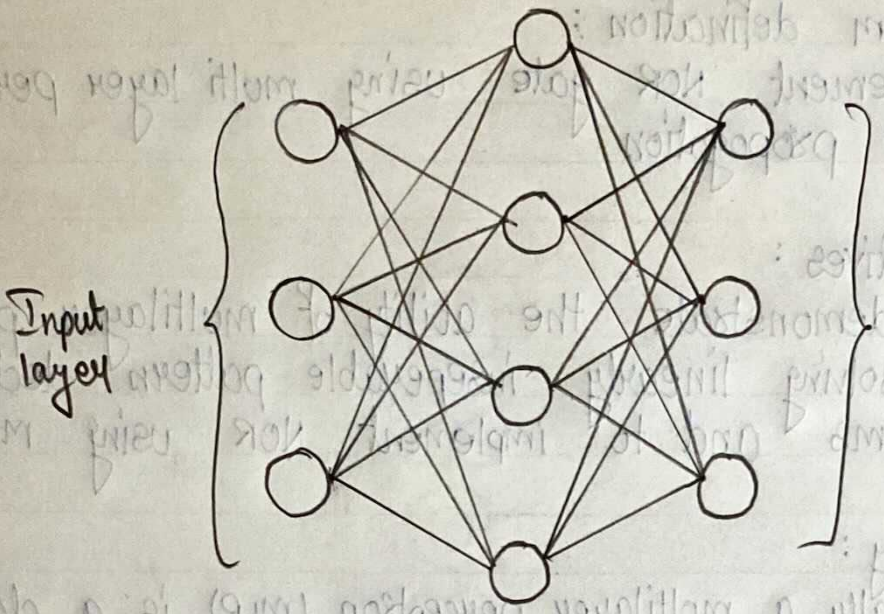
```
hidden_layer_bias = sigmoid(hidden_layer_activation)

#Backpropogation
error = expected_output - predicted_output
d_predicted_output = error * sigmoid_derivative
                     (predicted_output)
error_hidden_layer = d_predicted_output.dot(output_
                     weights.T)
d_hidden_layer = error_hidden_layer * sigmoid_
                 derivative(hidden_layer_output)

#Updating weights and biases
output_weights += hidden_layer_output.T.dot(d_predicted
                  _output.dot(output_weights.T) * lr.
dutput_bias += np.sum(d_predicted_output, axis=0,
               keepdims=True) * lr.
hidden_weights += inputs.T.dot(d_hidden_layer) * lr.
hidden_bias += np.sum(d_hidden_layer, axis=0,
               keepdims=True) * lr.

print("Final hidden weights: ", end='')
print(*hidden_weights)
print("Final hidden bias: ", end='')
print(*hidden_bias)
print("Final output weights: ", end='')
print(*output_weights)
print("Final output bias: ", end='')
print(*output_bias)
```

```
print("\n Output from neural network after 10000 epochs: ",
      end = '')
print(*predicted_output)
```

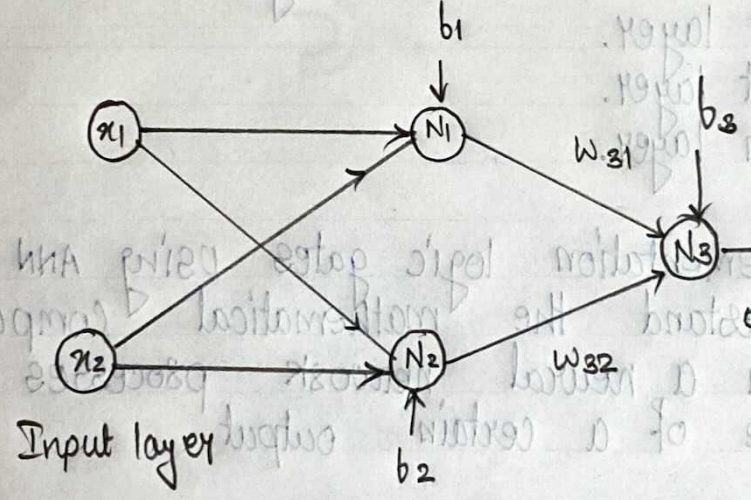
* Github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/Hw4a.py.

* Conclusion:
Through this experiment, we learnt classification of linearly inseperable problems by taking example of XOR and using MLP/EBP.

* Reference:
AI illuminated by Ben Coppin.



general diagram of MLP:



Neural Network model for XOR:

```
Initial hidden weights: [0.41465228 0.3630467 ] [0.77553507 0.08045423]
Initial hidden biases: [0.11104104 0.93874837]
Initial output weights: [0.70577545] [0.56541599]
Initial output biases: [0.02674812]
Final hidden weights: [5.92333998 3.65104122] [5.86972939 3.64133575]
Final hidden bias: [-2.44752239 -5.57608436]
Final output weights: [7.47061027] [-8.13664207]
Final output bias: [-3.34689509]

Output from neural network after 10,000 epochs: [0.05826709] [0.94585437] [0.94601877] [0.05875393]
```

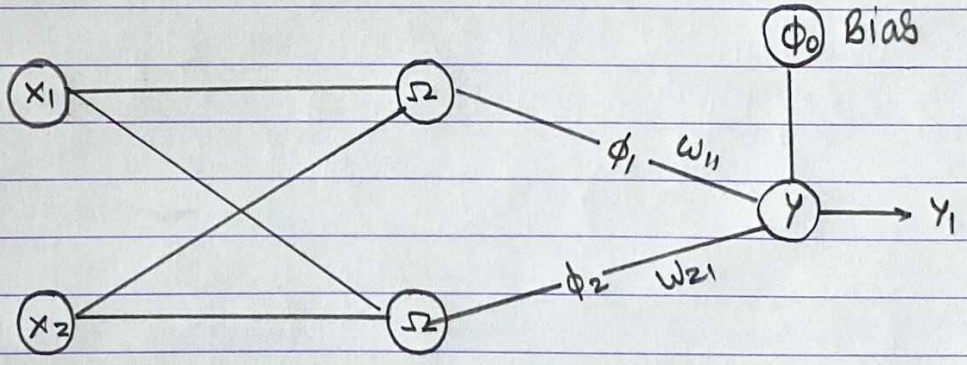
Term Work 4b:

* Problem Defination:
Implementation of XOR gate using radial Basis function network (RBFN)

* Objectives:
- Understand working of RBFN
- Implement XOR using RBFN.

* Theory:
The RBFN is a particular type of NN. The approach of RBFN is more initiative : I/p.s similarity to examples from training set.

Gradial radiation function : $\phi(x) = \exp(-x^2/2\sigma^2)$
where : $\sigma > 0$.

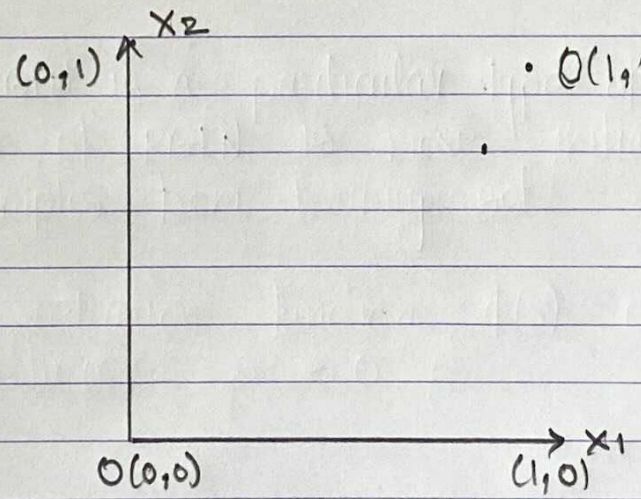


Algorithm:
//considering RBFN architecture
1. P - No of input features/values.
2. m - No of transformed vectors dimensions (hidden layer width) ($m \geq P$)
3. Each node in hidden layers, performs a set of

non linear radian basis function.

4. Only hidden layer nodes perform radial basis function.
5. Output layer performs the linear combination of the output of the hidden layer to give a final probiolistic value at o/p layer.

Example :



$\phi \rightarrow t_1 \rightarrow (0,0)$; $\sigma_1 = 1$

$\Rightarrow \phi_1(x) = e^{\frac{-|x - t_1|^2}{2}}$

$\Rightarrow \phi_2(x) = e^{\frac{-|x - t_2|^2}{2}}$

$p = 2$

$\phi_3 \rightarrow t_3 = (1,0)$; $\sigma_3 = 1$

$\Rightarrow \phi_3(x) = e^{\frac{-|x - t_3|^2}{2}}$

$\phi_4 \rightarrow t_4 = (1,1)$; $\sigma_4 = 1$

$\Rightarrow \phi_4(x) = e^{\frac{-|x - t_4|^2}{2}}$

input	ϕ_1	ϕ_2	ϕ_3	ϕ_4	$\Sigma w_i \phi_i$	Output
0.1	1.0	0.6	0.6	0.4	-0.2	0
0.1	0.6	1.0	0.4	0.6	0.2	1
1.0	0.6	0.4	1.0	0.6	0.2	1
1.1	0.4	0.6	0.6	1.0	-0.2	0

* Source Code :

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.initializers import Initializer
from tensorflow.keras.layers import Layer
from tensorflow.keras.initializers import RandomUniform,
Initializer, Constant

def gaussian_rbf(x, landmark, gamma=1):
    return np.exp(-gamma * np.linalg.norm(x-landmark)
                  **2)

def end_to_end(x1, x2, ys, mu1, mu2):
    from_1 = [gaussian_rbf(i, mu1) for i in zip(x1, x2)]
    from_2 = [gaussian_rbf(i, mu2) for i in zip(x1, x2)]

    plt.figure(figsize=(13,5))
    plt.subplot(1,2,1)
    plt.scatter((x1[0], x1[3]), (x2[0], x2[3]), label=
                "class_0")
    plt.scatter((x1[1], x1[2]), (x2[1], x2[2]), label=
                "class_1")
```

```
plt.xlabel("$X1$", fontsize=15)
plt.ylabel("$X2$", fontsize=15)
plt.title("Xor: Linearly Inseparable", fontsize=15)
plt.legend()

plt.subplot(1,2,2)
plt.scatter(from_1[0], from_2[0], label="Class_0")
plt.scatter(from_1[1], from_2[1], label="Class_1")
plt.scatter(from_1[2], from_2[2], label="Class_1")
plt.scatter(from_1[3], from_2[3], label="Class_0")
plt.plot([0,0.95],[0.95,0],"k--")
plt.annotate("seperating hyperplane", xy=(0.4,0.55),
    xytext=(0.55,0.66), arrowprops=dict(facecolor
    ='black', shrink=0.05))
plt.xlabel(f"$mu1$: {(mu1)}", fontsize=15)
plt.ylabel(f"$mu2$: {(mu2)}", fontsize=15)
plt.title("Transformed Inputs: Linearly seperable",
    fontsize=15)
plt.legend()

A=[]
for i,j in zip(from_1, from_2):
    temp=[]
    temp.append(i)
    temp.append(j)
    temp.append(1)
    A.append(temp)

A = np.array(A)
```

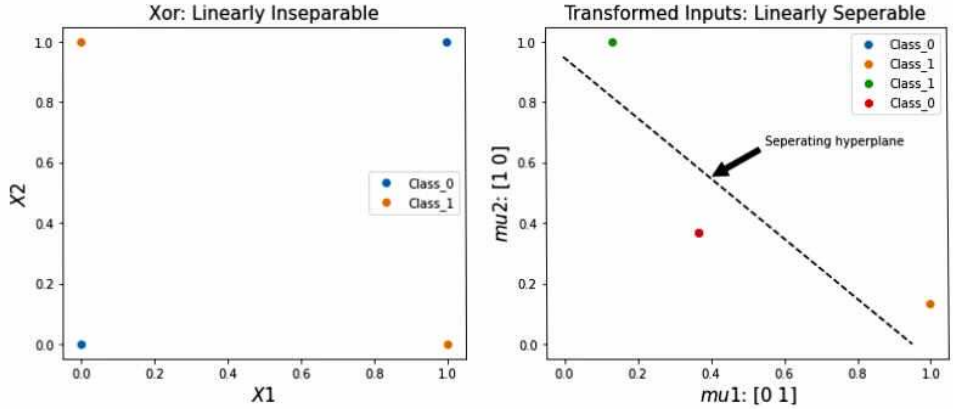
```
    w = np.linalg.inv(A.T.dot(A)).dot(A.T).dot(ys)
    print(np.round(A.dot(w)))
    print(ys)
    print(f"weights: {w}")
    return w

def predict_matrix(point, weights):
    gaussian_rbf_0 = gaussian_rbf(np.array(point), mu1)
    gaussian_rbf_1 = gaussian_rbf(np.array(point), mu2)
    A = np.array([gaussian_rbf_0, gaussian_rbf_1, 1])
    return np.round(A.dot(weights))

x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
ys = np.array([0,1,1,0])

mu1 = np.array([0,1])
mu2 = np.array([1,0])

w = end_to_end(x1, x2, ys, mu1, mu2)

print(f"Input: {np.array([0,0])}, predicted: {predict_
    matrix(np.array([0,0]), w)}")
print(f"Input: {np.array([0,1])}, predicted: {predict_
    matrix(np.array([0,1]), w)}")
print(f"Input: {np.array([1,0])}, predicted: {predict_
    matrix(np.array([1,0]), w)}")
printf(f"Input: {np.array([1,1])}, predicted: {predict_
    matrix(np.array([1,1]), w)}")
```

* github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/TW4b.py.

* Conclusion :
Through this, we learnt the working of RBFN and also implemented XOR base on RBFN.

* References:
Virtual labs, towardsdatascience.com.

```
[0. 1. 1. 0.]
[0 1 1 0]
Weights: [ 2.5026503   2.5026503  -1.84134719]
Input:[0 0], Predicted: 0.0
Input:[0 1], Predicted: 1.0
Input:[1 0], Predicted: 1.0
Input:[1 1], Predicted: 0.0
```



Termwork 5

* Problem defination :
Implement Hebbian rule and correlation rule.

* Objectives :
To understand the Hebbian rule learning be applied for supervised learning of neural network. To understand the correlation rule that can be applied for supervised learning of neural network.

* Theory
1. Hebbian learning rule :
It is one of the oldest learning algorithms and is based on large part of dynamics of biological systems.
The general idea is that any two cell systems of cells are repetedly active at the same time will lend to become associated so that the activity one facilitates activity in the other.

According to the hebbian rule,

$$\Delta w_i = n(o_i)x$$

i.e the change in the synoptic weights of the i^{th} neuron Δw_i is equal to the learning rate n times the i^{th} output o_i times the input x.

2. Correlation learning rule :
Its rules are based on a similar principle to the Hebbian learning rule. It assumes that weights between simultaneously responding neurons

should be largely positive and weights b/w neurons with opposite reaction should be largely negative.
The correlation rule is co supervised learning instead of an actual response o, the desired response P is used for the weight change calculation.

$$\Delta w_{ij} = n(o_i) x_j.$$

Where Di is the desired value of the alphabet signal. This learning algorithm usually starts with the initialization of the weight to years.
This simple rule states that if Di is the desired response due to x, then corresponding weight increase is proportional to this product.

* Source Code:

```
#xi = [ip1, ip2]
x1 = [1,1].
x2 = [1,-1].
x3 = [-1,1]
x4 = [-1,-1].
xilist = [x1, x2, x3, x4]
y = [1,-1,-1,-1].

w1 = w2 = bw = 0
b = 1

def heb_learn():
```

```
global w1, w2, bw
print("dw1\tdw2\tdb\tw1\tw2\tb")
i=0
for xi in xilist:
    dw1 = xi[0]*y[i]
    dw2 = xi[1]*y[i]
    db = y[i]

    w1 = w1 + dw1
    w2 = w2 + dw2
    bw += db

    print(dw1, dw2, db, w1, w2, bw, sep='\t')

    i+=1

print("Learning ...")
heb_learn()
print("\nlearning complete")
print("Output of AND gate using obtained w1,w2,
      bw :")
print("x1\tx2\ty")
for xi in xilist:
    print(xi[0], xi[1], w1*wi[0]+w2*xi[1]+b*bw,
          sep='\t')
print("lets define the function F as F(x) = 1
      if F(x)>1 and F(x) = -1 if F(x)<=1")
for xi in xilist:
    print(xi[0], xi[1], 1 if w1*xi[0]+w2*xi[1]+
```

b*bw>0 else -1, sep='\t')

* github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/TW5.py.

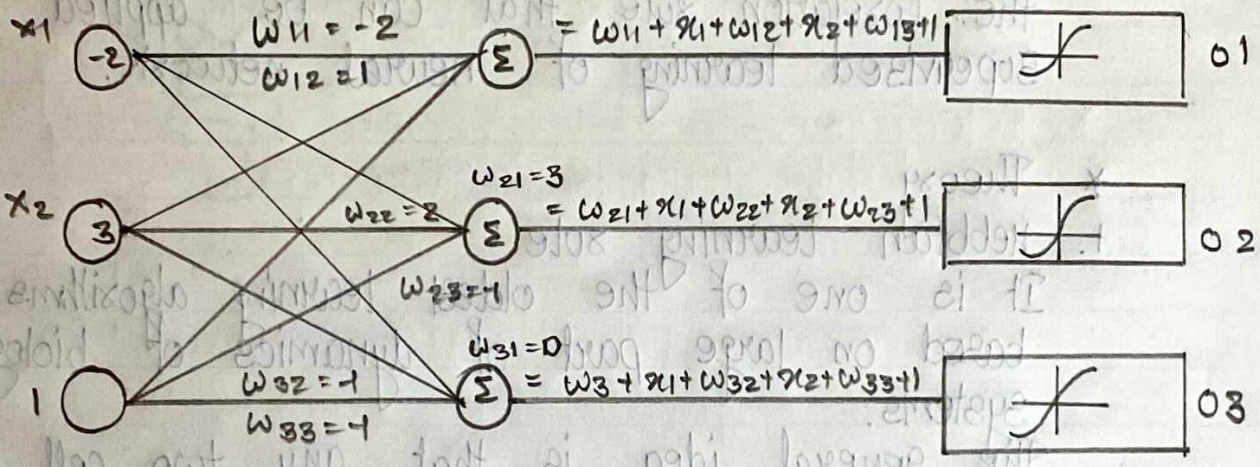
* Conclusion:
Through this experiment we implemented Hebbian rule and coxelation rule.

* Reference:
A.I illuminated by Ben Coppin.

Illustration :

Activation Function used : $x_1 = 2$ $x_2 = 3$, bias fixed ip = 1

$$F(x) = \frac{2}{1+e^{-x}} - 1.$$



Calculate output as $O = F(w \times X)$

$$O = F_c \begin{bmatrix} -2 & 1 & -6.5 \\ 3 & 2 & 1 \\ 0 & -1 & -1.5 \end{bmatrix} + \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

$$= F_c \begin{bmatrix} 0.5 \\ 1 \\ -4.5 \end{bmatrix} = \begin{bmatrix} 0.2449 \\ 0.4621 \\ -0.978 \end{bmatrix}$$

Desired output D is given by [1 -1 -1]

Since $\Delta w_i = n(O_i)x$

$w_{new} = w_{old} + \Delta w$.

Applying the Formula

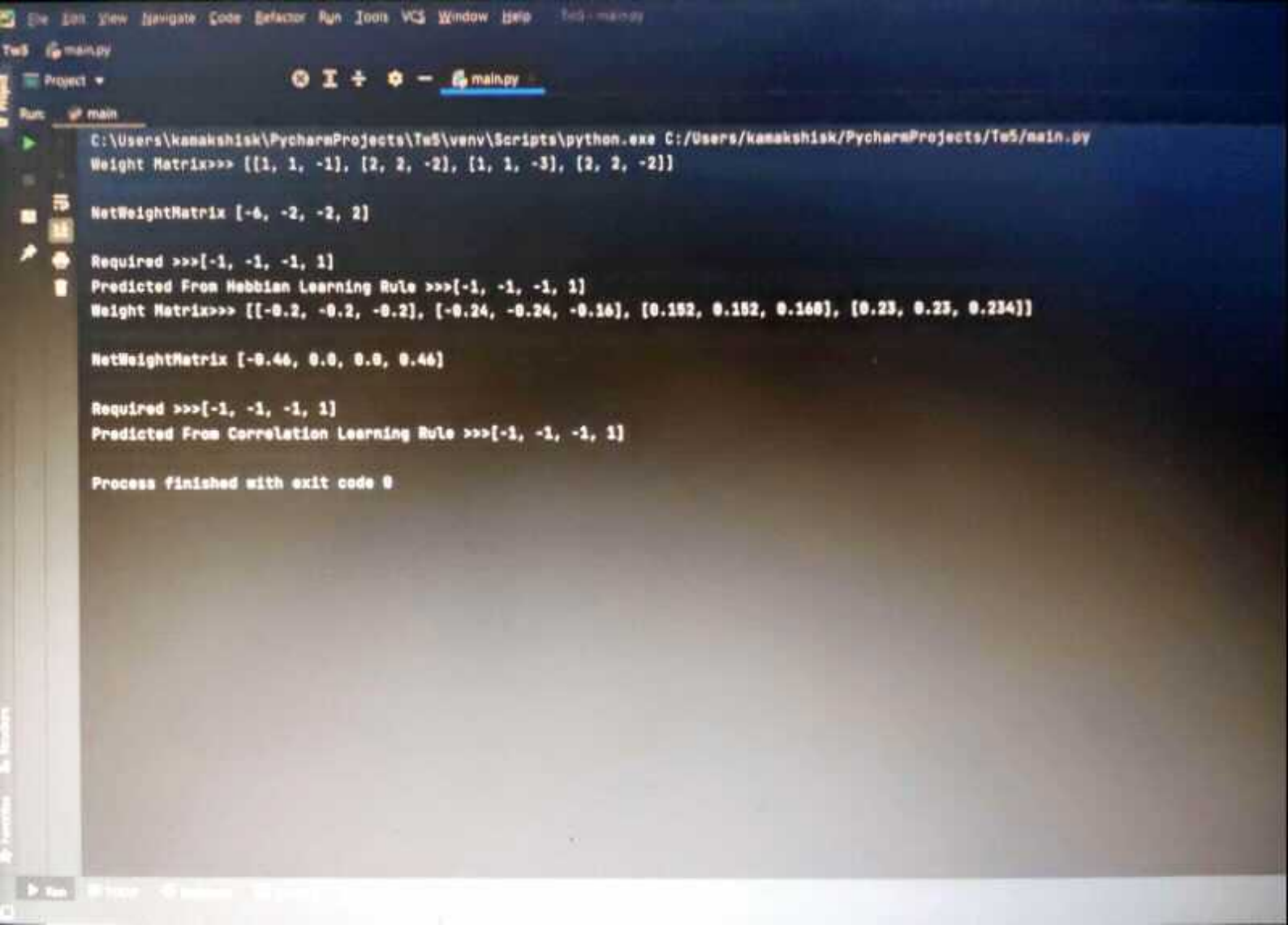
$$\begin{bmatrix} -2.1956 & 1.2939 & -6.402 \\ 2.6303 & 2.5545 & 1.1848 \\ 0.7829 & -2.1736 & -1.8912 \end{bmatrix}$$

In correlation learning rule, the activation function used is signum function.

let $x_1 = -2$ & $x_2 = 8$

bias = 1

$$f(x) = \begin{cases} 1 & x >= 0 \\ -1 & x < 0 \end{cases}$$ signum function.


File Edit View Navigate Code Refactor Run Tools VCS Window Help
Tw5 main.py
Project
main.py
Run: main
C:\Users\kamakshisk\PycharmProjects\Tw5\venv\Scripts\python.exe C:/Users/kamakshisk/PycharmProjects/Tw5/main.py
Weight Matrix>>> [[1, 1, -1], [2, 2, -2], [1, 1, -3], [2, 2, -2]]
NetWeightMatrix [-6, -2, -2, 2]
Required >>>[-1, -1, -1, 1]
Predicted From Hebbian Learning Rule >>>[-1, -1, -1, 1]
Weight Matrix>>> [[-0.2, -0.2, -0.2], [-0.24, -0.24, -0.16], [0.152, 0.152, 0.168], [0.23, 0.23, 0.234]]
NetWeightMatrix [-0.46, 0.0, 0.0, 0.46]
Required >>>[-1, -1, -1, 1]
Predicted From Correlation Learning Rule >>>[-1, -1, -1, 1]
Process finished with exit code 0

Term work 6

* Problem defination:
Implement Find-s and candidate elimination algorithm.

* Objectives:
1. To understand the Find-s algorithm
2. To understand the concept of candidate elimination algorithm.
3. To write the python program to implement both Find-s and candidate elimination algorithm.

* Theory:
1. Candidate elimination:
This will operate in a similar manner to the simple algorithm. Candidate elimination method stores two set of hypothesis. In addition to maintaining a set of most specific hypothesis that match the training data, this method also maintains a set of hypothesis that starts out as set of with the single item <? ? ? ? ? ?> and ends up being a set of the most general hypothesis that match all the training data.
This algorithm is thus able to make use of negative training data as well positive training data.

2. Find s Algorithm:
It finds the most specific hypothesis that fits all the positive examples. We have to note here the algorithm considers only those positive training example. It starts with the most specific hypothesis &

generalises this hypothesis each time it fails to classify an observed positive training data. Hence Find S algorithm moves from the most specific hypothesis to the most general hypothesis.

* Steps involved in Find S algorithm :

1. Start with most specific hypothesis
   $h = \{\phi, \phi, \phi, \phi, \phi, \phi\}$.
2. Take the next example and if it is negative, then no changes occur to the hypothesis.
3. If the example is positive and we find that our initial hypothesis is too specific then we update our current hypothesis to general conditions.
4. Keep repeating the above steps, till all the training examples are complete.

* Source code for candidate elimination method

```
import numpy as np
import pandas as pd

data = pd.read_csv('C:/Users/parth/Downloads/candata.csv')
concepts = np.array(data.iloc[:,0:-1]
print("\n Instances are :\n", concepts)
target = np.array(data.iloc[:,-1]
print('\n Target values are :", target)
```

```
def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("\n initialization of specific_h and
          general_h")
    print("\n specific boundary :", specific_h)
    general_h = [["?" for i in range(len(specific_h))]

for i in range(len(specific_h))]
    print("\n Generic Boundary :", general_h)

    for i,h in enumarate(concepts):
        print("\n Instance", i+1, "is", h)
        if target[i] == "yes":
            print("Instance is positive")
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = "?"
                    generate [x][x] = '?'

        if target[i] == "no":
            print("Instance is negative")
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else
                    general_h[x][x] = '?'

    print("specific Boundary after", i+1, "Instance
          is", specific_h)
```

```
        print(" Generic Boundary after", i+1,
              "Instance is" general_h)
        print('\n')

    indices = [i for i,val in enumarate(general_h) if val
               == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
            general_h.remove (['?', '?', '?', '?', '?', '?'])
    return. specific_h, general_h

s_final, g_final = learn( concepts, target)

print("Final. specific_h :", s_final, sep="\n")
print("Final_generate :", g.final, sep="\n")
```

* Source code for find s algorithm :

```
impost pandas as pd
impost numpy as np

data = pd.read_csv ("data.csv")
print( data, "n")

d = np. array (data) [:, :-1]
print("n the attributes are :", d)

-les
target = np. array (data) [ :-1]
print ("n the target is :", target)
```

```
def train(c,t):
    for i, val in enumarate(t):
        if val == "yes":
            specific_hypothesis = [[i].copy()]
            break
    for i,val in enumarate(c):
        if t[i] == "yes":
            for x in range(len(specific_hypothesis)):
                if val[x] != specific_hypothesis[x]:
                    specific_hypothesis[x] = '?'
                else
                    pass

    return specific_hypothesis

print("n the final hypothesis is:", train(d, target1)
```

* github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/tree/main/TCS6.

* Conclusion:
At the end of this experiment we are able to understand the concept of Find S and candidate elimination algorithm and we successfully built the python program to implement the Find S and candidate elimination algorithm.

* Reference :

Ben coppin, Artificial Intelligence Illuminated, Jones & Bartlet 2004.

```
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.19.0 -- An enhanced Interactive Python.

Populating the interactive namespace from numpy and matplotlib

runfile('C:/Users/umk/.spyder-py3/temp.py', wdir='C:/Users/umk/.spyder-py3')
   Sunny  Warm Normal  Strong Warm.1    Same  Yes
0  Sunny  Warm   High  Strong   Warm    Same  Yes
1  Rainy  Cold   High  Strong   Warm  Change   No
2  Sunny  Warm   High  Strong   Cool  Change  Yes n
n The attributes are:  [['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]
n The target is:  ['Yes' 'No' 'Yes']
n The final hypothesis is: ['Sunny' 'Warm' 'High' 'Strong' '?' '?']
```

Term work 7:

* Problem defination:
Build a linear regression model using housing prices.

* Objective:
1. To understand the concept of linear regression.
2. To build program to implement linear regression for the housing prices model.

* Theory:
Linear regression:
It is a statical model that estimates the relationship between one dependent variable and one or more independent variables using a line.

It can be used to predict the continuous numerical value of dependent variable and one or more independent variables using a line.

Linear regression may be simple or multiple regression. The simple linear regression has only one independent variable whereas multiple regression has 2 or more independent variables.

Relationship b/w two variables x & y is given by

$$cov(x,y) = \frac{\sum_{i=1}^{n} (x-\bar{x})(y_i-\bar{y})}{n-1}$$

- In correlation, the two variables are treated as equals.
- In regression, one variable is considered independent (= predictor) variable (x) and the other the dependent (= outcome) variable y.

* What's slope?
A slope of 2 means that every 1-unit change in x yields a 2-unit change in y.

Prediction: If you something about X, this knowledge helps you about predicting something about Y.

Regression equation: $E(Y_i|x_i) = \alpha + \beta x_i$.

Predicted value for an individual

$$\hat{y}_i = \underbrace{\alpha + \beta * x_i}_{\text{Fixed exactly on the line}} + \underbrace{\text{random error}}_{\text{Follows normal distribution}}$$

Linear regression assumes that:
a. The relationship b/w X & Y is linear.
b. Y is distributed normally at each value of x.
c. The variance of Y at every value of x is the same.
d. The observations are independent.

Slope (beta coefficient) :

$$\hat{\beta} = \frac{Cov(x,y)}{var(x)}.$$

Intercept calculate : $\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$.

Regression line always goes through the point : $(\bar{x}, \bar{y})$

* Relationship with correlation :

$$\hat{\beta} = \hat{\beta} \cdot \frac{SD_x}{SD_y}.$$

* Source code :

```
import pandas as pd
import numpy as np
import matplotlib.py plot as plt
from sklearn_datasets import load boston
from sklearn_model_selection import traintest_split.
from sklearn.preprocessing import standardScalar
from sklearn_linear_model import LinearRegression
import seaborn as sns
import statsmodels.formula.api as smf
from statsmodels.stats.outliers_influence import
    variance_inflation_factos.
import warnings
warnings.filterwarnings("ignore")
```

```
plt.style.use("ggplot")

data=load_boston()

df = pd.DataFrame(data.data,columns = data.feature_names)
df['MEDV'] = data.target
print('Details of the data base:):
df.head()
df.describe()
df.info()

df.isna().sum()

#using data visualization
plt.figure(figsize=(20,15))
plotnumber=1.

for column in df:
    if plotnumber<=14:
        ax = plt.subplot(3,5,plotnumber)
        sns.scatterplot(x=df['MEDV'],y=df[column])

    plotnumber+=1
plt.tight_layout()
plt.show()

plt.figure(figsize=(20,8))
sns.boxplot(data=df, width=0.8)
plt.show()
```

```
x = df.drop(columns = "MEDV", axis = 0)
y = df["MEDV"]

scalar = standardScalar()
x_scaled = scalar.fit_transform(x)
x_scaled
vif = pd.DataFrame()
vif["VIF"] = [variance_inflation_factor(x_scaled,i)
for i in range (x_scaled.shape[0])]
vif["Features"] = x.columns

vif
fig,ax = plt.subplots(figsize=(16,8))
sns.heatmap(df.corr(), annot = True, fmt = '1.1F'
annot_kws = {size:10}, linewidth = 1)
plt.show()

lm = smf.ols(formula = 'MEDV~RAD', data = df).fit()
lm.summary()

df.drop(columns = 'RAD', axis = 1, inplace=True)
df.head()

x_train, x_test, y_train = train_test_split(x_scaled,
    y, test_size = 0.30, random_state = 0)

lr_linearRegression
lr.fit(x_train, y_train)
y_pred = lr.predict(x_test)
```

```
print ("Training accuracy of the model = ", lr.score(
                    (X_train, y_train))
print ("Testing accuracy of the model = ", lr.score(
                    (X_train, y_train))

def adj_r2 (x,y, model)
    r2 = model.score(x,y)
    n = x.shape[0]
    p = x.shape[1]
    adjusted_r2 = 1- (1.r2) * (n-1) / (n*p*1)

    return adjusted_r2

print ("In After adjusted R-squared new value of
        accuracy In"):
print ("Training accuracy of the model = ",
        adj- r2 (X_train, y_train.lr))
print ("Testing accuracy of the model = " adj_r2(X_test,
        y_test, lr))
```

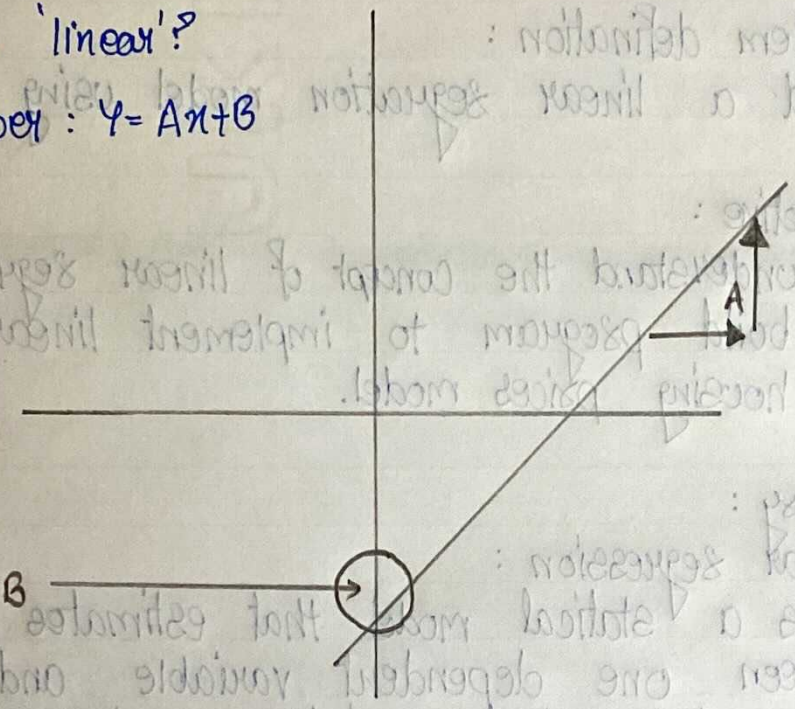
* github link
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/hw7.py.

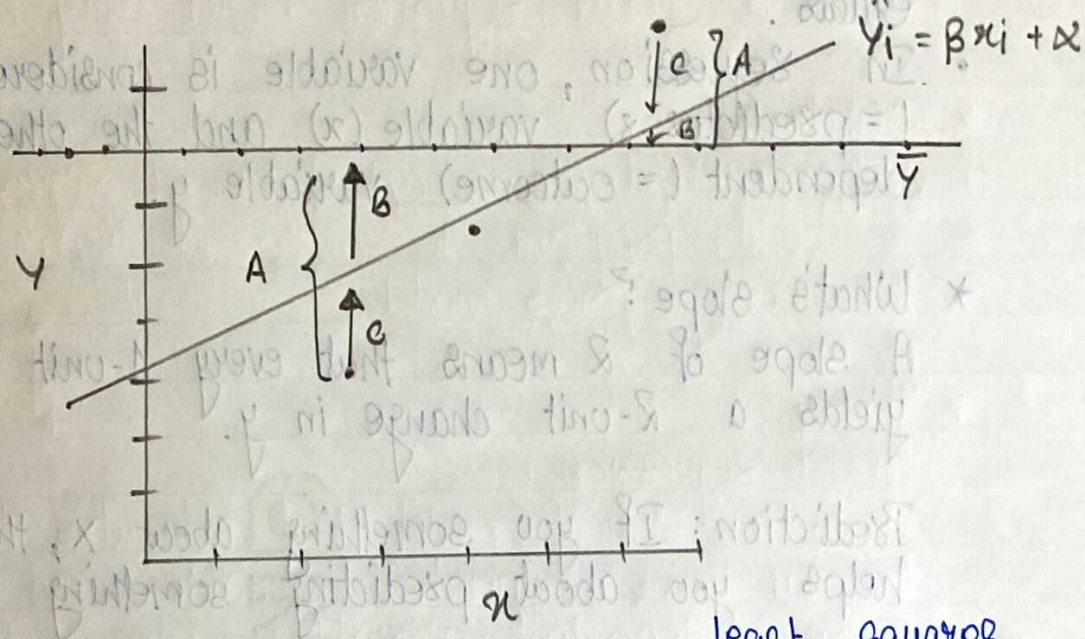
* Conclusion :
We are able to develop program to illustrate linear regression.

* Reference :
Ben Coppin, AI illuminated Jones and bartlet 2004.

What is 'linear'?

Remember : $Y = AX + B$





least squares estimation give us the line (β) that minimized c^2

$$\sum_{i=1}^{n} (y_1 - \bar{y})^2 = \sum_{i=1}^{n} (\hat{y}_1 - \bar{y})^2 + \sum_{i=1}^{n} (\hat{y} - y_i)$$

A^2_{ss} B^2_{ss} C^2_{ss}

Total squared distance of observation from naive mean of y.

```
[ ] model.coefficients
```

```
SUCCESS: Optimal solution found.
```

name	index	value	stderr
(intercept)	None	101.13831392475674	5.272933357982522
features	None	50.007088185477485	1.1598906969719902

[2 rows x 4 columns]

```
[ ] new_point = tc.SFrame({'features': [4]})

    model.predict(new_point)
```

```
dtype: float
Rows: 1
[301.1666666666667]
```

```
[ ]
```

TermWork 8:

* Problem defination:
Implement spam detection using NAIVE Bias Algorithm.

* Objective:
1. Implement spam detection using NAIVE Bias Algorithm.
2. Understand NAIVE Bias algorithm.

* Theory:
Naive bias classifier is a simple but effective learning system. Each piece of data that is to be classified consists of a set of attributes, each of which can take on a number of possible values.

Posterior probability of each possible classification is calculated

$$P(C_i | d_2 \_ d_n)$$

The hypothesis that has highest posterior probability known as the maximum a posterior or MAP hypothesis

$$\frac{P(d_1 \ldots d_n | C_i) \cdot P(C_i)}{P(d_1 \ldots . d_n)}$$

Because we are simply toying to find the highest probability and because $P(d_1 \_ \_ d_n)$ is a constant independent of C_i, we can eliminate it & simply and aim to find the classification C_i, for which

is maximized

$$P(d_1 \cdots d_n | C_i) \cdot P(C_i)$$

The naive bayes classifier now assumes that each of attributes in the data items is independent of others, in which above equation can be rewritten as

$$P(C_i) \cdot \prod_{j=1}^{n} P(d_j | C_i)$$

Zero frequency problem?
What if any of count is 0?
- Add 1 to all columns
- It is a form of landscape smoothing.

Applications are:
1. Real time prediction
   Naive Bayes is an eager learning classifier and it is sure fast. Thus it could be used for making prediction in real time.

2. multi class prediction
   well known for multi class prediction feature.

3. Text classification / spam filtering / sentiment analysis
   Mostly used in text classification.
   Has higher success rate as compared to other algorithms.

* Source code :

```
import    numpy as np
import    pandas as pd
import    nltk
import    string
import    pandas
import    matplotlib.pyplot as plt
import    seaborn as sns
from      sklearn.feature_extraction.text import
          CountVectorizer
from      sklearn.model_selection import train_test_split
from      sklearn.naive_bayers import MultinomialNB
from      sklearn.metrics import accuracy_score,
          precision_score, recall_score, f1_score

df_sms = pd.read_csv('spam.csv', encoding = 'latin-1')
df_sms = df_sms.drop(['Unamed:2", "unnamed:3",
         "unnamed:4"], axis =1)
df_sms = df_sms.rename(columns = {"V1" : "label",
         "V2" : "sms"})

print(len(df_sms))
print(df_sms.head())
print(df_sms.tail())

print(df_sms.label.value_counts())
df_sms['length'] = df_sms['sms'].apply(len)
```

```
df_sms['length'].plot(bins=50, kind='hist')
df_sms.loc[:,'label'] = df_sms.label.map({'ham':
                                0, 'spam':1})

print(df_sms.shape)

documents = ["Hello, how are you!","Win money, win from",
             "call me now","Hello call hello you
             tomorrow?"].

lower_case_documents = []
lower_case_documents = [d.lower() for d in documents]
sans_punctuation=documents = []

for i in lower_case_documents:
        sans punctuation_documents.append(i,translate.
         (str.maketrans("","",string.punctuation)))

sans_punctuation_on_documents:
preprocessing_documents = [[w.for w in d.split()] for d
in sans_punctuation_documents]
preprocessed_documents

count_vectos = CountVectorizer()
count_vectos.fit(documents)
count_vectos.get_Feature_name()
doc_array = count_vectos, transform(documents).toarray()
doc_array
```

```
Frequency_matrix = pd.DataFrame(doc_array, columns =
                   countvectors.get_feature_names())
Frequency_matrix

x_train, x_test, y_train, y_test = train_test_split
        (df_sms['sms'],
         df_sms['label'], test_size = 0.20,
         random_state = 1)

count_vectors = CountVectorizer()
training_data = count_vectors.fit_transform(x_train)
training_data = count_vectors.transform(x_test)

naive_bayes = MultinominalNB()
naive_bayes.fit(training_data, y_train)
predictions = naivebayes.predict(testing_data)

print("Accuracy score : {}", format(accuracy = score
        (y_test, predictions)))
print('precision score : {}', format(precision_score
        (y_test, predictions)))
print('Recall score : {}', format(recall_score
        (y_test, predictions)))
print("F1 score : {}', format[F]_score(y_test,
        predictions)).
```

* github link:
https://github.com/parth-zanvar/Machine-Learning-Lab/blob/main/Tws.py.

* Conclusion :
Implemented and understood the spam dectection using naive Bias algosithm.

* Reference :
Ben Coppin, AI illuminated Jones and bartlet 2004.

```
[ ] predict_naive_bayes('lottery sale')

    0.9638144992048691

[ ] predict_naive_bayes('Hi mom how are you')

    0.12554358867164467

[ ] predict_naive_bayes('Hi MOM how aRe yoU afdjsaklfsdhgjasdhfjklsd')

    0.12554358867164467

[ ] predict_naive_bayes('meet me at the lobby of the hotel at nine am')

    6.964603508395967e-05

[ ] predict_naive_bayes('enter the lottery to win three million dollars')

    0.9995234218677428

[ ] predict_naive_bayes('buy cheap lottery easy money now')

    0.999973472265966
```

Termwork 9:

* Title:
Implement hand writing classification using support vectos machines.

* Objective:
- To understand the concept of suppost vectos machine
- To understand the concept of classification and as well as segression by making case svm.

* Theory:
svm is one of the most popular supervised learning algorithms, which is used for classification as well as segression problems.
To goal of svm algorithm is to create the best line or decision boundary that can segregerate n-dimensional space into classes so that we can easily put the data point in the correct category in future. This best decision boundary is called a hyperplane.

svm chooses the extreme points that help in creating the hyperplane. These extreme cases are called as suppost vectors and hence algorithm is called svm. svm Algorith can be used for

1. Face dectection.
2. Image classification.
3. Text categorization.

Consider the beside diagram in which there are two different categories that are classified using a decision boundary or hyperplane

Types of svm

1. Linear svm :

Linear svm is used for linearly seperable ~~datas~~ data, which means if a dataset can be classified into two classes by using a single straight line then such data is termed as linearly seperable data and classifier is used called as Linear svm classifier.

2. Non linear svm :

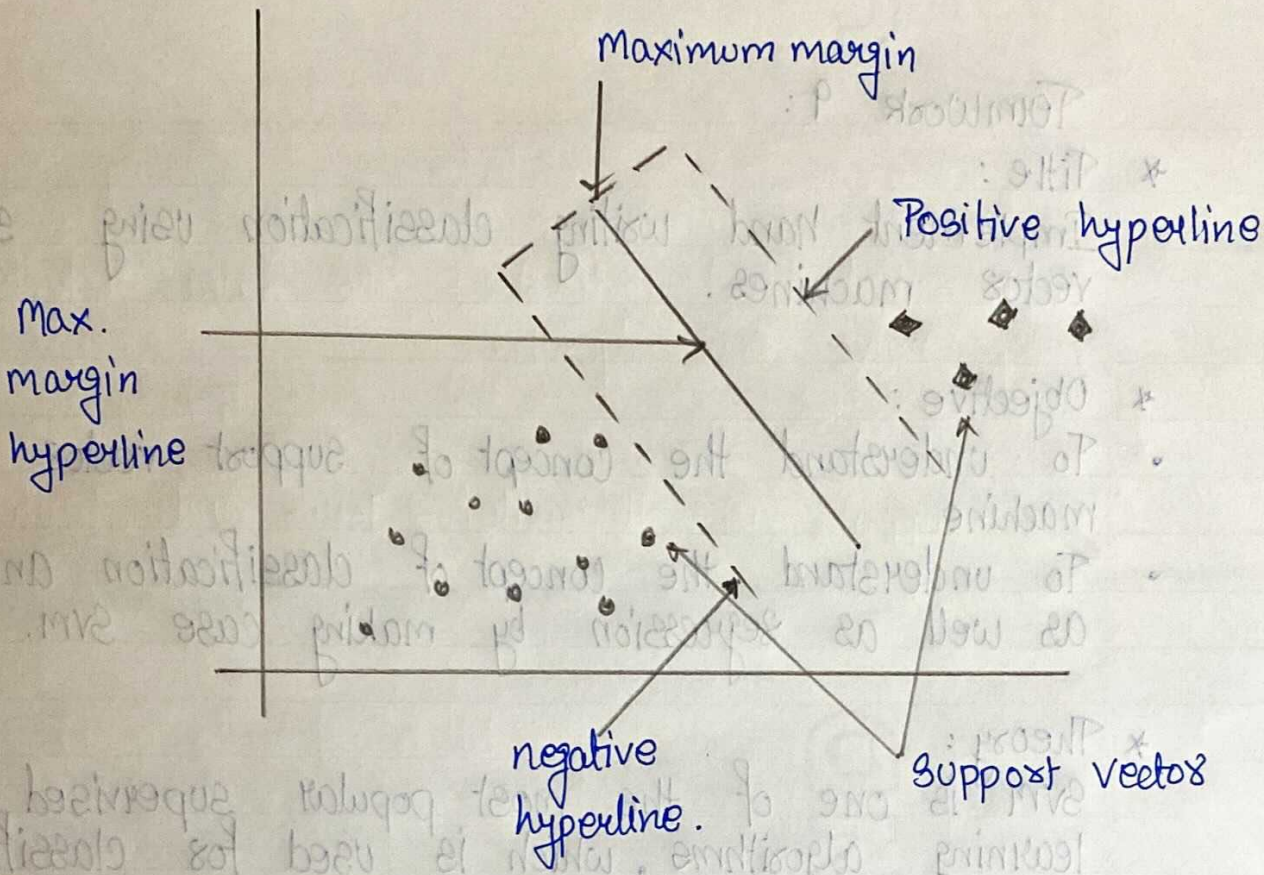
It is used for non linearly seperated data which means if a dataset cannot be classified by using a straight line then such data is termed as non linear data classifier used is called as non-linear classifier.

* Conclusion :

At the end of this termwork we can be able to understand the concept of support vector machine Algorithm and we implemented that algorithm using python program.

* Reference :

www.javapoint.com.


Maximum margin
Positive hyperline
Max.
margin
hyperline
negative
hyperline.
Support vectors

Termwork 10:

* Title:
Implement FP-tree for finding co-occuring words in a twitter feed.

* objectives:
To understand the concept of frequent pattern growth algorithm. in detail

* Theory:
FP growth algorithm is an improvement of apriori algorithm. It is used to find the frequent itemset in a transaction database without candidate generation. FP growth represents frequent items to FP-tree.

The requisites for FP tree are apriori algo and trie-data structure

Consider the following example:

Transaction ID	Items
T1	{E K M N O Y}
T2	{D E K N O Y}
T3	{A E K M}
T4	{C K M U Y}
T5	{C E I K O O}

The above given data is a hypothetical dataset of transactions with each letter representing an item.

is computed. let us minimum support be 3. A frequent pattern set is built will contain all the elements whose frequency is greater than or equal to minimum support.

item	Frequency
A	1
C	2
D	1
E	4
I	1
K	5
M	3
N	2
O	3
U	1
Y	3

After insertion of the relevant items, the set L looks like this:

$$L = \{K:5, E:4, M:3, O:3, Y:3\}$$

Transaction ID	items	ordered item set
T1	{E, K, M, N, O, Y}	{K, E, M, O, Y}
T2	{D, E, K, N, O, Y}	{K, E, O, Y}
T3	{A, K, E, M}	{K, E, M}
T4	{C, K, M, U, Y}	{K, M, Y}
T5	{C, E, I, K, O, O}	{K, E, O}

Now for each item, the conditional frequent pattern tree built. It is done by taking set of elements which is common in all paths, in the conditional pattern base of that item and calculating its support count by summing the support counts of all paths in the conditional pattern base

Items	conditional pattern base	Conditional FP-Tree
Y	{{K,E,M,O:1}, {K,E,O:1} {K,M:1}}	{K:3}
O	{{K,E,M:1}, {K,E:2}}	{K,E:3}
M	{{K,E:2}, {K:1}}	{K:3}
E	{K:4}	{K:4}
K		

From the conditional FP tree the FP are generated by pairing the items of each conditional FP. tree corresponding to the item is given in below table.

items	Frequent Pattern Generated
Y	{<K,Y:3>}
O	{<K,O:3>, <F,O:3>, <E,K,O:3>}
M	{<K,M:3>}
E	{<K,E:3>}
K	

* Conclusion:
At the end of this homework we understood the

concept of frequent pattern tree and we implemented that successfully.

* Reference:
www.GeeksforGeeks.com (FP-growth algorithm).