



MALAD KANDIVALI EDUCATION SOCIETY'S

**NAGINDAS KHANDWALA COLLEGE OF COMMERCE, ARTS &
MANAGEMENT STUDIES & SHANTABEN NAGINDAS KHANDWALA COLLEGE
OF SCIENCE**

MALAD [W], MUMBAI – 64

AUTONOMOUS INSTITUTION

(Affiliated To University Of Mumbai)

Reaccredited 'A' Grade by NAAC | ISO 9001:2015 Certified

CERTIFICATE

Name: Ms. Zeenat Hafeez Fazale Rab

Roll No: 578

Programme: BSc IT

Semester: V

This is certified to be a bonafide record of practical works done by the above student in the college laboratory for the course **PRINCIPLES OF ARTIFICIAL INTELLIGENCE PRACTICAL (Course Code: 2151UITPA)** for the partial fulfilment of Fifth Semester of BSc IT during the academic year 2021-22.

The journal work is the original study work that has been duly approved in the year 2021- 22 by the undersigned.

External Examiner

Mrs. Elizabeth Leah George
(Subject-In-Charge)

Date of Examination:

(College Stamp)

Name: Zeenat

Class: TYIT

Roll no: 578

Name: Zeenat

Class: T.Y. B.Sc. IT Sem- V

Roll No: 578

Subject: PRINCIPLES OF ARTIFICIAL INTELLIGENCE PRACTICAL

[Course Code: 2151UITPA]

INDEX

Sr No	Name	Date	Sign
1	Implement Breadth First Search and Depth First Search algorithms.	30/06/2021	
2	To implement Hill Climbing Algorithm in Travelling Salesman Problem (TSP)	07/07/2021	
3a	To implement Constraint Satisfaction Problem. (CSP).	14/07/2021	
3b	To implement Minimax algorithm	22/07/2021	
4a	To implement the working of decision tree based ID3 algorithm.	28/07/2021	
4b	To implement the working of Naive Bayes Algorithm	04/08/2021	
5	To implement the Bayesian Classifier using the Medical data.	11/08/2021	
6	To apply Markov Property to generate Donald's Trump's speech by considering each word used in the speech and for each word, create a dictionary of words that are used next.	14/08/2021	
7	Prediction Algorithm - Use of different packages on dataset of Cat and Non-Cat images	18/08/2021	
8	Neural Representation of AND and OR Logic Gates Perceptron.	23/08/2021	
9	Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set.	25/08/2021	
10	Implementation of basic neural network model with 4 activation functions on Pima Indians onset of diabetes dataset.	27/08/2021	

Practical 1

Aim: Implement Breadth first search and Depth first search

Theory:

Breadth First Search(BFS)

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

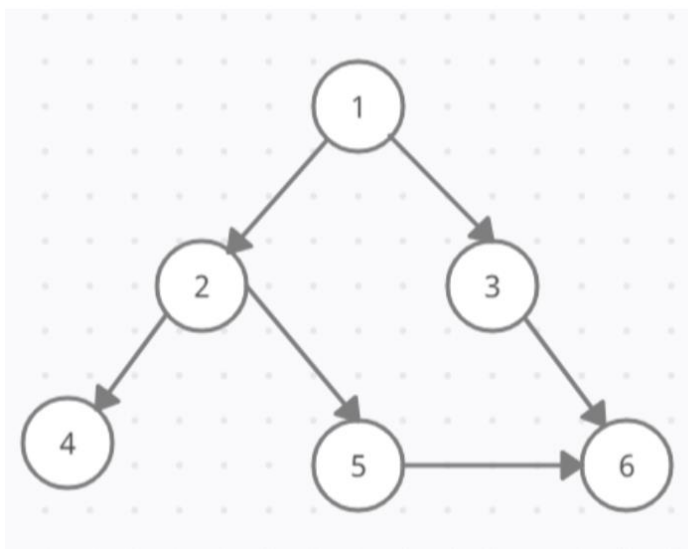
implemented Queue (FIFO)

Give always optimal solution

provide shallowest path

no backtracking required

Graph:



Code:

```
[ ] # BFS
graph = {
    '1' : ['2', '3'],
    '2' : ['4', '5'],
    '3' : ['6'],
    '4' : [],
    '5' : ['6'],
    '6' : []
}

visited = []
queue = []

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)

    while queue:
        s = queue.pop(0)
        print (s, end = " ")

        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)
```

```
bfs(visited, graph, '1')
```

Output:

```
1 2 3 4 5 6
```

Google Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=Ui7dmws2OF7R>

Theory:**Depth First Search (DFS)**

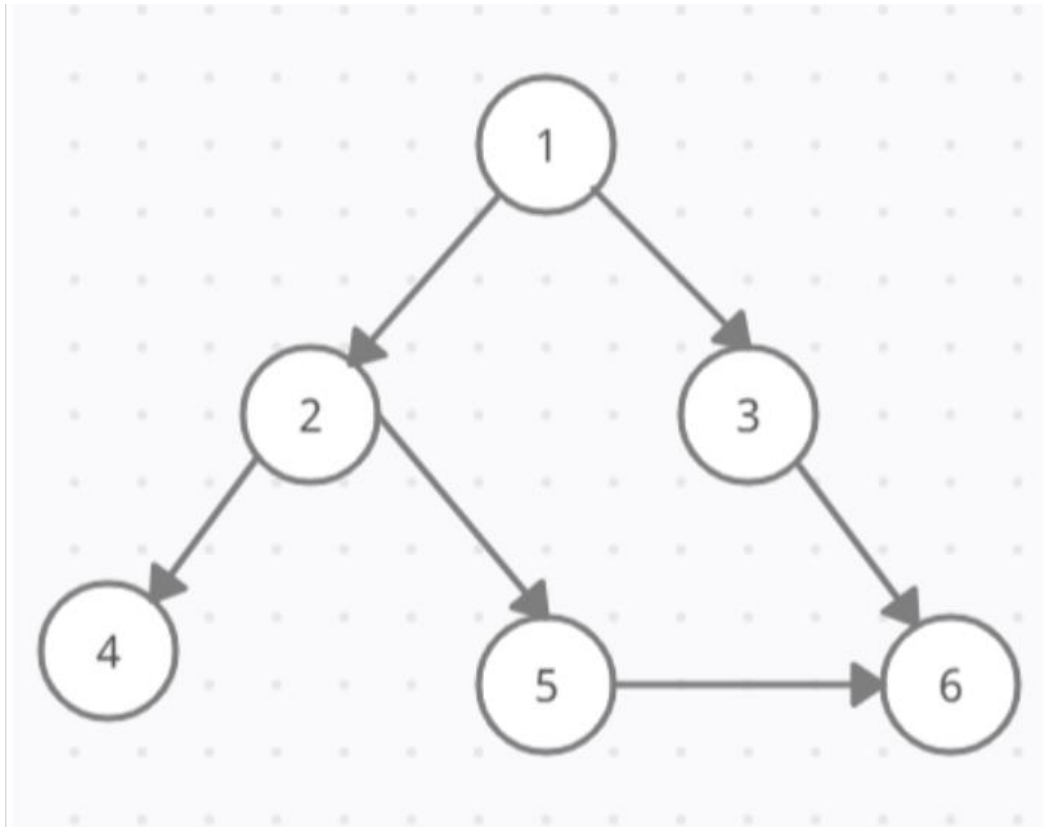
The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. This recursive nature of DFS can be implemented using stacks. Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

implemented using Stack(LIFO)

provide Deepest Node

some time not give complete ans

backtracking required.

Graph:**Code:**

```
[ ] # DFS
graph = {
    '1' : ['2', '3'],
    '2' : ['4', '5'],
    '3' : ['6'],
    '4' : [],
    '5' : ['6'],
    '6' : []
}

visited = set()

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

dfs(visited, graph, '1')
```

Output:

```
☞ 1
   2
   4
   5
   6
   3
```

Colab Link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=3FY2sfcuGbl->

Practical 2

Aim: To implement Hill Climbing Algorithm in Travelling Salesman Problem (TSP)

Theory:

Hill Climbing is a heuristic search used for mathematical optimisation problems in the field of Artificial Intelligence. It is one such Algorithm is one that will find the best possible solution to your problem in the most reasonable period of time.

Given a large set of inputs and a good heuristic function, the algorithm tries to find the best possible solution to the problem in the most reasonable time period. This solution may not be the absolute best but it is sufficiently good considering the time allotted.

The definition above implies that hill-climbing solves the problems where we need to maximise or minimise a given real function by selecting values from the given inputs. example of this is the Travelling Salesman Problem where we need to minimise the distance travelled by the salesman.

Code:

```
class Node:
    """A simple node """
    def __declare_instance_variables(this) -> None:
        this.parent: Node = None
        this.root: Node = None
        this.__children: list = []
    def __init__(this, child = None, children: list = None, value: float = None, tag: str = None):
        """child: Node, children: List[Node]"""
        this.__declare_instance_variables()
        this.tag = tag
        this.value = value
        if (child != None):
            this.add(child)
        if (children != None):
            this.add_children(children)

    def get_neighbors(this) -> list:
        """Returns the neighbor nodes"""
        if this.parent == None:
            return [this]
        children = this.parent.get_children()
        if children == None:
            return []
```

```
[ ]    return children

def get_first(this):
    """Returns the first children of this node"""
    if (this.is_empty): return None
    return this.__children[0]

def is_root(this) -> bool:
    return this.parent == None

def is_leaf(this) -> bool:
    if (this.__children == None): return True
    return this.is_empty()

def is_inner(this) -> bool:
    return not (this.is_leaf() or this.is_root())

def get_children(this) -> list:
    return this.__children

def get_root(this):
    """Returns -> Node"""
    if (this.is_root()):
        return this
    else:
```



```
[ ]         return this.parent.root

def get_height(this) -> int:
    if (this.is_empty()):
        return 0
    maxHeight: int = 0
    children: list = this.get_children()
    for element in children:
        height: int = element.get_height()
        if (height > maxHeight):
            maxHeight = height
    return maxHeight + 1

def get_depth(this) -> int:
    if (this.is_root()):
        return 0
    return this.parent.get_depth() + 1

def is_empty(this) -> bool:
    return len(this.__children) == 0

def is_not_empty(this) -> bool:
    return not this.is_empty()

def add(this, child) -> None:
```

```
[ ] """child: Node"""
    assert child != None
    if (this.__children == None):
        this.__children = []
    child.parent = this
    child.root = this.get_root()
    this.__children.append(child)

def add_children(this, children: list) -> None:
    assert children != None
    if (len(children) == 0):
        return
    if (this.__children == None):
        this.__children = []
    for element in children:
        element.parent = this
        element.root = this.get_root()
        this.__children.append(element)

def __len__(this) -> int:
    if (len(this.__children) != 0 and this.__children != None):
        maxLength: int = 1
        for child in this.__children:
            maxLength += len(child)
        return maxLength
```

```
[ ]     else:
        return 1
    def __str__(this) -> str:
        return f"Node({this.value})"
```

```
[ ] def node_to_string(node: Node, islast=False):
    pretab = '' if node.get_depth() == 0 else ' ' * (node.get_depth())
    prefix = f'{pretab}:{node.get_depth()} ——'
    value = node.value
    depthTab: str = ' ' * (node.get_depth() + 1)
    children_str = ''
    for child in node.get_children():
        ischildlast = node.get_children()[-1] == child
        children_str += f'{depthTab}{node_to_string(child, ischildlast)}'
    return (
        f'{prefix} {node.tag} = {value}\n'
        f'{children_str}'
    )
```

```
[ ] def evaluate(node: Node):
    """returns the value of the node"""
    if(isinstance(node.value, float) or isinstance(node.value, int)):
        assert node.value != None, "Node must have a value"
        return node.value
```

```
[ ] elif(isinstance(node.value, str)):  
    raise NotImplementedError
```

Simple hill climbing algorithm

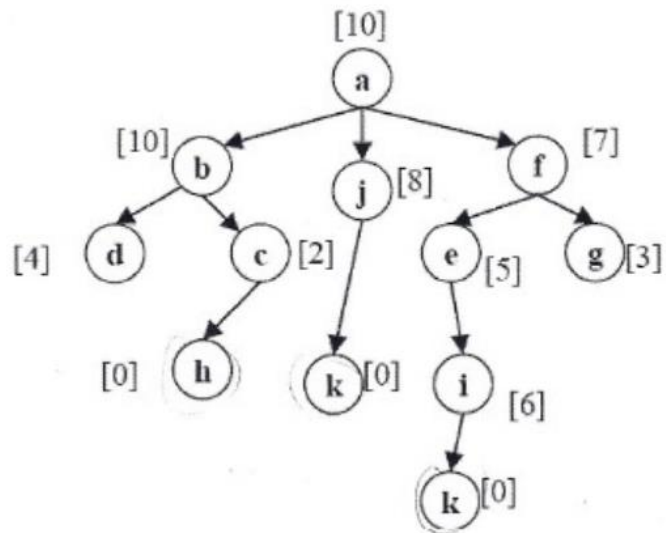
```
[ ] from math import inf  
def hill_climbing(start_node) -> Node:  
    """  
    Pseudo-code for the algorithm  
    ...  
    algorithm hill Climbing is  
        currentNode := startNode  
        loop do  
            L := NEIGHBORS(currentNode)  
            nextEval := -INF  
            nextNode := NULL  
            for all x in L do  
                if EVAL(x) > nextEval then  
                    nextNode := x  
                    nextEval := EVAL(x)  
            if nextEval ≤ EVAL(currentNode) then  
                // Return current node since no better neighbors exist  
                return currentNode
```

```
current_node = start_node
best_value = -inf
best_node = None
while True:
    current_value = evaluate(current_node)
    if current_value > best_value:
        best_node = current_node
        best_value = current_value
    else:
        # this node has a value smaller than the best node.
        # stopping search with local maxima
        return best_node
childrens = current_node.get_children()
for child in childrens:
    child_value = evaluate(child)
    if child_value > best_value:
        best_value = child_value
        best_node = child
    else:
        return best_node
# every neighbour of this child is traversed.
# Setting current_node as the last child traversed
```



```
current_node = child
```

Graph



```
[ ] # implemented the above tree of nodes
tree1: Node = Node(
    value=10, tag='a',
    children = f
```

```
[ ]    children=[
        Node(
            value=10, tag='b',
            children=[
                Node(value=4, tag='d'),
                Node(value=2, tag='c',
                    child=Node(value=0, tag='h'))
            ],
        ),
        Node(value=8, tag='j',
            child=Node(value=0, tag='k'))
    ),
    Node(value=7, tag='f',
        children=[
            Node(value=5, tag='e',
                child=Node(
                    value=6, tag='i',
                    child=Node(value=0, tag='k'))
            ),
            Node(value=3, tag='g')
        ]
    )
]
```

```

▶ # implemented an another tree of nodes
tree2: Node = Node(
    value=2, tag='a',
    children=[
        Node(
            value=4, tag='b',
            children=[
                Node(value=5, tag='d'),
                Node(value=6, tag='c',
                    child=Node(value=8, tag='h'))
            ],
        ),
        Node(value=9, tag='j',
            child=Node(value=0, tag='k'))
    ),
    Node(value=7, tag='f',
        children=[
            Node(value=12, tag='e',
                child=Node(
                    value=6, tag='i',
                    child=Node(value=0, tag='k'))
            )
        ],
    ),
),
Node(value=3, tag='g')
]
)
]
)

```

```

[ ] print('Tree - 1: representation')
    print('pattern -> :<depth> — <value>', end='\n\n')
    print(node_to_string(tree1))

```

Output:

Tree - 1: representation
pattern -> :<depth> — <value>

```
:0 — a = 10
  :1 — b = 10
    :2 — d = 4
    :2 — c = 2
      :3 — h = 0
  :1 — j = 8
    :2 — k = 0
:1 — f = 7
  :2 — e = 5
    :3 — i = 6
      :4 — k = 0
  :2 — g = 3
```

```
[ ] print('Tree - 2: representation')
    print('pattern -> :<depth> — <value>', end='\n\n')
    print(node_to_string(tree2))
```

Output:

Tree - 2: representation
pattern -> :<depth> — <value>

```
:0 — a = 2
  :1 — b = 4
    :2 — d = 5
    :2 — c = 6
      :3 — h = 8
  :1 — j = 9
    :2 — k = 0
:1 — f = 7
  :2 — e = 12
    :3 — i = 6
      :4 — k = 0
  :2 — g = 3
```

```
[ ] print('For Tree - 1')
    best_solution = hill_climbing(tree1)
    print(f"Best solution is {best_solution.value} with tag {best_solution.tag}")
```

Output:

```
For Tree - 1
Best solution is 10 with tag a
```

```
[ ] print('For Tree - 2')
    best_solution = hill_climbing(tree2)
    print(f"Best solution is {best_solution.value} with tag {best_solution.tag}")
```

```
For Tree - 2
Best solution is 9 with tag j
```

Colab Link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=OArpTnZZvRJE&line=1&uniqifier=1>

Practical 3

A) Aim: To implement Constraint Satisfaction Problem. (CSP).

Theory:

The constraint is the process to find the solution to the constraint. It is part of Artificial Intelligence. It defines that a set of objects whose state must satisfy a number of constraints. It is also called the Constraint Satisfaction Problem. The objectives are to assign a value for each variable such that all constraints are satisfied. Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem. Every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking.

Constraint Satisfaction Problem

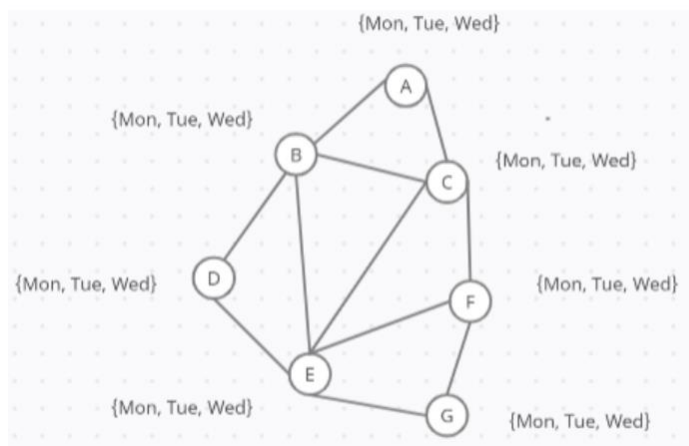
- 1 Set of variables $\{X_1, X_2, \dots, X_n\}$
- 2 Set of domains for each variable $\{D_1, D_2, \dots, D_n\}$
- 3 Set of constraints C

CSPs as Search Problems:

- initial state: empty assignment (no variables)
- actions: add a $\{\text{variable} = \text{value}\}$ to assignment
- transition model: shows how adding an assignment changes the assignment
- goal test: check if all variables assigned and constraints all satisfied
- path cost function: all paths have same cost

In this problem, we are implementing to constraint satisfaction rule and backtracking rule to solve this problem.

Graph:



Code:

"""

Naive backtracking search without any heuristics or inference.

"""

VARIABLES = ["A", "B", "C", "D", "E", "F", "G"]

CONSTRAINTS = [

("A", "B"),

("A", "C"),

("B", "C"),

("B", "D"),

("B", "E"),

("C", "E"),

("C", "F"),

("D", "E"),

("E", "F"),

("E", "G"),

("F", "G")

]

def backtrack(assignment):

"""Runs backtracking search to find an assignment."""

Check if assignment is complete

if len(assignment) == len(VARIABLES):

return assignment

Try a new variable

var = select_unassigned_variable(assignment)

for value in ["Monday", "Tuesday", "Wednesday"]:

new_assignment = assignment.copy()

new_assignment[var] = value

if consistent(new_assignment):

result = backtrack(new_assignment)

if result is not None:

return result

return None

def select_unassigned_variable(assignment):

"""Chooses a variable not yet assigned, in order."""

for variable in VARIABLES:

if variable not in assignment:

return variable

return None

def consistent(assignment):

```
"""Checks to see if an assignment is consistent."""  
for (x, y) in CONSTRAINTS:  
  
    # Only consider arcs where both are assigned  
    if x not in assignment or y not in assignment:  
        continue  
  
    # If both have same value, then not consistent  
    if assignment[x] == assignment[y]:  
        return False  
  
    # If nothing inconsistent, then assignment is consistent  
    return True
```

```
solution = backtrack(dict())  
print(solution)
```

Output:

```
{'A': 'Monday', 'B': 'Tuesday', 'C': 'Wednesday', 'D': 'Wednesday', 'E': 'Monday', 'F': 'Tuesday',  
'G': 'Wednesday'}
```

Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=77sRqSE6cBMR>

3B Aim: To implement Minimax algorithm**Theory:**

Alpha beta pruning is an optimisation technique for the minimax algorithm. Alpha-beta pruning is nothing but the pruning of useless branches in decision trees. This alpha-beta pruning algorithm was discovered independently by researchers in the 1900s. The need for pruning came from the fact that in some cases decision trees become very complex. In that tree, some useless branches increase the complexity of the model. So, to avoid this, Alpha-Beta pruning comes to play so that the computer does not have to look at the entire tree.

Minimax is a classic depth-first search technique for a sequential two-player game. The two players are called MAX and MIN. The minimax algorithm is designed for finding the optimal move for MAX, the player at the root node. The search tree is created by recursively expanding all nodes from the root in a depth-first manner until either the end of the game or the maximum search depth is reached.

Alpha: Alpha is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is $-\infty$.

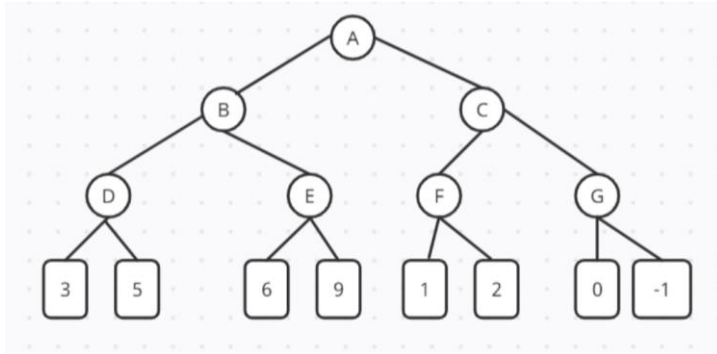
Beta: Beta is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is $+\infty$.

The condition for Alpha-beta Pruning is that $\alpha \geq \beta$.

MAX will update only alpha values and MIN player will update only beta values

Alpha and Beta values only be passed to child nodes.

Graph:



Code:

working of Alpha-Beta Pruning

Initial values of Alpha and Beta

MAX, MIN = 1000, -1000

```
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):
```

```
    if depth == 3:
        return values[nodeIndex]
```

```
    if maximizingPlayer:
```

```
        best = MIN
```

```
        # Recur for left and right children
```

```
        for i in range(0, 2):
```

```
            val = minimax(depth + 1, nodeIndex * 2 + i,
                           False, values, alpha, beta)
```

```
            best = max(best, val)
```

```
            alpha = max(alpha, best)
```

```
        # Alpha Beta Pruning
```

```
        if beta <= alpha:
```

```
            break
```

```
    return best
```

```
else:
    best = MAX

    for i in range(0, 2):

        val = minimax(depth + 1, nodeIndex * 2 + i,
                       True, values, alpha, beta)
        best = min(best, val)
        beta = min(beta, best)

    # Alpha Beta Pruning
    if beta <= alpha:
        break

    return best

if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:

The optimal value is : 5

Colab Link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=UE1ecnoc8Xt6&line=1&uniqifier=1>

Practical 4

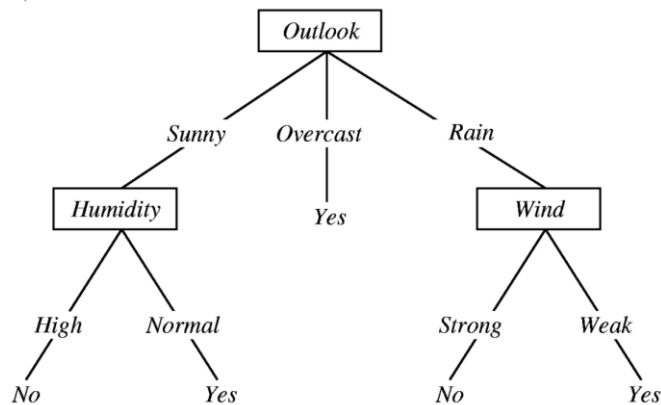
4A) Aim: To implement the working of decision tree based ID3 algorithm.

Theory:

Decision Tree

Decision trees are assigned to the information based learning algorithms which use different measures of information gain for learning it is the most powerful and popular tool for classification and prediction. We can use decision trees for issues where we have continuous but also categorical input and target features. Decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.

Graph:



Code:

```
# code for decision tree

import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

data = load_iris()
print('Classes to predict: ', data.target_names)

X = data.data

y = data.target

print('Number of examples in the data:', X.shape[0])
```



```
X[:4]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 47, test_size = 0.25)
```

```
from sklearn.tree import DecisionTreeClassifier  
clf = DecisionTreeClassifier(criterion = 'entropy')
```

```
clf.fit(X_train, y_train)
```

```
y_pred = clf.predict(X_test)
```

```
from sklearn.metrics import accuracy_score  
print('Accuracy Score on train data: ', accuracy_score(y_true=y_train, y_pred=clf.predict(X_train)))  
print('Accuracy Score on test data: ', accuracy_score(y_true=y_test, y_pred=y_pred))
```

Output:

```
:
```

```
Classes to predict: ['setosa' 'versicolor' 'virginica']  
Number of examples in the data: 150  
Accuracy Score on train data: 1.0  
Accuracy Score on test data: 0.9473684210526315
```

Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=Xbb2SZUcC3vH>

4B) Aim: To implement the working of Naive Bayes Algorithm

Theory:

Naive Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems. It is mainly used in text classification that includes a high-dimensional training dataset. Naive bayes does quite well when the training data doesn't contain all possibilities so it can be very good with low amounts of data. Decision trees work better with lots of data compared to Naive Bayes. Naive Bayes is used a lot in robotics and computer vision, and does quite well with those tasks.

Code:

```
from sklearn.datasets import load_breast_cancer
```

```
data_loaded = load_breast_cancer()
```

```
X = data_loaded.data
```

```
y = data_loaded.target
```

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(data_loaded.data, data_loaded.target, test_size=0.2, random_state=20)
```

```
from sklearn.naive_bayes import GaussianNB
```

```
naive_bayes = GaussianNB()
```

```
naive_bayes.fit(X_train, y_train)
```

```
y_predicted = naive_bayes.predict(X_test)
```

```
from sklearn import metrics
```

```
metrics.accuracy_score(y_predicted, y_test)
```

Output:

0.956140350877193

Colab link: https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=xP5JkxC0_bxO&line=1&uniquifier=1

Practical 5

Aim: To implement the Bayesian Classifier using the Medical data.

Theory:

Bayesian network theory can be thought of as a fusion of incidence diagrams and Bayes' theorem. Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph.

It is also called a Bayes network, belief network, decision network, or Bayesian model. The probability of an event occurring given that another event has already occurred is called a conditional probability.

Dataset:

1	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
2	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
3	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
4	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
5	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
6	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
7	57	1	0	140	192	0	1	148	0	0.4	1	0	1	1
8	56	0	1	140	294	0	0	153	0	1.3	1	0	2	1
9	44	1	1	120	263	0	1	173	0	0	2	0	3	1
10	52	1	2	172	199	1	1	162	0	0.5	2	0	3	1
11	57	1	2	150	168	0	1	174	0	1.6	2	0	2	1
12	54	1	0	140	239	0	1	160	0	1.2	2	0	2	1
13	48	0	2	130	275	0	1	139	0	0.2	2	0	2	1
14	49	1	1	130	266	0	1	171	0	0.6	2	0	2	1
15	64	1	3	110	211	0	0	144	1	1.8	1	0	2	1
16	58	0	3	150	283	1	0	162	0	1	2	0	2	1
17	50	0	2	120	219	0	1	158	0	1.6	1	0	2	1
18	58	0	2	120	340	0	1	172	0	0	2	0	2	1
19	66	0	3	150	226	0	1	114	0	2.6	0	0	2	1
20	43	1	0	150	247	0	1	171	0	1.5	2	0	2	1
21	69	0	3	140	239	0	1	151	0	1.8	2	2	2	1
22	59	1	0	135	234	0	1	161	0	0.5	1	0	3	1

Code:

```
from google.colab import files
uploaded = files.upload()
```

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

```
heart = pd.read_csv('heart.csv')
```

```
import io
!pip install pgmpy
```

```
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator, BayesianEstimator
```

heart.columns

```
model = BayesianModel([('age','trestbps'),('age','fbs'),('sex','trestbps'),('exang','trestbps'),('trestbps','target'),('fbs','target'),
('target','restecg'),('target','thalach'),('target','chol')])
```

```
model.fit(heart,estimator=MaximumLikelihoodEstimator)
```

```
print(model.get_cpds('age'))
```

Output:

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

Saving heart.csv to heart (1).csv

Requirement already satisfied: pgmpy in /usr/local/lib/python3.7/dist-packages (0.1.15)

Requirement already satisfied: networkx in /usr/local/lib/python3.7/dist-packages (from pgmpy) (2.5.1)

Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from pgmpy) (1.4.1)

Requirement already satisfied: scikit-learn in /usr/local/lib/python3.7/dist-packages (from pgmpy) (0.22.2)

Requirement already satisfied: statsmodels in /usr/local/lib/python3.7/dist-packages (from pgmpy) (0.10.2)

Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from pgmpy) (1.1.5)

Requirement already satisfied: pyparsing in /usr/local/lib/python3.7/dist-packages (from pgmpy) (2.4.7)

Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from pgmpy) (1.19.5)

Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (from pgmpy) (1.9.0+cu102)

Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from pgmpy) (4.41.1)

Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages (from pgmpy) (1.0.1)

Requirement already satisfied: decorator<5,>=4.3 in /usr/local/lib/python3.7/dist-packages (from networkx->pgmpy)

Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages (from pandas->pgmpy) (2)

Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-packages (from pandas->six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.7.

Requirement already satisfied: patsy>=0.4.0 in /usr/local/lib/python3.7/dist-packages (from statsmodels->pgmpy)

Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from torch->pgmpy)

```
+-----+
| age(29) | 0.00330033 |
+-----+

| age(34) | 0.00660066 |
+-----+
| age(35) | 0.0132013 |
+-----+
| age(37) | 0.00660066 |
+-----+
| age(38) | 0.00990099 |
+-----+
| age(39) | 0.0132013 |
+-----+
| age(40) | 0.00990099 |
+-----+
| age(41) | 0.0330033 |
+-----+
| age(42) | 0.0264026 |
+-----+
| age(43) | 0.0264026 |
+-----+
| age(44) | 0.0363036 |
+-----+
| age(45) | 0.0264026 |
+-----+
| age(46) | 0.0231023 |
+-----+
```

Colab link: https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=cpb_f_2SDYCT

Practical 6

Aim: To apply Markov Property to generate Donald's Trump's speech by considering each word used in the speech and for each word, create a dictionary of words that are used next.

Theory:

A Markov model is a stochastic method for randomly changing systems where it is assumed that future states do not depend on past states. These models show all possible states as well as the transitions, rate of transitions and probabilities between them.

For instance, Hidden Markov Models are similar to Markov chains, but they have a few hidden states. Since they're hidden, you can't see them directly in the chain, only through the observation of another process that depends on it.

A Markov chain is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, like the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state.

Code:

```
import numpy as np
from google.colab import files
files.upload()
trump = open('DonaldTs.txt', encoding='utf8').read()
#display the data

print(trump)

corpus = trump.split()

#Display the corpus
print(corpus)
def make_pairs(corpus):
    for i in range(len(corpus) - 1):
        yield (corpus[i], corpus[i + 1])

pairs = make_pairs(corpus)
word_dict={}
for word_1, word_2 in pairs:
    if word_1 in word_dict.keys():
        word_dict[word_1].append(word_2)
    else:
        word_dict[word_1] = [word_2]

#randomly pick the first word
```

```
first_word = np.random.choice(corpus)
```

```
#Pick the first word as a capitalized word so that the picked word is not taken from in between a sentence
```

```
while first_word.islower():
```

```
#Start the chain from the picked word
```

```
chain = [first_word]
```

```
#Initialize the number of stimulated words
```

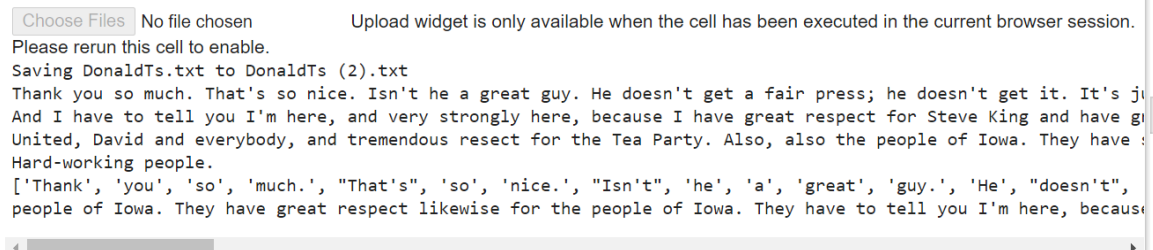
```
n_words = 60
```

```
for i in range(n_words):
```

```
chain.append(np.random.choice(word_dict[chain[-1]]))
```

```
print(" ".join(chain))
```

Output:



Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

Saving DonaldTs.txt to DonaldTs (2).txt

Thank you so much. That's so nice. Isn't he a great guy. He doesn't get a fair press; he doesn't get it. It's j

And I have to tell you I'm here, and very strongly here, because I have great respect for Steve King and have gr

United, David and everybody, and tremendous respect for the Tea Party. Also, also the people of Iowa. They have :

Hard-working people.

['Thank', 'you', 'so', 'much.', 'That's', 'so', 'nice.', 'Isn't', 'he', 'a', 'great', 'guy.', 'He', 'doesn't',

people of Iowa. They have great respect likewise for the people of Iowa. They have to tell you I'm here, because

Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=PkOnDo-s444h>

Practical 7

Aim: Prediction Algorithm - Use of different packages on dataset of Cat and Non-Cat images

Theory:

Predictive analytics in the Trendskout AI Platform Prediction through machine learning or deep learning can be done in a number of different ways, depending on the underlying algorithm that is used. As the name suggests, predictive models are designed to predict unknown values, properties or events. If the predictions from the algorithm proved to be accurate, the algorithm could theoretically be used in the future to identify people at high risk of suicide, and deliver targeted programs to them. That would be a very good thing. Predictive algorithms are everywhere.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
%matplotlib inline
def load_dataset():
    train_dataset = h5py.File('train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

    test_dataset = h5py.File('test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

    classes = np.array(test_dataset["list_classes"][:]) # the list of classes

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes

train_set_x_orig, train_set_y, test_set_x_orig, test_set_y_orig, classes = load_dataset()
#index = 26
#plt.imshow(train_set_x_orig[index])
#print ("y = " + str(train_set_y[:, index]) + ", it's a '" + classes[np.squeeze(train_set_y[:, index
])).decode("utf-8") + "' picture.")

index= 25
plt.imshow(test_set_x_orig[index])
```

```
print ("y = " + str(test_set_y[:, index]) + ", it's a " + classes[np.squeeze(test_set_y[:, index]).  
decode("utf-8") + " picture.")
```

```
#plt.imshow(train_set_y_orig[index])  
#print ("y = " + str(train_set_y[:, index]) + ", it's a " + classes[np.squeeze(train_set_y[:, index  
)].decode("utf-8") + " picture.")
```

```
import numpy as np  
import matplotlib.pyplot as plt  
import h5py  
import scipy  
from PIL import Image  
from scipy import ndimage  
#from lr_utils import load_dataset
```

```
%matplotlib inline
```

```
def load_dataset():  
    train_dataset = h5py.File('train_catvnoncat.h5', "r")  
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features  
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels  
  
    test_dataset = h5py.File('test_catvnoncat.h5', "r")  
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features  
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels  
  
    classes = np.array(test_dataset["list_classes"][:]) # the list of classes  
  
    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))  
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))  
  
    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes
```

```
#Data will be loaded from the test_catvnoncat.h5 and train_catvnoncat.h5 files  
#The load_dataset function below is responsible for loading the above mentioned data files.  
#lr_utils file includes the function load_dataset()
```

```
# Loading the data (cat/non-cat)
```

```
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y_orig, classes = load_dataset()
```

```
# We added "_orig" at the end of image datasets (train and test) because we are going to prepr  
ocess them. After preprocessing, we will end up with train_set_x and test_set_x (the labels tra  
in_set_y and test_set_y don't need any preprocessing).
```

```
# Each line of your train_set_x_orig and test_set_x_orig is an array representing an image. Y  
ou can visualize an example by running the following code. Feel free also to change the `inde  
x` value and re-run to see other images.
```

```
# Example of a picture
```

```
#change the index value below to check if the image at that particular index is cat or non cat
```

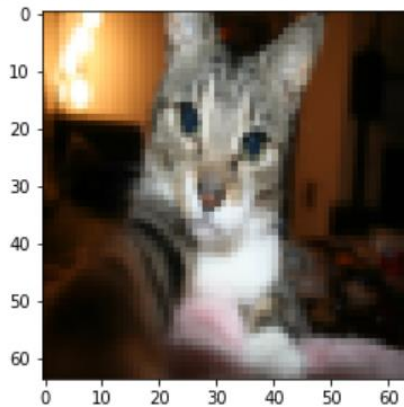


```
index = 26
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a " + classes[np.squeeze(train_set_y[:, index])].decode("utf-8") + " picture.")

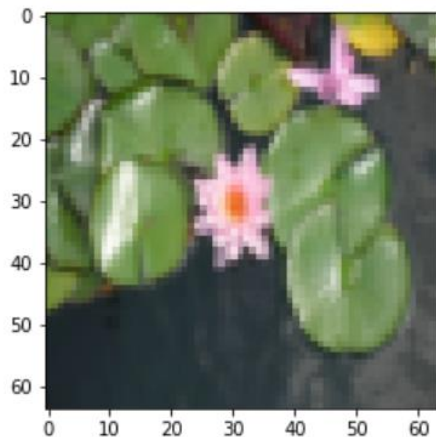
#plt.imshow(train_set_y_orig[index])
#print ("y = " + str(train_set_y[:, index]) + ", it's a " + classes[np.squeeze(train_set_y[:, index])].decode("utf-8") + " picture.")
```

Output:

↳ y = [1], it's a 'cat' picture.



y = [0], it's a 'non-cat' picture.



```
import numpy as np
import h5py
```

```
# Loading the data (cat/non-cat)
```

```
train_dataset = h5py.File('train_catvnoncat.h5', "r")
train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # train set features
train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # train set labels

test_dataset = h5py.File('test_catvnoncat.h5', "r")
test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # test set features
test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # test set labels
```

```
classes = np.array(test_dataset["list_classes"][:]) # the list of classes

train_set_y = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
test_set_y = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

print ("Dataset dimensions:")
print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
↳ Dataset dimensions:
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

Reshape the training and test examples

```
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

```
↳ train_set_x_flatten shape: (12288, 209)
   train_set_y shape: (1, 209)
   test_set_x_flatten shape: (12288, 50)
   test_set_y shape: (1, 50)
   sanity check after reshaping: [17 31 56 22 33]
```

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
from sklearn.linear_model import LogisticRegression
```

```
lr = LogisticRegression(C=1000.0, random_state=0)
```

```
lr.fit(train_set_x.T, train_set_y.T.ravel())
```

```
lr.coef_.shape
```

```
lr.coef_
```

```
lr.intercept_
```

```
Y_prediction = lr.predict(test_set_x.T)
```

```
Y_prediction.shape
```

```
print("test accuracy: { } %".format(100 - np.mean(np.abs(Y_prediction - test_set_y)) * 100))
```

```
test accuracy: 70.0 %
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

Increase the number of iterations (max_iter) or scale the data as shown in:

<https://scikit-learn.org/stable/modules/preprocessing.html>
Please also refer to the documentation for alternative solver options:
https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)

```
[ ] lr.coef_.shape
```

```
(1, 12288)
```

```
[ ] lr.coef_
```

```
array([[ 0.07723427, -0.10503755, -0.01932438, ..., -0.05192317,
        -0.11902044,  0.17739674]])
```

```
[ ] lr.intercept_
```

```
array([-0.1060208])
```

```
[ ] Y_prediction = lr.predict(test_set_x.T)
Y_prediction.shape
```

```
[ ] print("test accuracy: { } %".format(100 - np.mean(np.abs(Y_prediction - test_set_y)) * 100))
```

```
test accuracy: 70.0 %
```

Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=vUMLRIoWD0hb>

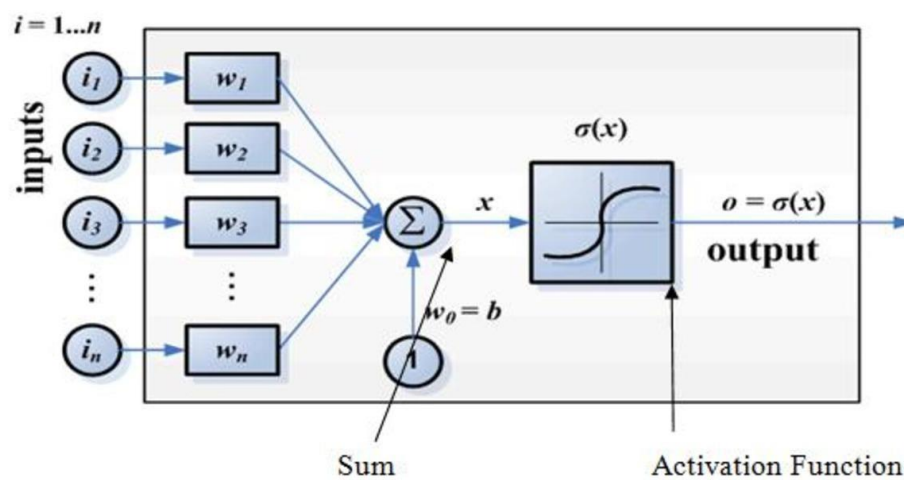
Practical 8

Aim: Neural Representation of AND and OR Logic Gates Perceptron.

Theory:

A Perceptron is an Artificial Neuron. It is the simplest possible Neural Network. Neural Networks are the building blocks of Artificial Intelligence. Perceptron is one of the earliest—and most elementary—artificial neural network models. The perceptron is extremely simple by modern deep learning model standards. However the concepts utilised in its design apply more broadly to sophisticated deep network architectures. The perceptron is a supervised learning binary classification algorithm, originally developed by Frank Rosenblatt in 1957. It categorises input data into one of two separate states based a training procedure carried out on prior input data.

Graph:



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([1,1])
    b=-0.5
    return perceptronModel(x,w,b)

def AND_logicFunction(x):
```

```
w = np.array([1, 1])
bAND = -1.5
return perceptronModel(x, w, bAND)

test1=np.array([0,0])
test2=np.array([0,1])
test3=np.array([1,0])
test4=np.array([1,1])
print("OR({}, {})={}".format(0,0,OR_logicfunction(test1)))
print("OR({}, {})={}".format(0,1,OR_logicfunction(test2)))
print("OR({}, {})={}".format(1,0,OR_logicfunction(test3)))
print("OR({}, {})={}".format(1,1,OR_logicfunction(test4)))

print("AND({}, {})={}".format(0,1,AND_logicFunction(test1)))
print("AND({}, {})={}".format(1,1,AND_logicFunction(test2)))
print("AND({}, {})={}".format(0,0,AND_logicFunction(test3)))
print("AND({}, {})={}".format(1,0,AND_logicFunction(test4)))
```

Output:

```
OR(0,0)=0
OR(0,1)=1
OR(1,0)=1
OR(1,1)=1
AND(0,1)=0
AND(1,1)=0
AND(0,0)=0
AND(1,0)=1
```

Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=3TiXACHqOwMy&line=1&uniquifier=1>

Practical 9

Aim: Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set.

Theory:

The k-nearest neighbors (KNN) algorithm is a simple, supervised machine learning algorithm that can be used to solve both classification and regression problems. It's easy to implement and understand, but has a major drawback of becoming significantly slower as the size of that data in use grows.

It is also flexible because the number of K neighbors and the distance between them are chosen for what is appropriate for the data being analyzed. It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data. We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

The Iris Dataset contains four features (length and width of sepals and petals) of 50 samples of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). These measures were used to create a linear discriminant model to classify the species. The dataset is often used in data mining, classification and clustering examples and to test algorithms.



Iris Versicolor



Iris Setosa



Iris Virginica

Code:

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

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
```

```
# Assign column names to the dataset
```


```
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
```

```
# Read dataset to pandas dataframe
```

```
dataset = pd.read_csv(url, names=names)
```

```
dataset.head()
```

Output:



	sepal-length	sepal-width	petal-length	petal-width	Class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
X = dataset.iloc[:, :-1].values
```

```
y = dataset.iloc[:, 4].values
```

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
```

```
scaler.fit(X_train)
```

```
X_train = scaler.transform(X_train)
```

```
X_test = scaler.transform(X_test)
```

```
from sklearn.neighbors import KNeighborsClassifier
```

```
classifier = KNeighborsClassifier(n_neighbors=5)
```

```
classifier.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',  
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,  
                     weights='uniform')
```

```
y_pred = classifier.predict(X_test)
```

```
from sklearn.metrics import classification_report, confusion_matrix
```

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

```
print(classification_report(y_test, y_pred))
```

Output:


```
[[11  0  0]
 [ 0  8  0]
 [ 0  1 10]]
```

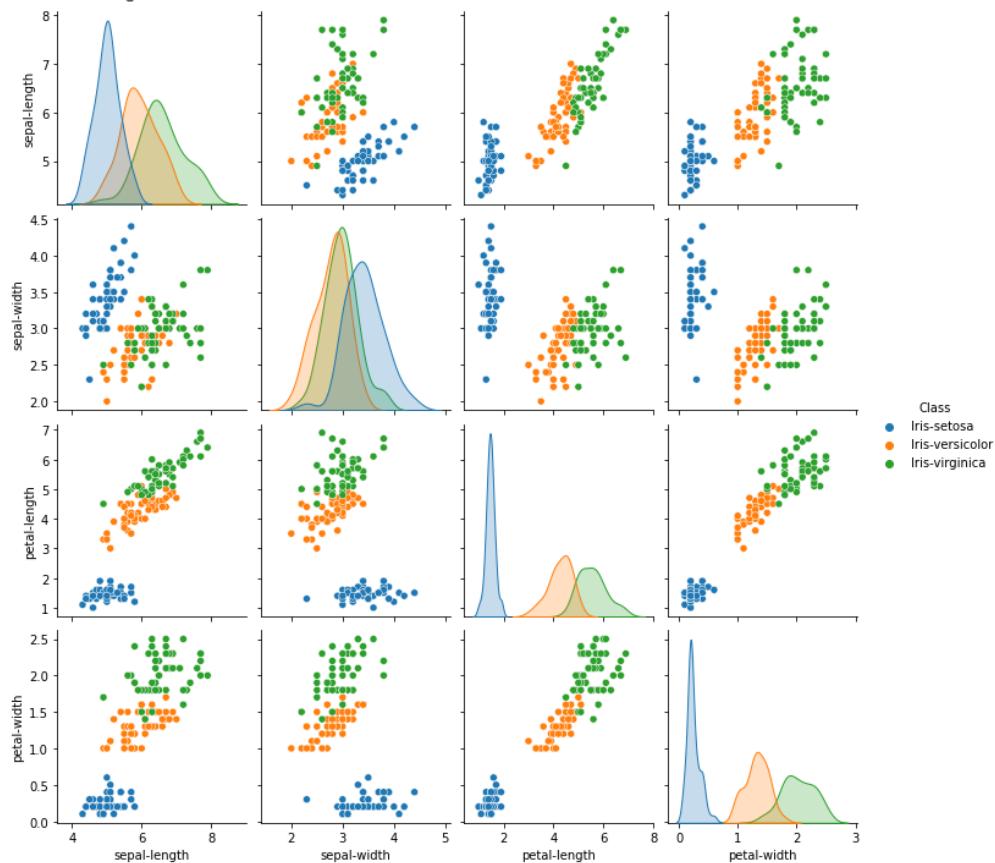
	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	11
Iris-versicolor	0.89	1.00	0.94	8
Iris-virginica	1.00	0.91	0.95	11
accuracy			0.97	30
macro avg	0.96	0.97	0.96	30
weighted avg	0.97	0.97	0.97	30

```
[10] dataset.columns
```

```
Index(['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class'], dtype='object')
```

```
[9] sns.pairplot(dataset,hue="Class")
```

```
<seaborn.axisgrid.PairGrid at 0x7fcc46dbf1d0>
```



Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb051TteRVziiPhTm8OGM#scrollTo=0xLvGk88Djli>

Practical 10

Aim: Implementation of basic neural network model with 4 activation functions on Pima Indians onset of diabetes dataset.

Theory:

The Role of Artificial Intelligence made it possible for machines to learn from experience to perform tasks more efficiently. The Artificial neural network is one of its advancements which is inspired by the structure of the human brain that helps computers and machines more like a human.

This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years.

As such, it is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks that expect numerical input and output values, and ideal for our first neural network in Keras.

Dataset:

1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0
5	0	137	40	35	168	43.1	2.288	33	1
6	5	116	74	0	0	25.6	0.201	30	0
7	3	78	50	32	88	31	0.248	26	1
8	10	115	0	0	0	35.3	0.134	29	0
9	2	197	70	45	543	30.5	0.158	53	1
10	8	125	96	0	0	0	0.232	54	1
11	4	110	92	0	0	37.6	0.191	30	0
12	10	168	74	0	0	38	0.537	34	1
13	10	139	80	0	0	27.1	1.441	57	0
14	1	189	60	23	846	30.1	0.398	59	1
15	5	166	72	19	175	25.8	0.587	51	1
16	7	100	0	0	0	30	0.484	32	1
17	0	118	84	47	230	45.8	0.551	31	1
18	7	107	74	0	0	29.6	0.254	31	1
19	1	103	30	38	83	43.3	0.183	33	0
20	1	115	70	30	96	34.6	0.529	32	1
21	3	126	88	41	235	39.3	0.704	27	0
22	8	99	84	0	0	35.4	0.388	50	0

Code:

```
from google.colab import files
files.upload()
```

```
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
```

```
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
...
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
```

Output:

```
Epoch 6/150
77/77 [=====] - 0s 2ms/step - loss: 0.6810 - accuracy: 0.6549
Epoch 7/150
77/77 [=====] - 0s 2ms/step - loss: 0.6736 - accuracy: 0.6549
Epoch 8/150
77/77 [=====] - 0s 1ms/step - loss: 0.6666 - accuracy: 0.6589
Epoch 9/150
77/77 [=====] - 0s 2ms/step - loss: 0.6607 - accuracy: 0.6628
Epoch 10/150
77/77 [=====] - 0s 2ms/step - loss: 0.6557 - accuracy: 0.6641
Epoch 11/150
77/77 [=====] - 0s 1ms/step - loss: 0.6519 - accuracy: 0.6641
Epoch 12/150
77/77 [=====] - 0s 2ms/step - loss: 0.6489 - accuracy: 0.6654
Epoch 13/150
77/77 [=====] - 0s 1ms/step - loss: 0.6470 - accuracy: 0.6628
Epoch 14/150
77/77 [=====] - 0s 1ms/step - loss: 0.6438 - accuracy: 0.6615
Epoch 15/150
77/77 [=====] - 0s 2ms/step - loss: 0.6419 - accuracy: 0.6602
Epoch 16/150
77/77 [=====] - 0s 2ms/step - loss: 0.6399 - accuracy: 0.6615
Epoch 17/150
77/77 [=====] - 0s 2ms/step - loss: 0.6385 - accuracy: 0.6641
Epoch 18/150
77/77 [=====] - 0s 1ms/step - loss: 0.6367 - accuracy: 0.6667
Epoch 19/150
77/77 [=====] - 0s 2ms/step - loss: 0.6359 - accuracy: 0.6628
```

```
[ ] # evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

24/24 [=====] - 0s 1ms/step - loss: 0.4778 - accuracy: 0.7708
Accuracy: 77.08

```
[ ] model.fit(X, y, epochs=150, batch_size=10, verbose=0)
# evaluate the keras model
_, accuracy = model.evaluate(X, y, verbose=0)
```

```
# make probability predictions with the model
predictions = model.predict(X)
# round predictions
rounded = [round(x[0]) for x in predictions]
```

```
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
# make class predictions with the model

# summarize the first 5 cases
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Output:

```
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 0 (expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 0 (expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 0 (expected 1)
```

Colab link: <https://colab.research.google.com/drive/1kx2fQ-NvedFtb051TteRVziiPhTm8OGM#scrollTo=J4ePRdp2PFJ6>

Name: Zeenat

Class: TYIT

Roll no: 578