2G118CS076.

Termwork - 1

Title :- Implement DFID algorithm and compare its performance with DFS and BFS algorithm.

Theory:- In AI, search techniques are universal solving methods. Properties of search Algorithms are:-

1.) Completeness :- A search algorithm is said to be complete if it guarantees to return a solution if atleast any solution exists for any random input.

2.) Optimality :- If a solution found for an algorithm is guaranteed to be the best solution (longest path cost) among all the other solutions; then such solution for is said to be optimal solution.

3.) Time Complexity :- Time complexity is a measure of time for an algorithm to complete task.

4.) Space Complexity :- It is maximum storage space required at any point during the work i.e search as complexity of the problem.

Types of search algorithms are:-

1. BFS (Breadth First Search) Algorithm.
2. DFS (Depth First Search) Algorithm.

1. BFS uses queue data structure.
   * It can be used to find single source shortest path in an unweighted graph.
   * It's more suitable for searching vertices which are closer to given source.
   * The time complexity of BFS is O(V+E).

2. DFS uses stack data structure.
   * In DFS we might transverse more edges to reach a destination vertex from a source. It might be more

Suitable when solutions are away from the source.

* The time complexity of DFS is O(V+E)
* When adjacent list is used O(V^2)

Algorithm

1.) DFS

Step 1: Create a recursive function that takes the index of nodes and a visted array.

Step 2: Mark the current node a visited and print the nodes.

Step 3: Transverse all the adjacent nodes and unmarked nodes and call the recursive function with index of adjacent node.

2.) BFS

Step 1: Put anyone of the graph's vertices at the back of a queue.

Step 2: Take the front item of the queue and add it to the visited list.

Step 3: Create a list of that vertex's adjacent nodes. Add the one's which aren't in the visited list to the back of the queue.

Step 4: Keep repeating step 2 and step 3 until the queue is empty.

3.) DFID :- It combines the DFS's space efficiency and BFS. It calls DFS for different depths starting from an initial value. The DFS is restricted from going beyond given length.

```
// Returns true if target is reachable from src
// within max-depth
bool IDDFS (src, target, max-depth):
    for limit from 0 to max-depth:
        if DLS (src, target, limit) == true:
            return true
    return false
```

```
//if reachable the maximum depth, stop recursing
    if limit <0:
        return false
    for each adjacent i of src:
        if DFS (v, target, limit-1):
            return true
    return false.
```

BFS Code:- from collections import defaultdict.

```
# This collection represents a directed graph using
  adjacent list representation.
class graph:
    def __init__ (self):
        self.graph = defaultdict (list)
    def addedge (self, u, v):
        self.graph [u] .append [v]
    def DFSutil (self, v, visited):
        visited. add (v)
        print (v, end = " ")
# Recur for all vertices adjacent to this vertex
    for neighbour in self graph [v]:
        if neighbour not is visited
            self. DFS util (neighbour visited)
# The function to do DFS traversal using recursive
  DFSutil ()
    def DFS (self, v):
        visited = set ()
        self. DFSutil (v, visited)
# Driver Code.
# Create a graph in given above.
    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 ("The following is in DFS for from vertex 2"): g.DFS(2)
```

BFS Code.

```
from collections import defaultdict
class Graph
  # constructor
  def__ init__(self):
      self.graph = defaultdict (list)
  def addEdge (self, u, v).
      self.graph [u].append [v].
  def BFS (self, s)
  # mark all vertices as non-visited.
    visited = [False]* (max (self.graph) + 1)
  # Create a queue for BFS.
    queue = []
  # Mark source node as visited and enqueue.
    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:
```

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

print("following is Breadth first starting from vertex 2")
g.gBFS(2).

DFIB code.

```
From collections import defaultdict,
Class Graph:
    def __init__(self,vertices):
        self.v = vertices
        self.graph = defaultdict(list)
    def addEdge(self, v, u)
        self.graph[u].append(v)
    def DLS(self, src, target, maxDepth):
        if src == target
            return True
        if maxDepth <= 0
            return False
        for i in self.graph[src]:
            if (self.DLS(i, target, maxDepth - 1)):
                return true
            return false
    def IDDFS(self, src, target, maxDepth)
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return true
        return false
    def DFSUtil(self, v, visited):
        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):
    visited = set()
        set.DFSutil (v, visited)
def BFS (self, s):
    visited = [False]* max (self)
queue = []
queue.append (s)
visited [s] = true
print ()
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 [7]:                g.addedge (1,4)
g.addedge (0,1)               g.addedge (2,5)
g.addedge (0,2)               g.addedge (2,6).
g.addedge (1,3)
target = 6.
max-depth = 3
src = 0.        if g.IDDFS (src, target, max-depth == true.
                    print ("Target " + "within max Depth"):
                g.DFS (0)       g.BFS (0)
```

Conclusion: DFS is for traversing or searching a tree or graph. and BFS is for traversing a tree or graph. DFID is a combination of DFS & BFS by traversing all possible paths.

References :-

Ben Coppin Artificial Intelligence illumited Jones & Bartlett 2004

Teamwork - 2.

Title : Implement Best - First search algorithm.

Theory :- In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node, so both DFS and BFS blindly explore paths without considering any cost function. The idea of Best First search is to use an evaluation function to decide which adjacent is most promising and then explore. Best first search falls under the category of Heuristic search or informed search. We use a priority queue to store costs of nodes so the implementation is a variation of BFS, we get just need to change queue to priority queue.

Algorithm :- Best First Search (Graph g, Node start).

Step 1 :- Create an empty priority queue.

Priority Queue P-q.

Step 2 : Insert "Start" in pq.

pq. insert (start).

Step 3 : Until priority Queue is empty -

u - priority Queue. Delete min.

If u is the goal.

Exit

Else.

For each neighbour v of u

If v "unvisited"

mark v "visited"

pq. insert (v)

mark u "Examined".

Step 4 : End procedure.

Source Code :-

from queue import Priority Queue

```
v = 14.
graph = [ [] for i in range (v)]
# Function for implementing best first Search.
# Carries output path having lowest cost.
def best-first-Search (source, target, n):
    visited = [0] * n.
    visited = True.
    pq = PriorityQueue ()
    pq.put ((0, source)).
    while pq.empty () == False:
        u = pq.get () [1]
# Displaying path having lowest cost.
    print (v, end = " ")
    if u == target :
        break.
    for v, c in graph [u]:
        if visited [v] == False:
            visited [v] == True
            pq.put ((c, v)).
    print ()
# Function for adding edges to graph.
def addedge (x, y, cost).
    graph [x]. append ((y, cost))
    graph [y]. append ((x, cost))
# The nodes shown in above examples
(alphabets) are implemented using integers.
addedge (x, y, cost)
addedge (0, 1, 3)
addedge (0, 2, 6)
addedge (0, 3, 5)
addedge (1, 4, 9)
addedge (1, 5, 8)
addedge (2, 6, 12)
addedge (3, 8, 4)
addedge (8, 9, 5)
addedge (8, 9, 5)
addedge (8, 10, 6)
addedge (9, 11, 1)
addedge (9, 12, 10)
addedge (9, 13, 2)
```

Source = 0
target = 9
best - first - search (source, target, v)

Conclusion :-

We use best first Search as an evaluation function to decide which adjacent its most promising and then explore performance of the algorithm depends on how well the cost or evaluation function is designed.

Reference :-

Ben coppin Artifical Intelligence illuminated Jones and Bartlett 2004.

Termwork - 3

Title: Implementation of AND/OR/NOT Gate using single layer perceptron.

Theory:- To build up towards the (useful) multi-layer Neural Networks, we will start with considering the single layer Neural Network. This is called as Perception. In the field of Machine learning, the perception is a supervised learning Algorithm for binary classifiers. The perceptron Model implements the following function:

$$\hat{y} = \phi(w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b)$$
$$= \phi(x \cdot x + b)$$

where $\phi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{otherwise} \end{cases}$

Algorithm:- Basically there is no specific algorithm to follow, but the general working of perception can be stated in

1.) Initialize weights, bias, learning rate, and threshold values.
2.) Forward propagate.
3.) check the error.
4.) Back propagate and update weights and bias.
5.) Repeat for all training examples.

Examples.

1.) AND gate :

Inputs		Desired Output
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Let $w_1 = w_2 = 0.4$ (weights)

$a = 0.2$ (learning rate) and $t = 1$ (threshold)

* Row 1: $x_1 = 0$ $x_2 = 0$

$d = x_1 \cdot x_1 + x_2 \cdot w_2 = 0$ (calculated value)

$\Rightarrow d = y$.

Similarly Row 2 and Row 3 will follow the desired o/p.

* Row 4: $x_1 = 1$ $x_2 = 1$

$d = 1 * 0.4 + 1 * 0.4 = 0.8 < t$

ie $d = 0$ but $y = 1$

$\rightarrow$ updating of weights.

$\rightarrow w_1 = w_1 + (a \cdot x_1 \cdot t) = 0.4 + (0.2 * 1 * 1) = 0.6$.

$\rightarrow w_2 = w_2 + (a \cdot x_2 \cdot t) = 0.4 + (0.2 * 1 * 1) = 0.6$.

Now,

for $x_1 = 1$ & $x_2 = 1$

$d = 0.6 * 1 + 0.6 * 1 = 1.2 < t$

$\Rightarrow d = 1 = y$ and all other rows will follow.

So for $\boxed{w_1 = 0.6}$ and $\boxed{w_2 = 0.6}$

given perception builds / provides AND gate output.

2.) OR gate:

Inputs		Desired output
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

$\Rightarrow$ Let $w_1 = w_2 = 0.2$

$a = 0.2$ $t = 0$

* Row 1: $x_1 = 0$ $x_2 = 0$

$d = 0 * 0.2 + 0 * 0.2 = 0 = y$.

* Row 4: $x_1 = 1$ $x_2 = 1$

$d = 1 * 0.2 + 1 * 0.2 = 0.4 > t \Rightarrow d = 1 = y$.

So for $\boxed{w_1 = 0.2}$ and $\boxed{w_2 = 0.2}$

given perception gives OR gate outputs.

3.) NOT gate:

Input	Desired output
x	y
0	1
1	0

→ Let, $w = 1$, $b = -1$

$d = w \cdot x + b$.

* Row 1 : $x = 0$

$d = 1 * 0 + (-1) = -1 < 0 \Rightarrow d = 0$

Let $b = 1$, now, $d = 1 > 0 \Rightarrow d = y$.

* Row 2 : $x = 1$, $d = 1 * 1 + 1 = 2 \Rightarrow d = 1 \neq y$.

Let $w = -1$

$\therefore d = -1 * 1 + 1 = 0 \Rightarrow d = y$.

finally after learning, $w = -1$, $b = 1$

$d_1 = (-1) * (0) + 1 = 1 = y$.

$d_2 = (-1) * (1) + 1 = 0 = y$.

So far $w = -1$ & $b = 1$: perceptron follows NOT gate output

Source Code:

```
import numpy as np.
def unitstep(v):
    if v >= 0:
        return 1
    else
        return 0
def perceptronModel(x, w, b)
    v = np.dot(w, x) + b
    y = unitstep(v)
    return y.
def OR-LogicFunction(x)
    w = np.array([0.5, 0.5])
    b = -0.3
    return perceptronModel(x, w, b)
test1 = np.array([0, 0])
test2 = np.array([0, 1])
test3 = np.array([1, 0])
test4 = np.array([1, 1])
```

```
print ("OR {0, 0} = {}".format(0,0,OR-LogicFunction(test1)))
print ("OR {0, 1} = {}".format(0,1,OR-LogicFunction(test2)))
print ("OR {1, 1} = {}".format(1,0,OR-LogicFunction(test3)))
print ("OR {1, 1} = {}".format(1,1,OR-LogicFunction(test4))).
```

Conclusion :

Through this experiments, we learnt to implement three basic login gates using simple single layer perceptron

References.

Ben Coppin Artificial Intelligence illuminate Jones and Bartlett 2004.

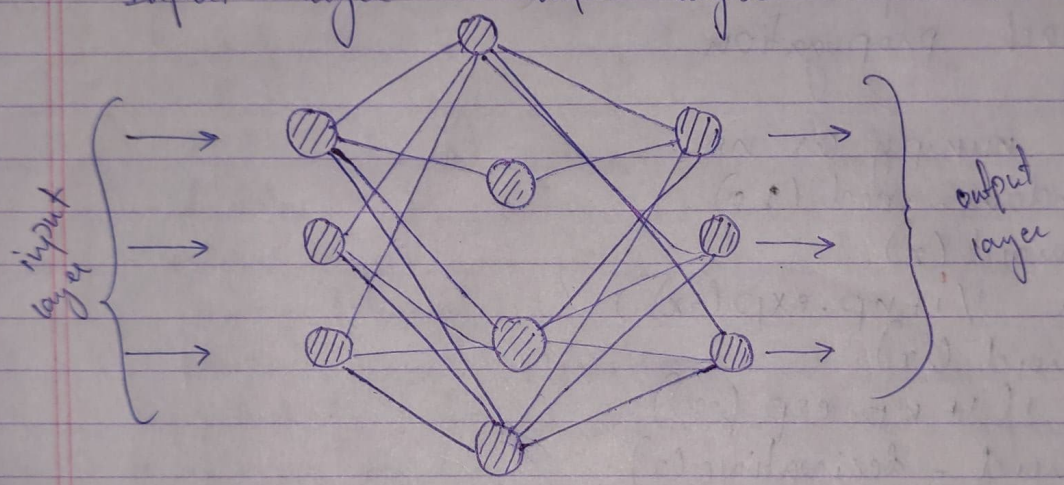
Teamwork 4 (a)

Title : Implement NOR gate using multi-layer perception/ error propogation.

Objective: To demonstrate the ability of multilayer present-ation (MLP) in solving linearly inseparable patteen declassification problems. To implement NOR using MLP/EBP.

Theory: Basically a multilayer perception (MLP) is a class of "feedforward" artificial neural network (ANN). It consists:

- Input layer
- Output layer
- Hidden layer

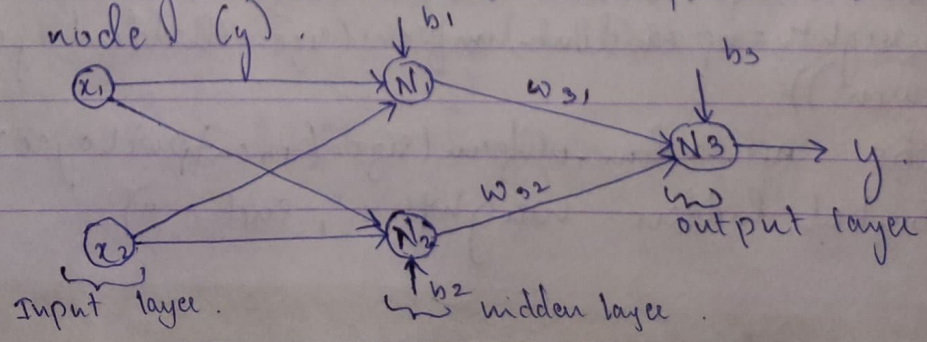


General diagram of MLP.

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

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

For above XOR implementation, the ANN will consists 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.

// Basic working logic

1.) Initialize the weight and biases immediately randomly.

2.) Iterate over data.

i.) Compute the predicted output using sigment 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 complete algorithm is divided into two parts.

- forward pass and
- back propogation.

Source Code :-

```
import numpy as np.
# np.random.seed(10).
def sigmoid(x)
    return 1/1+np.exp(-x))
def sigmoid(x)
    return 1(1+np.exp(-x))
def sigmoid-derivative(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
input Layer Neurons, hidden Layer Neurons, Output Layer Neurons = 2,2,1
# Random weights and bias initialization hidden-weights =
np.random.uniform(size = (input Layer Neurons, hidden Layer Neurons)
hidden-bias = np.random.uniform(size = (1, hidden Layer Neurons))
output weights = np.random.uniform(size = (hidden Layer Neurons, output
Layer Neurons))
output-bias = np.random.uniform(size=(1, output Layer Neuron))
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 algorithms
for - in range (epochs):
    # forward propogation -
    hidden-layer-activation = np.dot(inputs, hidden-weights)
    hidden-layer-activation += hidden-bias
    hidden-layer-output = sigmoid (hidden-layer-activation)
    output-layer-activation = np.dot(hidden-layer-output,
      output-weights)
    output-layer-activation += output-bias
    predicted-output = sigmoid(output-layer-activation)
  # Back propogation.
    error = expected-output predicted-output
    d-predicted-output = error * sigmoid-derative (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) * lr
    hidden-weights += inputs.T.dot(d-hidden-layer)
    hidden-bias += np.sum (d.hidden-layer, axis = 0, keepdims=
      true) * lr
print (" Final hidden weights :", end=")
```

```
print("Hidden- weight)
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)
```

Conclusion:-

Through this experiment, we learnt classification of linearly inseparable problems by taking examples of XOR & using MLP/EBP.

References:-

- AI illuminated by Ben Coppin
- Virtual labs
- towards datascience.com.

Teamwork - 5

Title:- Implement Hebbian rule and correlation rule.

Objection:- 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) Habbian 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 repeatedly active at the same time will tend to become associated so that the activity one facilitates activity in the other. According to Habbian rule,

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

ie the change in the synoptic weights of the ith neuron w_i is equal to the learning rate n times the ith output o_i times the input x.

2.) Correlation learning rule:- Its rule are based on a similar principle to the Habbian learning rule. It assumes that weights between simultaneously responding neurons should be largely positive and weights between neurons with opposite reaction should be largely negative. The correlation rule is supervised learning instead of an actual response o, the desired response D is used for the weight change calculation.

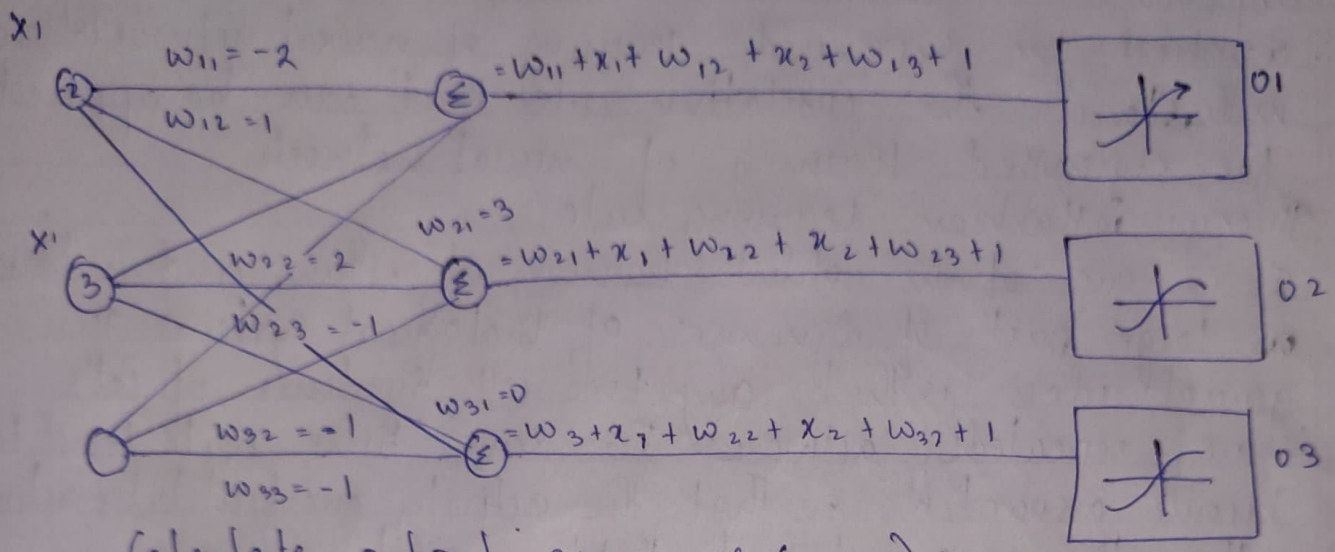
$$\Delta w_{ij} = n(o_i)x_j$$

where D_i is the desired value of the alphabet signal. This learning algorithm usually starts with the initialization of the weight to zeros. This simple rule states that if D_i is the desired response due to x, then corresponding weight increase is proportional to their product

Illustrations:-

Activation function used: $x_1 = 2$ $x_2 = 3$, bias fixed i p = 1.

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



Calculate output as $o = f(w \times x)$

$$o = f\left(\begin{bmatrix} -2 & 1 & -6.5 \\ 3 & 2 & 1 \\ 0 & -1 & -1.5 \end{bmatrix} + \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}\right)$$

$$= f\left(\begin{bmatrix} 0.5 \\ 1 \\ -4.5 \end{bmatrix}\right) = \begin{bmatrix} 0.2449 \\ 0.4621 \\ -0.978 \end{bmatrix}$$

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

Since $\Delta w_i = n(D - o_i)\, x$

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

Applying the formula.

$$\begin{bmatrix} -2.1958 & 1.2939 & -6.402 \\ 2.6303 & 2.5545 & 1.1848 \\ 0.7829 & -2.1736 & -1.8912 \end{bmatrix}$$

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

Let $x_1 = 2$ & $x_2 = 3$

bias = 1

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

Calculations:

$$O = f(w \times x)$$

$$= \text{ofc} \begin{bmatrix} -2 & 1 & -6.5 \\ 3 & 2 & 1 \\ 0 & -1 & -1.5 \end{bmatrix} \times \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

$$= \text{fc} \begin{bmatrix} 0.5 \\ 1 \\ -4.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$$

$$O = \begin{bmatrix} \phi \\ i \\ -1 \end{bmatrix} \quad D = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

Since $w_{i\,new} = w_{i\,old} + \eta (P_i) x$

$w_{i\,new} = [-2.8 \quad 2.2 \quad -6.1]$

$w_{2\,new} = [3.8 \quad 0.8 \quad 0.6]$

$w_{3\,new} = [0.8 \quad -2.2 \quad -1.9]$

$\therefore$ The new matrix will be

$$\begin{bmatrix} -2.8 & 2.2 & -6.1 \\ 3.8 & 6.8 & 0.6 \\ -0.8 & -2.2 & -1.5 \end{bmatrix}$$

Source Code:-

```
# xi = [ip1, ip2]
x1 = [1, 1]                x3 = [-1, 1]
x2 = [1, -1]               x4 = [-1, -1]
xlist = [x1, x2, x3, x4]
y = [1, -1, -1, -1]
w1 = w2 = w = 0 ,  b = 1
def heb learn ()
global w1, w2, bw
print (" dw1 \t dw2 \t db \t w1 \t w2 \t b")
i = 0
```

```
for xi in list:
    dw1 = xi[0] * y[i]
    dw2 = xi[1] * y[i]
    db = y[i]
  w1 = w1 + dw1
  w2 = w2 + dw2
  bw + = db.
print(dw1, dw2, w1, w2, bw, sep = '\t')
    it = 1
print("learning...")
heb.learning()
    print("learning completed")
print("output of AND gate using obtained w1, w2, bw:")
print("x1\tx2\ty")
  for xi in xilist:
    print(xi[0], xi[1], 1 if w1 * xi[0] + w2 + xi[1] + b
        * bw > 0
    else -1, sep = '\t')
```

Conclusion :-

Through this experiment, we implemented Hebbian rule and correlation rule.

Reference :-

- AI Illuminated by Ben Coppin
- Virtual Labs
- towards data science.com.

Teamwork - 1 (Part B)

Title:- Implement Find-s and candidate elimination algorithms.

Objectives :- i) To understand the Find-s algorithm.

ii) To understand the concept of candidate elimination algorithms

iii) To write the python program to implement both Find-s and candidate elimination algorithm.

Theory:- i) Candidate Elimination ii) Find-s

i) Candidate Elimination:- This will operates in a similar manner to the simple algorithm. Candidate elimination methode 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 hypotheses that starts out as set of with the single item <???, ???> and ends up being a set of the most general hypotheses that match call the training data. This algorithm is than able to make use of negative training data as well positive training data.

This method operates as:- Two sets are maintained of hypothese hs and hg: hs is initialized as {<ϕ, ϕ, ϕ, ϕ, ϕ, ϕ>} and hg is initialized as {< ?, ?, ?, ?, ?, ? >}.

ii) Find-s algorithm It finds the most specific hypothesis that fits all the positive egs. we have to note here that the algorithm considers only those positive training egs. It starts with the most specific hypothesis and 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 most general hypothesis.

1.) ? indicate that any value is acceptable for the attribute

2.) Specify a single required value for the attribute

3.) ϕ indicates that no value is acceptable

4.) The most general hypothesis is represented by {???, ???}

5.) The most specific hypothesis is represented by {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}

Find-S algorithm.

Step 1:- Load data set.

Step 2:- Initialize h to the most specific hypothesis in h.

Step 3: For each positive training intance x.

- For each attribute constraint a in h:

  If the constraint a is satisfied by x then do nothing

  else replace a in h by the next more general constrain

Step 4: Output hypothesis h.

Source Code:-

```
import pandas as pd
import numpy as np
# to read the data in csv files
data = pd.read_csv("data.csv")
print(data, "h")
# making an array of all attributes
d= np.array(data)[:,:-1]
print("n The attributes are :",d)
# Segregating the target that has +ve & -ve examples
target = np.array(data)[:,-1]
print("n The target is : ", target)
# training function
def train(c, t):
for i, val in enumerate(t):
    if val == "yes":
        specific_hypothesis = c[i].copy()
        break

for i, val in enumerate(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
```

```
#obtaining the final hypothesis.
print("\n the final hypothesis is :", train(d, target))
```

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.

References:

i) Ben coppin, Artificial intelligence Illuminated Jones and bartel 2004.

ii) www.geeksforgeeks.com.

Teamwork - 02 (Part B)

Title :- Build a linear regression model using housing prices.

Objectives :- 1.) To understand the concept of linear regression.
2.) To build program to implement linear regression for the housing prices model.

Theory :- Linear regression model is a statistical model that estimates the relationship between one dependant variable and one or more independent variables using a line.

* It can be used to predict the continous numerical value of dependant variable for given value of independent variable
* Linear regression may be simple or multiple regression. The simple linear regression has only one independent variable where as multiple regression has 2 or more independent variables.
* Relationship b/w two variables x & y is given by.

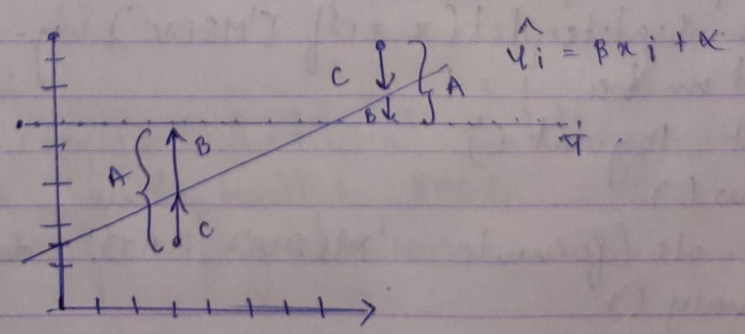
$$cov(x, y) = \frac{\sum_{i=1}^{n} (x - \bar{x})(y_i - \bar{y})}{n-1}$$

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

Prediction :- If you something about x, this knowledge helps you about predict something about y.

* Regression equation :-

$$E(Y_i | x_i) = \alpha + \beta x_i$$



$$\underbrace{\sum_{i=1}^{n} (y_i - \bar{y})^2}_{A^2_{SS}} = \underbrace{\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2}_{B^2_{SS}} + \underbrace{\sum_{i=1}^{n} (\hat{y} - y_i)}_{C^2_{SS}}$$

Source Code: -

```
import pandas as pd.
import numpy as np.
import matplotlib.pyplot as plt.
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split.
from sklearn.preprocessing import standard scalar
from sklearn.linear model import linearRegression
import seaborn as sns
import statmodels.stats.outliers_influence import variance_
        inflation_factor.
warnings.filterwarnings("ignore").
plt.style.use("ggplot")
data = load_boston()
# adding target value to the data.
df['MEDV'] = data.target
print('Details of the data base\n').
df.head()
df.describe()
df.info()
# ~~tt~~ lets see how data is distributed for every column using da
plt.figure(figsize = (20, 15))
plot number = 1
for column in df:
    if plot number <= 14:
        ax = plt.subplot(3,5, plotnumber)
        sns.scatterplot(x = df['MEDV'], y = df[column])
        plot number += 1
plt.tight_layout()
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 - linear Regression
lr.fit (x_train , y_train)
y_pred = lr.predict (x_test)
print ("Training accuracy of the model = ", lr.score (x_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]
    adjust_r2 = 1.(1. r2) * (n-1) / (n+p+1)
    return adjusted_r2
print ("After R-squared new value of accuracy \n")
print (" Training Accuracy of the model = ', adj_r2(x_train, y_train
print (" Testing Accuracy of the model = 'adj_r2 (x_test, y_test, lr)
```

Conclusion:-

At the end of this teamwork we are able to write the program for illustrate the concept of linear Regression. We understood the concept of linear Regression.

References:

1.) Ben coppin, Artificial Intelligence, Illuminated Jones and bartlet 2004.

2.) www.geeksforgeeks.com.

Teamwork 3 (A)

Title: Implement spam detection using 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 no of possible values.

$$P(c_i / d_1 \ldots d_n)$$

* 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 trying to find the highest probability, & because $P(d_1 \ldots d_n)$ is a constant independent of c_i we can eliminate it & simply aim to find the classification c_i, for which is maximized

$$P(d_1, \ldots d_n | c_i) \cdot P(c_i)$$

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

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

* Application.

1) Real time prediction.
   → naive Bayes is an eager learning classifier and it is sure fast. Thus it could be used for making prediction.

2) Multi class prediction.
   → well known for multi class prediction feature.

3) Text classification / spam filtering / sentiment Analysis
   → Mostly used in text classification
   → Have higher success rate as compared to other algorithm.

Source Code.

```
import ~~all the~~ 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.
from sklearn.model-selection.
from sklearn.naive_bayes
from sklearn.metrics import accuracy-score, precision-score, recall
df-sms = pd.read-csv('spam.csv', encoding='latin-1')
df-sms = df-sms.drop(['unamed:2', "unamed:3", "unamed:4"], axis=1)
df-sms = df-sms.rename(columns={"v1": label", "v2": "sms"})
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!."]
=
naive-bayes-MultinomialNB()
naive-bayes.fit(training-data, y-train)
prediction= naivebayes.predict(testing-data)
print("Accuracy score: {}", format(accuracy.score(y-test, prediction)))
print('precision score: {}', format(precision-score(y-test, prediction)))
print('Recall score: {}', format(recall-score(y-test, prediction)))
print('F1 score: {}', format(F1-score(y-test, prediction)))
```

Conclusion: At the end of this teamwork we understood the concept ~~of frequent~~ pattern ~~the~~ and implemented that successfully.

References:-

www.geeksforgeeks.com.

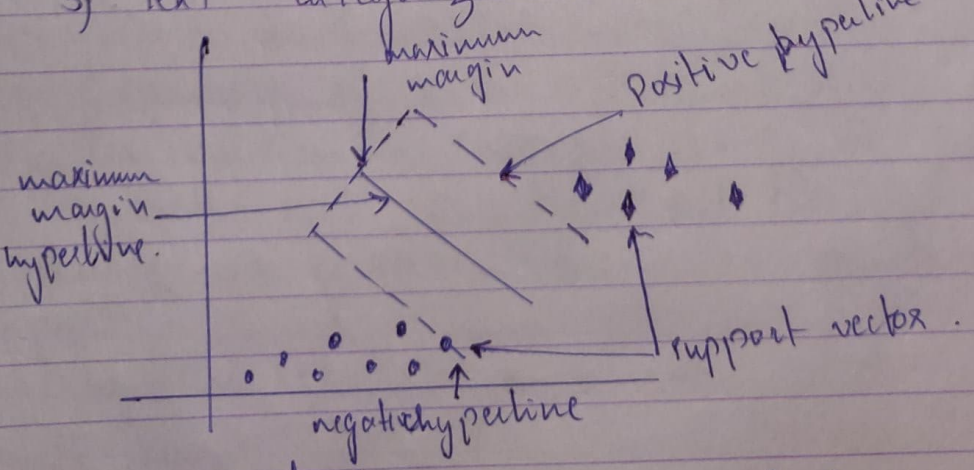
Teamwork - 4 (B)

Title:- Implement hand writing classification using support vector machine (SVM)

Objective:- To understand the concept of support vector machine To understand the concept of classification and as well as regression by making use SVM.

Theory:- Support Vector Machine (SVM) is one of the most popular supervised learning algorithms which is used for classification as well as regression. The goal of SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in future. This best decision boundary is called a hyperplane SVM chooses the extreme points / vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence Algorithm is called SVM. SVM algorithm can be used for

1) Face detection.
2) Image classification.
3) Text categorization.



Consider the ~~beside~~ above diagram in which there are 2 different categories that are classified using a decision boundary or hyperplane

Types of SVM.

1.) Linear SVM: It is used for linearly seperable 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, & classifiers is used called as linear SVM classifier.

2.) Non linear SVM: Non-linear SVM 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 teamwork we are able to understand the concept of support Vector Machine Algorithm and we implemented that algorithm using python program.

References:

www.javapoint.com.

Teamwork 5 (B).

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

Objective:- To understand the concept of Frequent pattern growth.

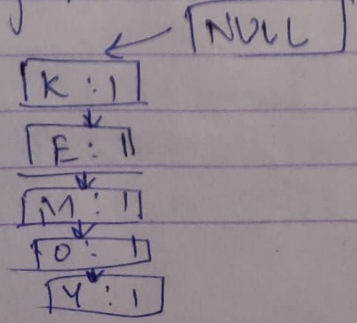
Theory:- FP growth algorithm is an improvement of apriori algorithm. It is used to find the frequent interest in a transaction database without candidate generation FP growth represents frequent items in FP-tree.

* Pre-requisities for FP-tree are Apriori Algo and trie-data structure.
* Consider the following table

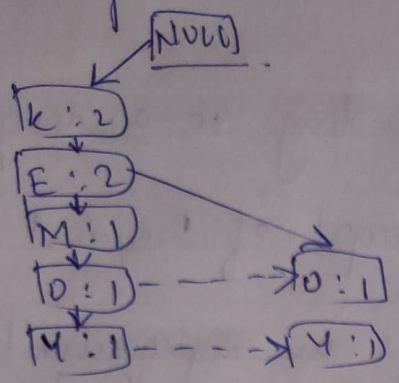
Transaction ID	Item
T1	{E, K M N O Y}
T2	{D E K N O Y}
T3	{A E K M}
T4	{C. K, M O Y}
T5	{C E I K O O}

The above-given data is a hypothetical dataset of transactions with each letter representing an item. The frequency of each individual item is computed. Let the 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

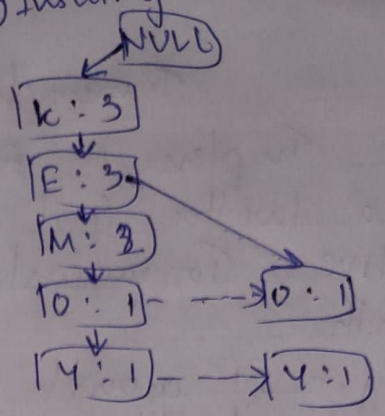
1.) Inserting the set <K, K, O, Y, M>



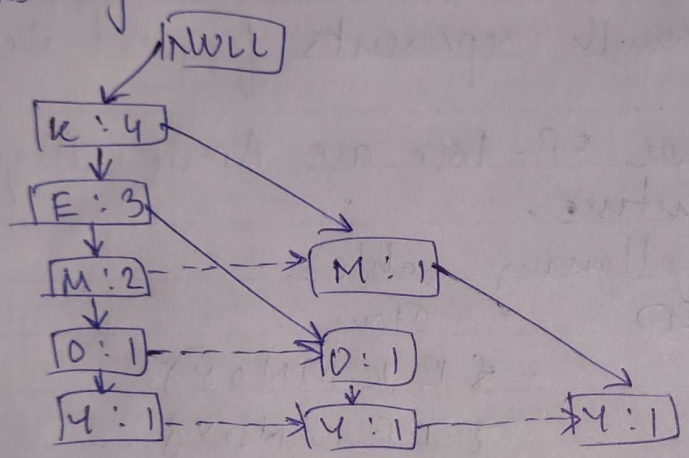
2.) Inserting the set {K, E, O, Y}



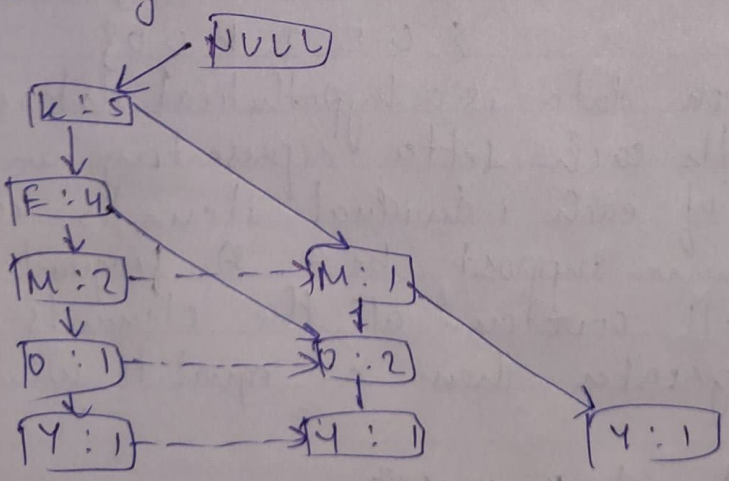
3.) Inserting the set {K E M}.



4.) Inserting the set {K, M, Y}.



5.) Inserting {K, E, O}.



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

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}	{E, M, Y}
T5	{C, E, I, K, O, O}	{K, E, O}.

Items	Conditional pattern Base
Y	{{KEMO:1} {KEO:1} {K,M:1}}
O	{{KEM:1} {KE:2}}
M	{{K,E:2} {K:1}}
E	{K:4}.
K	

Items	Conditional pattern Base	Conditional FP-Tree.
Y	{{KEMO:1}{KEO:1} {KM:1}}	{K:3}
O	{{K,E,M:1} {KE:2}}	{KE:3}

M	{{k, E : 2} {k : 1}}	{k : 3}
E	{k : 4}	{k : 4}
K.		

Items	Frequent Pattern Generated.
Y	{<k, Y : 3}}
O	{<k, O : 3}, <E, O : 3>, <E, k, O : 3>}
M	{<k, M : 3>}
E	{<E, k : 3>}
K.	

Conclusion : At the end of this teamwork we understood the concept of frequent pattern tree and we implemented that successfully.

References :

www.geeksforgeeks.com.